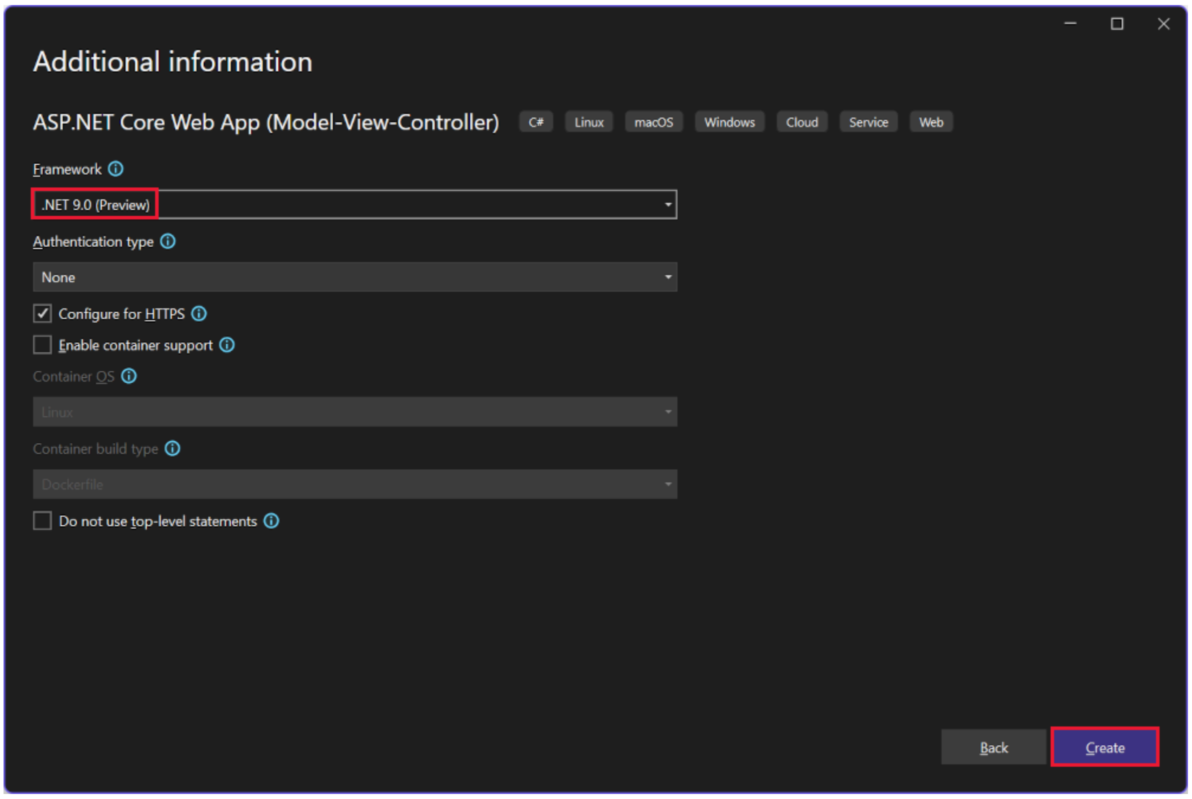

Additional information

ASP.NET Core Web App (Model-View-Controller)

Framework: .NET 9.0 (Preview)

Authentication type: None

☑ Configure for HTTPS

☐ Enable container support

Container OS: Linux

Container build type: Dockerfile

☐ Do not use top-level statements

For more information, including alternative approaches to create the project, see Create a new project in Visual Studio.

Visual Studio uses the default project template for the created MVC project. The created project:

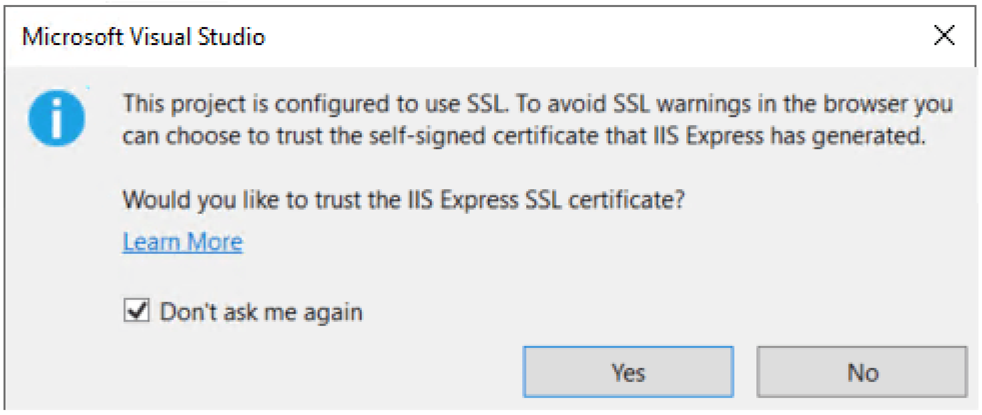
- Is a working app.
- Is a basic starter project.

Run the app

Visual Studio

- Press `Ctrl`+`F5` to run the app without the debugger.

  Visual Studio displays the following dialog when a project is not yet configured to use SSL:


Microsoft Visual Studio

This project is configured to use SSL. To avoid SSL warnings in the browser you can choose to trust the self-signed certificate that IIS Express has generated.

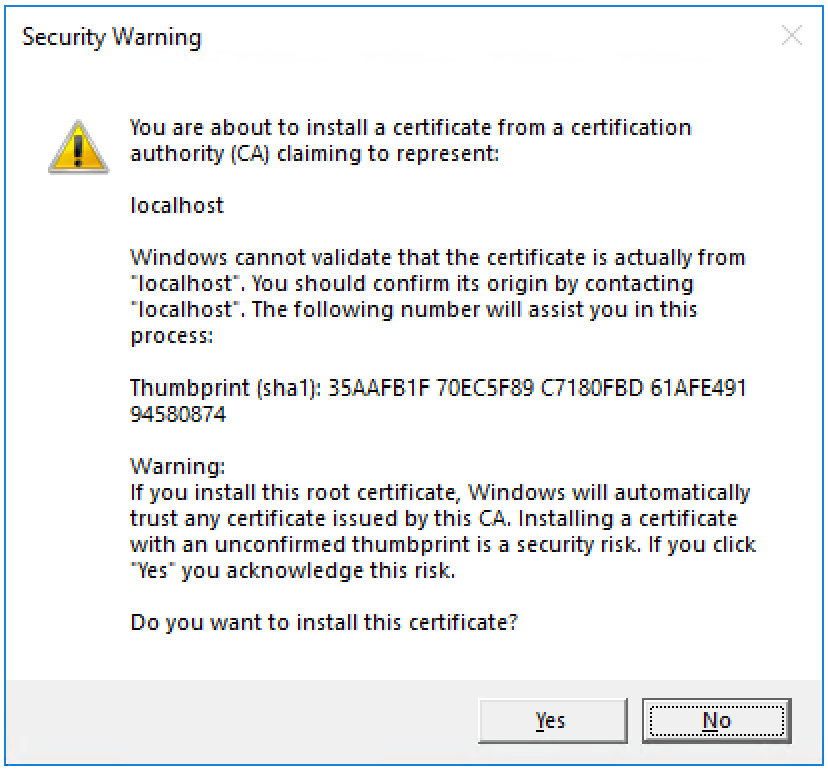
Would you like to trust the IIS Express SSL certificate?

Learn More

☑ Don't ask me again

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

The following dialog is displayed:


Security Warning

You are about to install a certificate from a certification authority (CA) claiming to represent:

localhost

Windows cannot validate that the certificate is actually from "localhost". You should confirm its origin by contacting "localhost". The following number will assist you in this process:

Thumbprint (sha1): 35AAFB1F 70EC5F89 C7180FBD 61AFE491 94580874

Warning:
If you install this root certificate, Windows will automatically trust any certificate issued by this CA. Installing a certificate with an unconfirmed thumbprint is a security risk. If you click "Yes" you acknowledge this risk.

Do you want to install this certificate?

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

For information on trusting the Firefox browser, see Firefox SEC_ERROR_INADEQUATE_KEY_USAGE certificate error.

Visual Studio runs the app and opens the default browser.

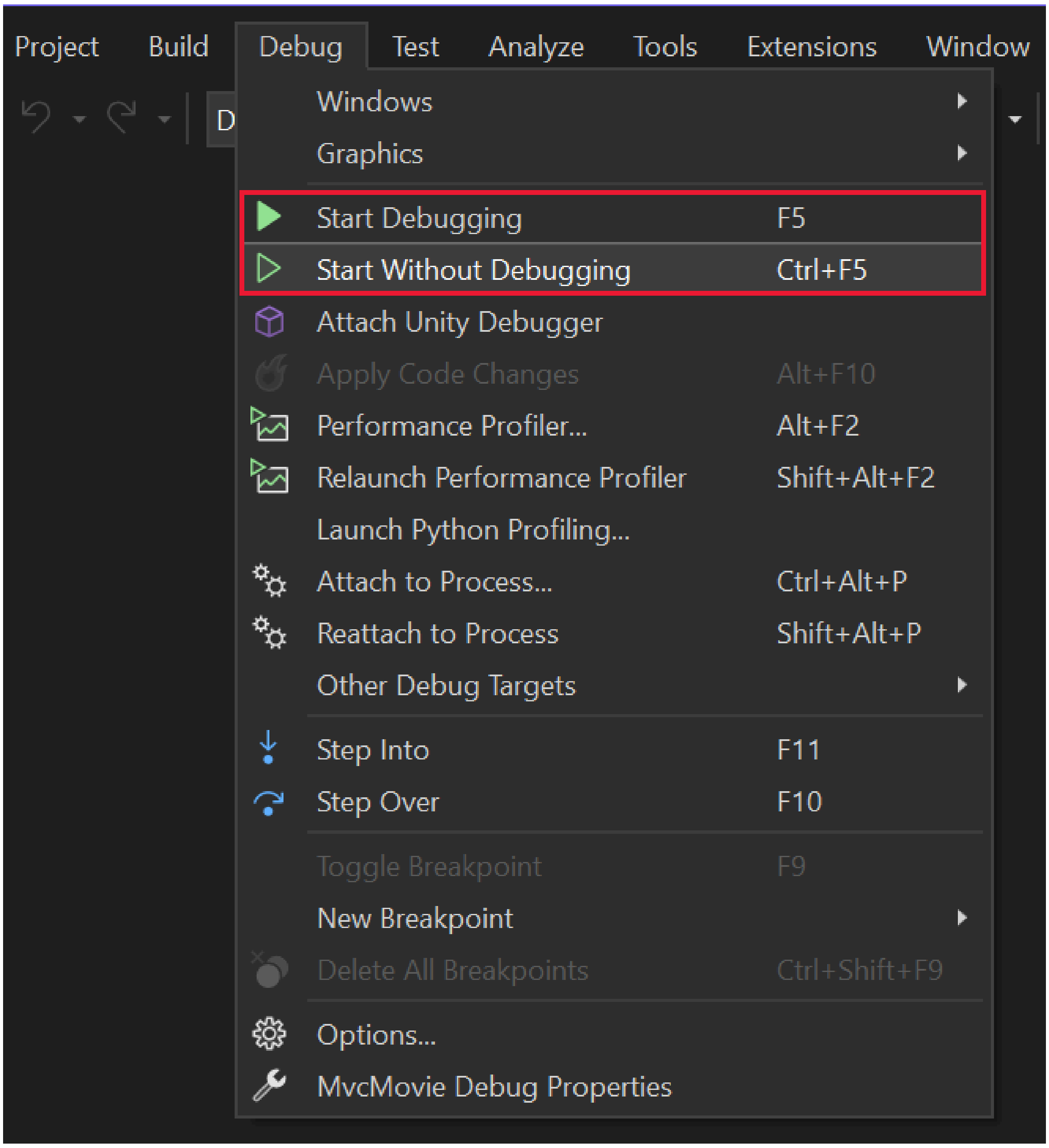
The address bar shows `localhost:<port#>` and not something like `example.com`. The standard hostname for your local computer is `localhost`. When Visual Studio creates a web project, a random port is used for the web server.

Launching the app without debugging by pressing `Ctrl`+`F5` allows you to:

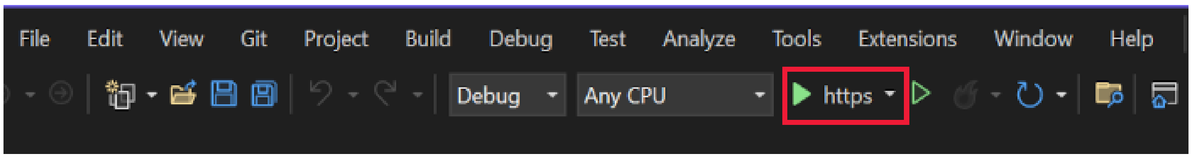
- Make code changes.

- Save the file.
- Quickly refresh the browser and see the code changes.

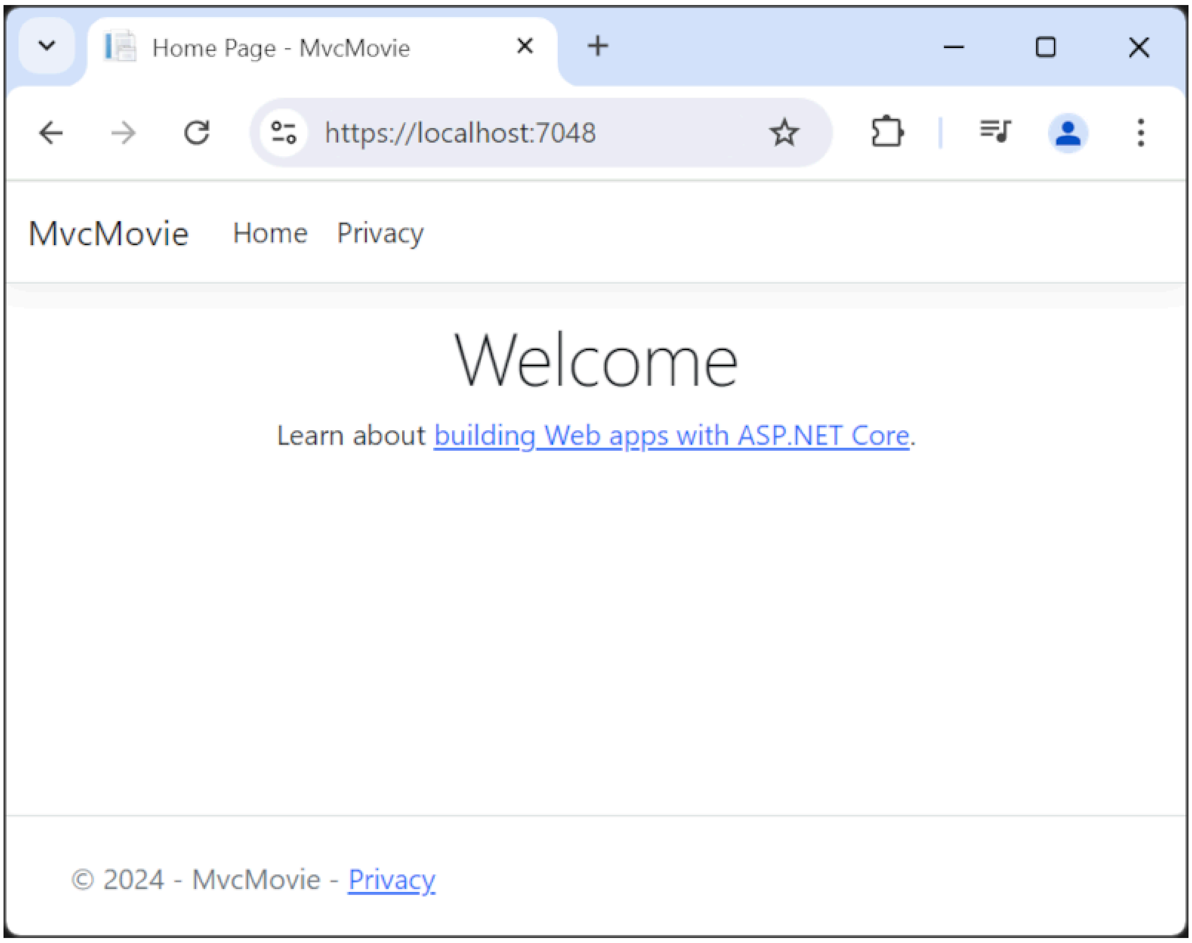
You can launch the app in debug or non-debug mode from the **Debug** menu:


Project
Build
Debug
Test
Analyze
Tools
Extensions
Window
Windows
Graphics
Start Debugging F5
Start Without Debugging Ctrl+F5
Attach Unity Debugger
Apply Code Changes Alt+F10
Performance Profiler... Alt+F2
Relaunch Performance Profiler Shift+Alt+F2
Launch Python Profiling...
Attach to Process... Ctrl+Alt+P
Reattach to Process Shift+Alt+P
Other Debug Targets
Step Into F11
Step Over F10
Toggle Breakpoint F9
New Breakpoint
Delete All Breakpoints Ctrl+Shift+F9
Options...
MvcMovie Debug Properties

You can debug the app by selecting the **https** button in the toolbar:



The following image shows the app:


Home Page - MvcMovie

https://localhost:7048

MvcMovie Home Privacy

Welcome

Learn about building Web apps with ASP.NET Core.

© 2024 - MvcMovie - Privacy

- Close the browser window. Visual Studio will stop the application.

Visual Studio

Visual Studio help

- Learn to debug C# code using Visual Studio
- Introduction to the Visual Studio IDE

In the next tutorial in this series, you learn about MVC and start writing some code.

Next: Add a controller

Part 2, add a controller to an ASP.NET Core MVC app

Article • 07/30/2024

Important

This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

For the current release, see the .NET 9 version of this article.

By Rick Anderson

The Model-View-Controller (MVC) architectural pattern separates an app into three main components: **M**odel, **V**iew, and **C**ontroller. The MVC pattern helps you create apps that are more testable and easier to update than traditional monolithic apps.

MVC-based apps contain:

- **M**odels: Classes that represent the data of the app. The model classes use validation logic to enforce business rules for that data. Typically, model objects retrieve and store model state in a database. In this tutorial, a `Movie` model retrieves movie data from a database, provides it to the view or updates it. Updated data is written to a database.
- **V**iews: Views are the components that display the app's user interface (UI). Generally, this UI displays the model data.
- **C**ontrollers: Classes that:
  - Handle browser requests.
  - Retrieve model data.
  - Call view templates that return a response.

In an MVC app, the view only displays information. The controller handles and responds to user input and interaction. For example, the controller handles URL segments and query-string values, and passes these values to the model. The model might use these values to query the database. For example:

- `https://localhost:5001/Home/Privacy`: specifies the `Home` controller and the `Privacy` action.

- `https://localhost:5001/Movies/Edit/5`: is a request to edit the movie with ID=5 using the `Movies` controller and the `Edit` action, which are detailed later in the tutorial.

Route data is explained later in the tutorial.

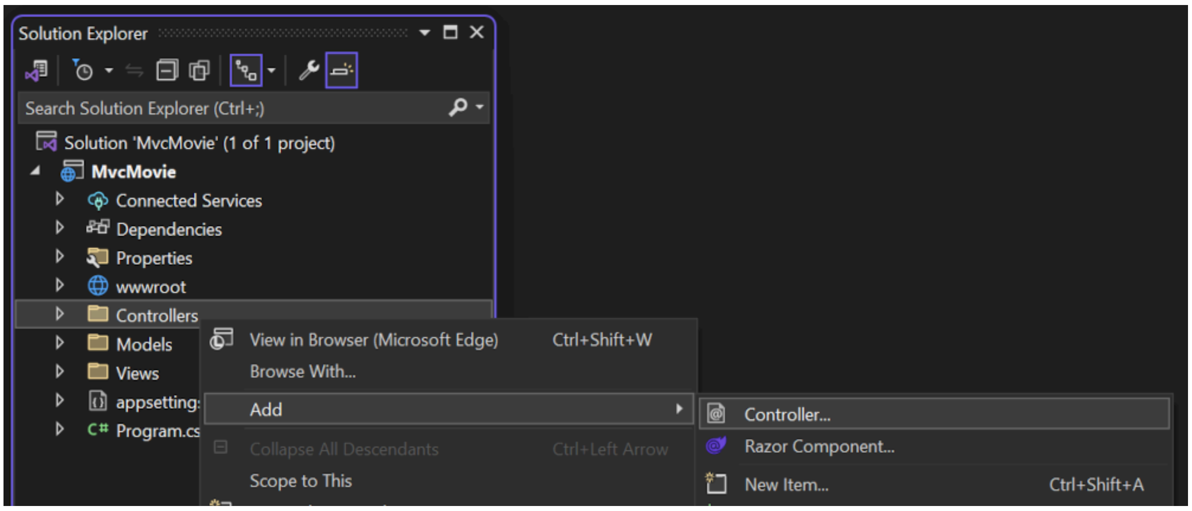
The MVC architectural pattern separates an app into three main groups of components: Models, Views, and Controllers. This pattern helps to achieve separation of concerns: The UI logic belongs in the view. Input logic belongs in the controller. Business logic belongs in the model. This separation helps manage complexity when building an app, because it enables work on one aspect of the implementation at a time without impacting the code of another. For example, you can work on the view code without depending on the business logic code.

These concepts are introduced and demonstrated in this tutorial series while building a movie app. The MVC project contains folders for the *Controllers* and *Views*.

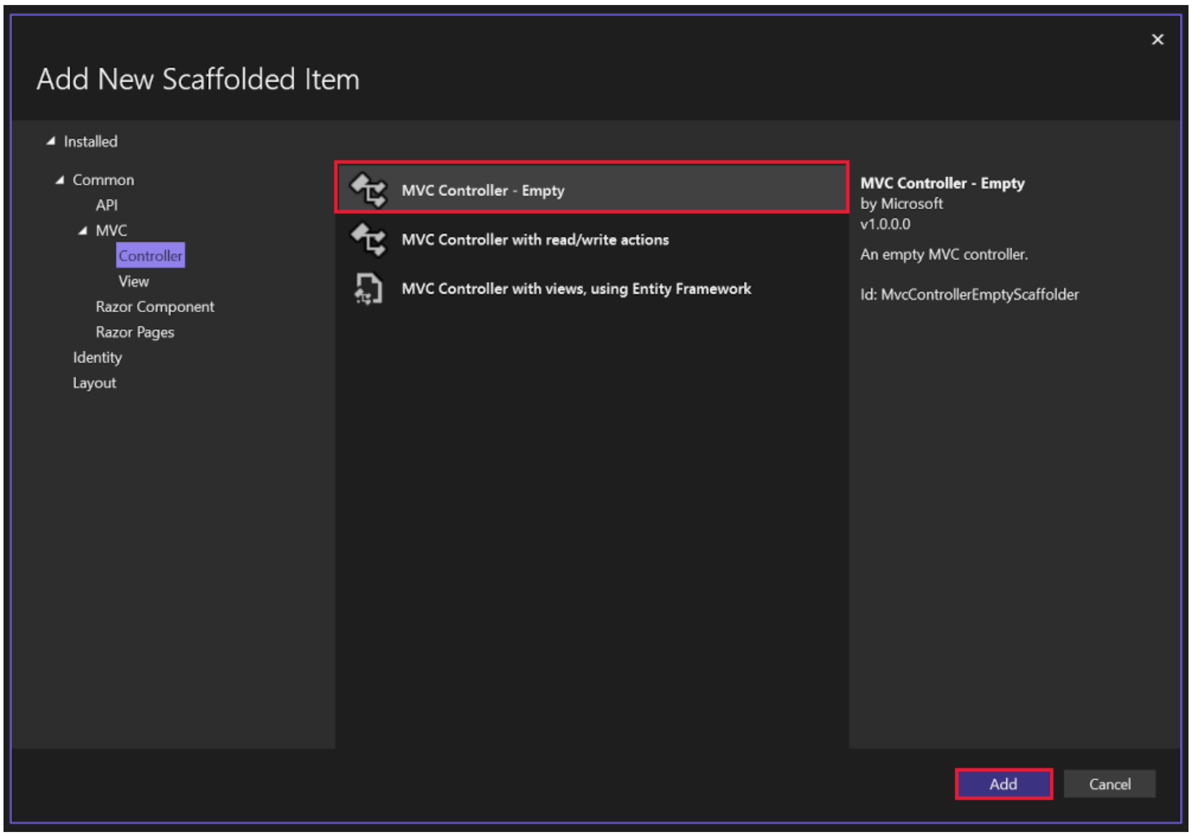
Add a controller

Visual Studio

In **Solution Explorer**, right-click **Controllers** > **Add** > **Controller**.



In the **Add New Scaffolded Item** dialog box, select **MVC Controller - Empty** > **Add**.



MVC Controller - Empty
by Microsoft
v1.0.0.0

An empty MVC controller.

Id: MvcControllerEmptyScaffolder

In the **Add New Item - MvcMovie** dialog, enter `HelloWorldController.cs` and select **Add**.

Replace the contents of `Controllers/HelloWorldController.cs` with the following code:

C#

```csharp
using Microsoft.AspNetCore.Mvc;
using System.Text.Encodings.Web;

namespace MvcMovie.Controllers;

public class HelloWorldController : Controller
{
    //
    // GET: /HelloWorld/
    public string Index()
    {
        return "This is my default action...";
    }
    //
    // GET: /HelloWorld/Welcome/
    public string Welcome()
    {
        return "This is the Welcome action method...";
    }
}
```

Every `public` method in a controller is callable as an HTTP endpoint. In the sample above, both methods return a string. Note the comments preceding each method.

An HTTP endpoint:

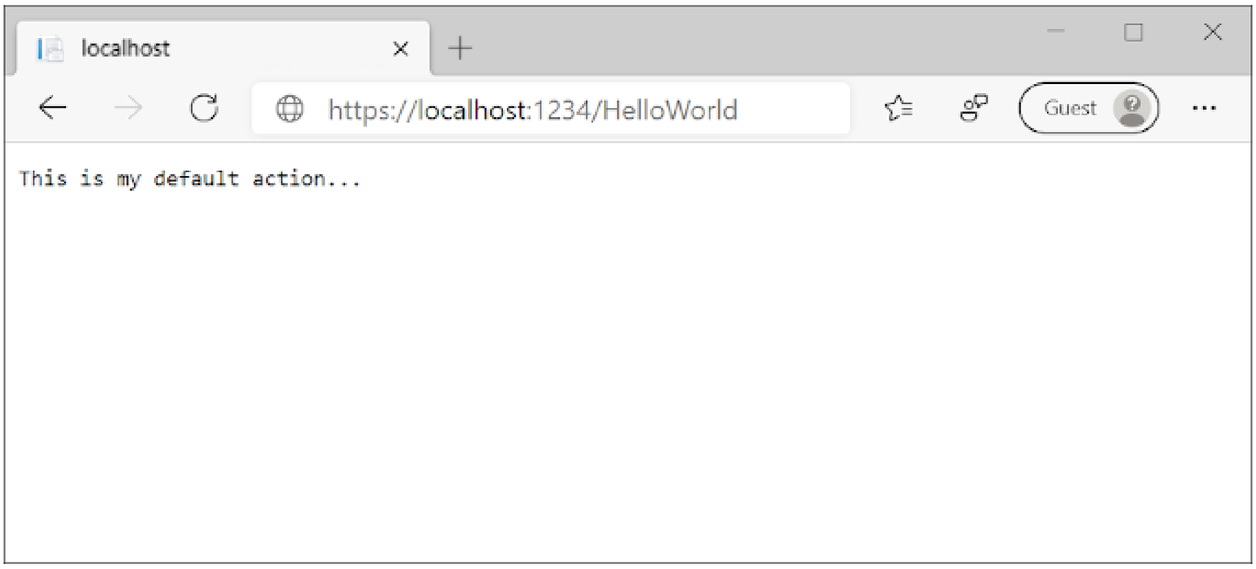
- Is a targetable URL in the web application, such as `https://localhost:5001/HelloWorld`.
- Combines:
  - The protocol used: `HTTPS`.
  - The network location of the web server, including the TCP port: `localhost:5001`.
  - The target URI: `HelloWorld`.

The first comment states this is an HTTP GET method that's invoked by appending `/HelloWorld/` to the base URL.

The second comment specifies an HTTP GET method that's invoked by appending `/HelloWorld/Welcome/` to the URL. Later on in the tutorial, the scaffolding engine is used to generate `HTTP POST` methods, which update data.

Run the app without the debugger by pressing Ctrl+F5 (Windows) or ⌘+F5 (macOS).

Append `/HelloWorld` to the path in the address bar. The `Index` method returns a string.



```
This is my default action...
```

MVC invokes controller classes, and the action methods within them, depending on the incoming URL. The default URL routing logic used by MVC, uses a format like this to determine what code to invoke:

`/[Controller]/[ActionName]/[Parameters]`

The routing format is set in the `Program.cs` file.

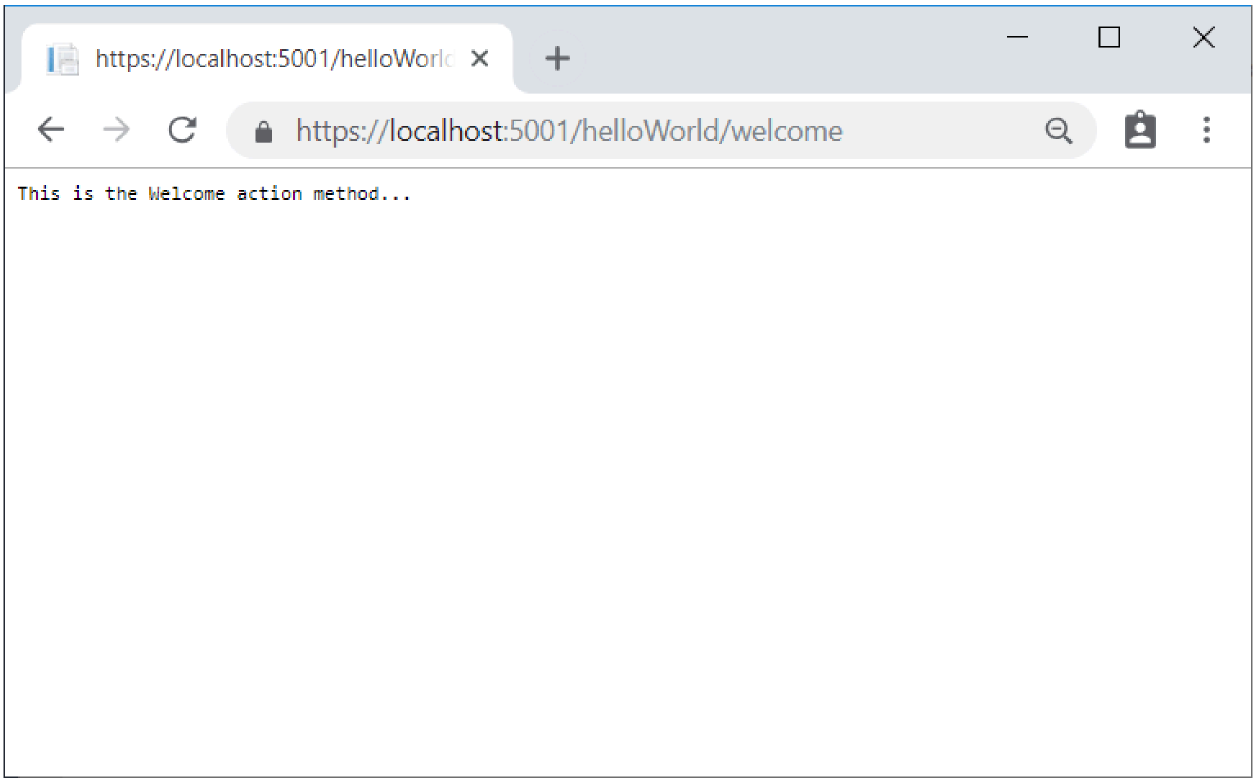
```C#
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
```

When you browse to the app and don't supply any URL segments, it defaults to the "Home" controller and the "Index" method specified in the template line highlighted above. In the preceding URL segments:

- The first URL segment determines the controller class to run. So `localhost:5001/HelloWorld` maps to the **HelloWorld** Controller class.
- The second part of the URL segment determines the action method on the class. So `localhost:5001/HelloWorld/Index` causes the `Index` method of the `HelloWorldController` class to run. Notice that you only had to browse to `localhost:5001/HelloWorld` and the `Index` method was called by default. `Index` is the default method that will be called on a controller if a method name isn't explicitly specified.
- The third part of the URL segment (`id`) is for route data. Route data is explained later in the tutorial.

Browse to: `https://localhost:{PORT}/HelloWorld/Welcome`. Replace `{PORT}` with your port number.

The `Welcome` method runs and returns the string `This is the Welcome action method...`. For this URL, the controller is `HelloWorld` and `Welcome` is the action method. You haven't used the `[Parameters]` part of the URL yet.



```
This is the Welcome action method...
```

Modify the code to pass some parameter information from the URL to the controller. For example, `/HelloWorld/Welcome?name=Rick&numtimes=4`.

Change the `Welcome` method to include two parameters as shown in the following code.

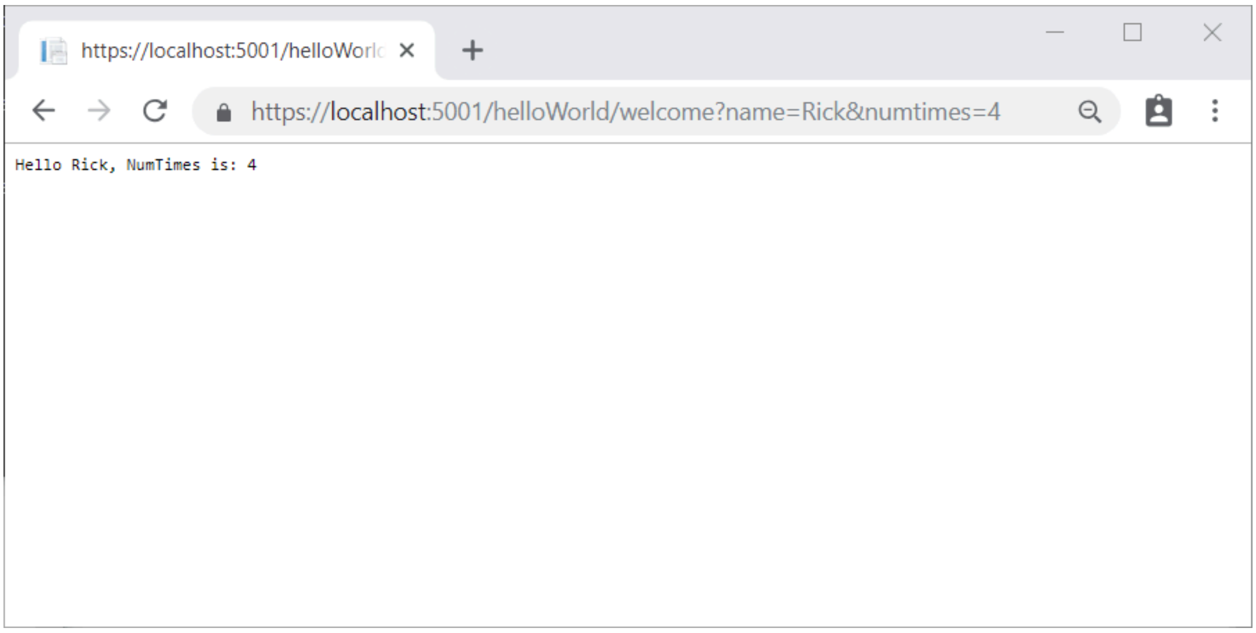
```C#
// GET: /HelloWorld/Welcome/
// Requires using System.Text.Encodings.Web;
public string Welcome(string name, int numTimes = 1)
{
    return HtmlEncoder.Default.Encode($"Hello {name}, NumTimes is:
{numTimes}");
}
```

The preceding code:

- Uses the C# optional-parameter feature to indicate that the `numTimes` parameter defaults to 1 if no value is passed for that parameter.
- Uses `HtmlEncoder.Default.Encode` to protect the app from malicious input, such as through JavaScript.
- Uses Interpolated Strings in `$"Hello {name}, NumTimes is: {numTimes}"`.

Run the app and browse to: `https://localhost:{PORT}/HelloWorld/Welcome?name=Rick&numtimes=4`. Replace `{PORT}` with your port number.

Try different values for `name` and `numtimes` in the URL. The MVC model binding system automatically maps the named parameters from the query string to parameters in the method. See Model Binding for more information.


https://localhost:5001/helloWorld/welcome?name=Rick&numtimes=4

```
Hello Rick, NumTimes is: 4
```

In the previous image:

- The URL segment `Parameters` isn't used.
- The `name` and `numTimes` parameters are passed in the query string.
- The `?` (question mark) in the above URL is a separator, and the query string follows.
- The `&` character separates field-value pairs.

Replace the `Welcome` method with the following code:

C#

```csharp
public string Welcome(string name, int ID = 1)
{
    return HtmlEncoder.Default.Encode($"Hello {name}, ID: {ID}");
}
```

Run the app and enter the following URL: `https://localhost:{PORT}/HelloWorld/Welcome/3?name=Rick`

In the preceding URL:

- The third URL segment matched the route parameter `id`.
- The `Welcome` method contains a parameter `id` that matched the URL template in the `MapControllerRoute` method.
- The trailing `?` starts the query string.

C#

```
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
```

In the preceding example:

- The third URL segment matched the route parameter `id`.
- The `Welcome` method contains a parameter `id` that matched the URL template in the `MapControllerRoute` method.
- The trailing `?` (in `id?`) indicates the `id` parameter is optional.

Previous: Get Started Next: Add a View

Part 3, add a view to an ASP.NET Core MVC app

Article • 07/30/2024

ⓘ **Important**

This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

For the current release, see the .NET 9 version of this article.

By Rick Anderson

In this section, you modify the `HelloWorldController` class to use Razor view files. This cleanly encapsulates the process of generating HTML responses to a client.

View templates are created using Razor. Razor-based view templates:

- Have a `.cshtml` file extension.
- Provide an elegant way to create HTML output with C#.

Currently the `Index` method returns a string with a message in the controller class. In the `HelloWorldController` class, replace the `Index` method with the following code:

```C#
public IActionResult Index()
{
    return View();
}
```

The preceding code:

- Calls the controller's View method.
- Uses a view template to generate an HTML response.

Controller methods:

- Are referred to as *action methods*. For example, the `Index` action method in the preceding code.

- Generally return an IActionResult or a class derived from ActionResult, not a type like `string`.

Add a view

Visual Studio

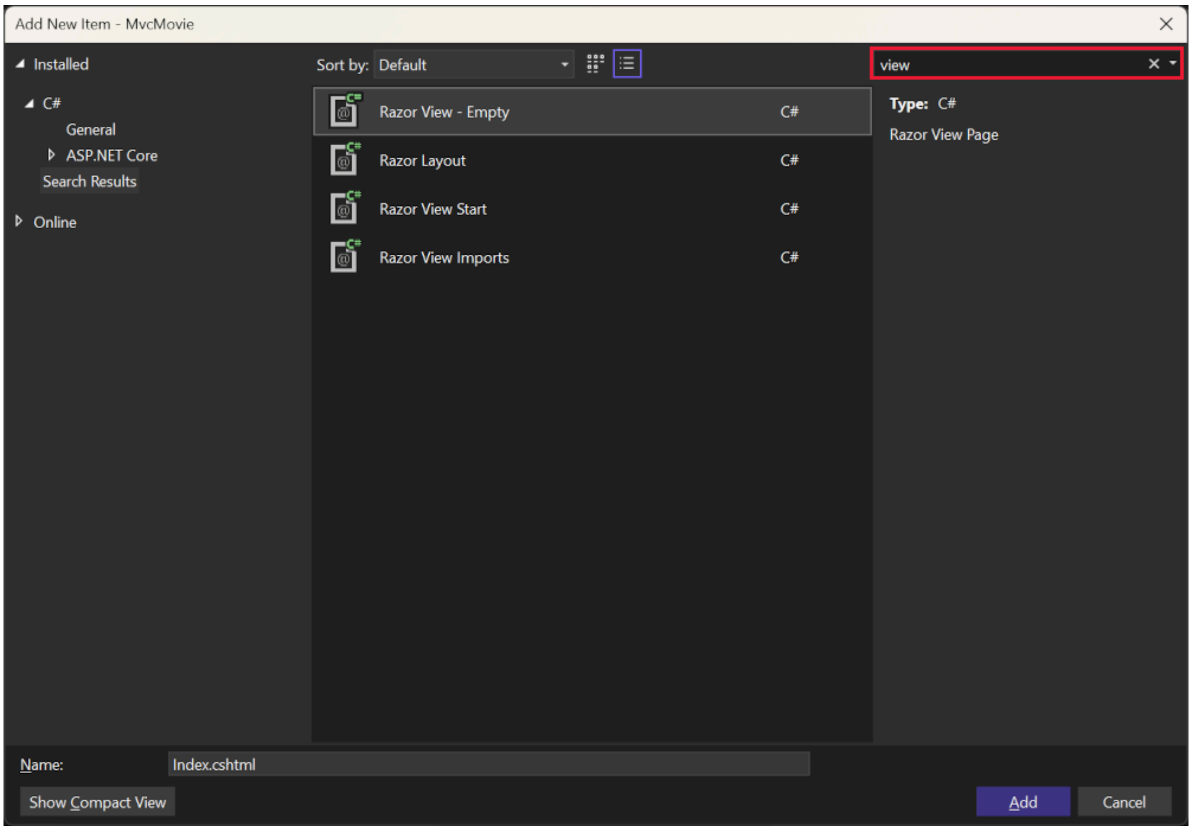
Right-click on the *Views* folder, and then **Add** > **New Folder** and name the folder *HelloWorld*.

Right-click on the *Views/HelloWorld* folder, and then **Add** > **New Item**.

In the **Add New Item** dialog select **Show All Templates**.

In the **Add New Item - MvcMovie** dialog:

- In the search box in the upper-right, enter *view*
- Select **Razor View - Empty**
- Keep the **Name** box value, `Index.cshtml`.
- Select **Add**


Add New Item - MvcMovie

Search: view

Sort by: Default

Razor View - Empty	C#	
Razor Layout	C#	
Razor View Start	C#	
Razor View Imports	C#	

Type: C#
Razor View Page

Name: Index.cshtml

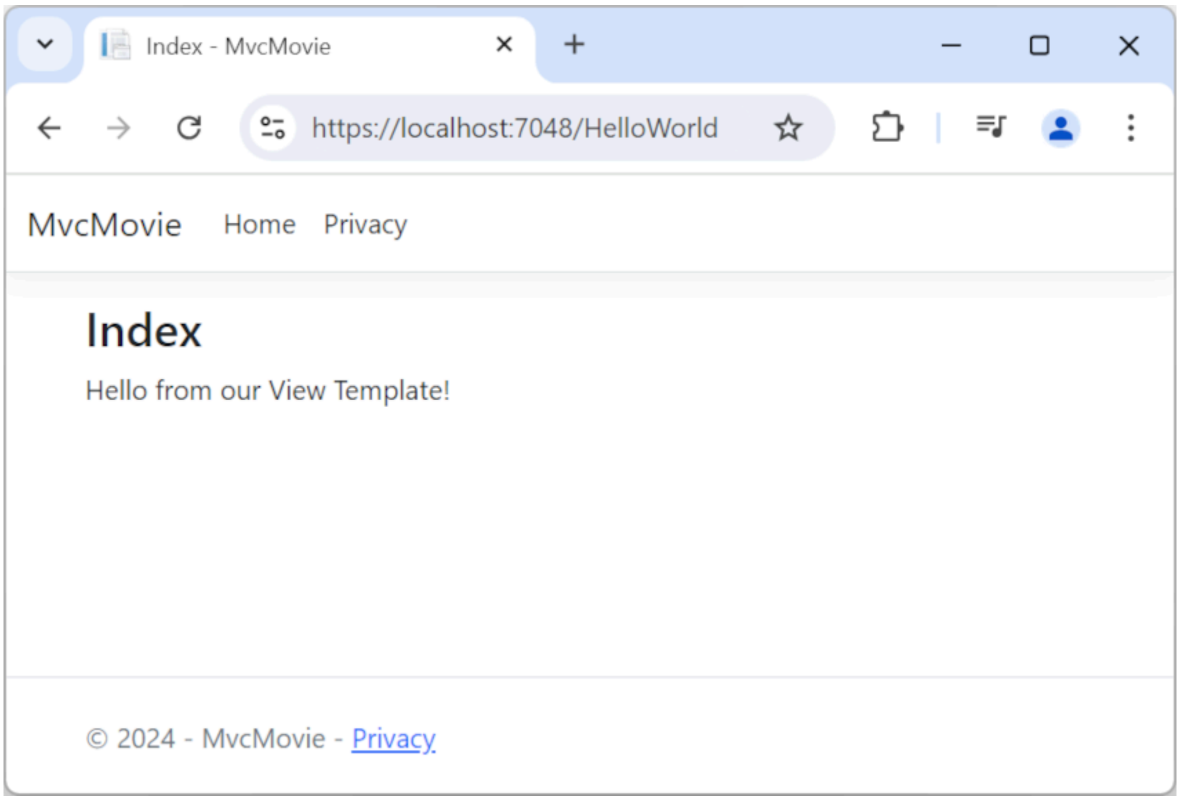
Replace the contents of the `Views/HelloWorld/Index.cshtml` Razor view file with the following:

CSHTML

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

<h2>Index</h2>

<p>Hello from our View Template!</p>
```

Navigate to `https://localhost:{PORT}/HelloWorld`:

- The `Index` method in the `HelloWorldController` ran the statement `return View();`, which specified that the method should use a view template file to render a response to the browser.
- A view template file name wasn't specified, so MVC defaulted to using the default view file. When the view file name isn't specified, the default view is returned. The default view has the same name as the action method, `Index` in this example. The view template `/Views/HelloWorld/Index.cshtml` is used.
- The following image shows the string "Hello from our View Template!" hard-coded in the view:



Index

Hello from our View Template!

© 2024 - MvcMovie - Privacy

Change views and layout pages

Select the menu links **MvcMovie**, **Home**, and **Privacy**. Each page shows the same menu layout. The menu layout is implemented in the `Views/Shared/_Layout.cshtml` file.

Open the `Views/Shared/_Layout.cshtml` file.

Layout templates allow:

- Specifying the HTML container layout of a site in one place.
- Applying the HTML container layout across multiple pages in the site.

Find the `@RenderBody()` line. `RenderBody` is a placeholder where all the view-specific pages you create show up, *wrapped* in the layout page. For example, if you select the **Privacy** link, the `Views/Home/Privacy.cshtml` view is rendered inside the `RenderBody` method.

Change the title, footer, and menu link in the layout file

Replace the content of the `Views/Shared/_Layout.cshtml` file with the following markup. The changes are highlighted:

CSHTML

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>@ViewData["Title"] - Movie App</title>
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
    <link rel="stylesheet" href="~/css/site.css" asp-append-version="true"
/>
    <link rel="stylesheet" href="~/MvcMovie.styles.css" asp-append-
version="true" />
</head>
<body>
    <header>
        <nav class="navbar navbar-expand-sm navbar-toggleable-sm navbar-
light bg-white border-bottom box-shadow mb-3">
            <div class="container-fluid">
                <a class="navbar-brand" asp-area="" asp-controller="Movies"
asp-action="Index">Movie App</a>
                <button class="navbar-toggler" type="button" data-bs-
toggle="collapse" data-bs-target=".navbar-collapse" aria-
controls="navbarSupportedContent"
                        aria-expanded="false" aria-label="Toggle
navigation">
                    <span class="navbar-toggler-icon"></span>
                </button>
                <div class="navbar-collapse collapse d-sm-inline-flex
justify-content-between">
                    <ul class="navbar-nav flex-grow-1">
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
controller="Home" asp-action="Index">Home</a>
                        </li>
                        <li class="nav-item">
                            <a class="nav-link text-dark" asp-area="" asp-
controller="Home" asp-action="Privacy">Privacy</a>
                        </li>
                    </ul>
                </div>
            </div>
        </nav>
    </header>
    <div class="container">
        <main role="main" class="pb-3">
            @RenderBody()
        </main>
    </div>

    <footer class="border-top footer text-muted">
        <div class="container">
            &copy; 2024 - Movie App - <a asp-area="" asp-controller="Home"
asp-action="Privacy">Privacy</a>
        </div>
    </footer>
    <script src="~/lib/jquery/dist/jquery.js"></script>
    <script src="~/lib/bootstrap/dist/js/bootstrap.js"></script>
    <script src="~/js/site.js" asp-append-version="true"></script>
    @await RenderSectionAsync("Scripts", required: false)
</body>
</html>
```

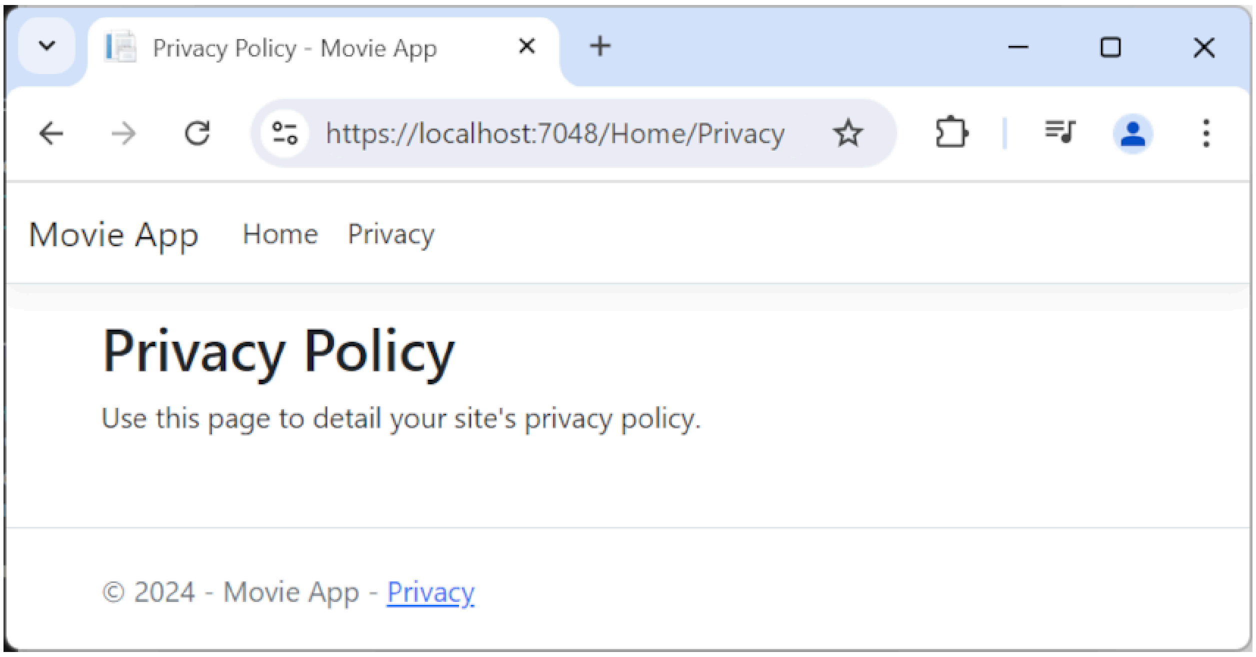
The preceding markup made the following changes:

- Three occurrences of `MvcMovie` to `Movie App`.
- The anchor element `<a class="navbar-brand" asp-area="" asp-controller="Home" asp-action="Index">MvcMovie</a>` to `<a class="navbar-brand" asp-controller="Movies" asp-action="Index">Movie App</a>`.

In the preceding markup, the `asp-area=""` anchor Tag Helper attribute and attribute value was omitted because this app isn't using Areas.

Note: The `Movies` controller hasn't been implemented. At this point, the `Movie App` link isn't functional.

Save the changes and select the **Privacy** link. Notice how the title on the browser tab displays **Privacy Policy - Movie App** instead of **Privacy Policy - MvcMovie**


Privacy Policy - Movie App
https://localhost:7048/Home/Privacy
Movie App
Home
Privacy

Privacy Policy

Use this page to detail your site's privacy policy.

© 2024 - Movie App - Privacy

Select the **Home** link.

Notice that the title and anchor text display **Movie App**. The changes were made once in the layout template and all pages on the site reflect the new link text and new title.

Examine the `Views/_ViewStart.cshtml` file:

CSHTML

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

The `Views/_ViewStart.cshtml` file brings in the `Views/Shared/_Layout.cshtml` file to each view. The `Layout` property can be used to set a different layout view, or set it to `null` so no layout file will be used.

Open the `Views/HelloWorld/Index.cshtml` view file.

Change the title and `<h2>` element as highlighted in the following:

CSHTML

```
@{
    ViewData["Title"] = "Movie List";
}

<h2>My Movie List</h2>
```

```
<p>Hello from our View Template!</p>
```

The title and `<h2>` element are slightly different so it's clear which part of the code changes the display.

`ViewData["Title"] = "Movie List";` in the code above sets the `Title` property of the `ViewData` dictionary to "Movie List". The `Title` property is used in the `<title>` HTML element in the layout page:

CSHTML

```
<title>@ViewData["Title"] - Movie App</title>
```

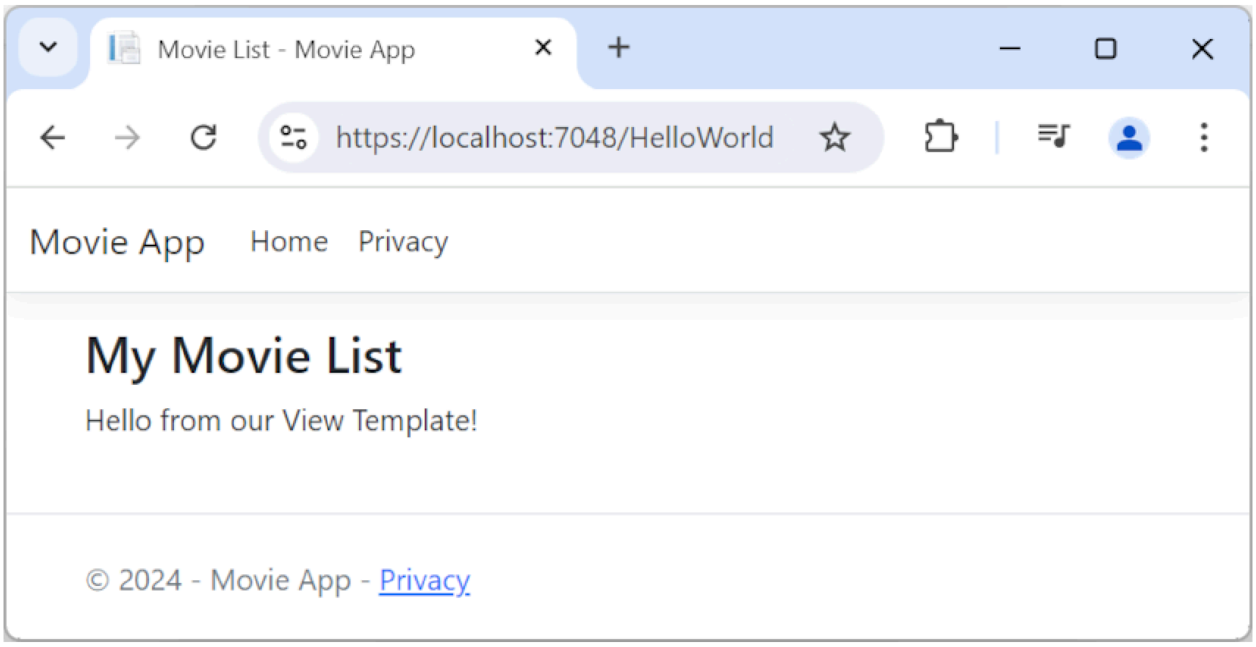
Save the change and navigate to `https://localhost:{PORT}/HelloWorld`.

Notice that the following have changed:

- Browser title.
- Primary heading.
- Secondary headings.

If there are no changes in the browser, it could be cached content that is being viewed. Press Ctrl+F5 in the browser to force the response from the server to be loaded. The browser title is created with `ViewData["Title"]` we set in the `Index.cshtml` view template and the additional "- Movie App" added in the layout file.

The content in the `Index.cshtml` view template is merged with the `Views/Shared/_Layout.cshtml` view template. A single HTML response is sent to the browser. Layout templates make it easy to make changes that apply across all of the pages in an app. To learn more, see Layout.



My Movie List

Hello from our View Template!

© 2024 - Movie App - Privacy

The small bit of "data", the "Hello from our View Template!" message, is hard-coded however. The MVC application has a "V" (view), a "C" (controller), but no "M" (model) yet.

Passing Data from the Controller to the View

Controller actions are invoked in response to an incoming URL request. A controller class is where the code is written that handles the incoming browser requests. The controller retrieves data from a data source and decides what type of response to send back to the browser. View templates can be used from a controller to generate and format an HTML response to the browser.

Controllers are responsible for providing the data required in order for a view template to render a response.

View templates should **not**:

- Do business logic
- Interact with a database directly.

A view template should work only with the data that's provided to it by the controller. Maintaining this "separation of concerns" helps keep the code:

- Clean.
- Testable.
- Maintainable.

Currently, the `Welcome` method in the `HelloWorldController` class takes a `name` and an `ID` parameter and then outputs the values directly to the browser.

Rather than have the controller render this response as a string, change the controller to use a view template instead. The view template generates a dynamic response, which means that appropriate data must be passed from the controller to the view to generate the response. Do this by having the controller put the dynamic data (parameters) that the view template needs in a `ViewData` dictionary. The view template can then access the dynamic data.

In `HelloWorldController.cs`, change the `Welcome` method to add a `Message` and `NumTimes` value to the `ViewData` dictionary.

The `ViewData` dictionary is a dynamic object, which means any type can be used. The `ViewData` object has no defined properties until something is added. The MVC model binding system automatically maps the named parameters `name` and `numTimes` from the query string to parameters in the method. The complete `HelloWorldController`:

```C#
using Microsoft.AspNetCore.Mvc;
using System.Text.Encodings.Web;

namespace MvcMovie.Controllers;

public class HelloWorldController : Controller
{
    public IActionResult Index()
    {
        return View();
    }
    public IActionResult Welcome(string name, int numTimes = 1)
    {
        ViewData["Message"] = "Hello " + name;
        ViewData["NumTimes"] = numTimes;
        return View();
    }
}
```

The `ViewData` dictionary object contains data that will be passed to the view.

Create a Welcome view template named `Views/HelloWorld/Welcome.cshtml`.

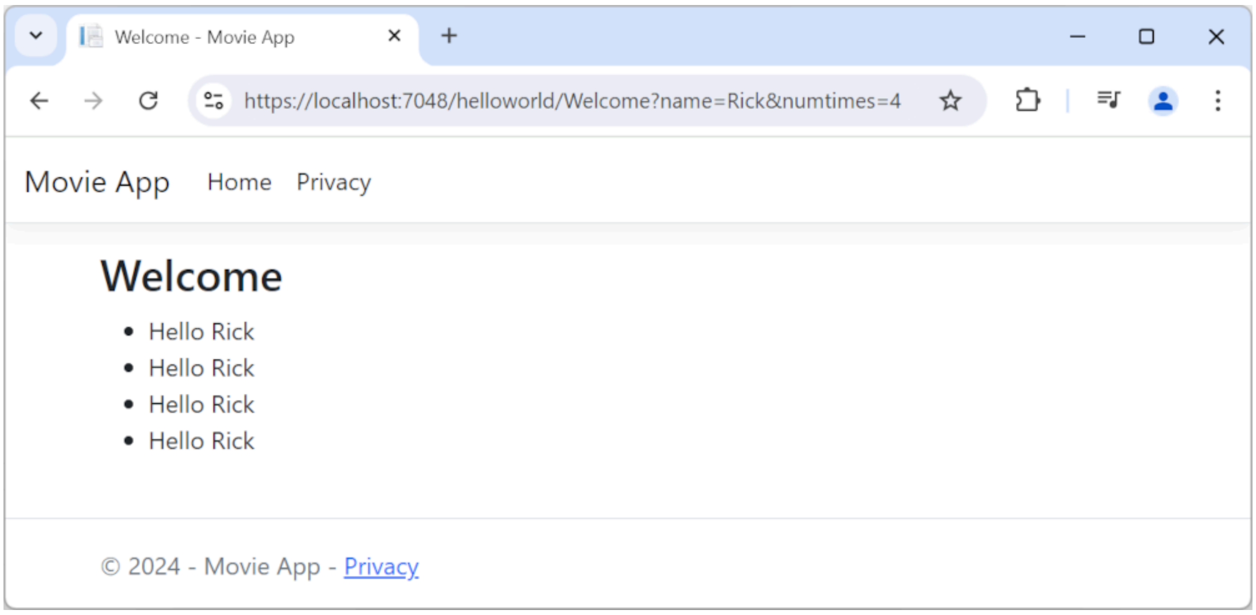
You'll create a loop in the `Welcome.cshtml` view template that displays "Hello" `NumTimes`. Replace the contents of `Views/HelloWorld/Welcome.cshtml` with the following:

```CSHTML
@{
    ViewData["Title"] = "Welcome";
}

<h2>Welcome</h2>

<ul>
    @for (int i = 0; i < (int)ViewData["NumTimes"]!; i++)
    {
        <li>@ViewData["Message"]</li>
    }
</ul>
```

Save your changes and browse to the following URL:

`https://localhost:{PORT}/HelloWorld/Welcome?name=Rick&numtimes=4`

Data is taken from the URL and passed to the controller using the MVC model binder. The controller packages the data into a `ViewData` dictionary and passes that object to the view. The view then renders the data as HTML to the browser.


Welcome - Movie App
https://localhost:7048/helloworld/Welcome?name=Rick&numtimes=4
Movie App
Home
Privacy

Welcome

- Hello Rick
- Hello Rick
- Hello Rick
- Hello Rick

© 2024 - Movie App - Privacy

In the preceding sample, the `ViewData` dictionary was used to pass data from the controller to a view. Later in the tutorial, a view model is used to pass data from a controller to a view. The view model approach to passing data is preferred over the `ViewData` dictionary approach.

In the next tutorial, a database of movies is created.

Previous: Add a Controller **Next: Add a Model**

Part 4, add a model to an ASP.NET Core MVC app

Article • 07/30/2024

Important

This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

For the current release, see the .NET 9 version of this article.

By Rick Anderson and Jon P Smith.

In this tutorial, classes are added for managing movies in a database. These classes are the "**M**odel" part of the **M**VC app.

These model classes are used with Entity Framework Core (EF Core) to work with a database. EF Core is an object-relational mapping (ORM) framework that simplifies the data access code that you have to write.

The model classes created are known as ***POCO*** classes, from **P**lain **O**ld **C**LR **O**bjects. POCO classes don't have any dependency on EF Core. They only define the properties of the data to be stored in the database.

In this tutorial, model classes are created first, and EF Core creates the database.

Add a data model class

Visual Studio

Right-click the *Models* folder > **Add** > **Class**. Name the file `Movie.cs`.

Update the `Models/Movie.cs` file with the following code:

```C#
using System.ComponentModel.DataAnnotations;

namespace MvcMovie.Models;

```

```
public class Movie
{
    public int Id { get; set; }
    public string? Title { get; set; }
    [DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }
    public string? Genre { get; set; }
    public decimal Price { get; set; }
}
```

The `Movie` class contains an `Id` field, which is required by the database for the primary key.

The DataType attribute on `ReleaseDate` specifies the type of the data (`Date`). With this attribute:

- The user isn't required to enter time information in the date field.
- Only the date is displayed, not time information.

DataAnnotations are covered in a later tutorial.

The question mark after `string` indicates that the property is nullable. For more information, see Nullable reference types.

Add NuGet packages

Visual Studio

Visual Studio automatically installs the required packages.

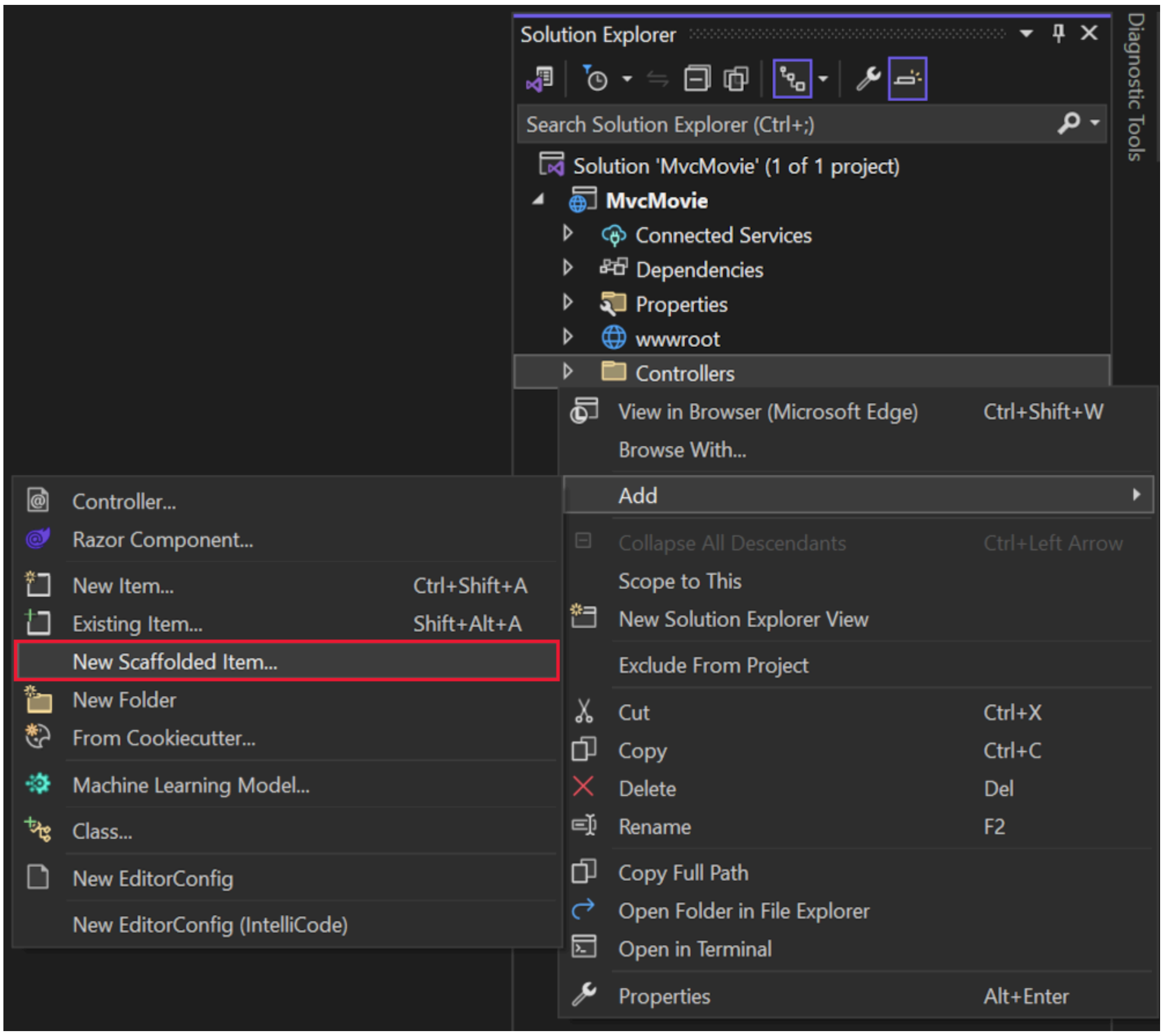
Build the project as a check for compiler errors.

Scaffold movie pages

Use the scaffolding tool to produce `Create`, `Read`, `Update`, and `Delete` (CRUD) pages for the movie model.

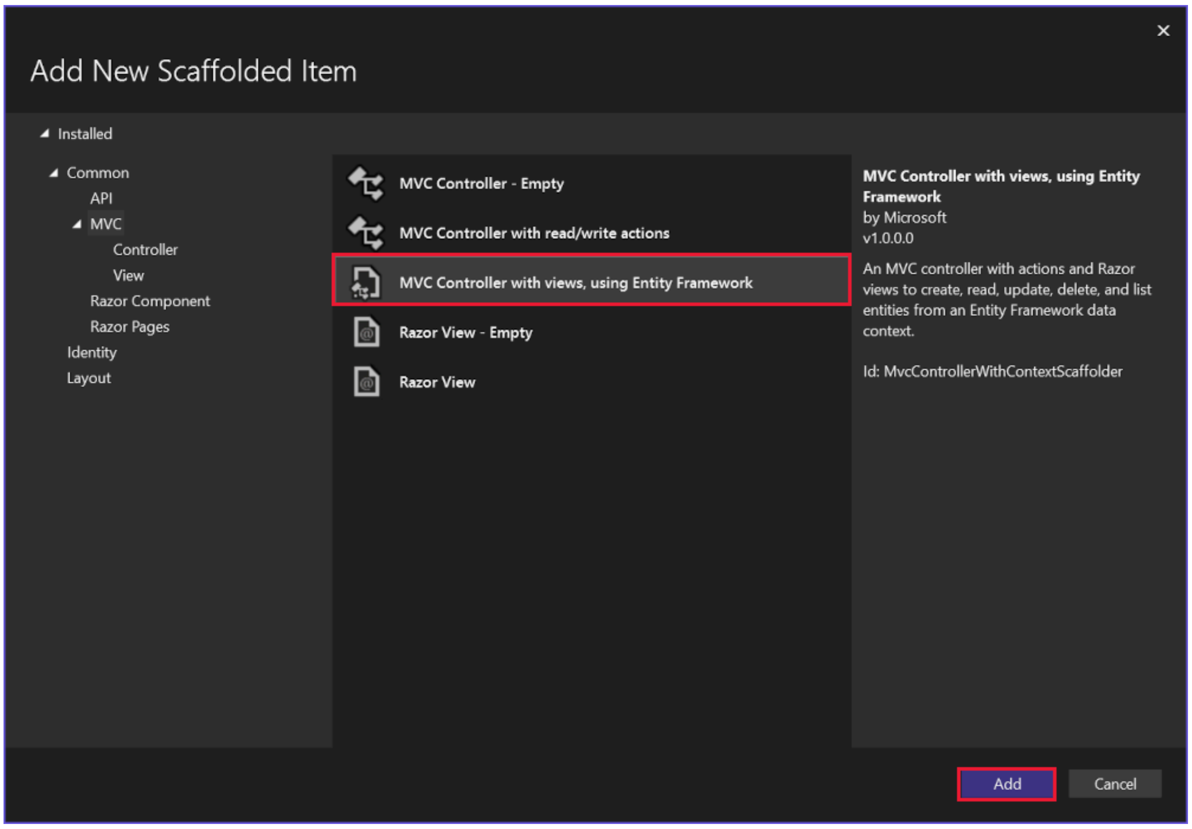
Visual Studio

In **Solution Explorer**, right-click the *Controllers* folder and select **Add** > **New Scaffolded Item**.



In the **Add New Scaffolded Item** dialog:

- In the left pane, select **Installed** > **Common** > **MVC**.
- Select **MVC Controller with views, using Entity Framework**.
- Select **Add**.



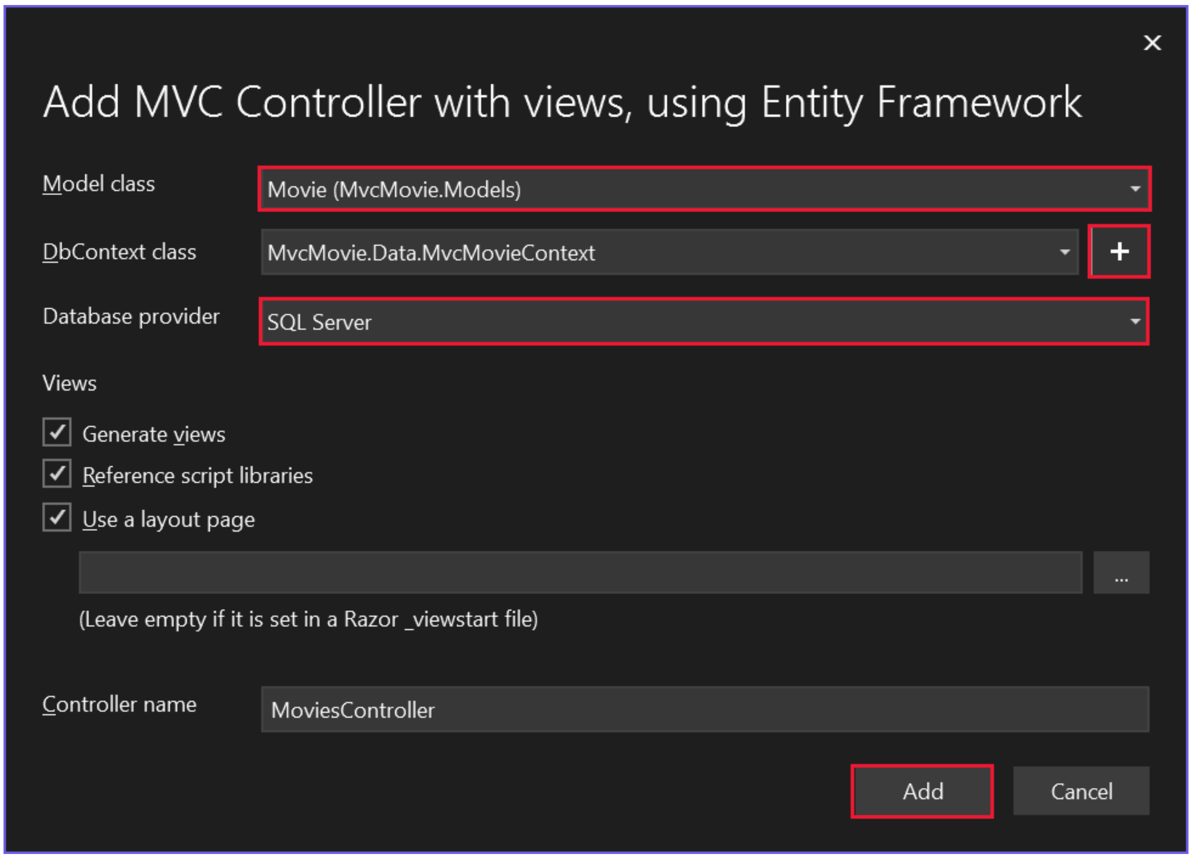
MVC Controller with views, using Entity Framework
by Microsoft
v1.0.0.0

An MVC controller with actions and Razor views to create, read, update, delete, and list entities from an Entity Framework data context.

Id: MvcControllerWithContextScaffolder

Complete the **Add MVC Controller with views, using Entity Framework** dialog:

- In the **Model class** drop down, select **Movie (MvcMovie.Models)**.
- In the **Data context class** row, select the **+** (plus) sign.
  - In the **Add Data Context** dialog, the class name *MvcMovie.Data.MvcMovieContext* is generated.
  - Select **Add**.
- In the **Database provider** drop down, select **SQL Server**.
- **Views** and **Controller name**: Keep the default.
- Select **Add**.


Add MVC Controller with views, using Entity Framework

Model class: Movie (MvcMovie.Models)

DbContext class: MvcMovie.Data.MvcMovieContext

Database provider: SQL Server

Views

- [✓] Generate views
- [✓] Reference script libraries
- [✓] Use a layout page

(Leave empty if it is set in a Razor _viewstart file)

Controller name: MoviesController

If you get an error message, select **Add** a second time to try it again.

Scaffolding adds the following packages:

- `Microsoft.EntityFrameworkCore.SqlServer`
- `Microsoft.EntityFrameworkCore.Tools`
- `Microsoft.VisualStudio.Web.CodeGeneration.Design`

Scaffolding creates the following:

- A movies controller: `Controllers/MoviesController.cs`
- Razor view files for **Create**, **Delete**, **Details**, **Edit**, and **Index** pages: `Views/Movies/*.cshtml`
- A database context class: `Data/MvcMovieContext.cs`

Scaffolding updates the following:

- Inserts required package references in the `MvcMovie.csproj` project file.
- Registers the database context in the `Program.cs` file.
- Adds a database connection string to the `appsettings.json` file.

The automatic creation of these files and file updates is known as *scaffolding*.

The scaffolded pages can't be used yet because the database doesn't exist. Running the app and selecting the **Movie App** link results in a *Cannot open database* or *no* *such table: Movie* error message.

Build the app to verify that there are no errors.

Initial migration

Use the EF Core Migrations feature to create the database. *Migrations* is a set of tools that create and update a database to match the data model.

Visual Studio

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

In the Package Manager Console (PMC), enter the following command:

PowerShell

```
Add-Migration InitialCreate
```

- `Add-Migration InitialCreate`: Generates a `Migrations/{timestamp}_InitialCreate.cs` migration file. The `InitialCreate` argument is the migration name. Any name can be used, but by convention, a name is selected that describes the migration. Because this is the first migration, the generated class contains code to create the database schema. The database schema is based on the model specified in the `MvcMovieContext` class.

The following warning is displayed, which is addressed in a later step:

> No store type was specified for the decimal property 'Price' on entity type 'Movie'. This will cause values to be silently truncated if they do not fit in the default precision and scale. Explicitly specify the SQL server column type that can accommodate all the values in 'OnModelCreating' using 'HasColumnType', specify precision and scale using 'HasPrecision', or configure a value converter using 'HasConversion'.

In the PMC, enter the following command:

PowerShell

```
Update-Database
```

- `Update-Database`: Updates the database to the latest migration, which the previous command created. This command runs the `Up` method in the `Migrations/{time-stamp}_InitialCreate.cs` file, which creates the database.

For more information on the PMC tools for EF Core, see EF Core tools reference - PMC in Visual Studio.

Test the app

Visual Studio

Run the app and select the **Movie App** link.

If you get an exception similar to the following, you may have missed the `Update-Database` command in the migrations step:

Console

```
SqlException: Cannot open database "MvcMovieContext-1" requested by the
login. The login failed.
```

Note

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

Examine the generated database context class and registration

With EF Core, data access is performed using a model. A model is made up of entity classes and a context object that represents a session with the database. The context object allows querying and saving data. The database context is derived from

Microsoft.EntityFrameworkCore.DbContext and specifies the entities to include in the data model.

Scaffolding creates the `Data/MvcMovieContext.cs` database context class:

C#

```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.EntityFrameworkCore;
using MvcMovie.Models;

namespace MvcMovie.Data
{
    public class MvcMovieContext : DbContext
    {
        public MvcMovieContext (DbContextOptions<MvcMovieContext> options)
            : base(options)
        {
        }

        public DbSet<MvcMovie.Models.Movie> Movie { get; set; } = default!;
    }
}
```

The preceding code creates a DbSet<Movie> property that represents the movies in the database.

Dependency injection

ASP.NET Core is built with dependency injection (DI). Services, such as the database context, are registered with DI in `Program.cs`. These services are provided to components that require them via constructor parameters.

In the `Controllers/MoviesController.cs` file, the constructor uses Dependency Injection to inject the `MvcMovieContext` database context into the controller. The database context is used in each of the CRUD methods in the controller.

Scaffolding generated the following highlighted code in `Program.cs`:

Visual Studio

C#

```
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddDbContext<MvcMovieContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("MvcMovie
Context") ?? throw new InvalidOperationException("Connection string
'MvcMovieContext' not found.")));
```

The ASP.NET Core configuration system reads the "MvcMovieContext" database connection string.

Examine the generated database connection string

Scaffolding added a connection string to the `appsettings.json` file:

Visual Studio

JSON

```json
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "MvcMovieContext": "Server=
(localdb)\\mssqllocaldb;Database=MvcMovieContext-4ebefa10-de29-4dea-
b2ad-8a8dc6bcf374;Trusted_Connection=True;MultipleActiveResultSets=true"
  }
}
```

For local development, the ASP.NET Core configuration system reads the `ConnectionString` key from the `appsettings.json` file.

The `InitialCreate` class

Examine the `Migrations/{timestamp}_InitialCreate.cs` migration file:

C#

```csharp
using System;
using Microsoft.EntityFrameworkCore.Migrations;
```

```
#nullable disable

namespace MvcMovie.Migrations
{
    /// <inheritdoc />
    public partial class InitialCreate : Migration
    {
        /// <inheritdoc />
        protected override void Up(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.CreateTable(
                name: "Movie",
                columns: table => new
                {
                    Id = table.Column<int>(type: "int", nullable: false)
                        .Annotation("SqlServer:Identity", "1, 1"),
                    Title = table.Column<string>(type: "nvarchar(max)",
nullable: true),
                    ReleaseDate = table.Column<DateTime>(type: "datetime2",
nullable: false),
                    Genre = table.Column<string>(type: "nvarchar(max)",
nullable: true),
                    Price = table.Column<decimal>(type: "decimal(18,2)",
nullable: false)
                },
                constraints: table =>
                {
                    table.PrimaryKey("PK_Movie", x => x.Id);
                });
        }

        /// <inheritdoc />
        protected override void Down(MigrationBuilder migrationBuilder)
        {
            migrationBuilder.DropTable(
                name: "Movie");
        }
    }
}
```

In the preceding code:

- `InitialCreate.Up` creates the Movie table and configures `Id` as the primary key.
- `InitialCreate.Down` reverts the schema changes made by the `Up` migration.

Dependency injection in the controller

Open the `Controllers/MoviesController.cs` file and examine the constructor:

C#

```csharp
public class MoviesController : Controller
{
    private readonly MvcMovieContext _context;

    public MoviesController(MvcMovieContext context)
    {
        _context = context;
    }
```

The constructor uses Dependency Injection to inject the database context (`MvcMovieContext`) into the controller. The database context is used in each of the CRUD methods in the controller.

Test the **Create** page. Enter and submit data.

Test the **Edit**, **Details**, and **Delete** pages.

Strongly typed models and the `@model` directive

Earlier in this tutorial, you saw how a controller can pass data or objects to a view using the `ViewData` dictionary. The `ViewData` dictionary is a dynamic object that provides a convenient late-bound way to pass information to a view.

MVC provides the ability to pass strongly typed model objects to a view. This strongly typed approach enables compile time code checking. The scaffolding mechanism passed a strongly typed model in the `MoviesController` class and views.

Examine the generated `Details` method in the `Controllers/MoviesController.cs` file:

C#

```csharp
// GET: Movies/Details/5
public async Task<IActionResult> Details(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var movie = await _context.Movie
        .FirstOrDefaultAsync(m => m.Id == id);
    if (movie == null)
    {
        return NotFound();
    }
    return View(movie);
}
```

The `id` parameter is generally passed as route data. For example, `https://localhost:5001/movies/details/1` sets:

- The controller to the `movies` controller, the first URL segment.
- The action to `details`, the second URL segment.
- The `id` to 1, the last URL segment.

The `id` can be passed in with a query string, as in the following example:

`https://localhost:5001/movies/details?id=1`

The `id` parameter is defined as a nullable type (`int?`) in cases when the `id` value isn't provided.

A lambda expression is passed in to the FirstOrDefaultAsync method to select movie entities that match the route data or query string value.

C#

```
var movie = await _context.Movie
    .FirstOrDefaultAsync(m => m.Id == id);
```

If a movie is found, an instance of the `Movie` model is passed to the `Details` view:

C#

```
return View(movie);
```

Examine the contents of the `Views/Movies/Details.cshtml` file:

CSHTML

```
@model MvcMovie.Models.Movie

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

<h1>Details</h1>

<div>
    <h4>Movie</h4>
    <hr />
    <dl class="row">
        <dt class = "col-sm-2">
            @Html.DisplayNameFor(model => model.Title)
        </dt>
        <dd class = "col-sm-10">
            @Html.DisplayFor(model => model.Title)
        </dd>
        <dt class = "col-sm-2">
            @Html.DisplayNameFor(model => model.ReleaseDate)
        </dt>
        <dd class = "col-sm-10">
            @Html.DisplayFor(model => model.ReleaseDate)
        </dd>
        <dt class = "col-sm-2">
            @Html.DisplayNameFor(model => model.Genre)
        </dt>
        <dd class = "col-sm-10">
            @Html.DisplayFor(model => model.Genre)
        </dd>
        <dt class = "col-sm-2">
            @Html.DisplayNameFor(model => model.Price)
        </dt>
        <dd class = "col-sm-10">
            @Html.DisplayFor(model => model.Price)
        </dd>
    </dl>
</div>
<div>
    <a asp-action="Edit" asp-route-id="@Model.Id">Edit</a> |
    <a asp-action="Index">Back to List</a>
</div>
```

The `@model` statement at the top of the view file specifies the type of object that the view expects. When the movie controller was created, the following `@model` statement was included:

CSHTML

```
@model MvcMovie.Models.Movie
```

This `@model` directive allows access to the movie that the controller passed to the view. The `Model` object is strongly typed. For example, in the `Details.cshtml` view, the code passes each movie field to the `DisplayNameFor` and `DisplayFor` HTML Helpers with the strongly typed `Model` object. The `Create` and `Edit` methods and views also pass a `Movie` model object.

Examine the `Index.cshtml` view and the `Index` method in the Movies controller. Notice how the code creates a `List` object when it calls the `View` method. The code passes this

`Movies` list from the `Index` action method to the view:

C#

```csharp
// GET: Movies
public async Task<IActionResult> Index()
{
    return View(await _context.Movie.ToListAsync());
}
```

The code returns problem details if the `Movie` property of the data context is null.

When the movies controller was created, scaffolding included the following `@model` statement at the top of the `Index.cshtml` file:

CSHTML

```cshtml
@model IEnumerable<MvcMovie.Models.Movie>
```

The `@model` directive allows access to the list of movies that the controller passed to the view by using a `Model` object that's strongly typed. For example, in the `Index.cshtml` view, the code loops through the movies with a `foreach` statement over the strongly typed `Model` object:

CSHTML

```cshtml
@model IEnumerable<MvcMovie.Models.Movie>

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

<h1>Index</h1>

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

Because the `Model` object is strongly typed as an `IEnumerable<Movie>` object, each item in the loop is typed as `Movie`. Among other benefits, the compiler validates the types used in the code.

Additional resources

- Entity Framework Core for Beginners
- Tag Helpers
- Globalization and localization

Previous: Adding a View   Next: Working with SQL

Part 5, work with a database in an ASP.NET Core MVC app

Article • 09/10/2024

Important

This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

For the current release, see the .NET 9 version of this article.

By Rick Anderson and Jon P Smith.

The `MvcMovieContext` object handles the task of connecting to the database and mapping `Movie` objects to database records. The database context is registered with the Dependency Injection container in the `Program.cs` file:

Visual Studio

C#

```csharp
var builder = WebApplication.CreateBuilder(args);
builder.Services.AddDbContext<MvcMovieContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("MvcMovie
Context") ?? throw new InvalidOperationException("Connection string
'MvcMovieContext' not found.")));
```

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

JSON

```json
"ConnectionStrings": {
  "MvcMovieContext": "Server=
(localdb)\\mssqllocaldb;Database=MvcMovieContext-4ebefa10-de29-4dea-
b2ad-8a8dc6bcf374;Trusted_Connection=True;MultipleActiveResultSets=true"
}
```

⚠ Warning

This article uses a local database that doesn't require the user to be authenticated. Production apps should use the most secure authentication flow available. For more information on authentication for deployed test and production apps, see Secure authentication flows.

Visual Studio

SQL Server Express LocalDB

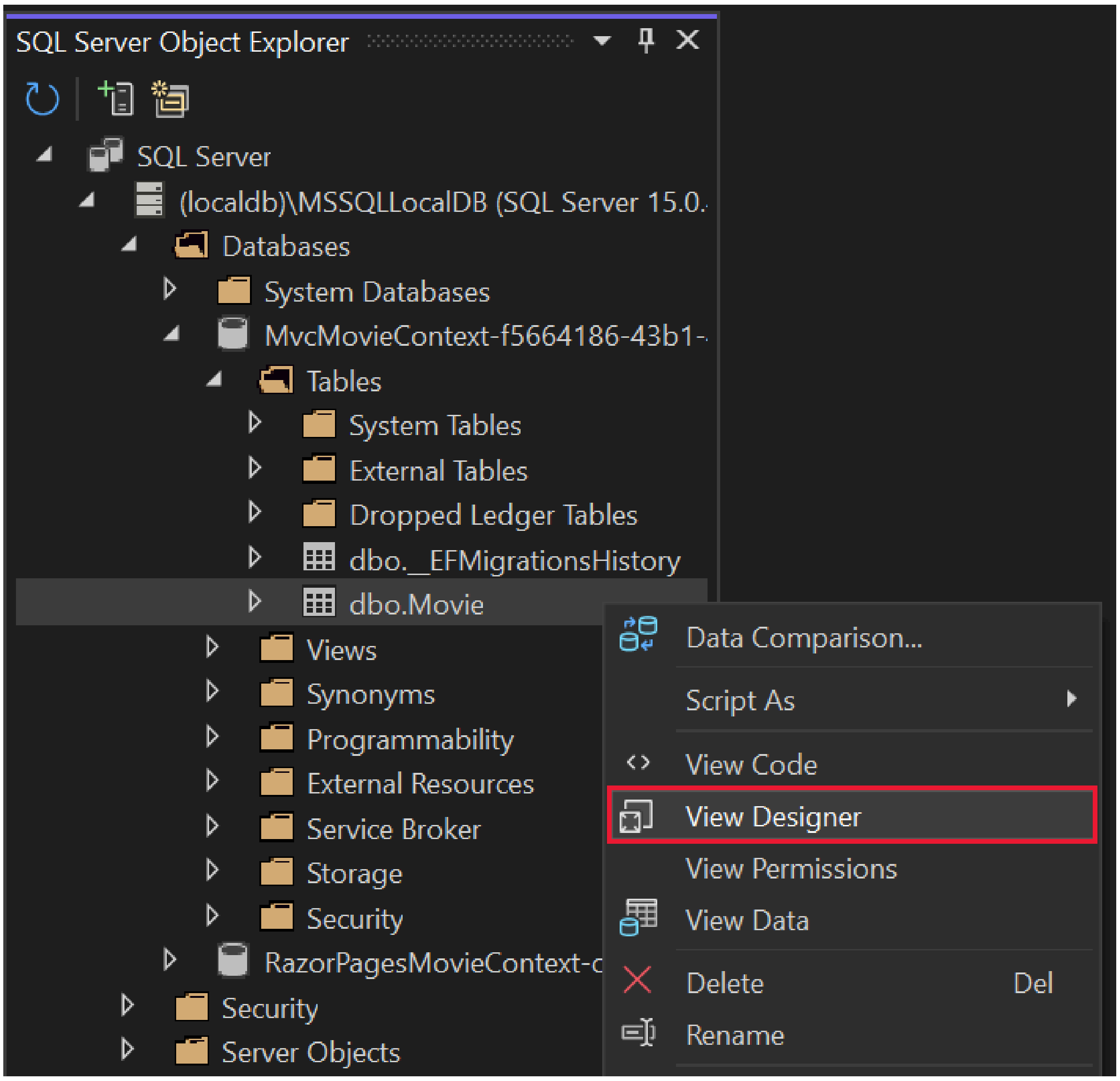
LocalDB:

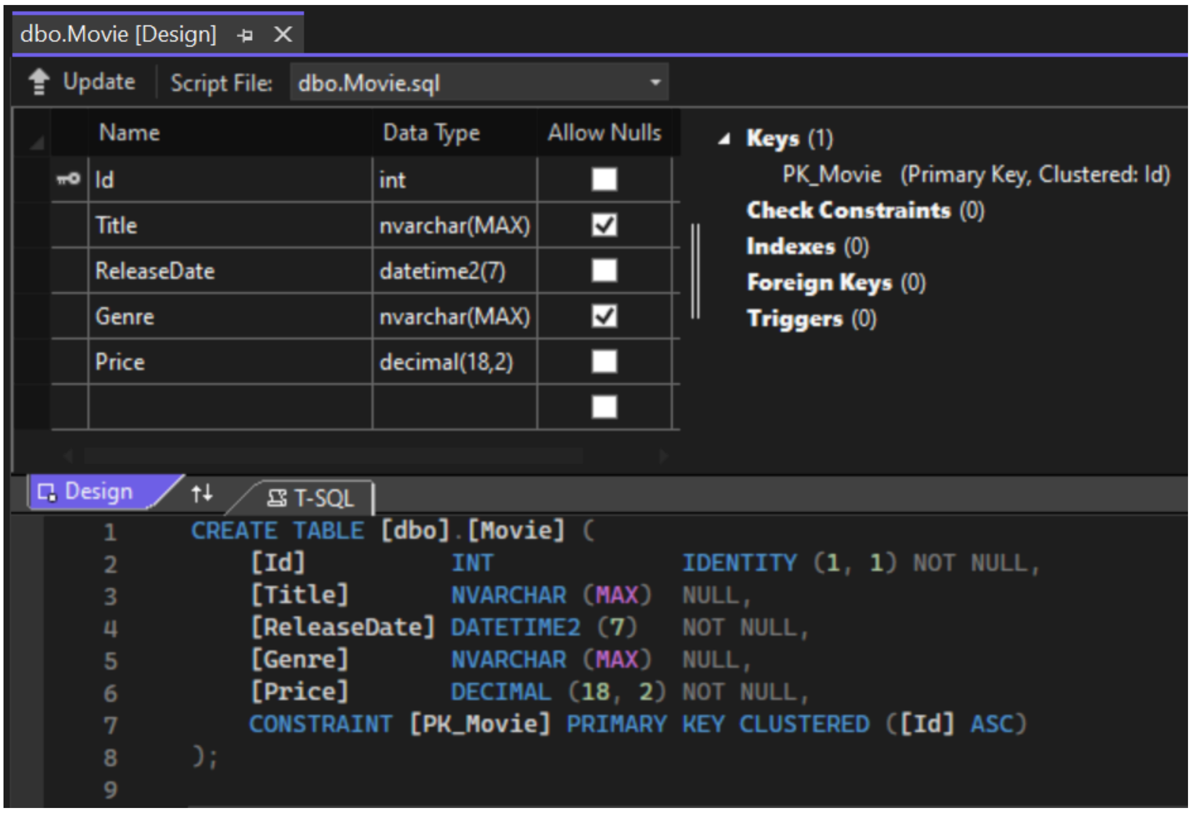
- Is a lightweight version of the SQL Server Express Database Engine, installed by default with Visual Studio.
- Starts on demand by using a connection string.
- Is targeted for program development. It runs in user mode, so there's no complex configuration.
- By default creates *.mdf* files in the *C:/Users/{user}* directory.

Examine the database

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

Right-click on the `Movie` table (`dbo.Movie`) > **View Designer**





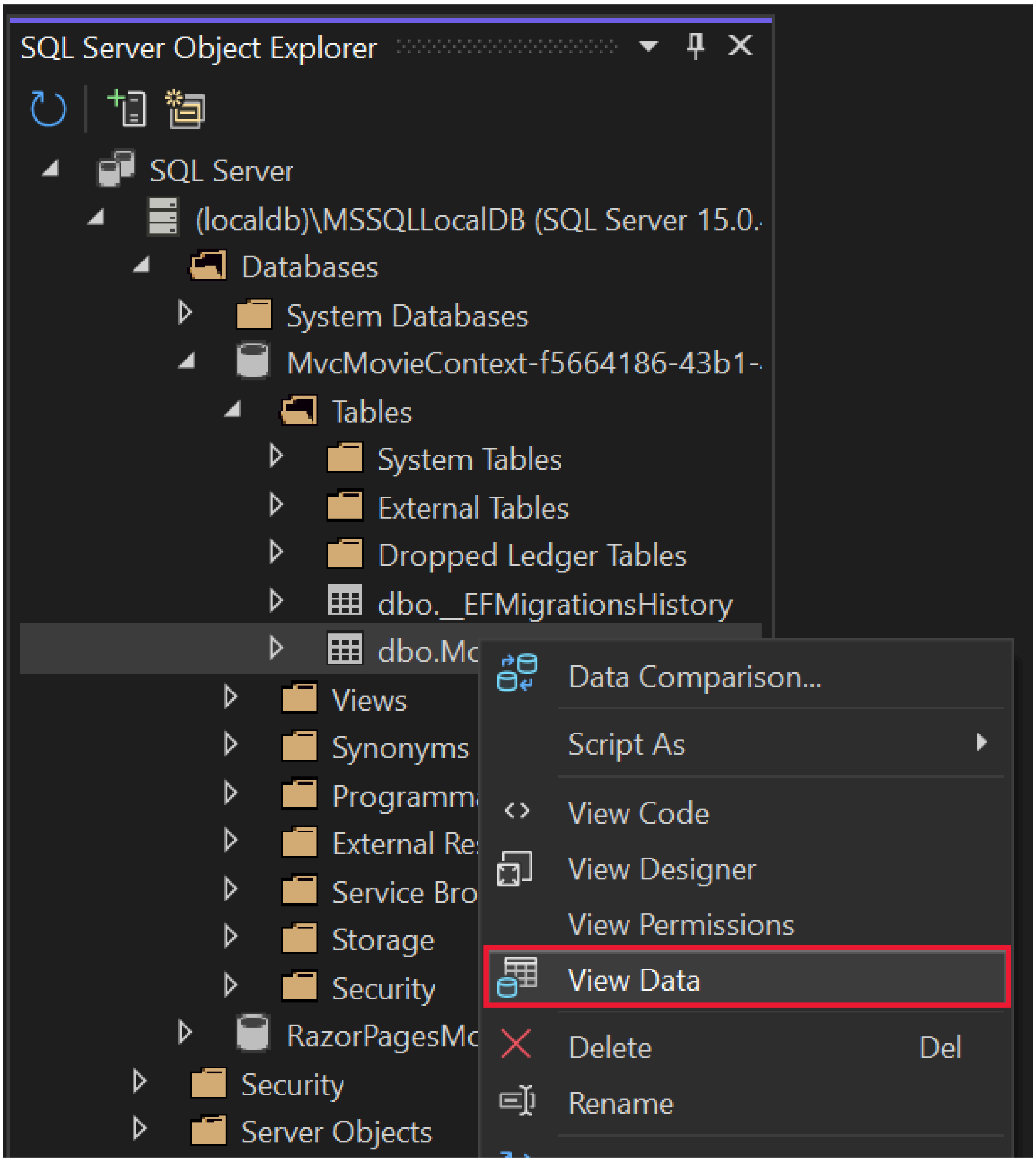
Name	Data Type	Allow Nulls
Id	int	☐
Title	nvarchar(MAX)	✓
ReleaseDate	datetime2(7)	☐
Genre	nvarchar(MAX)	✓
Price	decimal(18,2)	☐
		☐

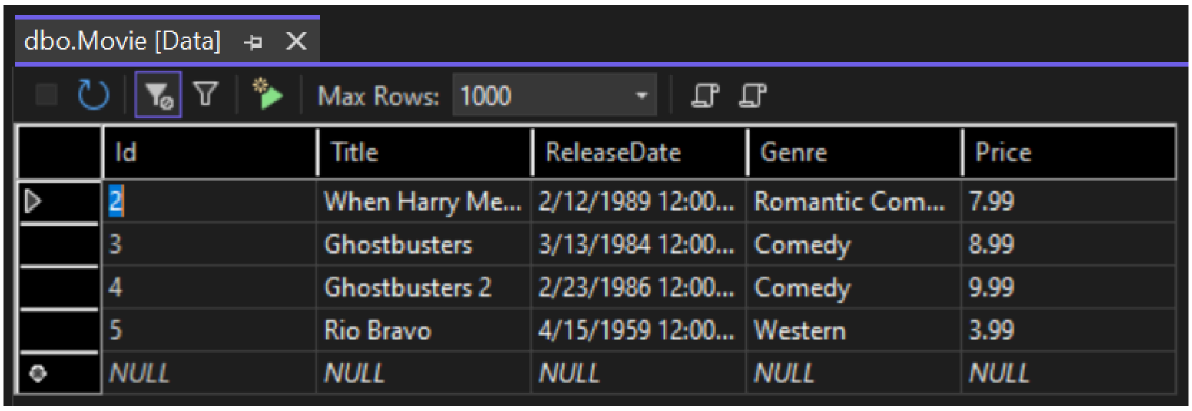
Keys (1)
PK_Movie (Primary Key, Clustered: Id)
Check Constraints (0)
Indexes (0)
Foreign Keys (0)
Triggers (0)

```
CREATE TABLE [dbo].[Movie] (
    [Id]          INT            IDENTITY (1, 1) NOT NULL,
    [Title]       NVARCHAR (MAX) NULL,
    [ReleaseDate] DATETIME2 (7)  NOT NULL,
    [Genre]       NVARCHAR (MAX) NULL,
    [Price]       DECIMAL (18, 2) NOT NULL,
    CONSTRAINT [PK_Movie] PRIMARY KEY CLUSTERED ([Id] ASC)
);
```

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

Right-click on the `Movie` table > **View Data**




dbo.Movie [Data]

Max Rows: 1000

	Id	Title	ReleaseDate	Genre	Price
▷	2	When Harry Me...	2/12/1989 12:00...	Romantic Com...	7.99
	3	Ghostbusters	3/13/1984 12:00...	Comedy	8.99
	4	Ghostbusters 2	2/23/1986 12:00...	Comedy	9.99
	5	Rio Bravo	4/15/1959 12:00...	Western	3.99
●	NULL	NULL	NULL	NULL	NULL

Seed the database

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

C#

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

namespace MvcMovie.Models;

public static class SeedData
{
    public static void Initialize(IServiceProvider serviceProvider)
    {
        using (var context = new MvcMovieContext(
            serviceProvider.GetRequiredService<
                DbContextOptions<MvcMovieContext>>()))
        {
            // Look for any movies.
            if (context.Movie.Any())
            {
                return;   // DB has been seeded
            }
            context.Movie.AddRange(
                new Movie
                {
                    Title = "When Harry Met Sally",
                    ReleaseDate = DateTime.Parse("1989-2-12"),
                    Genre = "Romantic Comedy",
                    Price = 7.99M
                },
                new Movie
                {
                    Title = "Ghostbusters ",
                    ReleaseDate = DateTime.Parse("1984-3-13"),
                    Genre = "Comedy",
                    Price = 8.99M
                },
                new Movie
                {
                    Title = "Ghostbusters 2",
                    ReleaseDate = DateTime.Parse("1986-2-23"),
                    Genre = "Comedy",
                    Price = 9.99M
                },
                new Movie
                {
                    Title = "Rio Bravo",
                    ReleaseDate = DateTime.Parse("1959-4-15"),
                    Genre = "Western",
                    Price = 3.99M
                }
            );
            context.SaveChanges();
        }
    }
}
```

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

C#

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

Add the seed initializer

Visual Studio

Replace the contents of `Program.cs` with the following code. The new code is highlighted.

C#

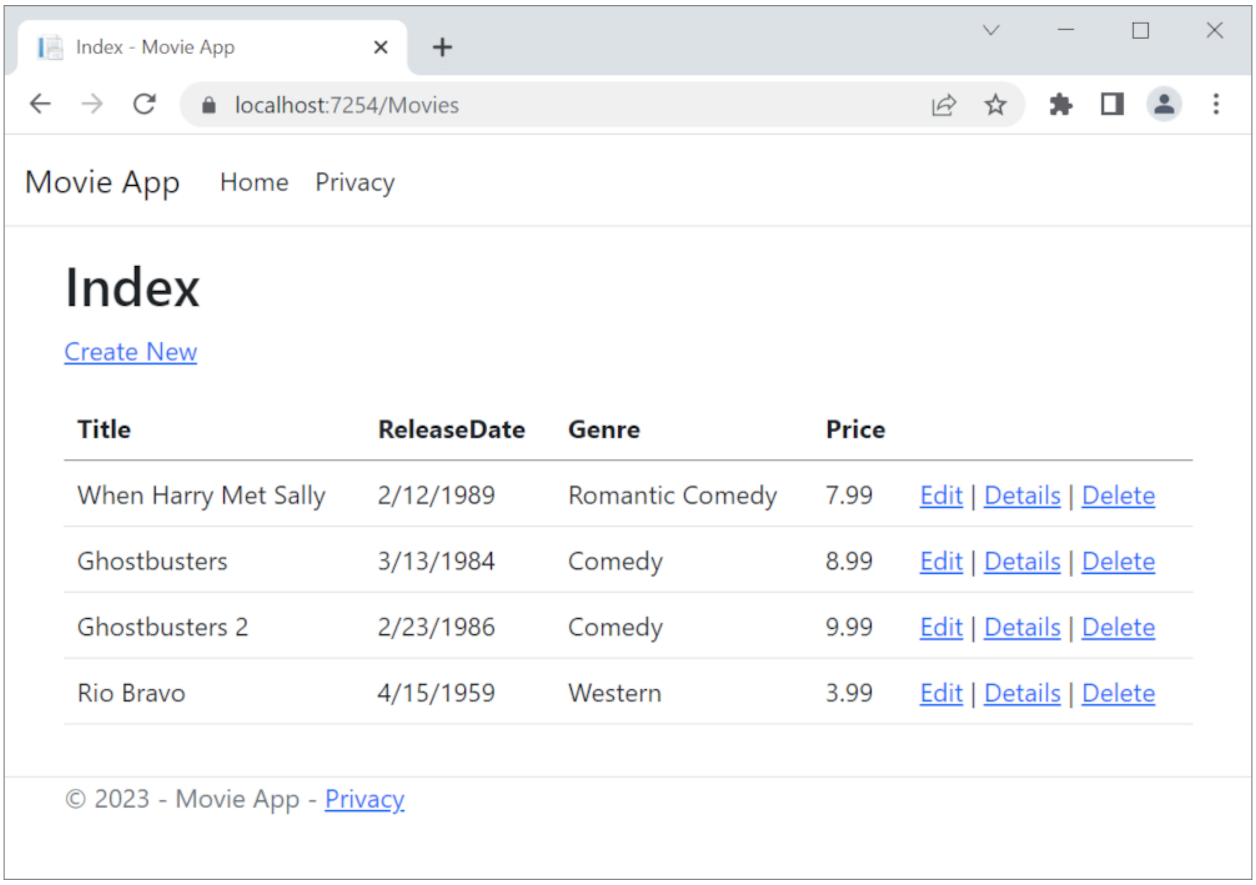
```
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using MvcMovie.Data;
using MvcMovie.Models;
var builder = WebApplication.CreateBuilder(args);
```

```
builder.Services.AddDbContext<MvcMovieContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("MvcMovie
Context") ?? throw new InvalidOperationException("Connection string
'MvcMovieContext' not found.")));

// Add services to the container.
builder.Services.AddControllersWithViews();

var app = builder.Build();

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

    SeedData.Initialize(services);
}
// Configure the HTTP request pipeline.
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Home/Error");
    // The default HSTS value is 30 days. You may want to change this
for production scenarios, see https://aka.ms/aspnetcore-hsts.
    app.UseHsts();
}

app.UseHttpsRedirection();

app.UseRouting();

app.UseAuthorization();

app.MapStaticAssets();

app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");

app.Run();
```

Delete all the records in the database. You can do this with the delete links in the browser or from SSOX.

Test the app. Force the app to initialize, calling the code in the `Program.cs` file, so the seed method runs. To force initialization, close the command prompt window that Visual Studio opened, and restart by pressing Ctrl+F5.

The app shows the seeded data.



Index

Create New

Title	ReleaseDate	Genre	Price	
When Harry Met Sally	2/12/1989	Romantic Comedy	7.99	Edit | Details | Delete
Ghostbusters	3/13/1984	Comedy	8.99	Edit | Details | Delete
Ghostbusters 2	2/23/1986	Comedy	9.99	Edit | Details | Delete
Rio Bravo	4/15/1959	Western	3.99	Edit | Details | Delete

© 2023 - Movie App - Privacy

Previous: Adding a model

Next: Adding controller methods and views

Part 6, controller methods and views in ASP.NET Core

Article • 09/18/2024

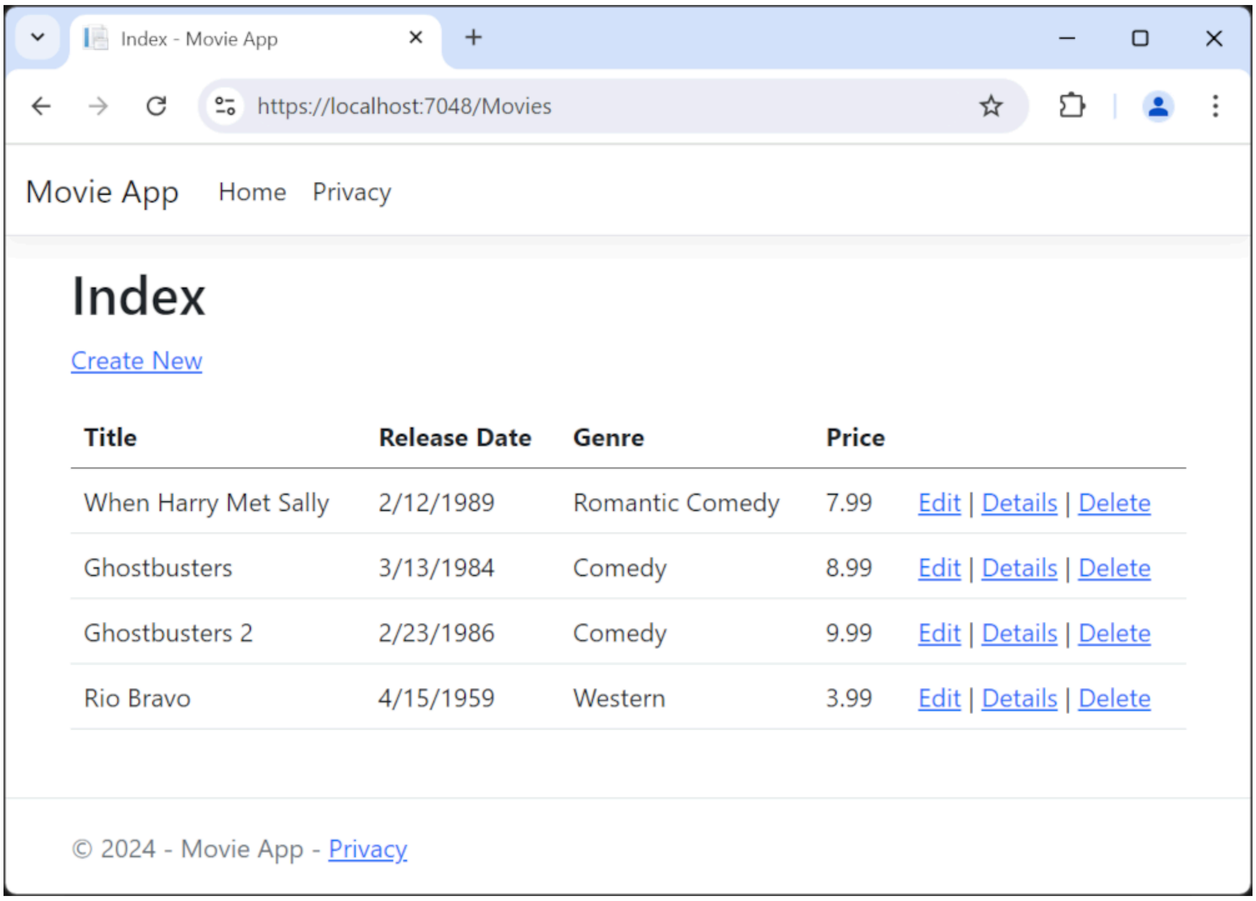
Important

This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

For the current release, see the .NET 9 version of this article.

By Rick Anderson

We have a good start to the movie app, but the presentation isn't ideal, for example, **ReleaseDate** should be two words.



Index

Create New

Title	Release Date	Genre	Price	
When Harry Met Sally	2/12/1989	Romantic Comedy	7.99	Edit | Details | Delete
Ghostbusters	3/13/1984	Comedy	8.99	Edit | Details | Delete
Ghostbusters 2	2/23/1986	Comedy	9.99	Edit | Details | Delete
Rio Bravo	4/15/1959	Western	3.99	Edit | Details | Delete

© 2024 - Movie App - Privacy

Open the `Models/Movie.cs` file and add the highlighted lines shown below:

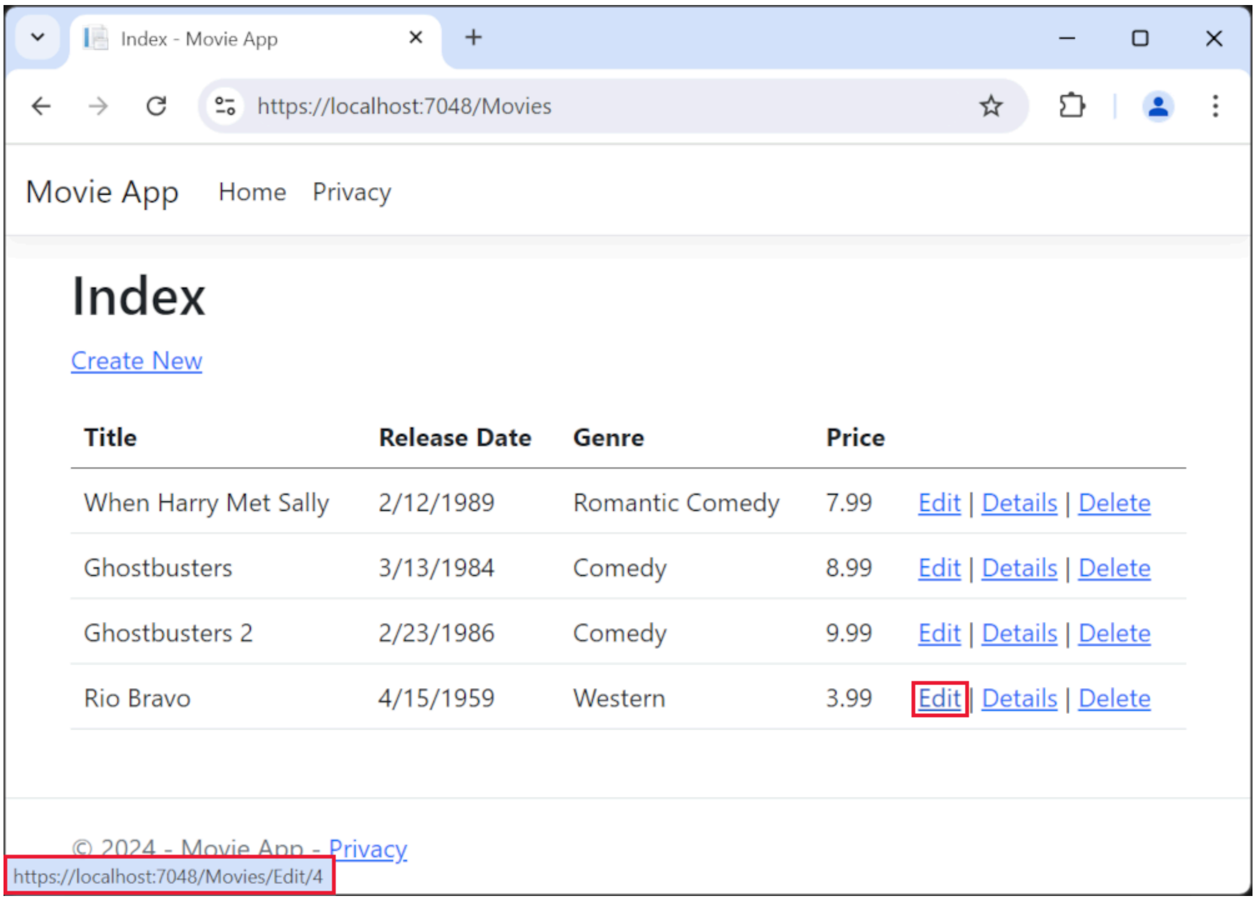
C#

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

namespace MvcMovie.Models;

public class Movie
{
    public int Id { get; set; }
    public string? Title { get; set; }

    [Display(Name = "Release Date")]
    [DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }
    public string? Genre { get; set; }
    [Column(TypeName = "decimal(18, 2)")]
    public decimal Price { get; set; }
}
```

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

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

Browse to the `Movies` controller and hold the mouse pointer over an **Edit** link to see the target URL.


Index - Movie App

https://localhost:7048/Movies

Movie App Home Privacy

Index

Create New

Title	Release Date	Genre	Price	
When Harry Met Sally	2/12/1989	Romantic Comedy	7.99	Edit | Details | Delete
Ghostbusters	3/13/1984	Comedy	8.99	Edit | Details | Delete
Ghostbusters 2	2/23/1986	Comedy	9.99	Edit | Details | Delete
Rio Bravo	4/15/1959	Western	3.99	Edit | Details | Delete

© 2024 - Movie App - Privacy

https://localhost:7048/Movies/Edit/4

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

CSHTML

```cshtml
            <a asp-action="Edit" asp-route-id="@item.Id">Edit</a> |
            <a asp-action="Details" asp-route-id="@item.Id">Details</a> |
            <a asp-action="Delete" asp-route-id="@item.Id">Delete</a>
        </td>
    </tr>
```

Tag Helpers enable server-side code to participate in creating and rendering HTML elements in Razor files. In the code above, the `AnchorTagHelper` dynamically generates the HTML `href` attribute value from the controller action method and route id. You use **View Source** from your favorite browser or use the developer tools to examine the generated markup. A portion of the generated HTML is shown below:

HTML

```html
 <td>
    <a href="/Movies/Edit/4"> Edit </a> |
    <a href="/Movies/Details/4"> Details </a> |
    <a href="/Movies/Delete/4"> Delete </a>
</td>
```

Recall the format for routing set in the `Program.cs` file:

```C#
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
```

ASP.NET Core translates `https://localhost:5001/Movies/Edit/4` into a request to the `Edit` action method of the `Movies` controller with the parameter `Id` of 4. (Controller methods are also known as action methods.)

Tag Helpers are one of the most popular new features in ASP.NET Core. For more information, see Additional resources.

Open the `Movies` controller and examine the two `Edit` action methods. The following code shows the `HTTP GET Edit` method, which fetches the movie and populates the edit form generated by the `Edit.cshtml` Razor file.

```C#
// GET: Movies/Edit/5
public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var movie = await _context.Movie.FindAsync(id);
    if (movie == null)
    {
        return NotFound();
    }
    return View(movie);
}
```

The following code shows the `HTTP POST Edit` method, which processes the posted movie values:

```C#
// POST: Movies/Edit/5
// To protect from overposting attacks, enable the specific properties you
want to bind to.
// For more details, see http://go.microsoft.com/fwlink/?LinkId=317598.
[HttpPost]
[ValidateAntiForgeryToken]
```

```
public async Task<IActionResult> Edit(int id,
[Bind("Id,Title,ReleaseDate,Genre,Price,Rating")] Movie movie)
{
    if (id != movie.Id)
    {
        return NotFound();
    }

    if (ModelState.IsValid)
    {
        try
        {
            _context.Update(movie);
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!MovieExists(movie.Id))
            {
                return NotFound();
            }
            else
            {
                throw;
            }
        }
        return RedirectToAction(nameof(Index));
    }
    return View(movie);
}
```

The `[Bind]` attribute is one way to protect against over-posting. You should only include properties in the `[Bind]` attribute that you want to change. For more information, see Protect your controller from over-posting. ViewModels provide an alternative approach to prevent over-posting.

Notice the second `Edit` action method is preceded by the `[HttpPost]` attribute.

```C#
// POST: Movies/Edit/5
// To protect from overposting attacks, enable the specific properties you
want to bind to.
// For more details, see http://go.microsoft.com/fwlink/?LinkId=317598.
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int id,
[Bind("Id,Title,ReleaseDate,Genre,Price,Rating")] Movie movie)
{
    if (id != movie.Id)
    {
        return NotFound();
    }

    if (ModelState.IsValid)
    {
        try
        {
            _context.Update(movie);
            await _context.SaveChangesAsync();
        }
        catch (DbUpdateConcurrencyException)
        {
            if (!MovieExists(movie.Id))
            {
                return NotFound();
            }
            else
            {
                throw;
            }
        }
        return RedirectToAction(nameof(Index));
    }
    return View(movie);
}
```

The `HttpPost` attribute specifies that this `Edit` method can be invoked *only* for `POST` requests. You could apply the `[HttpGet]` attribute to the first edit method, but that's not necessary because `[HttpGet]` is the default.

The `ValidateAntiForgeryToken` attribute is used to prevent forgery of a request and is paired up with an antiforgery token generated in the edit view file (`Views/Movies/Edit.cshtml`). The edit view file generates the antiforgery token with the Form Tag Helper.

CSHTML

```
<form asp-action="Edit">
```

The Form Tag Helper generates a hidden antiforgery token that must match the `[ValidateAntiForgeryToken]` generated antiforgery token in the `Edit` method of the Movies controller. For more information, see Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core.

The `HttpGet Edit` method takes the movie `ID` parameter, looks up the movie using the Entity Framework `FindAsync` method, and returns the selected movie to the Edit view. If a movie cannot be found, `NotFound` (HTTP 404) is returned.

C#

```
// GET: Movies/Edit/5
public async Task<IActionResult> Edit(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var movie = await _context.Movie.FindAsync(id);
    if (movie == null)
    {
        return NotFound();
    }
    return View(movie);
}
```

When the scaffolding system created the Edit view, it examined the `Movie` class and created code to render `<label>` and `<input>` elements for each property of the class. The following example shows the Edit view that was generated by the Visual Studio scaffolding system:

CSHTML

```
@model MvcMovie.Models.Movie

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

<h1>Edit</h1>

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

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

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

Notice how the view template has a `@model MvcMovie.Models.Movie` statement at the top of the file. `@model MvcMovie.Models.Movie` specifies that the view expects the model for the view template to be of type `Movie`.

The scaffolded code uses several Tag Helper methods to streamline the HTML markup. The Label Tag Helper displays the name of the field ("Title", "ReleaseDate", "Genre", or "Price"). The Input Tag Helper renders an HTML `<input>` element. The Validation Tag Helper displays any validation messages associated with that property.

Run the application and navigate to the `/Movies` URL. Click an **Edit** link. In the browser, view the source for the page. The generated HTML for the `<form>` element is shown below.

HTML

```
<form action="/Movies/Edit/7" method="post">
    <div class="form-horizontal">
        <h4>Movie</h4>
        <hr />
        <div class="text-danger" />
        <input type="hidden" data-val="true" data-val-required="The ID field
is required." id="ID" name="ID" value="7" />
        <div class="form-group">
            <label class="control-label col-md-2" for="Genre" />
            <div class="col-md-10">
```

```
                <input class="form-control" type="text" id="Genre"
name="Genre" value="Western" />
                <span class="text-danger field-validation-valid" data-
valmsg-for="Genre" data-valmsg-replace="true"></span>
            </div>
        </div>
        <div class="form-group">
            <label class="control-label col-md-2" for="Price" />
            <div class="col-md-10">
                <input class="form-control" type="text" data-val="true"
data-val-number="The field Price must be a number." data-val-required="The
Price field is required." id="Price" name="Price" value="3.99" />
                <span class="text-danger field-validation-valid" data-
valmsg-for="Price" data-valmsg-replace="true"></span>
            </div>
        </div>
        <!-- Markup removed for brevity -->
        <div class="form-group">
            <div class="col-md-offset-2 col-md-10">
                <input type="submit" value="Save" class="btn btn-default" />
            </div>
        </div>
    </div>
    <input name="__RequestVerificationToken" type="hidden"
value="CfDJ8Inyxgp63fRFqUePGvuI5jGZsloJu1L7X9le1gy7NCIlSduCRx9jDQClrV9pOTTmq
UyXnJBXhmrjcUVDJyDUMm7-
MF_9rK8aAZdRdlOri7FmKVkRe_2v5LIHGKFcTjPrWPYnc9AdSbomkiOSaTEg7RU" />
</form>
```

The `<input>` elements are in an `HTML <form>` element whose `action` attribute is set to post to the `/Movies/Edit/id` URL. The form data will be posted to the server when the `Save` button is clicked. The last line before the closing `</form>` element shows the hidden XSRF token generated by the Form Tag Helper.

Processing the POST Request

The following listing shows the `[HttpPost]` version of the `Edit` action method.

C#

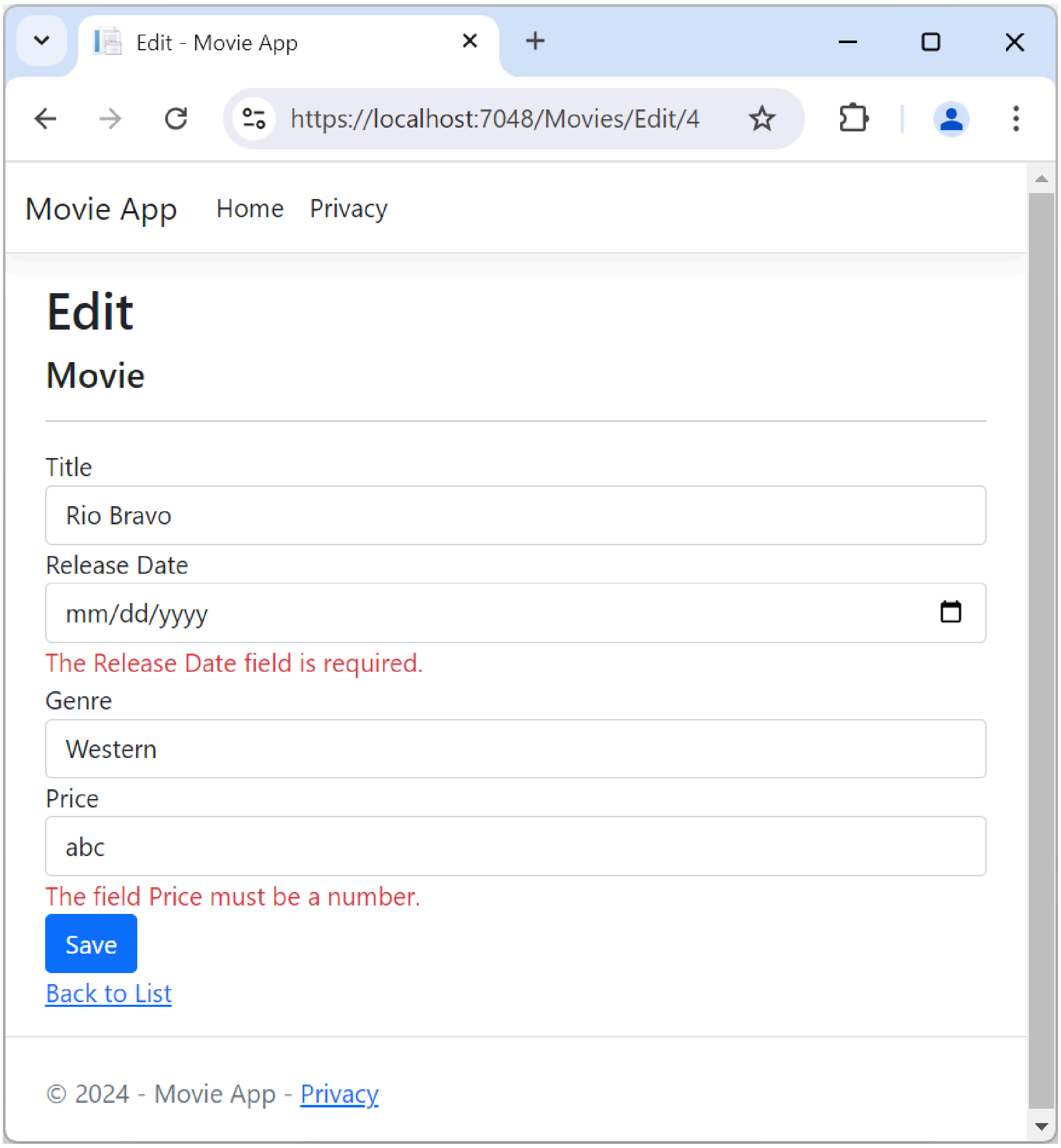
```
// POST: Movies/Edit/5
// To protect from overposting attacks, enable the specific properties you
want to bind to.
// For more details, see http://go.microsoft.com/fwlink/?LinkId=317598.
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Edit(int id,
[Bind("Id,Title,ReleaseDate,Genre,Price,Rating")] Movie movie)
{
    if (id != movie.Id)
```

```
        {
            return NotFound();
        }

        if (ModelState.IsValid)
        {
            try
            {
                _context.Update(movie);
                await _context.SaveChangesAsync();
            }
            catch (DbUpdateConcurrencyException)
            {
                if (!MovieExists(movie.Id))
                {
                    return NotFound();
                }
                else
                {
                    throw;
                }
            }
            return RedirectToAction(nameof(Index));
        }
        return View(movie);
    }
```

The `[ValidateAntiForgeryToken]` attribute validates the hidden XSRF token generated by the antiforgery token generator in the Form Tag Helper

The model binding system takes the posted form values and creates a `Movie` object that's passed as the `movie` parameter. The `ModelState.IsValid` property verifies that the data submitted in the form can be used to modify (edit or update) a `Movie` object. If the data is valid, it's saved. The updated (edited) movie data is saved to the database by calling the `SaveChangesAsync` method of database context. After saving the data, the code redirects the user to the `Index` action method of the `MoviesController` class, which displays the movie collection, including the changes just made.

Before the form is posted to the server, client-side validation checks any validation rules on the fields. If there are any validation errors, an error message is displayed and the form isn't posted. If JavaScript is disabled, you won't have client-side validation but the server will detect the posted values that are not valid, and the form values will be redisplayed with error messages. Later in the tutorial we examine Model Validation in more detail. The Validation Tag Helper in the `Views/Movies/Edit.cshtml` view template takes care of displaying appropriate error messages.


Edit - Movie App
https://localhost:7048/Movies/Edit/4
Movie App
Home
Privacy

Edit

Movie

Title

Rio Bravo

Release Date

mm/dd/yyyy

The Release Date field is required.

Genre

Western

Price

abc

The field Price must be a number.

Save
Back to List
© 2024 - Movie App - Privacy

All the `HttpGet` methods in the movie controller follow a similar pattern. They get a movie object (or list of objects, in the case of `Index`), and pass the object (model) to the view. The `Create` method passes an empty movie object to the `Create` view. All the methods that create, edit, delete, or otherwise modify data do so in the `[HttpPost]` overload of the method. Modifying data in an `HTTP GET` method is a security risk. Modifying data in an `HTTP GET` method also violates HTTP best practices and the architectural REST pattern, which specifies that GET requests shouldn't change the state of your application. In other words, performing a GET operation should be a safe operation that has no side effects and doesn't modify your persisted data.

Additional resources

- Globalization and localization
- Introduction to Tag Helpers
- Author Tag Helpers
- Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core
- Protect your controller from over-posting
- ViewModels
- Form Tag Helper
- Input Tag Helper
- Label Tag Helper
- Select Tag Helper
- Validation Tag Helper

Part 7, add search to an ASP.NET Core MVC app

Article • 09/18/2024

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the .NET 9 version of this article.

By Rick Anderson

In this section, you add search capability to the `Index` action method that lets you search movies by *genre* or *name*.

Update the `Index` method found inside `Controllers/MoviesController.cs` with the following code:

```C#
public async Task<IActionResult> Index(string searchString)
{
    if (_context.Movie == null)
    {
        return Problem("Entity set 'MvcMovieContext.Movie'  is null.");
    }

    var movies = from m in _context.Movie
                 select m;

    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s =>
s.Title!.ToUpper().Contains(searchString.ToUpper()));
    }

    return View(await movies.ToListAsync());
}
```

The following line in the `Index` action method creates a LINQ query to select the movies:

C#

```csharp
var movies = from m in _context.Movie
             select m;
```

The query is *only defined* at this point, it has **not** been run against the database.

If the `searchString` parameter contains a string, the movies query is modified to filter on the value of the search string:

C#

```csharp
if (!String.IsNullOrEmpty(searchString))
{
    movies = movies.Where(s =>
s.Title!.ToUpper().Contains(searchString.ToUpper()));
}
```

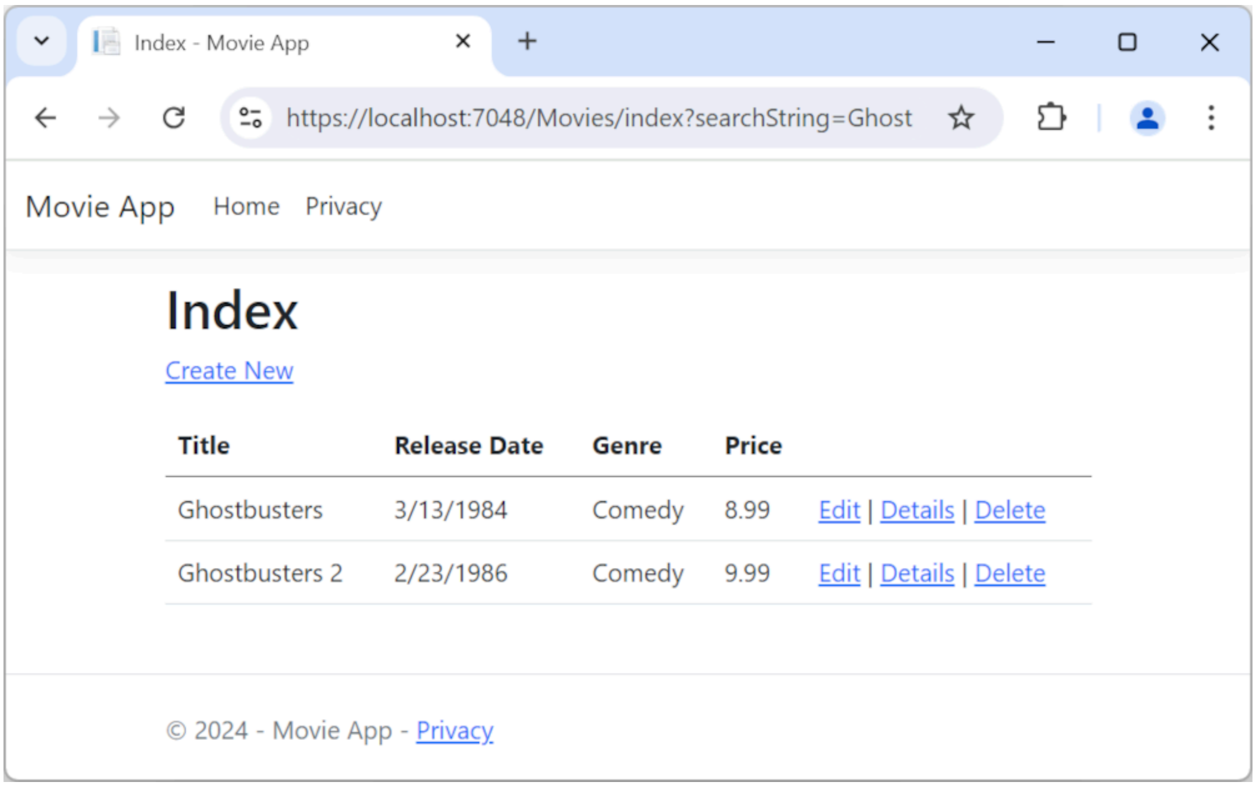
The `s => s.Title!.ToUpper().Contains(searchString.ToUpper())` code above is a Lambda Expression. Lambdas are used in method-based LINQ queries as arguments to standard query operator methods such as the Where method or `Contains` (used in the code above). LINQ queries are not executed when they're defined or when they're modified by calling a method such as `Where`, `Contains`, or `OrderBy`. Rather, query execution is deferred. That means that the evaluation of an expression is delayed until its realized value is actually iterated over or the `ToListAsync` method is called. For more information about deferred query execution, see Query Execution.

ⓘ Note

The Contains method is run on the database, not in the C# code. The case sensitivity on the query depends on the database and the collation. On SQL Server, `Contains` maps to SQL LIKE, which is case insensitive. SQLite with the default collation is a mixture of case sensitive and case ***IN***sensitive, depending on the query. For information on making case insensitive SQLite queries, see the following:

- How to use case-insensitive query with Sqlite provider? (dotnet/efcore #11414)
- How to make a SQLite column case insensitive (dotnet/AspNetCore.Docs #22314)
- Collations and Case Sensitivity

Navigate to `/Movies/Index`. Append a query string such as `?searchString=Ghost` to the URL. The filtered movies are displayed.



Index

Create New

Title	Release Date	Genre	Price	
Ghostbusters	3/13/1984	Comedy	8.99	Edit | Details | Delete
Ghostbusters 2	2/23/1986	Comedy	9.99	Edit | Details | Delete

© 2024 - Movie App - Privacy

If you change the signature of the `Index` method to have a parameter named `id`, the `id` parameter will match the optional `{id}` placeholder for the default routes set in `Program.cs`.

```C#
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
```

Change the parameter to `id` and change all occurrences of `searchString` to `id`.

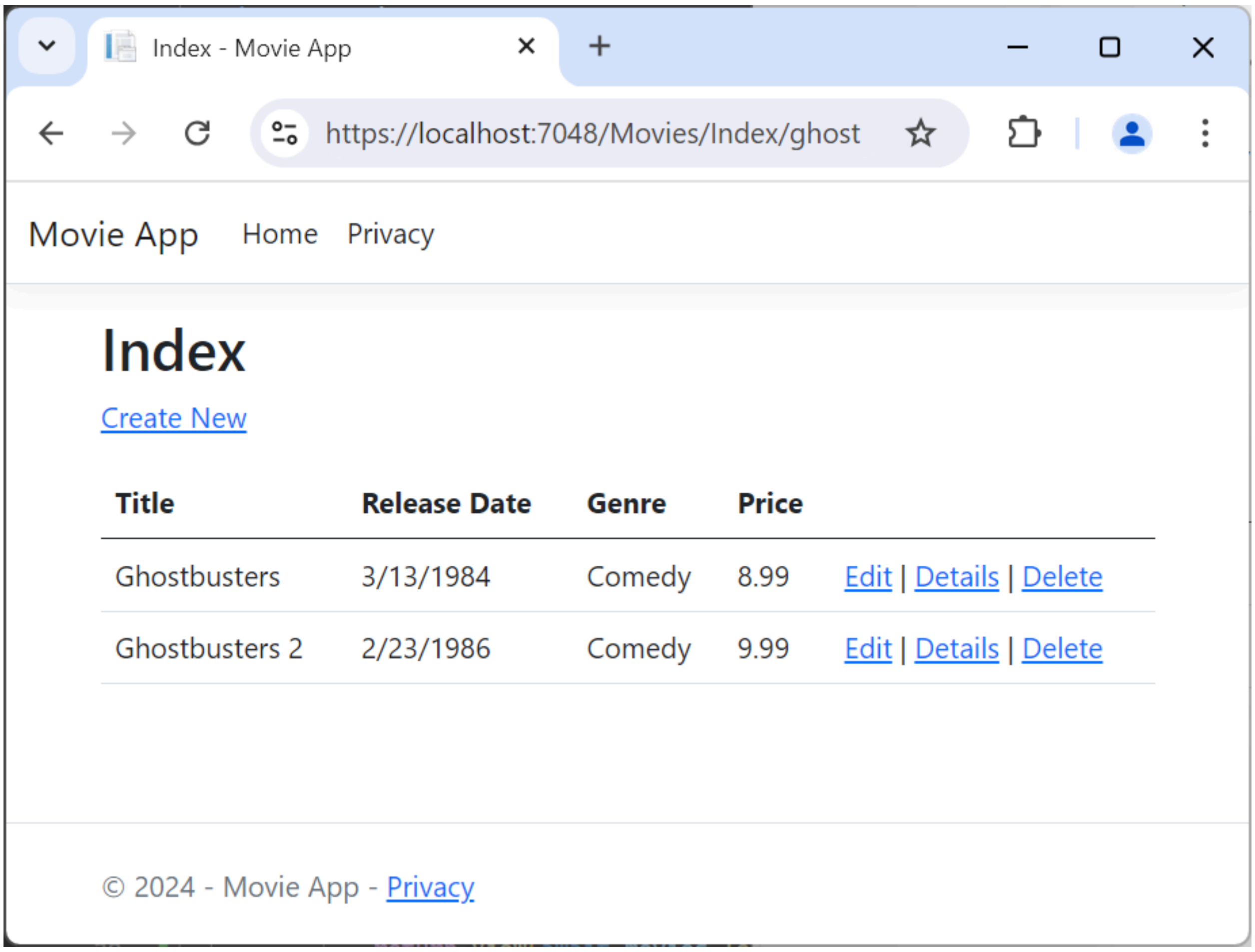
The previous `Index` method:

```C#
public async Task<IActionResult> Index(string searchString)
{
    if (_context.Movie == null)
    {
        return Problem("Entity set 'MvcMovieContext.Movie'  is null.");
    }

    var movies = from m in _context.Movie
                 select m;
    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s =>
s.Title!.ToUpper().Contains(searchString.ToUpper()));
    }

    return View(await movies.ToListAsync());
}
```

The updated `Index` method with `id` parameter:

```C#
public async Task<IActionResult> Index(string id)
{
    if (_context.Movie == null)
    {
        return Problem("Entity set 'MvcMovieContext.Movie'  is null.");
    }

    var movies = from m in _context.Movie
                 select m;

    if (!String.IsNullOrEmpty(id))
    {
        movies = movies.Where(s =>
s.Title!.ToUpper().Contains(id.ToUpper()));
    }

    return View(await movies.ToListAsync());
}
```

You can now pass the search title as route data (a URL segment) instead of as a query string value.


Index - Movie App

https://localhost:7048/Movies/Index/ghost

Movie App Home Privacy

Index

Create New

Title	Release Date	Genre	Price	
Ghostbusters	3/13/1984	Comedy	8.99	Edit | Details | Delete
Ghostbusters 2	2/23/1986	Comedy	9.99	Edit | Details | Delete

© 2024 - Movie App - Privacy

However, you can't expect users to modify the URL every time they want to search for a movie. So now you'll add UI elements to help them filter movies. If you changed the signature of the `Index` method to test how to pass the route-bound `ID` parameter, change it back so that it takes a parameter named `searchString`:

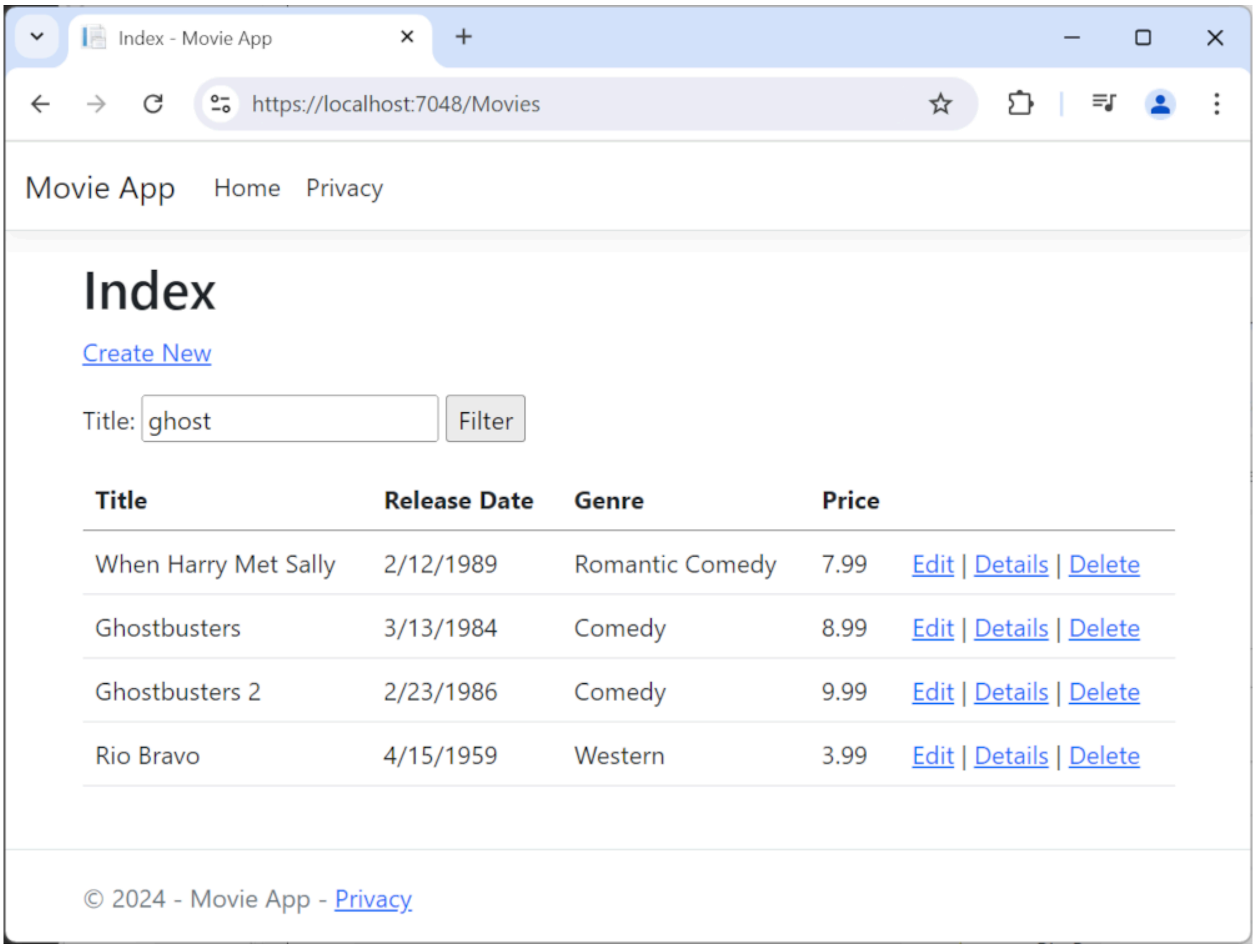
C#

```csharp
public async Task<IActionResult> Index(string searchString)
{
    if (_context.Movie == null)
    {
        return Problem("Entity set 'MvcMovieContext.Movie'  is null.");
    }

    var movies = from m in _context.Movie
                 select m;

    if (!String.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s =>
s.Title!.ToUpper().Contains(searchString.ToUpper()));
    }

    return View(await movies.ToListAsync());
}
```

Open the `Views/Movies/Index.cshtml` file, and add the `<form>` markup highlighted below:

```
CSHTML

@model IEnumerable<MvcMovie.Models.Movie>

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

<h1>Index</h1>

<p>
    <a asp-action="Create">Create New</a>
</p>

<form asp-controller="Movies" asp-action="Index">
    <p>
        <label>Title: <input type="text" name="SearchString" /></label>
        <input type="submit" value="Filter" />
    </p>
</form>
<table class="table">
```

The HTML `<form>` tag uses the Form Tag Helper, so when you submit the form, the filter string is posted to the `Index` action of the movies controller. Save your changes and then test the filter.



Index

Create New

Title: ghost Filter

Title	Release Date	Genre	Price	
When Harry Met Sally	2/12/1989	Romantic Comedy	7.99	Edit | Details | Delete
Ghostbusters	3/13/1984	Comedy	8.99	Edit | Details | Delete
Ghostbusters 2	2/23/1986	Comedy	9.99	Edit | Details | Delete
Rio Bravo	4/15/1959	Western	3.99	Edit | Details | Delete

© 2024 - Movie App - Privacy

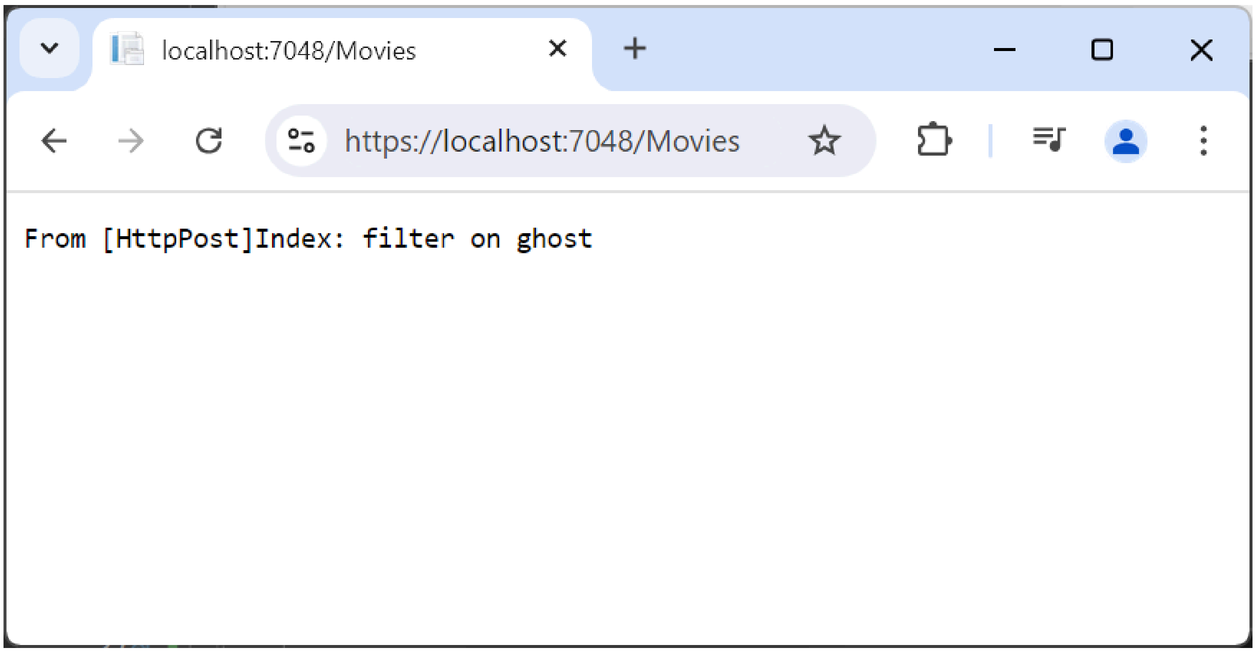
There's no `[HttpPost]` overload of the `Index` method as you might expect. You don't need it, because the method isn't changing the state of the app, just filtering data.

You could add the following `[HttpPost] Index` method.

```C#
[HttpPost]
public string Index(string searchString, bool notUsed)
{
    return "From [HttpPost]Index: filter on " + searchString;
}
```

The `notUsed` parameter is used to create an overload for the `Index` method. We'll talk about that later in the tutorial.

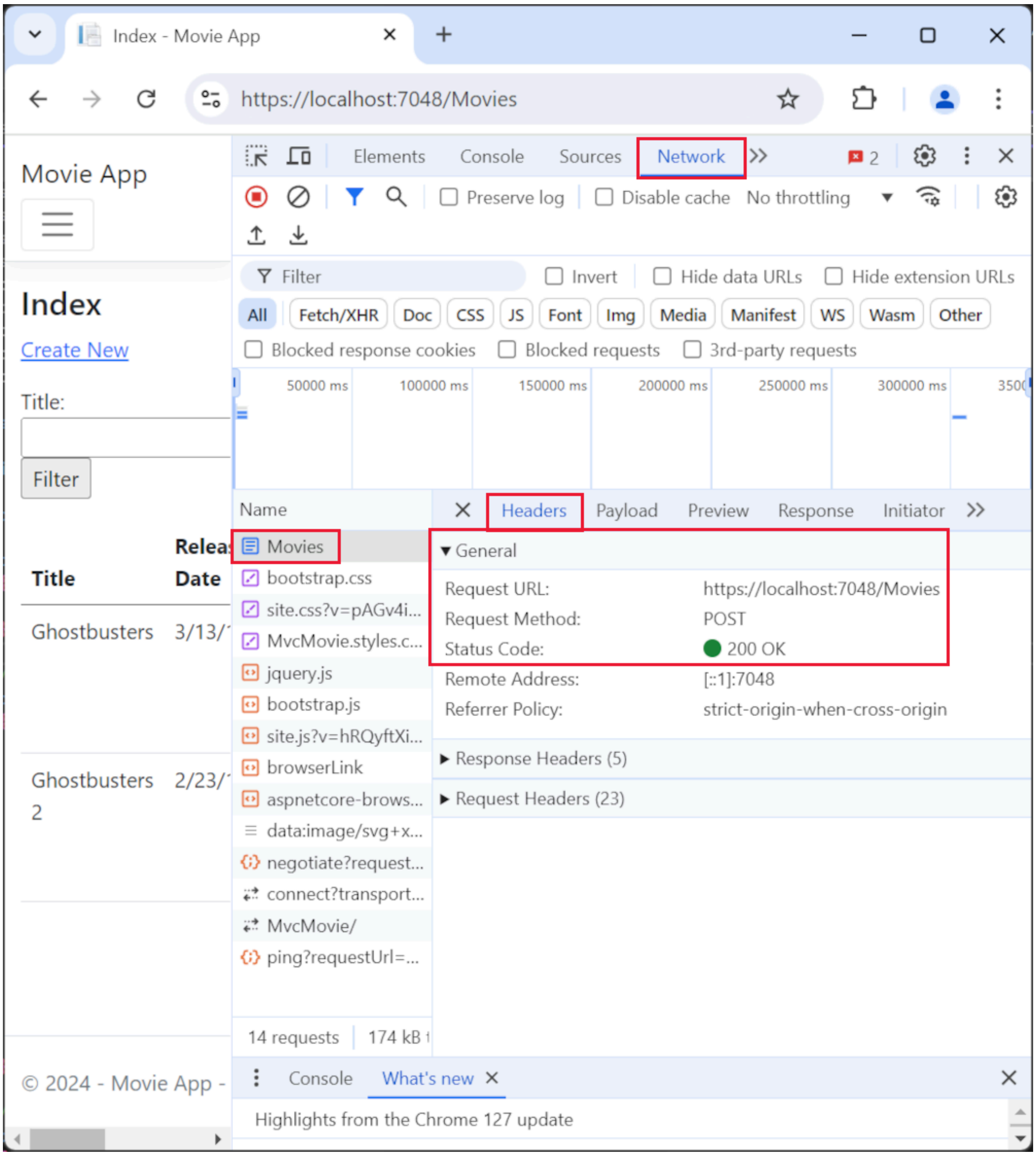
If you add this method, the action invoker would match the `[HttpPost] Index` method, and the `[HttpPost] Index` method would run as shown in the image below.



```
From [HttpPost]Index: filter on ghost
```

However, even if you add this `[HttpPost]` version of the `Index` method, there's a limitation in how this has all been implemented. Imagine that you want to bookmark a particular search or you want to send a link to friends that they can click in order to see the same filtered list of movies. Notice that the URL for the HTTP POST request is the same as the URL for the GET request (localhost:{PORT}/Movies/Index) -- there's no search information in the URL. The search string information is sent to the server as a form field value. You can verify that with the browser Developer tools or the excellent Fiddler tool.

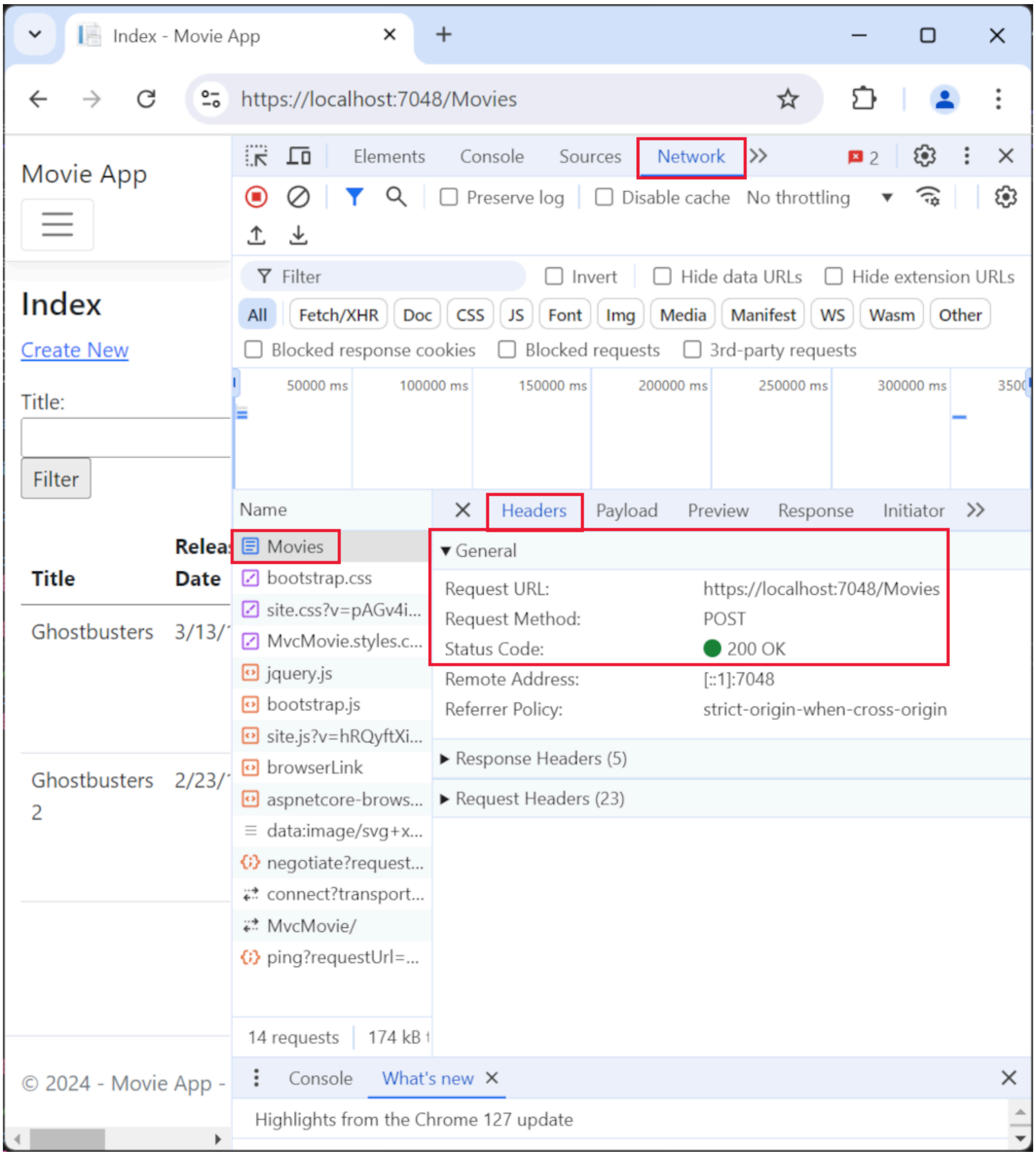
The following image shows the Chrome browser Developer tools with the **Network** and **Headers** tabs selected:



General

Request URL:	https://localhost:7048/Movies
Request Method:	POST
Status Code:	● 200 OK
Remote Address:	[::1]:7048
Referrer Policy:	strict-origin-when-cross-origin

The **Network** and **Payload** tabs are selected to view form data:



General

Request URL:	https://localhost:7048/Movies
Request Method:	POST
Status Code:	200 OK
Remote Address:	[::1]:7048
Referrer Policy:	strict-origin-when-cross-origin

▶ Response Headers (5)

▶ Request Headers (23)

You can see the search parameter and XSRF token in the request body. Note, as mentioned in the previous tutorial, the Form Tag Helper generates an XSRF antiforgery token. We're not modifying data, so we don't need to validate the token in the controller method.

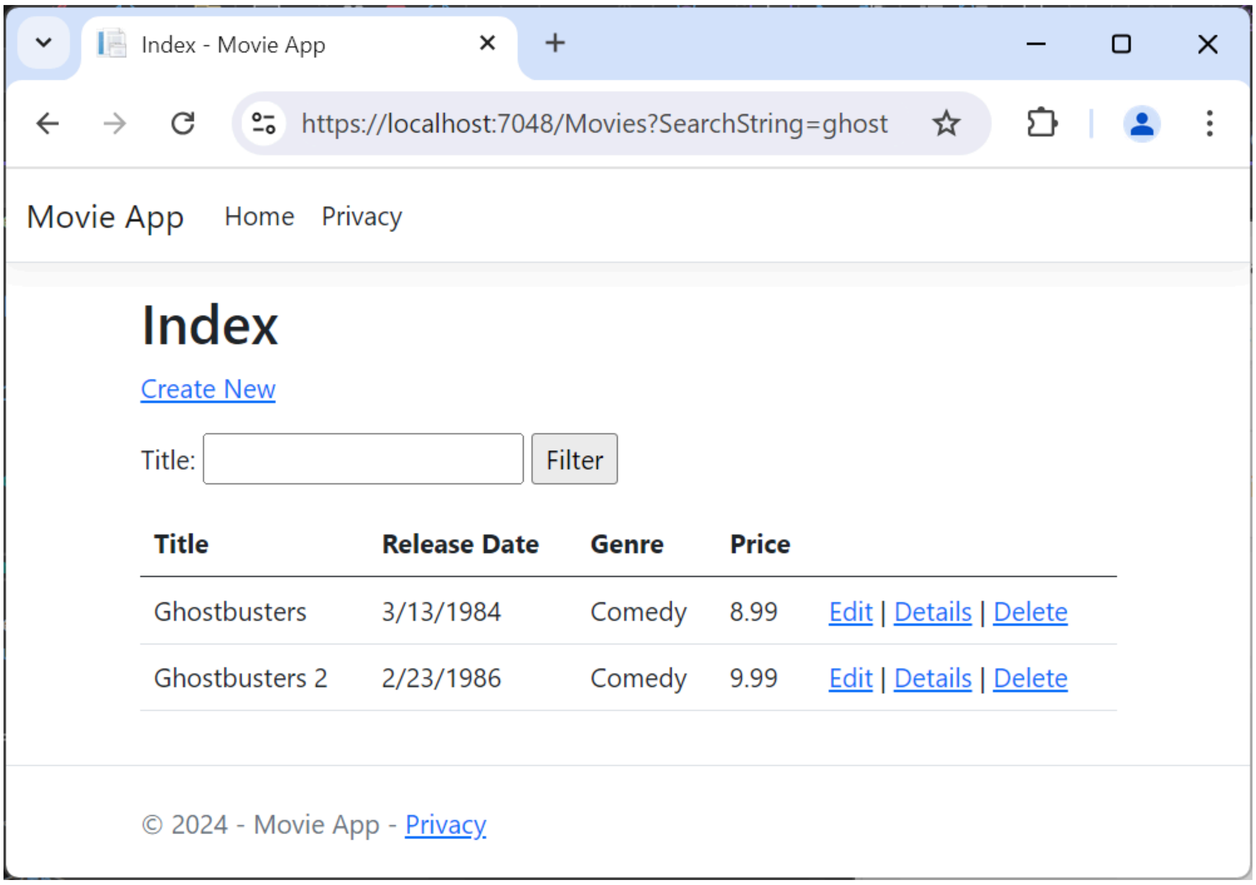
Because the search parameter is in the request body and not the URL, you can't capture that search information to bookmark or share with others. Fix this by specifying the request should be `HTTP GET` in the `form` tag found in the `Views/Movies/Index.cshtml` file.

CSHTML

```
@model IEnumerable<MvcMovie.Models.Movie>
```

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

<h1>Index</h1>

<p>
    <a asp-action="Create">Create New</a>
</p>

<form asp-controller="Movies" asp-action="Index" method="get">
    <p>
        <label>Title: <input type="text" name="SearchString" /></label>
        <input type="submit" value="Filter" />
    </p>
</form>
<table class="table">
```

Now when you submit a search, the URL contains the search query string. Searching will also go to the `HttpGet Index` action method, even if you have a `HttpPost Index` method.



Index

Create New

Title: Filter

Title	Release Date	Genre	Price	
Ghostbusters	3/13/1984	Comedy	8.99	Edit | Details | Delete
Ghostbusters 2	2/23/1986	Comedy	9.99	Edit | Details | Delete

© 2024 - Movie App - Privacy

Add Search by genre

Add the following `MovieGenreViewModel` class to the *Models* folder:

C#

```
using Microsoft.AspNetCore.Mvc.Rendering;
using System.Collections.Generic;

namespace MvcMovie.Models;

public class MovieGenreViewModel
{
    public List<Movie>? Movies { get; set; }
    public SelectList? Genres { get; set; }
    public string? MovieGenre { get; set; }
    public string? SearchString { get; set; }
}
```

The movie-genre view model will contain:

- A list of movies.
- A `SelectList` containing the list of genres. This allows the user to select a genre from the list.
- `MovieGenre`, which contains the selected genre.
- `SearchString`, which contains the text users enter in the search text box.

Replace the `Index` method in `MoviesController.cs` with the following code:

C#

```
// GET: Movies
public async Task<IActionResult> Index(string movieGenre, string
searchString)
{
    if (_context.Movie == null)
    {
        return Problem("Entity set 'MvcMovieContext.Movie'  is null.");
    }

    // Use LINQ to get list of genres.
    IQueryable<string> genreQuery = from m in _context.Movie
                                    orderby m.Genre
                                    select m.Genre;
    var movies = from m in _context.Movie
                 select m;

    if (!string.IsNullOrEmpty(searchString))
    {
        movies = movies.Where(s =>
s.Title!.ToUpper().Contains(searchString.ToUpper()));
    }

    if (!string.IsNullOrEmpty(movieGenre))
    {
        movies = movies.Where(x => x.Genre == movieGenre);
    }

    var movieGenreVM = new MovieGenreViewModel
    {
        Genres = new SelectList(await genreQuery.Distinct().ToListAsync()),
        Movies = await movies.ToListAsync()
    };

    return View(movieGenreVM);
}
```

The following code is a `LINQ` query that retrieves all the genres from the database.

C#

```
// Use LINQ to get list of genres.
IQueryable<string> genreQuery = from m in _context.Movie
                                orderby m.Genre
                                select m.Genre;
```

The `SelectList` of genres is created by projecting the distinct genres (we don't want our select list to have duplicate genres).

When the user searches for the item, the search value is retained in the search box.

Add search by genre to the Index view

Update `Index.cshtml` found in *Views/Movies/* as follows:

CSHTML

```
@model MvcMovie.Models.MovieGenreViewModel

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

<h1>Index</h1>

<p>
    <a asp-action="Create">Create New</a>
</p>
<form asp-controller="Movies" asp-action="Index" method="get">
    <p>

        <select asp-for="MovieGenre" asp-items="Model.Genres">
            <option value="">All</option>
        </select>
```

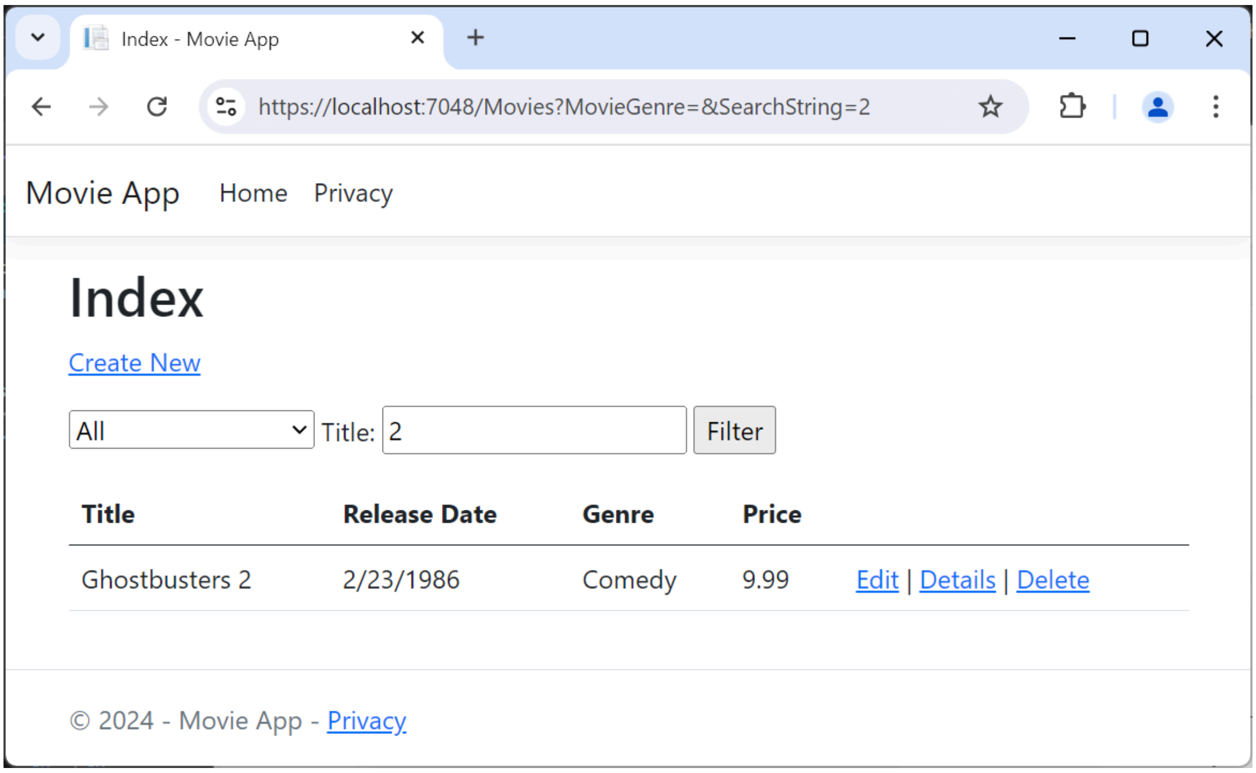
```
          <label>Title: <input type="text" asp-for="SearchString" /></label>
          <input type="submit" value="Filter" />
      </p>
  </form>

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

Examine the lambda expression used in the following HTML Helper:

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

In the preceding code, the `DisplayNameFor` HTML Helper inspects the `Title` property referenced in the lambda expression to determine the display name. Since the lambda expression is inspected rather than evaluated, you don't receive an access violation when `model`, `model.Movies`, or `model.Movies[0]` are `null` or empty. When the lambda expression is evaluated (for example, `@Html.DisplayFor(modelItem => item.Title)`), the model's property values are evaluated. The `!` after `model.Movies` is the null-forgiving operator, which is used to declare that `Movies` isn't null.

Test the app by searching by genre, by movie title, and by both:



Index

Create New

All Title: 2 Filter

Title	Release Date	Genre	Price	
Ghostbusters 2	2/23/1986	Comedy	9.99	Edit | Details | Delete

© 2024 - Movie App - Privacy

Previous Next

Part 8, add a new field to an ASP.NET Core MVC app

Article • 07/30/2024

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the .NET 9 version of this article.

By Rick Anderson

In this section Entity Framework Migrations is used to:

- Add a new field to the model.
- Migrate the new field to the database.

When Entity Framework (EF) is used to automatically create a database from model classes:

- A table is added to the database to track the schema of the database.
- The database is verified to be in sync with the model classes it was generated from. If they aren't in sync, EF throws an exception. This makes it easier to find inconsistent database/code issues.

Add a Rating Property to the Movie Model

Add a `Rating` property to `Models/Movie.cs`:

C#

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

namespace MvcMovie.Models;

public class Movie
{
    public int Id { get; set; }
    public string? Title { get; set; }
    [Display(Name = "Release Date")]
    [DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }
    public string? Genre { get; set; }

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

Build the app

Visual Studio

Press `Ctrl`+`Shift`+`B`

Because you've added a new field to the `Movie` class, you need to update the property binding list so this new property will be included. In `MoviesController.cs`, update the `[Bind]` attribute for both the `Create` and `Edit` action methods to include the `Rating` property:

C#

```
[Bind("Id,Title,ReleaseDate,Genre,Price,Rating")]
```

Update the view templates in order to display, create, and edit the new `Rating` property in the browser view.

Edit the `/Views/Movies/Index.cshtml` file and add a `Rating` field:

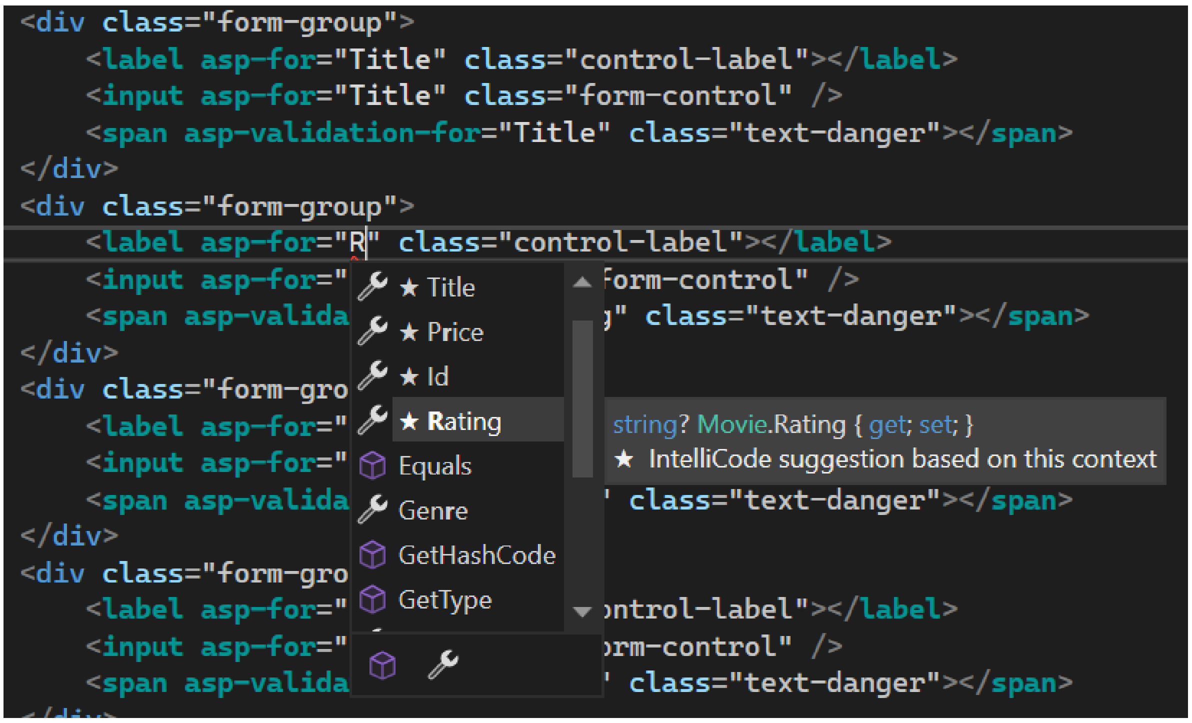
CSHTML

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

Update the `/Views/Movies/Create.cshtml` with a `Rating` field.

Visual Studio

You can copy/paste the previous "form group" and let intelliSense help you update the fields. IntelliSense works with Tag Helpers.



```
<div class="form-group">
    <label asp-for="Title" class="control-label"></label>
    <input asp-for="Title" class="form-control" />
    <span asp-validation-for="Title" class="text-danger"></span>
</div>
<div class="form-group">
    <label asp-for="R" class="control-label"></label>
    <input asp-for="                  form-control" />
    <span asp-valida                  g" class="text-danger"></span>
</div>
<div class="form-gro
    <label asp-for="
    <input asp-for="
    <span asp-valida                  ' class="text-danger"></span>
</div>
<div class="form-gro
    <label asp-for="                  ontrol-label"></label>
    <input asp-for="                  orm-control" />
    <span asp-valida                  ' class="text-danger"></span>
```

★ Title
★ Price
★ Id
★ Rating
Equals
Genre
GetHashCode
GetType
string? Movie.Rating { get; set; }
★ IntelliCode suggestion based on this context

Add the `Rating` property to the remaining `Create.cshtml`, `Delete.cshtml`, `Details.cshtml`, and `Edit.cshtml` view templates.

Update the `SeedData` class so that it provides a value for the new column. A sample change is shown below, but you'll want to make this change for each `new Movie`.

C#

```csharp
new Movie
{
    Title = "When Harry Met Sally",
    ReleaseDate = DateTime.Parse("1989-1-11"),
    Genre = "Romantic Comedy",
    Rating = "R",
    Price = 7.99M
},
```

The app won't work until the DB is updated to include the new field. If it's run now, the following `SqlException` is thrown:

```
SqlException: Invalid column name 'Rating'.
```

This error occurs because the updated Movie model class is different than the schema of the Movie table of the existing database. (There's no `Rating` column in the database table.)

There are a few approaches to resolving the error:

1. Have the Entity Framework automatically drop and re-create the database based on the new model class schema. This approach is very convenient early in the development cycle when you're doing active development on a test database; it allows you to quickly evolve the model and database schema together. The downside, though, is that you lose existing data in the database — so you don't want to use this approach on a production database! Using an initializer to automatically seed a database with test data is often a productive way to develop an application. This is a good approach for early development and when using SQLite.

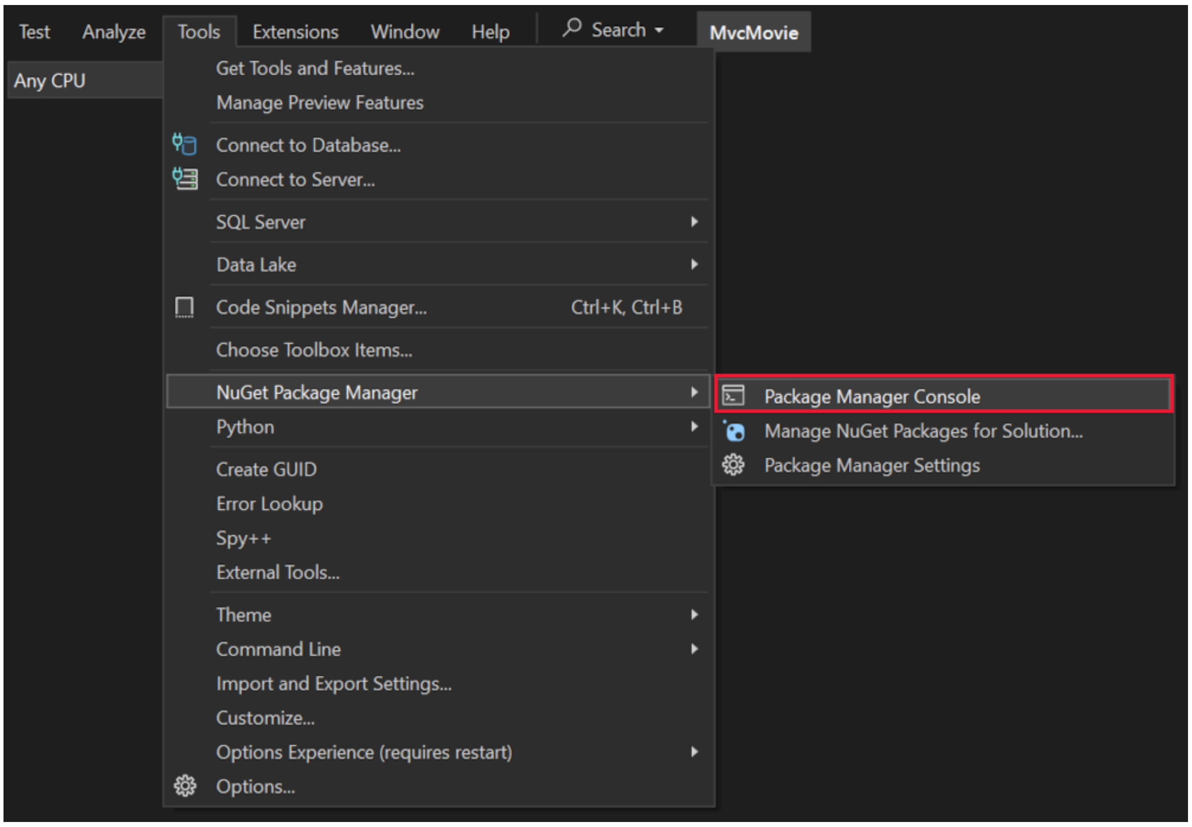
2. Explicitly modify the schema of the existing database so that it matches the model classes. The advantage of this approach is that you keep your data. You can make this change either manually or by creating a database change script.

3. Use Entity Framework Migrations to update the database schema.

For this tutorial, Entity Framework Migrations is used.

Visual Studio

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



In the Package Manager Console, enter the following command:

PowerShell

```
Add-Migration Rating
```

The `Add-Migration` command tells the migration framework to examine the current `Movie` model with the current `Movie` DB schema and create the necessary code to migrate the DB to the new model.

The name "Rating" is arbitrary and is used to name the migration file. It's helpful to use a meaningful name for the migration file.

If all the records in the DB are deleted, the initialize method will seed the DB and include the `Rating` field.

In the Package Manager Console, enter the following command:

PowerShell

```
Update-Database
```

The Update-Database command runs the Up method in migrations that have not been applied.

Run the app and verify you can create, edit, and display movies with a `Rating` field.

Previous Next

Part 9, add validation to an ASP.NET Core MVC app

Article • 07/30/2024

ⓘ **Important**

This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

For the current release, see the .NET 9 version of this article.

By Rick Anderson

In this section:

- Validation logic is added to the `Movie` model.
- You ensure that the validation rules are enforced any time a user creates or edits a movie.

Keeping things DRY

One of the design tenets of MVC is DRY ("Don't Repeat Yourself"). ASP.NET Core MVC encourages you to specify functionality or behavior only once, and then have it be reflected everywhere in an app. This reduces the amount of code you need to write and makes the code you do write less error prone, easier to test, and easier to maintain.

The validation support provided by MVC and Entity Framework Core is a good example of the DRY principle in action. You can declaratively specify validation rules in one place (in the model class) and the rules are enforced everywhere in the app.

Delete the previously edited data

In the next step, validation rules are added that don't allow null values. Run the app, navigate to `/Movies/Index`, delete all listed movies, and stop the app. The app will use the seed data the next time it is run.

Add validation rules to the movie model

The DataAnnotations namespace provides a set of built-in validation attributes that are applied declaratively to a class or property. DataAnnotations also contains formatting attributes like `DataType` that help with formatting and don't provide any validation.

Update the `Movie` class to take advantage of the built-in validation attributes `Required`, `StringLength`, `RegularExpression`, `Range` and the `DataType` formatting attribute.

C#

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

namespace MvcMovie.Models;

public class Movie
{
    public int Id { get; set; }

    [StringLength(60, MinimumLength = 3)]
    [Required]
    public string? Title { get; set; }

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

    [Range(1, 100)]
    [DataType(DataType.Currency)]
    [Column(TypeName = "decimal(18, 2)")]
    public decimal Price { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z\s]*$")]
    [Required]
    [StringLength(30)]
    public string? Genre { get; set; }

    [RegularExpression(@"^[A-Z]+[a-zA-Z0-9""'\s-]*$")]
    [StringLength(5)]
    [Required]
    public string? Rating { get; set; }
}
```

The validation attributes specify behavior that you want to enforce on the model properties they're applied to:

- The `Required` and `MinimumLength` attributes indicate that a property must have a value; but nothing prevents a user from entering white space to satisfy this validation.

- The `RegularExpression` attribute is used to limit what characters can be input. In the preceding code, "Genre":
  - Must only use letters.
  - The first letter is required to be uppercase. White spaces are allowed while numbers, and special characters are not allowed.

- The `RegularExpression` "Rating":
  - Requires that the first character be an uppercase letter.
  - Allows special characters and numbers in subsequent spaces. "PG-13" is valid for a rating, but fails for a "Genre".

- The `Range` attribute constrains a value to within a specified range.

- The `StringLength` attribute lets you set the maximum length of a string property, and optionally its minimum length.

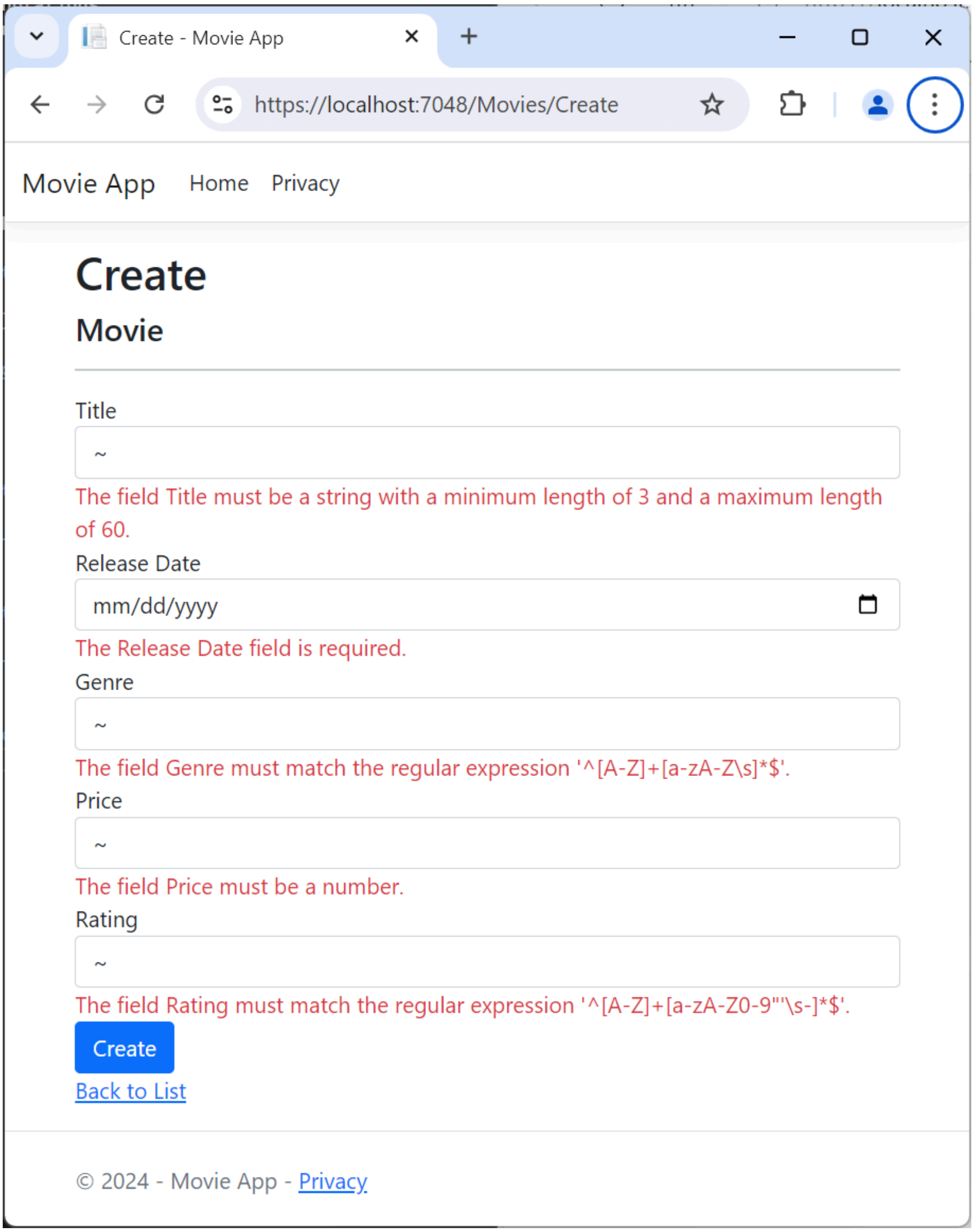
- Value types (such as `decimal`, `int`, `float`, `DateTime`) are inherently required and don't need the `[Required]` attribute.

Having validation rules automatically enforced by ASP.NET Core helps make your app more robust. It also ensures that you can't forget to validate something and inadvertently let bad data into the database.

Validation Error UI

Run the app and navigate to the Movies controller.

Select the **Create New** link to add a new movie. Fill out the form with some invalid values. As soon as jQuery client side validation detects the error, it displays an error message.



Create

Movie

Title

~

The field Title must be a string with a minimum length of 3 and a maximum length of 60.

Release Date

mm/dd/yyyy

The Release Date field is required.

Genre

~

The field Genre must match the regular expression '^[A-Z]+[a-zA-Z\s]*$'.

Price

~

The field Price must be a number.

Rating

~

The field Rating must match the regular expression '^[A-Z]+[a-zA-Z0-9"'\s-]*$'.

Create

Back to List

© 2024 - Movie App - Privacy

ⓘ **Note**

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

Notice how the form has automatically rendered an appropriate validation error message in each field containing an invalid value. The errors are enforced both client-side (using JavaScript and jQuery) and server-side (in case a user has JavaScript disabled).

A significant benefit is that you didn't need to change a single line of code in the `MoviesController` class or in the `Create.cshtml` view in order to enable this validation UI. The controller and views you created earlier in this tutorial automatically picked up the validation rules that you specified by using validation attributes on the properties of the `Movie` model class. Test validation using the `Edit` action method, and the same validation is applied.

The form data isn't sent to the server until there are no client side validation errors. You can verify this by putting a break point in the `HTTP Post` method, by using the Fiddler tool, or the F12 Developer tools.

How validation works

You might wonder how the validation UI was generated without any updates to the code in the controller or views. The following code shows the two `Create` methods.

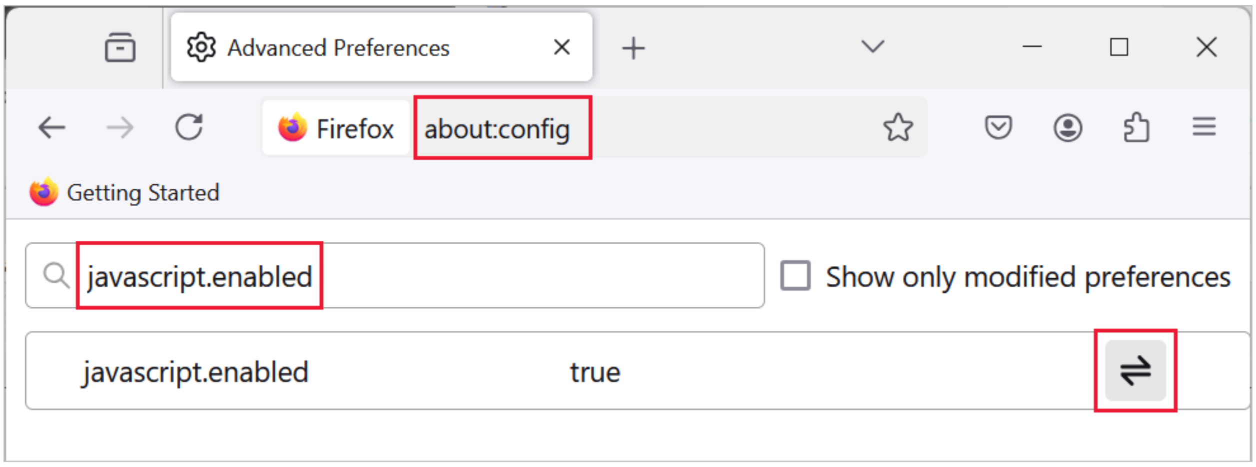
C#

```
// GET: Movies/Create
public IActionResult Create()
{
    return View();
}

// POST: Movies/Create
// To protect from overposting attacks, enable the specific properties you
want to bind to.
// For more details, see http://go.microsoft.com/fwlink/?LinkId=317598.
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult>
Create([Bind("Id,Title,ReleaseDate,Genre,Price,Rating")] Movie movie)
{
    if (ModelState.IsValid)
    {
        _context.Add(movie);
        await _context.SaveChangesAsync();
        return RedirectToAction(nameof(Index));
    }
    return View(movie);
}
```

The first (HTTP GET) `Create` action method displays the initial Create form. The second (`[HttpPost]`) version handles the form post. The second `Create` method (The `[HttpPost]` version) calls `ModelState.IsValid` to check whether the movie has any validation errors. Calling this method evaluates any validation attributes that have been applied to the object. If the object has validation errors, the `Create` method re-displays the form. If there are no errors, the method saves the new movie in the database. In our movie example, the form isn't posted to the server when there are validation errors detected on the client side; the second `Create` method is never called when there are client side validation errors. If you disable JavaScript in your browser, client validation is disabled and you can test the HTTP POST `Create` method `ModelState.IsValid` detecting any validation errors.

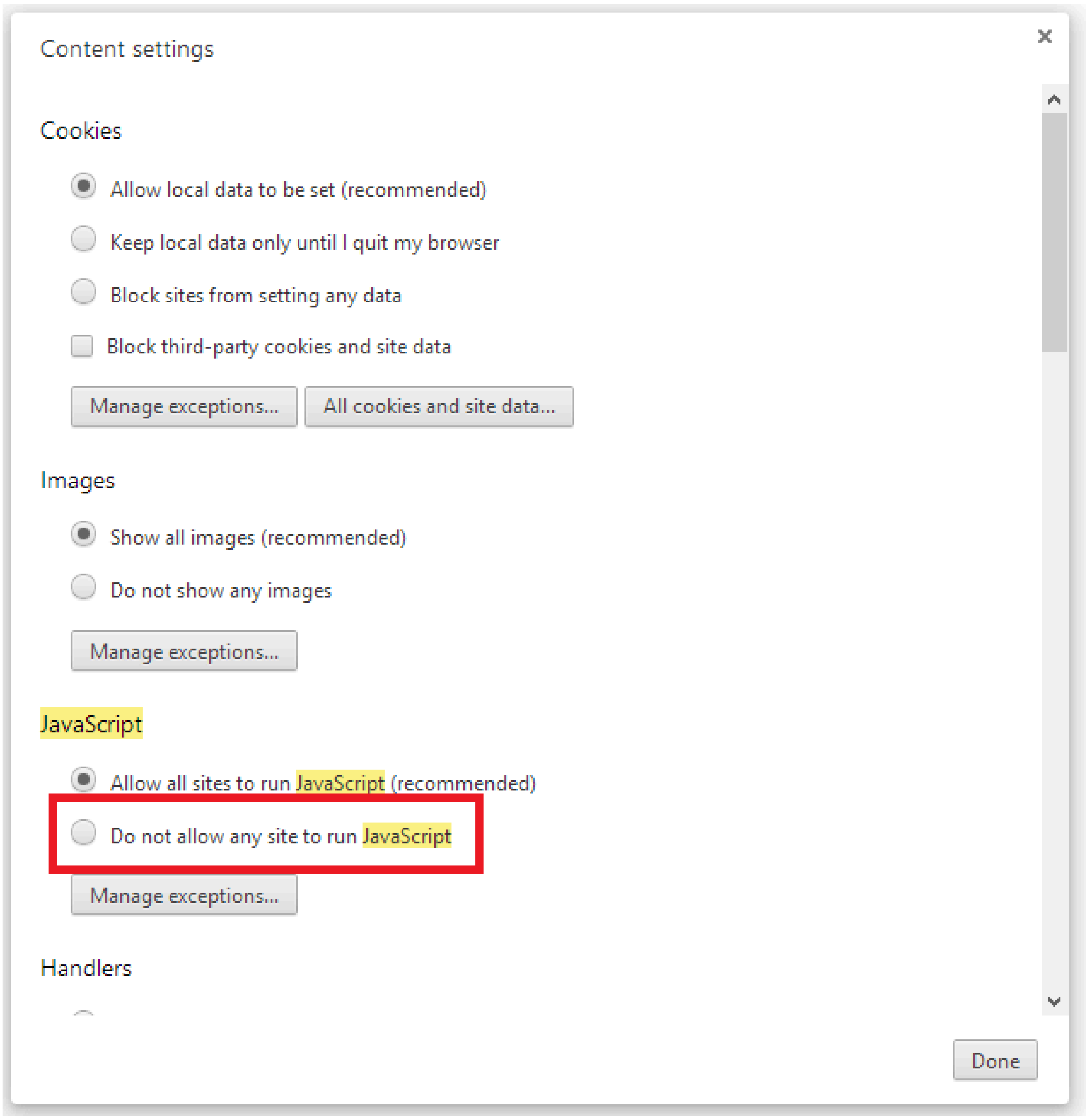
You can set a break point in the `[HttpPost] Create` method and verify the method is never called, client side validation won't submit the form data when validation errors are detected. If you disable JavaScript in your browser, then submit the form with errors, the break point will be hit. You still get full validation without JavaScript.

The following image shows how to disable JavaScript in the Firefox browser.



javascript.enabled true

The following image shows how to disable JavaScript in the Chrome browser.



Content settings

Cookies

- [●] Allow local data to be set (recommended)
- [ ] Keep local data only until I quit my browser
- [ ] Block sites from setting any data
- [ ] Block third-party cookies and site data

Images

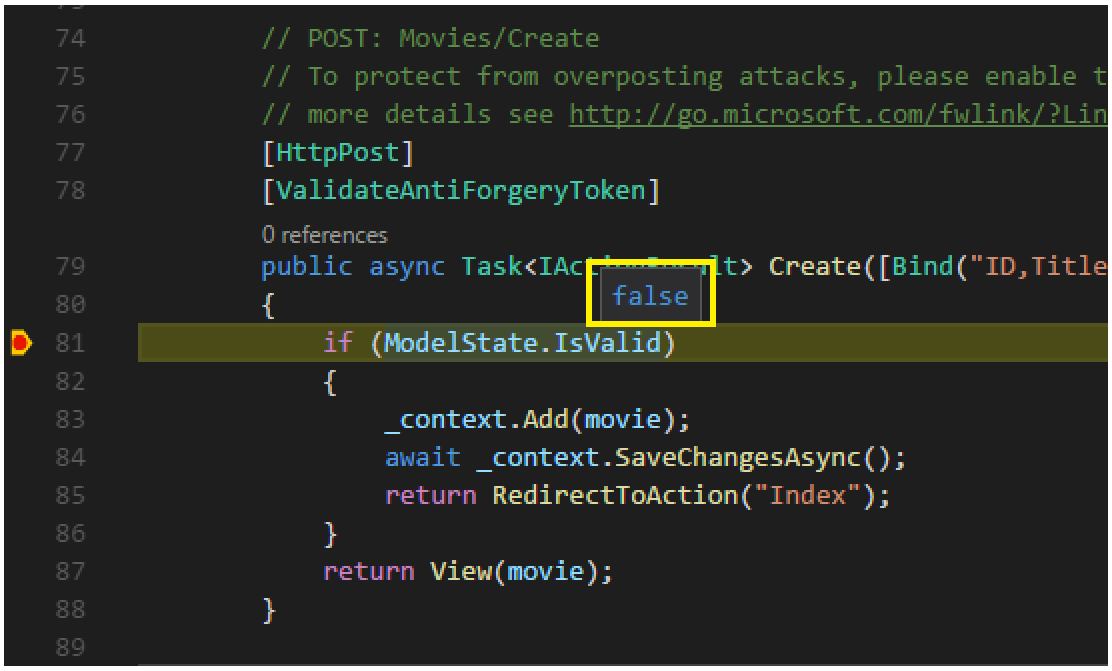
- [●] Show all images (recommended)
- [ ] Do not show any images

JavaScript

- [●] Allow all sites to run JavaScript (recommended)
- [ ] Do not allow any site to run JavaScript

Handlers

After you disable JavaScript, post invalid data and step through the debugger.



```
        // POST: Movies/Create
        // To protect from overposting attacks, please enable t
        // more details see http://go.microsoft.com/fwlink/?Lin
        [HttpPost]
        [ValidateAntiForgeryToken]
        0 references
        public async Task<IAc          t> Create([Bind("ID,Title
        {                   false
            if (ModelState.IsValid)
            {
                _context.Add(movie);
                await _context.SaveChangesAsync();
                return RedirectToAction("Index");
            }
            return View(movie);
        }
```

A portion of the `Create.cshtml` view template is shown in the following markup:

HTML

```html
<h4>Movie</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="Create">
            <div asp-validation-summary="ModelOnly" class="text-danger">
</div>
            <div class="form-group">
                <label asp-for="Title" class="control-label"></label>
                <input asp-for="Title" class="form-control" />
                <span asp-validation-for="Title" class="text-danger"></span>
            </div>

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

The preceding markup is used by the action methods to display the initial form and to redisplay it in the event of an error.

The Input Tag Helper uses the DataAnnotations attributes and produces HTML attributes needed for jQuery Validation on the client side. The Validation Tag Helper displays validation errors. See Validation for more information.

What's really nice about this approach is that neither the controller nor the `Create` view template knows anything about the actual validation rules being enforced or about the specific error messages displayed. The validation rules and the error strings are specified only in the `Movie` class. These same validation rules are automatically applied to the `Edit` view and any other views templates you might create that edit your model.

When you need to change validation logic, you can do so in exactly one place by adding validation attributes to the model (in this example, the `Movie` class). You won't have to worry about different parts of the application being inconsistent with how the rules are enforced — all validation logic will be defined in one place and used everywhere. This keeps the code very clean, and makes it easy to maintain and evolve. And it means that you'll be fully honoring the DRY principle.

Using DataType Attributes

Open the `Movie.cs` file and examine the `Movie` class. The `System.ComponentModel.DataAnnotations` namespace provides formatting attributes in addition to the built-in set of validation attributes. We've already applied a `DataType` enumeration value to the release date and to the price fields. The following code shows the `ReleaseDate` and `Price` properties with the appropriate `DataType` attribute.

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

[Range(1, 100)]
[DataType(DataType.Currency)]
[Column(TypeName = "decimal(18, 2)")]
public decimal Price { get; set; }
```

The `DataType` attributes only provide hints for the view engine to format the data and supplies elements/attributes such as `<a>` for URL's and `<a href="mailto:EmailAddress.com">` for email. You can use the `RegularExpression` attribute to validate the format of the data. The `DataType` attribute is used to specify a data type that's more specific than the database intrinsic type, they're not validation attributes. In this case we only want to keep track of the date, not the time. The `DataType` Enumeration provides for many data types, such as Date, Time, PhoneNumber, Currency, EmailAddress and more. The `DataType` attribute can also enable the application to automatically provide type-specific features. For example, a `mailto:` link can be created for `DataType.EmailAddress`, and a date selector can be provided for `DataType.Date` in browsers that support HTML5. The `DataType` attributes emit HTML 5 `data-` (pronounced data dash) attributes that HTML 5 browsers can understand. The `DataType` attributes do **not** provide any validation.

`DataType.Date` doesn't specify the format of the date that's displayed. By default, the data field is displayed according to the default formats based on the server's `CultureInfo`.

The `DisplayFormat` attribute is used to explicitly specify the date format:

```C#
[DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode =
true)]
public DateTime ReleaseDate { get; set; }
```

The `ApplyFormatInEditMode` setting specifies that the formatting should also be applied when the value is displayed in a text box for editing. (You might not want that for some fields — for example, for currency values, you probably don't want the currency symbol in the text box for editing.)

You can use the `DisplayFormat` attribute by itself, but it's generally a good idea to use the `DataType` attribute. The `DataType` attribute conveys the semantics of the data as opposed to how to render it on a screen, and provides the following benefits that you don't get with DisplayFormat:

- The browser can enable HTML5 features (for example to show a calendar control, the locale-appropriate currency symbol, email links, etc.)
- By default, the browser will render data using the correct format based on your locale.
- The `DataType` attribute can enable MVC to choose the right field template to render the data (the `DisplayFormat` if used by itself uses the string template).

ⓘ **Note**

jQuery validation doesn't work with the `Range` attribute and `DateTime`. For example, the following code will always display a client side validation error, even when the date is in the specified range:

`[Range(typeof(DateTime), "1/1/1966", "1/1/2020")]`

You will need to disable jQuery date validation to use the `Range` attribute with `DateTime`. It's generally not a good practice to compile hard dates in your models, so using the `Range` attribute and `DateTime` is discouraged.

The following code shows combining attributes on one line:

C#

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

namespace MvcMovie.Models;

public class Movie
{
    public int Id { get; set; }
    [StringLength(60, MinimumLength = 3)]
    public string Title { get; set; }
    [Display(Name = "Release Date"), DataType(DataType.Date)]
    public DateTime ReleaseDate { get; set; }
    [RegularExpression(@"^[A-Z]+[a-zA-Z\s]*$"), Required, StringLength(30)]
    public string Genre { get; set; }
    [Range(1, 100), DataType(DataType.Currency)]
    [Column(TypeName = "decimal(18, 2)")]
    public decimal Price { get; set; }
    [RegularExpression(@"^[A-Z]+[a-zA-Z0-9""'\s-]*$"), StringLength(5)]
    public string Rating { get; set; }
}
```

In the next part of the series, we review the app and make some improvements to the automatically generated `Details` and `Delete` methods.

Additional resources

- Working with Forms
- Globalization and localization
- Introduction to Tag Helpers
- Author Tag Helpers

Previous Next

Part 10, examine the Details and Delete methods of an ASP.NET Core app

Article • 08/05/2024

Important

This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

For the current release, see the .NET 9 version of this article.

By Rick Anderson

Open the Movie controller and examine the `Details` method:

C#

```
// GET: Movies/Details/5
public async Task<IActionResult> Details(int? id)
{
    if (id == null)
    {
        return NotFound();
    }

    var movie = await _context.Movie
        .FirstOrDefaultAsync(m => m.Id == id);
    if (movie == null)
    {
        return NotFound();
    }

    return View(movie);
}
```

The MVC scaffolding engine that created this action method adds a comment showing an HTTP request that invokes the method. In this case it's a GET request with three URL segments, the `Movies` controller, the `Details` method, and an `id` value. Recall these segments are defined in `Program.cs`.

C#

```
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
```

EF makes it easy to search for data using the `FirstOrDefaultAsync` method. An important security feature built into the method is that the code verifies that the search method has found a movie before it tries to do anything with it. For example, a hacker could introduce errors into the site by changing the URL created by the links from `http://localhost:{PORT}/Movies/Details/1` to something like `http://localhost:{PORT}/Movies/Details/12345` (or some other value that doesn't represent an actual movie). If you didn't check for a null movie, the app would throw an exception.

Examine the `Delete` and `DeleteConfirmed` methods.

```C#
// GET: Movies/Delete/5
public async Task<IActionResult> Delete(int? id)
{

    if (id == null)
    {
        return NotFound();
    }

    var movie = await _context.Movie
        .FirstOrDefaultAsync(m => m.Id == id);
    if (movie == null)
    {
        return NotFound();
    }

    return View(movie);
}

// POST: Movies/Delete/5
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
    var movie = await _context.Movie.FindAsync(id);
    if (movie != null)
    {
        _context.Movie.Remove(movie);
    }

    await _context.SaveChangesAsync();
    return RedirectToAction(nameof(Index));
}
```

Note that the `HTTP GET Delete` method doesn't delete the specified movie, it returns a view of the movie where you can submit (HttpPost) the deletion. Performing a delete operation in response to a GET request (or for that matter, performing an edit operation, create operation, or any other operation that changes data) opens up a security hole.

The `[HttpPost]` method that deletes the data is named `DeleteConfirmed` to give the HTTP POST method a unique signature or name. The two method signatures are shown below:

C#

```
// GET: Movies/Delete/5
public async Task<IActionResult> Delete(int? id)
{
```

C#

```
// POST: Movies/Delete/5
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public async Task<IActionResult> DeleteConfirmed(int id)
{
```

The common language runtime (CLR) requires overloaded methods to have a unique parameter signature (same method name but different list of parameters). However, here you need two `Delete` methods -- one for GET and one for POST -- that both have the same parameter signature. (They both need to accept a single integer as a parameter.)

There are two approaches to this problem, one is to give the methods different names. That's what the scaffolding mechanism did in the preceding example. However, this introduces a small problem: ASP.NET maps segments of a URL to action methods by name, and if you rename a method, routing normally wouldn't be able to find that method. The solution is what you see in the example, which is to add the `ActionName("Delete")` attribute to the `DeleteConfirmed` method. That attribute performs mapping for the routing system so that a URL that includes /Delete/ for a POST request will find the `DeleteConfirmed` method.

Another common work around for methods that have identical names and signatures is to artificially change the signature of the POST method to include an extra (unused) parameter. That's what we did in a previous post when we added the `notUsed` parameter. You could do the same thing here for the `[HttpPost] Delete` method:

C#

```
// POST: Movies/Delete/6
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Delete(int id, bool notUsed)
```

Publish to Azure

For information on deploying to Azure, see Tutorial: Build an ASP.NET Core and SQL Database app in Azure App Service.

Reliable web app patterns

See *The Reliable Web App Pattern for.NET* YouTube videos and article for guidance on creating a modern, reliable, performant, testable, cost-efficient, and scalable ASP.NET Core app, whether from scratch or refactoring an existing app.

Views in ASP.NET Core MVC

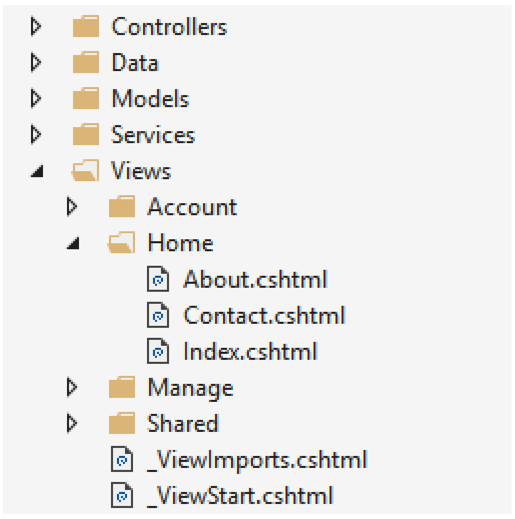
Article • 06/17/2024

By Steve Smith and Dave Brock

This document explains views used in ASP.NET Core MVC applications. For information on Razor Pages, see Introduction to Razor Pages in ASP.NET Core.

In the Model-View-Controller (MVC) pattern, the *view* handles the app's data presentation and user interaction. A view is an HTML template with embedded Razor markup. Razor markup is code that interacts with HTML markup to produce a webpage that's sent to the client.

In ASP.NET Core MVC, views are `.cshtml` files that use the C# programming language in Razor markup. Usually, view files are grouped into folders named for each of the app's controllers. The folders are stored in a `Views` folder at the root of the app:



The `Home` controller is represented by a `Home` folder inside the `Views` folder. The `Home` folder contains the views for the `About`, `Contact`, and `Index` (homepage) webpages. When a user requests one of these three webpages, controller actions in the `Home` controller determine which of the three views is used to build and return a webpage to the user.

Use layouts to provide consistent webpage sections and reduce code repetition. Layouts often contain the header, navigation and menu elements, and the footer. The header and footer usually contain boilerplate markup for many metadata elements and links to script and style assets. Layouts help you avoid this boilerplate markup in your views.

Partial views reduce code duplication by managing reusable parts of views. For example, a partial view is useful for an author biography on a blog website that appears in several views. An author biography is ordinary view content and doesn't require code to execute in order to produce the content for the webpage. Author biography content is available to the view by model binding alone, so using a partial view for this type of content is ideal.

View components are similar to partial views in that they allow you to reduce repetitive code, but they're appropriate for view content that requires code to run on the server in order to render the webpage. View components are useful when the rendered content requires database interaction, such as for a website shopping cart. View components aren't limited to model binding in order to produce webpage output.

Benefits of using views

Views help to establish separation of concerns within an MVC app by separating the user interface markup from other parts of the app. Following SoC design makes your app modular, which provides several benefits:

- The app is easier to maintain because it's better organized. Views are generally grouped by app feature. This makes it easier to find related views when working on a feature.
- The parts of the app are loosely coupled. You can build and update the app's views separately from the business logic and data access components. You can modify the views of the app without necessarily having to update other parts of the app.
- It's easier to test the user interface parts of the app because the views are separate units.
- Due to better organization, it's less likely that you'll accidentally repeat sections of the user interface.

Creating a view

Views that are specific to a controller are created in the `Views/[ControllerName]` folder. Views that are shared among controllers are placed in the `Views/Shared` folder. To create a view, add a new file and give it the same name as its associated controller action with the `.cshtml` file extension. To create a view that corresponds with the `About` action in the `Home` controller, create an `About.cshtml` file in the `Views/Home` folder:

CSHTML

```
@{
    ViewData["Title"] = "About";
}
<h2>@ViewData["Title"].</h2>
<h3>@ViewData["Message"]</h3>
```

```
<p>Use this area to provide additional information.</p>
```

Razor markup starts with the `@` symbol. Run C# statements by placing C# code within Razor code blocks set off by curly braces (`{ ... }`). For example, see the assignment of "About" to `ViewData["Title"]` shown above. You can display values within HTML by simply referencing the value with the `@` symbol. See the contents of the `<h2>` and `<h3>` elements above.

The view content shown above is only part of the entire webpage that's rendered to the user. The rest of the page's layout and other common aspects of the view are specified in other view files. To learn more, see the Layout topic.

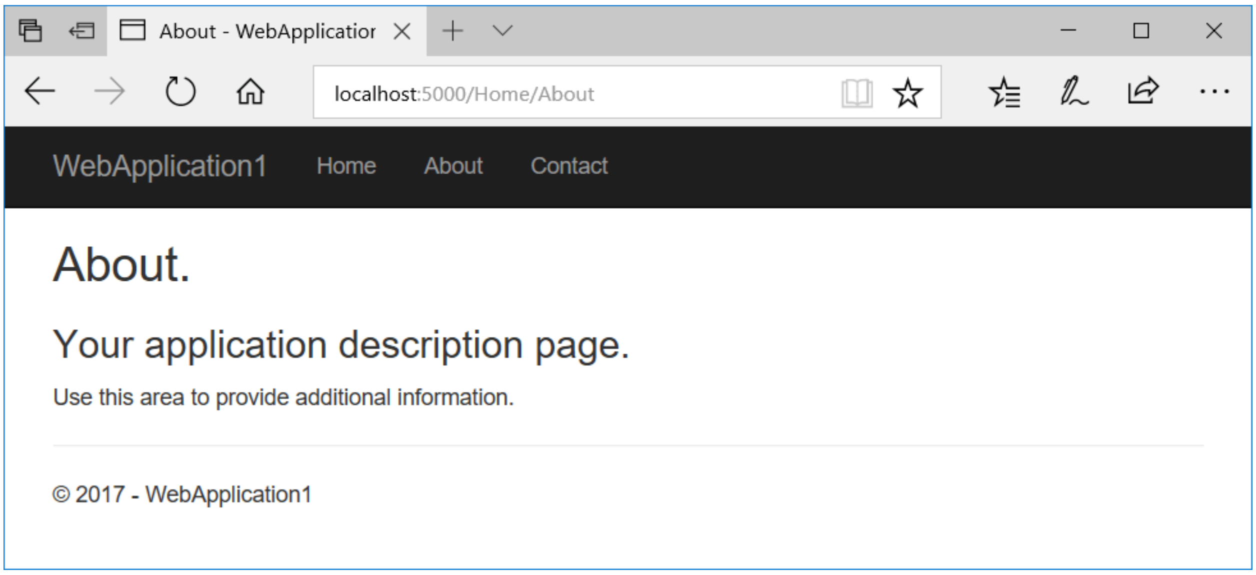
How controllers specify views

Views are typically returned from actions as a ViewResult, which is a type of ActionResult. Your action method can create and return a `ViewResult` directly, but that isn't commonly done. Since most controllers inherit from Controller, you simply use the `View` helper method to return the `ViewResult`:

`HomeController.cs`:

```C#
public IActionResult About()
{
    ViewData["Message"] = "Your application description page.";

    return View();
}
```

When this action returns, the `About.cshtml` view shown in the last section is rendered as the following webpage:



About.

Your application description page.

Use this area to provide additional information.

© 2017 - WebApplication1

The `View` helper method has several overloads. You can optionally specify:

- An explicit view to return:

```C#
return View("Orders");
```

- A model to pass to the view:

```C#
return View(Orders);
```

- Both a view and a model:

```C#
return View("Orders", Orders);
```

View discovery

When an action returns a view, a process called *view discovery* takes place. This process determines which view file is used based on the view name.

The default behavior of the `View` method (`return View();`) is to return a view with the same name as the action method from which it's called. For example, the `About` `ActionResult` method name of the controller is used to search for a view file named `About.cshtml`. First, the runtime looks in the `Views/[ControllerName]` folder for the view. If it doesn't find a matching view there, it searches the `Shared` folder for the view.

It doesn't matter if you implicitly return the `ViewResult` with `return View();` or explicitly pass the view name to the `View` method with `return View("<ViewName>");`. In both cases, view discovery searches for a matching view file in this order:

1. `Views/\[ControllerName]/\[ViewName].cshtml`
2. `Views/Shared/\[ViewName].cshtml`

A view file path can be provided instead of a view name. If using an absolute path starting at the app root (optionally starting with "/" or "~/"), the `.cshtml` extension must be specified:

```C#
return View("Views/Home/About.cshtml");
```

You can also use a relative path to specify views in different directories without the `.cshtml` extension. Inside the `HomeController`, you can return the `Index` view of your `Manage` views with a relative path:

```C#
return View("../Manage/Index");
```

Similarly, you can indicate the current controller-specific directory with the "./" prefix:

```C#
return View("./About");
```

Partial views and view components use similar (but not identical) discovery mechanisms.

You can customize the default convention for how views are located within the app by using a custom IViewLocationExpander.

View discovery relies on finding view files by file name. If the underlying file system is case sensitive, view names are probably case sensitive. For compatibility across operating systems, match case between controller and action names and associated view folders and file names. If you encounter an error that a view file can't be found while working with a case-sensitive file system, confirm that the casing matches between the requested view file and the actual view file name.

Follow the best practice of organizing the file structure for your views to reflect the relationships among controllers, actions, and views for maintainability and clarity.

Pass data to views

Pass data to views using several approaches:

- Strongly typed data: viewmodel
- Weakly typed data
  - `ViewData` (`ViewDataAttribute`)
  - `ViewBag`

Strongly-typed data (viewmodel)

The most robust approach is to specify a model type in the view. This model is commonly referred to as a *viewmodel*. You pass an instance of the viewmodel type to the view from the action.

Using a viewmodel to pass data to a view allows the view to take advantage of *strong* type checking. *Strong typing* (or *strongly typed*) means that every variable and constant has an explicitly defined type (for example, `string`, `int`, or `DateTime`). The validity of types used in a view is checked at compile time.

Visual Studio and Visual Studio Code list strongly typed class members using a feature called IntelliSense. When you want to see the properties of a viewmodel, type the variable name for the viewmodel followed by a period (`.`). This helps you write code faster with fewer errors.

Specify a model using the `@model` directive. Use the model with `@Model`:

CSHTML

```
@model WebApplication1.ViewModels.Address

<h2>Contact</h2>
<address>
    @Model.Street<br>
    @Model.City, @Model.State @Model.PostalCode<br>
    <abbr title="Phone">P:</abbr> 425.555.0100
</address>
```

To provide the model to the view, the controller passes it as a parameter:

C#

```
public IActionResult Contact()
{
    ViewData["Message"] = "Your contact page.";
    var viewModel = new Address()
    {
        Name = "Microsoft",
        Street = "One Microsoft Way",
        City = "Redmond",
        State = "WA",
        PostalCode = "98052-6399"
    };

    return View(viewModel);
}
```

There are no restrictions on the model types that you can provide to a view. We recommend using Plain Old CLR Object (POCO) viewmodels with little or no behavior (methods) defined. Usually, viewmodel classes are either stored in the `Models` folder or a separate `ViewModels` folder at the root of the app. The `Address` viewmodel used in the example above is a POCO viewmodel stored in a file named `Address.cs`:

```C#
namespace WebApplication1.ViewModels
{
    public class Address
    {
        public string Name { get; set; }
        public string Street { get; set; }
        public string City { get; set; }
        public string State { get; set; }
        public string PostalCode { get; set; }
    }
}
```

Nothing prevents you from using the same classes for both your viewmodel types and your business model types. However, using separate models allows your views to vary independently from the business logic and data access parts of your app. Separation of models and viewmodels also offers security benefits when models use model binding and validation for data sent to the app by the user.

Weakly typed data (`ViewData`, `[ViewData]` attribute, and `ViewBag`)

`ViewBag` *isn't available by default for use in Razor Pages* `PageModel` *classes.*

In addition to strongly typed views, views have access to a *weakly typed* (also called *loosely typed*) collection of data. Unlike strong types, *weak types* (or *loose types*) means that you don't explicitly declare the type of data you're using. You can use the collection of weakly typed data for passing small amounts of data in and out of controllers and views.

Expand table

Passing data between a ...	**Example**
Controller and a view	Populating a dropdown list with data.
View and a layout view	Setting the `<title>` element content in the layout view from a view file.
Partial view and a view	A widget that displays data based on the webpage that the user requested.

This collection can be referenced through either the `ViewData` or `ViewBag` properties on controllers and views. The `ViewData` property is a dictionary of weakly typed objects. The `ViewBag` property is a wrapper around `ViewData` that provides dynamic properties for the underlying `ViewData` collection. Note: Key lookups are case-insensitive for both `ViewData` and `ViewBag`.

`ViewData` and `ViewBag` are dynamically resolved at runtime. Since they don't offer compile-time type checking, both are generally more error-prone than using a viewmodel. For that reason, some developers prefer to minimally or never use `ViewData` and `ViewBag`.

`ViewData`

`ViewData` is a ViewDataDictionary object accessed through `string` keys. String data can be stored and used directly without the need for a cast, but you must cast other `ViewData` object values to specific types when you extract them. You can use `ViewData` to pass data from controllers to views and within views, including partial views and layouts.

The following is an example that sets values for a greeting and an address using `ViewData` in an action:

C#

```csharp
public IActionResult SomeAction()
{
    ViewData["Greeting"] = "Hello";
    ViewData["Address"]  = new Address()
    {
        Name = "Steve",
        Street = "123 Main St",
        City = "Hudson",
        State = "OH",
        PostalCode = "44236"
    };

    return View();
}
```

Work with the data in a view:

CSHTML

```
@{
    // Since Address isn't a string, it requires a cast.
    var address = ViewData["Address"] as Address;
}

@ViewData["Greeting"] World!

<address>
    @address.Name<br>
    @address.Street<br>
    @address.City, @address.State @address.PostalCode
</address>
```

`[ViewData]` attribute

Another approach that uses the ViewDataDictionary is ViewDataAttribute. Properties on controllers or Razor Page models marked with the `[ViewData]` attribute have their values stored and loaded from the dictionary.

In the following example, the Home controller contains a `Title` property marked with `[ViewData]`. The `About` method sets the title for the About view:

C#

```
public class HomeController : Controller
{
    [ViewData]
    public string Title { get; set; }

    public IActionResult About()
    {
        Title = "About Us";
        ViewData["Message"] = "Your application description page.";
        return View();
    }
}
```

In the layout, the title is read from the ViewData dictionary:

CSHTML

```cshtml
<!DOCTYPE html>
<html lang="en">
<head>
    <title>@ViewData["Title"] - WebApplication</title>
    ...
```

`ViewBag`

`ViewBag` isn't available by default for use in Razor Pages `PageModel` classes.

`ViewBag` is a `Microsoft.AspNetCore.Mvc.ViewFeatures.Internal.DynamicViewData` object that provides dynamic access to the objects stored in `ViewData`. `ViewBag` can be more convenient to work with, since it doesn't require casting. The following example shows how to use `ViewBag` with the same result as using `ViewData` above:

C#

```csharp
public IActionResult SomeAction()
{
    ViewBag.Greeting = "Hello";
    ViewBag.Address  = new Address()
    {
        Name = "Steve",
        Street = "123 Main St",
        City = "Hudson",
        State = "OH",
        PostalCode = "44236"
    };

    return View();
}
```

CSHTML

```cshtml
@ViewBag.Greeting World!

<address>
    @ViewBag.Address.Name<br>
    @ViewBag.Address.Street<br>
    @ViewBag.Address.City, @ViewBag.Address.State
@ViewBag.Address.PostalCode
</address>
```

Using `ViewData` and `ViewBag` simultaneously

`ViewBag` *isn't available by default for use in Razor Pages* `PageModel` *classes.*

Since `ViewData` and `ViewBag` refer to the same underlying `ViewData` collection, you can use both `ViewData` and `ViewBag` and mix and match between them when reading and writing values.

Set the title using `ViewBag` and the description using `ViewData` at the top of an `About.cshtml` view:

CSHTML

```
@{
    Layout = "/Views/Shared/_Layout.cshtml";
    ViewBag.Title = "About Contoso";
    ViewData["Description"] = "Let us tell you about Contoso's philosophy
and mission.";
}
```

Read the properties but reverse the use of `ViewData` and `ViewBag`. In the `_Layout.cshtml` file, obtain the title using `ViewData` and obtain the description using `ViewBag`:

CSHTML

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>@ViewData["Title"]</title>
    <meta name="description" content="@ViewBag.Description">
    ...
```

Remember that strings don't require a cast for `ViewData`. You can use `@ViewData["Title"]` without casting.

Using both `ViewData` and `ViewBag` at the same time works, as does mixing and matching reading and writing the properties. The following markup is rendered:

HTML

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>About Contoso</title>
    <meta name="description" content="Let us tell you about Contoso's 
philosophy and mission.">
    ...
```

Summary of the differences between `ViewData` and `ViewBag`

`ViewBag` *isn't available by default for use in Razor Pages* `PageModel` *classes.*

- `ViewData`
  - Derives from ViewDataDictionary, so it has dictionary properties that can be useful, such as `ContainsKey`, `Add`, `Remove`, and `Clear`.
  - Keys in the dictionary are strings, so whitespace is allowed. Example: `ViewData["Some Key With Whitespace"]`
  - Any type other than a `string` must be cast in the view to use `ViewData`.
- `ViewBag`
  - Derives from `Microsoft.AspNetCore.Mvc.ViewFeatures.Internal.DynamicViewData`, so it allows the creation of dynamic properties using dot notation (`@ViewBag.SomeKey = <value or object>`), and no casting is required. The syntax of `ViewBag` makes it quicker to add to controllers and views.
  - Simpler to check for null values. Example: `@ViewBag.Person?.Name`

When to use `ViewData` or `ViewBag`

Both `ViewData` and `ViewBag` are equally valid approaches for passing small amounts of data among controllers and views. The choice of which one to use is based on preference. You can mix and match `ViewData` and `ViewBag` objects, however, the code is easier to read and maintain with one approach used consistently. Both approaches are dynamically resolved at runtime and thus prone to causing runtime errors. Some development teams avoid them.

Dynamic views

Views that don't declare a model type using `@model` but that have a model instance passed to them (for example, `return View(Address);`) can reference the instance's properties dynamically:

CSHTML

```cshtml
<address>
    @Model.Street<br>
    @Model.City, @Model.State @Model.PostalCode<br>
    <abbr title="Phone">P:</abbr> 425.555.0100
</address>
```

This feature offers flexibility but doesn't offer compilation protection or IntelliSense. If the property doesn't exist, webpage generation fails at runtime.

More view features

Tag Helpers make it easy to add server-side behavior to existing HTML tags. Using Tag Helpers avoids the need to write custom code or helpers within your views. Tag helpers are applied as attributes to HTML elements and are ignored by editors that can't process them. This allows you to edit and render view markup in a variety of tools.

Generating custom HTML markup can be achieved with many built-in HTML Helpers. More complex user interface logic can be handled by View Components. View components provide the same SoC that controllers and views offer. They can eliminate the need for actions and views that deal with data used by common user interface elements.

Like many other aspects of ASP.NET Core, views support dependency injection, allowing services to be injected into views.

CSS isolation

Isolate CSS styles to individual pages, views, and components to reduce or avoid:

- Dependencies on global styles that can be challenging to maintain.
- Style conflicts in nested content.

To add a *scoped CSS file* for a page or view, place the CSS styles in a companion `.cshtml.css` file matching the name of the `.cshtml` file. In the following example, an `Index.cshtml.css` file supplies CSS styles that are only applied to the `Index.cshtml` page or view.

`Pages/Index.cshtml.css` (Razor Pages) or `Views/Index.cshtml.css` (MVC):

CSS

```css
h1 {
    color: red;
}
```

CSS isolation occurs at build time. The framework rewrites CSS selectors to match markup rendered by the app's pages or views. The rewritten CSS styles are bundled and produced as a static asset, `{APP ASSEMBLY}.styles.css`. The placeholder `{APP ASSEMBLY}` is the assembly name of the project. A link to the bundled CSS styles is placed in the app's layout.

In the `<head>` content of the app's `Pages/Shared/_Layout.cshtml` (Razor Pages) or `Views/Shared/_Layout.cshtml` (MVC), add or confirm the presence of the link to the bundled CSS styles:

HTML

```html
<link rel="stylesheet" href="~/{APP ASSEMBLY}.styles.css" />
```

In the following example, the app's assembly name is `WebApp`:

HTML

```html
<link rel="stylesheet" href="WebApp.styles.css" />
```

The styles defined in a scoped CSS file are only applied to the rendered output of the matching file. In the preceding example, any `h1` CSS declarations defined elsewhere in the app don't conflict with the `Index`'s heading style. CSS style cascading and inheritance rules remain in effect for scoped CSS files. For example, styles applied directly to an `<h1>` element in the `Index.cshtml` file override the scoped CSS file's styles in `Index.cshtml.css`.

Note

In order to guarantee CSS style isolation when bundling occurs, importing CSS in Razor code blocks isn't supported.

CSS isolation only applies to HTML elements. CSS isolation isn't supported for Tag Helpers.

Within the bundled CSS file, each page, view, or Razor component is associated with a scope identifier in the format `b-{STRING}`, where the `{STRING}` placeholder is a ten-character string generated by the framework. The following example provides the style for the preceding `<h1>` element in the `Index` page of a Razor Pages app:

```css
/* /Pages/Index.cshtml.rz.scp.css */
h1[b-3xxtam6d07] {
    color: red;
}
```

In the `Index` page where the CSS style is applied from the bundled file, the scope identifier is appended as an HTML attribute:

```html
<h1 b-3xxtam6d07>
```

The identifier is unique to an app. At build time, a project bundle is created with the convention `{STATIC WEB ASSETS BASE PATH}/Project.lib.scp.css`, where the placeholder `{STATIC WEB ASSETS BASE PATH}` is the static web assets base path.

If other projects are utilized, such as NuGet packages or Razor class libraries, the bundled file:

- References the styles using CSS imports.
- Isn't published as a static web asset of the app that consumes the styles.

CSS preprocessor support

CSS preprocessors are useful for improving CSS development by utilizing features such as variables, nesting, modules, mixins, and inheritance. While CSS isolation doesn't natively support CSS preprocessors such as Sass or Less, integrating CSS preprocessors is seamless as long as preprocessor compilation occurs before the framework rewrites the CSS selectors during the build process. Using Visual Studio for example, configure existing preprocessor compilation as a **Before Build** task in the Visual Studio Task Runner Explorer.

Many third-party NuGet packages, such as AspNetCore.SassCompiler, can compile SASS/SCSS files at the beginning of the build process before CSS isolation occurs, and no additional configuration is required.

CSS isolation configuration

CSS isolation permits configuration for some advanced scenarios, such as when there are dependencies on existing tools or workflows.

Customize scope identifier format

In this section, the `{Pages|Views}` placeholder is either `Pages` for Razor Pages apps or `Views` for MVC apps.

By default, scope identifiers use the format `b-{STRING}`, where the `{STRING}` placeholder is a ten-character string generated by the framework. To customize the scope identifier format, update the project file to a desired pattern:

```xml
<ItemGroup>
  <None Update="{Pages|Views}/Index.cshtml.css" CssScope="custom-scope-identifier" />
</ItemGroup>
```

In the preceding example, the CSS generated for `Index.cshtml.css` changes its scope identifier from `b-{STRING}` to `custom-scope-identifier`.

Use scope identifiers to achieve inheritance with scoped CSS files. In the following project file example, a `BaseView.cshtml.css` file contains common styles across views. A `DerivedView.cshtml.css` file inherits these styles.

```xml
<ItemGroup>
  <None Update="{Pages|Views}/BaseView.cshtml.css" CssScope="custom-scope-identifier" />
  <None Update="{Pages|Views}/DerivedView.cshtml.css" CssScope="custom-scope-identifier" />
</ItemGroup>
```

Use the wildcard (`*`) operator to share scope identifiers across multiple files:

```xml
<ItemGroup>
  <None Update="{Pages|Views}/*.cshtml.css" CssScope="custom-scope-identifier" />
</ItemGroup>
```

Change base path for static web assets

The scoped CSS file is generated at the root of the app. In the project file, use the `StaticWebAssetBasePath` property to change the default path. The following example places the scoped CSS file, and the rest of the app's assets, at the `_content` path:

XML

```xml
<PropertyGroup>
  <StaticWebAssetBasePath>_content/$(PackageId)</StaticWebAssetBasePath>
</PropertyGroup>
```

Disable automatic bundling

To opt out of how framework publishes and loads scoped files at runtime, use the `DisableScopedCssBundling` property. When using this property, other tools or processes are responsible for taking the isolated CSS files from the `obj` directory and publishing and loading them at runtime:

XML

```xml
<PropertyGroup>
  <DisableScopedCssBundling>true</DisableScopedCssBundling>
</PropertyGroup>
```

Razor class library (RCL) support

When a Razor class library (RCL) provides isolated styles, the `<link>` tag's `href` attribute points to `{STATIC WEB ASSET BASE PATH}/{PACKAGE ID}.bundle.scp.css`, where the placeholders are:

- `{STATIC WEB ASSET BASE PATH}`: The static web asset base path.
- `{PACKAGE ID}`: The library's package identifier. The package identifier defaults to the project's assembly name if the package identifier isn't specified in the project file.

In the following example:

- The static web asset base path is `_content/ClassLib`.
- The class library's assembly name is `ClassLib`.

`Pages/Shared/_Layout.cshtml` (Razor Pages) or `Views/Shared/_Layout.cshtml` (MVC):

HTML

```html
<link href="_content/ClassLib/ClassLib.bundle.scp.css" rel="stylesheet">
```

For more information on RCLs, see the following articles:

- Reusable Razor UI in class libraries with ASP.NET Core
- Consume ASP.NET Core Razor components from a Razor class library (RCL)

For information on Blazor CSS isolation, see ASP.NET Core Blazor CSS isolation.

Partial views in ASP.NET Core

Article • 05/17/2023

By Steve Smith, Maher JENDOUBI, Rick Anderson, and Scott Sauber

A partial view is a Razor markup file (`.cshtml`) without a @page directive that renders HTML output *within* another markup file's rendered output.

The term *partial view* is used when developing either an MVC app, where markup files are called *views*, or a Razor Pages app, where markup files are called *pages*. This topic generically refers to MVC views and Razor Pages pages as *markup files*.

View or download sample code (how to download)

When to use partial views

Partial views are an effective way to:

- Break up large markup files into smaller components.

  In a large, complex markup file composed of several logical pieces, there's an advantage to working with each piece isolated into a partial view. The code in the markup file is manageable because the markup only contains the overall page structure and references to partial views.

- Reduce the duplication of common markup content across markup files.

  When the same markup elements are used across markup files, a partial view removes the duplication of markup content into one partial view file. When the markup is changed in the partial view, it updates the rendered output of the markup files that use the partial view.

Partial views shouldn't be used to maintain common layout elements. Common layout elements should be specified in _Layout.cshtml files.

Don't use a partial view where complex rendering logic or code execution is required to render the markup. Instead of a partial view, use a view component.

Declare partial views

A partial view is a `.cshtml` markup file without an @page directive maintained within the *Views* folder (MVC) or *Pages* folder (Razor Pages).

In ASP.NET Core MVC, a controller's ViewResult is capable of returning either a view or a partial view. In Razor Pages, a PageModel can return a partial view represented as a PartialViewResult object. Referencing and rendering partial views is described in the Reference a partial view section.

Unlike MVC view or page rendering, a partial view doesn't run `_ViewStart.cshtml`. For more information on `_ViewStart.cshtml`, see Layout in ASP.NET Core.

Partial view file names often begin with an underscore (`_`). This naming convention isn't required, but it helps to visually differentiate partial views from views and pages.

Reference a partial view

Use a partial view in a Razor Pages PageModel

In ASP.NET Core 2.0 or 2.1, the following handler method renders the *_AuthorPartialRP.cshtml* partial view to the response:

```C#
public IActionResult OnGetPartial() =>
    new PartialViewResult
    {
        ViewName = "_AuthorPartialRP",
        ViewData = ViewData,
    };
```

In ASP.NET Core 2.2 or later, a handler method can alternatively call the Partial method to produce a `PartialViewResult` object:

```C#
public IActionResult OnGetPartial() =>
    Partial("_AuthorPartialRP");
```

Use a partial view in a markup file

Within a markup file, there are several ways to reference a partial view. We recommend that apps use one of the following asynchronous rendering approaches:

- Partial Tag Helper
- Asynchronous HTML Helper

Partial Tag Helper

The Partial Tag Helper requires ASP.NET Core 2.1 or later.

The Partial Tag Helper renders content asynchronously and uses an HTML-like syntax:

CSHTML

```
<partial name="_PartialName" />
```

When a file extension is present, the Tag Helper references a partial view that must be in the same folder as the markup file calling the partial view:

CSHTML

```
<partial name="_PartialName.cshtml" />
```

The following example references a partial view from the app root. Paths that start with a tilde-slash (`~/`) or a slash (`/`) refer to the app root:

Razor Pages

CSHTML

```
<partial name="~/Pages/Folder/_PartialName.cshtml" />
<partial name="/Pages/Folder/_PartialName.cshtml" />
```

MVC

CSHTML

```
<partial name="~/Views/Folder/_PartialName.cshtml" />
<partial name="/Views/Folder/_PartialName.cshtml" />
```

The following example references a partial view with a relative path:

CSHTML

```
<partial name="../Account/_PartialName.cshtml" />
```

For more information, see Partial Tag Helper in ASP.NET Core.

Asynchronous HTML Helper

When using an HTML Helper, the best practice is to use PartialAsync. `PartialAsync` returns an IHtmlContent type wrapped in a Task<TResult>. The method is referenced by prefixing the awaited call with an `@` character:

CSHTML

```cshtml
@await Html.PartialAsync("_PartialName")
```

When the file extension is present, the HTML Helper references a partial view that must be in the same folder as the markup file calling the partial view:

CSHTML

```cshtml
@await Html.PartialAsync("_PartialName.cshtml")
```

The following example references a partial view from the app root. Paths that start with a tilde-slash (`~/`) or a slash (`/`) refer to the app root:

Razor Pages

CSHTML

```cshtml
@await Html.PartialAsync("~/Pages/Folder/_PartialName.cshtml")
@await Html.PartialAsync("/Pages/Folder/_PartialName.cshtml")
```

MVC

CSHTML

```cshtml
@await Html.PartialAsync("~/Views/Folder/_PartialName.cshtml")
@await Html.PartialAsync("/Views/Folder/_PartialName.cshtml")
```

The following example references a partial view with a relative path:

CSHTML

```cshtml
@await Html.PartialAsync("../Account/_LoginPartial.cshtml")
```

Alternatively, you can render a partial view with RenderPartialAsync. This method doesn't return an IHtmlContent. It streams the rendered output directly to the response. Because the method doesn't return a result, it must be called within a Razor code block:

CSHTML

```
@{
    await Html.RenderPartialAsync("_AuthorPartial");
}
```

Since `RenderPartialAsync` streams rendered content, it provides better performance in some scenarios. In performance-critical situations, benchmark the page using both approaches and use the approach that generates a faster response.

Synchronous HTML Helper

Partial and RenderPartial are the synchronous equivalents of `PartialAsync` and `RenderPartialAsync`, respectively. The synchronous equivalents aren't recommended because there are scenarios in which they deadlock. The synchronous methods are targeted for removal in a future release.

> **Important**
>
> If you need to execute code, use a **view component** instead of a partial view.

Calling `Partial` or `RenderPartial` results in a Visual Studio analyzer warning. For example, the presence of `Partial` yields the following warning message:

> Use of IHtmlHelper.Partial may result in application deadlocks. Consider using <partial> Tag Helper or IHtmlHelper.PartialAsync.

Replace calls to `@Html.Partial` with `@await Html.PartialAsync` or the Partial Tag Helper. For more information on Partial Tag Helper migration, see Migrate from an HTML Helper.

Partial view discovery

When a partial view is referenced by name without a file extension, the following locations are searched in the stated order:

Razor Pages

1. Currently executing page's folder
2. Directory graph above the page's folder
3. `/Shared`
4. `/Pages/Shared`

5. `/Views/Shared`

MVC

1. `/Areas/<Area-Name>/Views/<Controller-Name>`
2. `/Areas/<Area-Name>/Views/Shared`
3. `/Views/Shared`
4. `/Pages/Shared`

The following conventions apply to partial view discovery:

- Different partial views with the same file name are allowed when the partial views are in different folders.
- When referencing a partial view by name without a file extension and the partial view is present in both the caller's folder and the *Shared* folder, the partial view in the caller's folder supplies the partial view. If the partial view isn't present in the caller's folder, the partial view is provided from the *Shared* folder. Partial views in the *Shared* folder are called *shared partial views* or *default partial views*.
- Partial views can be *chained*—a partial view can call another partial view if a circular reference isn't formed by the calls. Relative paths are always relative to the current file, not to the root or parent of the file.

> **Note**
>
> A Razor `section` defined in a partial view is invisible to parent markup files. The `section` is only visible to the partial view in which it's defined.

Access data from partial views

When a partial view is instantiated, it receives a *copy* of the parent's `ViewData` dictionary. Updates made to the data within the partial view aren't persisted to the parent view. `ViewData` changes in a partial view are lost when the partial view returns.

The following example demonstrates how to pass an instance of ViewDataDictionary to a partial view:

CSHTML

```cshtml
@await Html.PartialAsync("_PartialName", customViewData)
```

You can pass a model into a partial view. The model can be a custom object. You can pass a model with `PartialAsync` (renders a block of content to the caller) or `RenderPartialAsync` (streams the content to the output):

CSHTML

```
@await Html.PartialAsync("_PartialName", model)
```

Razor Pages

The following markup in the sample app is from the `Pages/ArticlesRP/ReadRP.cshtml` page. The page contains two partial views. The second partial view passes in a model and `ViewData` to the partial view. The `ViewDataDictionary` constructor overload is used to pass a new `ViewData` dictionary while retaining the existing `ViewData` dictionary.

CSHTML

```
@model ReadRPModel

<h2>@Model.Article.Title</h2>
@* Pass the author's name to Pages\Shared\_AuthorPartialRP.cshtml *@
@await Html.PartialAsync("../Shared/_AuthorPartialRP",
Model.Article.AuthorName)
@Model.Article.PublicationDate

@* Loop over the Sections and pass in a section and additional ViewData to
   the strongly typed Pages\ArticlesRP\_ArticleSectionRP.cshtml partial
view. *@
@{
    var index = 0;

    foreach (var section in Model.Article.Sections)
    {
        await Html.PartialAsync("_ArticleSectionRP",
                                section,
                                new ViewDataDictionary(ViewData)
                                {
                                    { "index", index }
                                });

        index++;
    }
}
```

`Pages/Shared/_AuthorPartialRP.cshtml` is the first partial view referenced by the `ReadRP.cshtml` markup file:

CSHTML

```
@model string
<div>
    <h3>@Model</h3>
    This partial view from /Pages/Shared/_AuthorPartialRP.cshtml.
</div>
```

`Pages/ArticlesRP/_ArticleSectionRP.cshtml` is the second partial view referenced by the `ReadRP.cshtml` markup file:

CSHTML

```
@using PartialViewsSample.ViewModels
@model ArticleSection

<h3>@Model.Title Index: @ViewData["index"]</h3>
<div>
    @Model.Content
</div>
```

MVC

The following markup in the sample app shows the `Views/Articles/Read.cshtml` view. The view contains two partial views. The second partial view passes in a model and `ViewData` to the partial view. The `ViewDataDictionary` constructor overload is used to pass a new `ViewData` dictionary while retaining the existing `ViewData` dictionary.

CSHTML

```
@model PartialViewsSample.ViewModels.Article

<h2>@Model.Title</h2>
@* Pass the author's name to Views\Shared\_AuthorPartial.cshtml *@
@await Html.PartialAsync("_AuthorPartial", Model.AuthorName)
@Model.PublicationDate

@* Loop over the Sections and pass in a section and additional ViewData to
   the strongly typed Views\Articles\_ArticleSection.cshtml partial view. *@
@{
    var index = 0;

    foreach (var section in Model.Sections)
    {
        @(await Html.PartialAsync("_ArticleSection",
                                section,
                                new ViewDataDictionary(ViewData)
                                {
                                    { "index", index }
                                }))
```

```
            index++;
        }
    }
```

`Views/Shared/_AuthorPartial.cshtml` is the first partial view referenced by the `Read.cshtml` markup file:

CSHTML

```
@model string
<div>
    <h3>@Model</h3>
    This partial view from /Views/Shared/_AuthorPartial.cshtml.
</div>
```

`Views/Articles/_ArticleSection.cshtml` is the second partial view referenced by the `Read.cshtml` markup file:

CSHTML

```
@using PartialViewsSample.ViewModels
@model ArticleSection

<h3>@Model.Title Index: @ViewData["index"]</h3>
<div>
    @Model.Content
</div>
```

At runtime, the partials are rendered into the parent markup file's rendered output, which itself is rendered within the shared `_Layout.cshtml`. The first partial view renders the article author's name and publication date:

> Abraham Lincoln
>
> This partial view from <shared partial view file path>. 11/19/1863 12:00:00 AM

The second partial view renders the article's sections:

> Section One Index: 0
>
> Four score and seven years ago ...
>
> Section Two Index: 1
>
> Now we are engaged in a great civil war, testing ...

> Section Three Index: 2
>
> But, in a larger sense, we can not dedicate ...

Additional resources

- Razor syntax reference for ASP.NET Core
- Tag Helpers in ASP.NET Core
- Partial Tag Helper in ASP.NET Core
- View components in ASP.NET Core
- Areas in ASP.NET Core

Handle requests with controllers in ASP.NET Core MVC

Article • 06/17/2024

By Steve Smith and Scott Addie

Controllers, actions, and action results are a fundamental part of how developers build apps using ASP.NET Core MVC.

What is a Controller?

A controller is used to define and group a set of actions. An action (or *action method*) is a method on a controller which handles requests. Controllers logically group similar actions together. This aggregation of actions allows common sets of rules, such as routing, caching, and authorization, to be applied collectively. Requests are mapped to actions through routing. Controllers are activated and disposed on a per request basis.

By convention, controller classes:

- Reside in the project's root-level *Controllers* folder.
- Inherit from `Microsoft.AspNetCore.Mvc.Controller`.

A controller is an instantiable class, usually public, in which at least one of the following conditions is true:

- The class name is suffixed with `Controller`.
- The class inherits from a class whose name is suffixed with `Controller`.
- The `[Controller]` attribute is applied to the class.

A controller class must not have an associated `[NonController]` attribute.

Controllers should follow the Explicit Dependencies Principle. There are a couple of approaches to implementing this principle. If multiple controller actions require the same service, consider using constructor injection to request those dependencies. If the service is needed by only a single action method, consider using Action Injection to request the dependency.

Within the **M**odel-**V**iew-**C**ontroller pattern, a controller is responsible for the initial processing of the request and instantiation of the model. Generally, business decisions should be performed within the model.

The controller takes the result of the model's processing (if any) and returns either the proper view and its associated view data or the result of the API call. Learn more at Overview of ASP.NET Core MVC and Get started with ASP.NET Core MVC and Visual Studio.

The controller is a *UI-level* abstraction. Its responsibilities are to ensure request data is valid and to choose which view (or result for an API) should be returned. In well-factored apps, it doesn't directly include data access or business logic. Instead, the controller delegates to services handling these responsibilities.

Defining Actions

Public methods on a controller, except those with the `[NonAction]` attribute, are actions. Parameters on actions are bound to request data and are validated using model binding. Model validation occurs for everything that's model-bound. The `ModelState.IsValid` property value indicates whether model binding and validation succeeded.

Action methods should contain logic for mapping a request to a business concern. Business concerns should typically be represented as services that the controller accesses through dependency injection. Actions then map the result of the business action to an application state.

Actions can return anything, but frequently return an instance of `IActionResult` (or `Task<IActionResult>` for async methods) that produces a response. The action method is responsible for choosing *what kind of response*. The action result *does the responding*.

Controller Helper Methods

Controllers usually inherit from Controller, although this isn't required. Deriving from `Controller` provides access to three categories of helper methods:

1. Methods resulting in an empty response body

No `Content-Type` HTTP response header is included, since the response body lacks content to describe.

There are two result types within this category: Redirect and HTTP Status Code.

- **HTTP Status Code**

This type returns an HTTP status code. A couple of helper methods of this type are `BadRequest`, `NotFound`, and `Ok`. For example, `return BadRequest();` produces a 400 status code when executed. When methods such as `BadRequest`, `NotFound`, and `Ok` are overloaded, they no longer qualify as HTTP Status Code responders, since content negotiation is taking place.

- **Redirect**

  This type returns a redirect to an action or destination (using `Redirect`, `LocalRedirect`, `RedirectToAction`, or `RedirectToRoute`). For example, `return RedirectToAction("Complete", new {id = 123});` redirects to `Complete`, passing an anonymous object.

  The Redirect result type differs from the HTTP Status Code type primarily in the addition of a `Location` HTTP response header.

2. Methods resulting in a non-empty response body with a predefined content type

Most helper methods in this category include a `ContentType` property, allowing you to set the `Content-Type` response header to describe the response body.

There are two result types within this category: View and Formatted Response.

- **View**

  This type returns a view which uses a model to render HTML. For example, `return View(customer);` passes a model to the view for data-binding.

- **Formatted Response**

  This type returns JSON or a similar data exchange format to represent an object in a specific manner. For example, `return Json(customer);` serializes the provided object into JSON format.

  Other common methods of this type include `File` and `PhysicalFile`. For example, `return PhysicalFile(customerFilePath, "text/xml");` returns PhysicalFileResult.

3. Methods resulting in a non-empty response body formatted in a content type negotiated with the client

This category is better known as **Content Negotiation**. Content negotiation applies whenever an action returns an ObjectResult type or something other than an

IActionResult implementation. An action that returns a non-`IActionResult` implementation (for example, `object`) also returns a Formatted Response.

Some helper methods of this type include `BadRequest`, `CreatedAtRoute`, and `Ok`. Examples of these methods include `return BadRequest(modelState);`, `return CreatedAtRoute("routename", values, newobject);`, and `return Ok(value);`, respectively. Note that `BadRequest` and `Ok` perform content negotiation only when passed a value; without being passed a value, they instead serve as HTTP Status Code result types. The `CreatedAtRoute` method, on the other hand, always performs content negotiation since its overloads all require that a value be passed.

Cross-Cutting Concerns

Applications typically share parts of their workflow. Examples include an app that requires authentication to access the shopping cart, or an app that caches data on some pages. To perform logic before or after an action method, use a *filter*. Using Filters on cross-cutting concerns can reduce duplication.

Most filter attributes, such as `[Authorize]`, can be applied at the controller or action level depending upon the desired level of granularity.

Error handling and response caching are often cross-cutting concerns:

- Handle errors
- Response Caching

Many cross-cutting concerns can be handled using filters or custom middleware.

Routing to controller actions in ASP.NET Core

Article • 06/17/2024

By Ryan Nowak, Kirk Larkin, and Rick Anderson

Important

This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

For the current release, see the .NET 9 version of this article.

ASP.NET Core controllers use the Routing middleware to match the URLs of incoming requests and map them to actions. Route templates:

- Are defined at startup in `Program.cs` or in attributes.
- Describe how URL paths are matched to actions.
- Are used to generate URLs for links. The generated links are typically returned in responses.

Actions are either conventionally-routed or attribute-routed. Placing a route on the controller or action makes it attribute-routed. See Mixed routing for more information.

This document:

- Explains the interactions between MVC and routing:
  - How typical MVC apps make use of routing features.
  - Covers both:
    - Conventional routing typically used with controllers and views.
    - *Attribute routing* used with REST APIs. If you're primarily interested in routing for REST APIs, jump to the Attribute routing for REST APIs section.
  - See Routing for advanced routing details.
- Refers to the default routing system called endpoint routing. It's possible to use controllers with the previous version of routing for compatibility purposes. See the 2.2-3.0 migration guide for instructions.

Set up conventional route

The ASP.NET Core MVC template generates conventional routing code similar to the following:

```C#
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllersWithViews();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Home/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");

app.Run();
```

MapControllerRoute is used to create a single route. The single route is named `default` route. Most apps with controllers and views use a route template similar to the `default` route. REST APIs should use attribute routing.

```C#
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
```

The route template `"{controller=Home}/{action=Index}/{id?}"`:

- Matches a URL path like `/Products/Details/5`
- Extracts the route values `{ controller = Products, action = Details, id = 5 }` by tokenizing the path. The extraction of route values results in a match if the app has a controller named `ProductsController` and a `Details` action:

C#

```
public class ProductsController : Controller
{
    public IActionResult Details(int id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }
}
```

MyDisplayRouteInfo is provided by the Rick.Docs.Samples.RouteInfo NuGet package and displays route information.

- `/Products/Details/5` model binds the value of `id = 5` to set the `id` parameter to `5`. See Model Binding for more details.
- `{controller=Home}` defines `Home` as the default `controller`.
- `{action=Index}` defines `Index` as the default `action`.
- The `?` character in `{id?}` defines `id` as optional.
  - Default and optional route parameters don't need to be present in the URL path for a match. See Route Template Reference for a detailed description of route template syntax.
- Matches the URL path `/`.
- Produces the route values `{ controller = Home, action = Index }`.

The values for `controller` and `action` make use of the default values. `id` doesn't produce a value since there's no corresponding segment in the URL path. `/` only matches if there exists a `HomeController` and `Index` action:

C#

```
public class HomeController : Controller
{
    public IActionResult Index() { ... }
}
```

Using the preceding controller definition and route template, the `HomeController.Index` action is run for the following URL paths:

- `/Home/Index/17`
- `/Home/Index`

- `/Home`
- `/`

The URL path `/` uses the route template default `Home` controllers and `Index` action. The URL path `/Home` uses the route template default `Index` action.

The convenience method MapDefaultControllerRoute:

```C#
app.MapDefaultControllerRoute();
```

Replaces:

```C#
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
```

ⓘ **Important**

Routing is configured using the UseRouting and UseEndpoints middleware. To use controllers:

- Call MapControllers to map attribute routed controllers.
- Call MapControllerRoute or MapAreaControllerRoute, to map both conventionally routed controllers and attribute routed controllers.

Apps typically don't need to call `UseRouting` or `UseEndpoints`. WebApplicationBuilder configures a middleware pipeline that wraps middleware added in `Program.cs` with `UseRouting` and `UseEndpoints`. For more information, see Routing in ASP.NET Core.

Conventional routing

Conventional routing is used with controllers and views. The `default` route:

```C#
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
```

The preceding is an example of a *conventional route*. It's called *conventional routing* because it establishes a *convention* for URL paths:

- The first path segment, `{controller=Home}`, maps to the controller name.
- The second segment, `{action=Index}`, maps to the action name.
- The third segment, `{id?}` is used for an optional `id`. The `?` in `{id?}` makes it optional. `id` is used to map to a model entity.

Using this `default` route, the URL path:

- `/Products/List` maps to the `ProductsController.List` action.
- `/Blog/Article/17` maps to `BlogController.Article` and typically model binds the `id` parameter to 17.

This mapping:

- Is based on the controller and action names **only**.
- Isn't based on namespaces, source file locations, or method parameters.

Using conventional routing with the default route allows creating the app without having to come up with a new URL pattern for each action. For an app with CRUD style actions, having consistency for the URLs across controllers:

- Helps simplify the code.
- Makes the UI more predictable.

⚠ **Warning**

The `id` in the preceding code is defined as optional by the route template. Actions can execute without the optional ID provided as part of the URL. Generally, when `id` is omitted from the URL:

- `id` is set to `0` by model binding.
- No entity is found in the database matching `id == 0`.

Attribute routing provides fine-grained control to make the ID required for some actions and not for others. By convention, the documentation includes optional parameters like `id` when they're likely to appear in correct usage.

Most apps should choose a basic and descriptive routing scheme so that URLs are readable and meaningful. The default conventional route `{controller=Home}/{action=Index}/{id?}`:

- Supports a basic and descriptive routing scheme.
- Is a useful starting point for UI-based apps.
- Is the only route template needed for many web UI apps. For larger web UI apps, another route using Areas is frequently all that's needed.

MapControllerRoute and MapAreaRoute :

- Automatically assign an **order** value to their endpoints based on the order they are invoked.

Endpoint routing in ASP.NET Core:

- Doesn't have a concept of routes.
- Doesn't provide ordering guarantees for the execution of extensibility, all endpoints are processed at once.

Enable Logging to see how the built-in routing implementations, such as Route, match requests.

Attribute routing is explained later in this document.

Multiple conventional routes

Multiple conventional routes can be configured by adding more calls to MapControllerRoute and MapAreaControllerRoute. Doing so allows defining multiple conventions, or to adding conventional routes that are dedicated to a specific action, such as:

```C#
app.MapControllerRoute(name: "blog",
                pattern: "blog/{*article}",
                defaults: new { controller = "Blog", action = "Article" });
app.MapControllerRoute(name: "default",
               pattern: "{controller=Home}/{action=Index}/{id?}");
```

The `blog` route in the preceding code is a **dedicated conventional route**. It's called a dedicated conventional route because:

- It uses conventional routing.
- It's dedicated to a specific action.

Because `controller` and `action` don't appear in the route template `"blog/{*article}"` as parameters:

- They can only have the default values `{ controller = "Blog", action = "Article" }`.
- This route always maps to the action `BlogController.Article`.

`/Blog`, `/Blog/Article`, and `/Blog/{any-string}` are the only URL paths that match the blog route.

The preceding example:

- `blog` route has a higher priority for matches than the `default` route because it is added first.
- Is an example of Slug style routing where it's typical to have an article name as part of the URL.

⚠ **Warning**

In ASP.NET Core, routing doesn't:

- Define a concept called a *route*. `UseRouting` adds route matching to the middleware pipeline. The `UseRouting` middleware looks at the set of endpoints defined in the app, and selects the best endpoint match based on the request.
- Provide guarantees about the execution order of extensibility like IRouteConstraint or IActionConstraint.

See Routing for reference material on routing.

Conventional routing order

Conventional routing only matches a combination of action and controller that are defined by the app. This is intended to simplify cases where conventional routes overlap. Adding routes using MapControllerRoute, MapDefaultControllerRoute, and MapAreaControllerRoute automatically assign an order value to their endpoints based on the order they are invoked. Matches from a route that appears earlier have a higher priority. Conventional routing is order-dependent. In general, routes with areas should be placed earlier as they're more specific than routes without an area. Dedicated conventional routes with catch-all route parameters like `{*article}` can make a route too greedy, meaning that it matches URLs that you intended to be matched by other routes. Put the greedy routes later in the route table to prevent greedy matches.

Resolving ambiguous actions

When two endpoints match through routing, routing must do one of the following:

- Choose the best candidate.
- Throw an exception.

For example:

```C#
public class Products33Controller : Controller
{
    public IActionResult Edit(int id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }

    [HttpPost]
    public IActionResult Edit(int id, Product product)
    {
        return ControllerContext.MyDisplayRouteInfo(id, product.name);
    }
}
```

The preceding controller defines two actions that match:

- The URL path `/Products33/Edit/17`
- Route data `{ controller = Products33, action = Edit, id = 17 }`.

This is a typical pattern for MVC controllers:

- `Edit(int)` displays a form to edit a product.
- `Edit(int, Product)` processes the posted form.

To resolve the correct route:

- `Edit(int, Product)` is selected when the request is an HTTP `POST`.
- `Edit(int)` is selected when the HTTP verb is anything else. `Edit(int)` is generally called via `GET`.

The HttpPostAttribute, `[HttpPost]`, is provided to routing so that it can choose based on the HTTP method of the request. The `HttpPostAttribute` makes `Edit(int, Product)` a better match than `Edit(int)`.

It's important to understand the role of attributes like `HttpPostAttribute`. Similar attributes are defined for other HTTP verbs. In conventional routing, it's common for actions to use the same action name when they're part of a show form, submit form workflow. For example, see Examine the two Edit action methods.

If routing can't choose a best candidate, an AmbiguousMatchException is thrown, listing the multiple matched endpoints.

Conventional route names

The strings `"blog"` and `"default"` in the following examples are conventional route names:

```C#
app.MapControllerRoute(name: "blog",
                pattern: "blog/{*article}",
                defaults: new { controller = "Blog", action = "Article" });
app.MapControllerRoute(name: "default",
               pattern: "{controller=Home}/{action=Index}/{id?}");
```

The route names give the route a logical name. The named route can be used for URL generation. Using a named route simplifies URL creation when the ordering of routes could make URL generation complicated. Route names must be unique application wide.

Route names:

- Have no impact on URL matching or handling of requests.
- Are used only for URL generation.

The route name concept is represented in routing as IEndpointNameMetadata. The terms **route name** and **endpoint name**:

- Are interchangeable.
- Which one is used in documentation and code depends on the API being described.

Attribute routing for REST APIs

REST APIs should use attribute routing to model the app's functionality as a set of resources where operations are represented by HTTP verbs.

Attribute routing uses a set of attributes to map actions directly to route templates. The following code is typical for a REST API and is used in the next sample:

```C#
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers();

var app = builder.Build();

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.Run();
```

In the preceding code, MapControllers is called to map attribute routed controllers.

In the following example:

- `HomeController` matches a set of URLs similar to what the default conventional route `{controller=Home}/{action=Index}/{id?}` matches.

```C#
public class HomeController : Controller
{
    [Route("")]
    [Route("Home")]
    [Route("Home/Index")]
    [Route("Home/Index/{id?}")]
    public IActionResult Index(int? id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }

    [Route("Home/About")]
    [Route("Home/About/{id?}")]
    public IActionResult About(int? id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }
}
```

The `HomeController.Index` action is run for any of the URL paths `/`, `/Home`, `/Home/Index`, or `/Home/Index/3`.

This example highlights a key programming difference between attribute routing and conventional routing. Attribute routing requires more input to specify a route. The conventional default route handles routes more succinctly. However, attribute routing allows and requires precise control of which route templates apply to each action.

With attribute routing, the controller and action names play no part in which action is matched, unless token replacement is used. The following example matches the same URLs as the previous example:

C#

```
public class MyDemoController : Controller
{
    [Route("")]
    [Route("Home")]
    [Route("Home/Index")]
    [Route("Home/Index/{id?}")]
    public IActionResult MyIndex(int? id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }

    [Route("Home/About")]
    [Route("Home/About/{id?}")]
    public IActionResult MyAbout(int? id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }
}
```

The following code uses token replacement for `action` and `controller`:

C#

```
public class HomeController : Controller
{
    [Route("")]
    [Route("Home")]
    [Route("[controller]/[action]")]
    public IActionResult Index()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }

    [Route("[controller]/[action]")]
    public IActionResult About()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }
}
```

The following code applies `[Route("[controller]/[action]")]` to the controller:

C#

```csharp
[Route("[controller]/[action]")]
public class HomeController : Controller
{
    [Route("~/")]
    [Route("/Home")]
    [Route("~/Home/Index")]
    public IActionResult Index()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }

    public IActionResult About()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }
}
```

In the preceding code, the `Index` method templates must prepend `/` or `~/` to the route templates. Route templates applied to an action that begin with `/` or `~/` don't get combined with route templates applied to the controller.

See Route template precedence for information on route template selection.

Reserved routing names

The following keywords are reserved route parameter names when using Controllers or Razor Pages:

- `action`
- `area`
- `controller`
- `handler`
- `page`

Using `page` as a route parameter with attribute routing is a common error. Doing that results in inconsistent and confusing behavior with URL generation.

C#

```
public class MyDemo2Controller : Controller
{
    [Route("/articles/{page}")]
    public IActionResult ListArticles(int page)
    {
        return ControllerContext.MyDisplayRouteInfo(page);
    }
}
```

The special parameter names are used by the URL generation to determine if a URL generation operation refers to a Razor Page or to a Controller.

The following keywords are reserved in the context of a Razor view or a Razor Page:

- `page`
- `using`
- `namespace`
- `inject`
- `section`
- `inherits`
- `model`
- `addTagHelper`
- `removeTagHelper`

These keywords shouldn't be used for link generations, model bound parameters, or top level properties.

HTTP verb templates

ASP.NET Core has the following HTTP verb templates:

- [HttpGet]
- [HttpPost]
- [HttpPut]
- [HttpDelete]
- [HttpHead]
- [HttpPatch]

Route templates

ASP.NET Core has the following route templates:

- All the HTTP verb templates are route templates.
- [Route]

Attribute routing with Http verb attributes

Consider the following controller:

C#

```
[Route("api/[controller]")]
[ApiController]
public class Test2Controller : ControllerBase
{
    [HttpGet]   // GET /api/test2
    public IActionResult ListProducts()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }

    [HttpGet("{id}")]   // GET /api/test2/xyz
    public IActionResult GetProduct(string id)
    {
       return ControllerContext.MyDisplayRouteInfo(id);
    }

    [HttpGet("int/{id:int}")] // GET /api/test2/int/3
    public IActionResult GetIntProduct(int id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }

    [HttpGet("int2/{id}")]  // GET /api/test2/int2/3
    public IActionResult GetInt2Product(int id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }
}
```

In the preceding code:

- Each action contains the `[HttpGet]` attribute, which constrains matching to HTTP GET requests only.
- The `GetProduct` action includes the `"{id}"` template, therefore `id` is appended to the `"api/[controller]"` template on the controller. The methods template is `"api/[controller]/{id}"`. Therefore this action only matches GET requests for the form `/api/test2/xyz`,`/api/test2/123`,`/api/test2/{any string}`, etc.

  C#

```
[HttpGet("{id}")]   // GET /api/test2/xyz
public IActionResult GetProduct(string id)
{
   return ControllerContext.MyDisplayRouteInfo(id);
}
```

- The `GetIntProduct` action contains the `"int/{id:int}"` template. The `:int` portion of the template constrains the `id` route values to strings that can be converted to an integer. A GET request to `/api/test2/int/abc`:
  - Doesn't match this action.
  - Returns a 404 Not Found error.

  C#

  ```
  [HttpGet("int/{id:int}")] // GET /api/test2/int/3
  public IActionResult GetIntProduct(int id)
  {
      return ControllerContext.MyDisplayRouteInfo(id);
  }
  ```

- The `GetInt2Product` action contains `{id}` in the template, but doesn't constrain `id` to values that can be converted to an integer. A GET request to `/api/test2/int2/abc`:
  - Matches this route.
  - Model binding fails to convert `abc` to an integer. The `id` parameter of the method is integer.
  - Returns a 400 Bad Request because model binding failed to convert `abc` to an integer.

  C#

  ```
  [HttpGet("int2/{id}")]  // GET /api/test2/int2/3
  public IActionResult GetInt2Product(int id)
  {
      return ControllerContext.MyDisplayRouteInfo(id);
  }
  ```

Attribute routing can use HttpMethodAttribute attributes such as HttpPostAttribute, HttpPutAttribute, and HttpDeleteAttribute. All of the HTTP verb attributes accept a route template. The following example shows two actions that match the same route template:

C#

```
[ApiController]
public class MyProductsController : ControllerBase
{
    [HttpGet("/products3")]
    public IActionResult ListProducts()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }

    [HttpPost("/products3")]
    public IActionResult CreateProduct(MyProduct myProduct)
    {
        return ControllerContext.MyDisplayRouteInfo(myProduct.Name);
    }
}
```

Using the URL path `/products3`:

- The `MyProductsController.ListProducts` action runs when the HTTP verb is `GET`.
- The `MyProductsController.CreateProduct` action runs when the HTTP verb is `POST`.

When building a REST API, it's rare that you'll need to use `[Route(...)]` on an action method because the action accepts all HTTP methods. It's better to use the more specific HTTP verb attribute to be precise about what your API supports. Clients of REST APIs are expected to know what paths and HTTP verbs map to specific logical operations.

REST APIs should use attribute routing to model the app's functionality as a set of resources where operations are represented by HTTP verbs. This means that many operations, for example, GET and POST on the same logical resource use the same URL. Attribute routing provides a level of control that's needed to carefully design an API's public endpoint layout.

Since an attribute route applies to a specific action, it's easy to make parameters required as part of the route template definition. In the following example, `id` is required as part of the URL path:

C#

```
[ApiController]
public class Products2ApiController : ControllerBase
{
    [HttpGet("/products2/{id}", Name = "Products_List")]
    public IActionResult GetProduct(int id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }
}
```

The `Products2ApiController.GetProduct(int)` action:

- Is run with URL path like `/products2/3`
- Isn't run with the URL path `/products2`.

The [Consumes] attribute allows an action to limit the supported request content types. For more information, see Define supported request content types with the Consumes attribute.

See Routing for a full description of route templates and related options.

For more information on `[ApiController]`, see ApiController attribute.

Route name

The following code defines a route name of `Products_List`:

```C#
[ApiController]
public class Products2ApiController : ControllerBase
{
    [HttpGet("/products2/{id}", Name = "Products_List")]
    public IActionResult GetProduct(int id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }
}
```

Route names can be used to generate a URL based on a specific route. Route names:

- Have no impact on the URL matching behavior of routing.
- Are only used for URL generation.

Route names must be unique application-wide.

Contrast the preceding code with the conventional default route, which defines the `id` parameter as optional (`{id?}`). The ability to precisely specify APIs has advantages, such as allowing `/products` and `/products/5` to be dispatched to different actions.

Combining attribute routes

To make attribute routing less repetitive, route attributes on the controller are combined with route attributes on the individual actions. Any route templates defined on the controller are prepended to route templates on the actions. Placing a route attribute on the controller makes **all** actions in the controller use attribute routing.

```C#
[ApiController]
[Route("products")]
public class ProductsApiController : ControllerBase
{
    [HttpGet]
    public IActionResult ListProducts()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }

    [HttpGet("{id}")]
    public IActionResult GetProduct(int id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }
}
```

In the preceding example:

- The URL path `/products` can match `ProductsApi.ListProducts`
- The URL path `/products/5` can match `ProductsApi.GetProduct(int)`.

Both of these actions only match HTTP `GET` because they're marked with the `[HttpGet]` attribute.

Route templates applied to an action that begin with `/` or `~/` don't get combined with route templates applied to the controller. The following example matches a set of URL paths similar to the default route.

```C#
[Route("Home")]
public class HomeController : Controller
{
    [Route("")]
    [Route("Index")]
    [Route("/")]
    public IActionResult Index()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }
    [Route("About")]
    public IActionResult About()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }
}
```

The following table explains the `[Route]` attributes in the preceding code:

Expand table

Attribute	Combines with `[Route("Home")]`	Defines route template
`[Route("")]`	Yes	`"Home"`
`[Route("Index")]`	Yes	`"Home/Index"`
`[Route("/")]`	**No**	`""`
`[Route("About")]`	Yes	`"Home/About"`

Attribute route order

Routing builds a tree and matches all endpoints simultaneously:

- The route entries behave as if placed in an ideal ordering.
- The most specific routes have a chance to execute before the more general routes.

For example, an attribute route like `blog/search/{topic}` is more specific than an attribute route like `blog/{*article}`. The `blog/search/{topic}` route has higher priority, by default, because it's more specific. Using conventional routing, the developer is responsible for placing routes in the desired order.

Attribute routes can configure an order using the Order property. All of the framework provided route attributes include `Order` . Routes are processed according to an ascending sort of the `Order` property. The default order is `0`. Setting a route using `Order = -1` runs before routes that don't set an order. Setting a route using `Order = 1` runs after default route ordering.

Avoid depending on `Order`. If an app's URL-space requires explicit order values to route correctly, then it's likely confusing to clients as well. In general, attribute routing selects the correct route with URL matching. If the default order used for URL generation isn't working, using a route name as an override is usually simpler than applying the `Order` property.

Consider the following two controllers which both define the route matching `/home`:

C#

```csharp
public class HomeController : Controller
{
    [Route("")]
    [Route("Home")]
    [Route("Home/Index")]
    [Route("Home/Index/{id?}")]
    public IActionResult Index(int? id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }

    [Route("Home/About")]
    [Route("Home/About/{id?}")]
    public IActionResult About(int? id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }
}
```

C#

```csharp
public class MyDemoController : Controller
{
    [Route("")]
    [Route("Home")]
    [Route("Home/Index")]
    [Route("Home/Index/{id?}")]
    public IActionResult MyIndex(int? id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }

    [Route("Home/About")]
    [Route("Home/About/{id?}")]
    public IActionResult MyAbout(int? id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }
}
```

Requesting `/home` with the preceding code throws an exception similar to the following:

text

```text
AmbiguousMatchException: The request matched multiple endpoints. Matches:
```

```
WebMvcRouting.Controllers.HomeController.Index
WebMvcRouting.Controllers.MyDemoController.MyIndex
```

Adding `Order` to one of the route attributes resolves the ambiguity:

C#

```
[Route("")]
[Route("Home", Order = 2)]
[Route("Home/MyIndex")]
public IActionResult MyIndex()
{
    return ControllerContext.MyDisplayRouteInfo();
}
```

With the preceding code, `/home` runs the `HomeController.Index` endpoint. To get to the `MyDemoController.MyIndex`, request `/home/MyIndex`. **Note**:

- The preceding code is an example or poor routing design. It was used to illustrate the `Order` property.
- The `Order` property only resolves the ambiguity, that template cannot be matched. It would be better to remove the `[Route("Home")]` template.

See Razor Pages route and app conventions: Route order for information on route order with Razor Pages.

In some cases, an HTTP 500 error is returned with ambiguous routes. Use logging to see which endpoints caused the `AmbiguousMatchException`.

Token replacement in route templates [controller], [action], [area]

For convenience, attribute routes support *token replacement* by enclosing a token in square-brackets (`[`, `]`). The tokens `[action]`, `[area]`, and `[controller]` are replaced with the values of the action name, area name, and controller name from the action where the route is defined:

C#

```
[Route("[controller]/[action]")]
public class Products0Controller : Controller
{
    [HttpGet]
    public IActionResult List()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }


    [HttpGet("{id}")]
    public IActionResult Edit(int id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }
}
```

In the preceding code:

C#

```
[HttpGet]
public IActionResult List()
{
    return ControllerContext.MyDisplayRouteInfo();
}
```

- Matches `/Products0/List`

C#

```
[HttpGet("{id}")]
public IActionResult Edit(int id)
{
    return ControllerContext.MyDisplayRouteInfo(id);
}
```

- Matches `/Products0/Edit/{id}`

Token replacement occurs as the last step of building the attribute routes. The preceding example behaves the same as the following code:

C#

```
public class Products20Controller : Controller
{
    [HttpGet("[controller]/[action]")]  // Matches '/Products20/List'
    public IActionResult List()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }

    [HttpGet("[controller]/[action]/{id}")]   // Matches
'/Products20/Edit/{id}'
```

```
    public IActionResult Edit(int id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }
}
```

If you are reading this in a language other than English, let us know in this GitHub discussion issue if you'd like to see the code comments in your native language.

Attribute routes can also be combined with inheritance. This is powerful combined with token replacement. Token replacement also applies to route names defined by attribute routes. `[Route("[controller]/[action]", Name="[controller]_[action]")]` generates a unique route name for each action:

```C#
[ApiController]
[Route("api/[controller]/[action]", Name = "[controller]_[action]")]
public abstract class MyBase2Controller : ControllerBase
{
}

public class Products11Controller : MyBase2Controller
{
    [HttpGet]                      // /api/products11/list
    public IActionResult List()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }

    [HttpGet("{id}")]             //    /api/products11/edit/3
    public IActionResult Edit(int id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }
}
```

To match the literal token replacement delimiter `[` or `]`, escape it by repeating the character (`[[` or `]]`).

Use a parameter transformer to customize token replacement

Token replacement can be customized using a parameter transformer. A parameter transformer implements IOutboundParameterTransformer and transforms the value of parameters. For example, a custom `SlugifyParameterTransformer` parameter transformer changes the `SubscriptionManagement` route value to `subscription-management`:

```
C#

using System.Text.RegularExpressions;

public class SlugifyParameterTransformer : IOutboundParameterTransformer
{
    public string? TransformOutbound(object? value)
    {
        if (value == null) { return null; }

        return Regex.Replace(value.ToString()!,
                             "([a-z])([A-Z])",
                             "$1-$2",
                             RegexOptions.CultureInvariant,

TimeSpan.FromMilliseconds(100)).ToLowerInvariant();
    }
}
```

The RouteTokenTransformerConvention is an application model convention that:

- Applies a parameter transformer to all attribute routes in an application.
- Customizes the attribute route token values as they are replaced.

```
C#

public class SubscriptionManagementController : Controller
{
    [HttpGet("[controller]/[action]")]
    public IActionResult ListAll()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }
}
```

The preceding `ListAll` method matches `/subscription-management/list-all`.

The `RouteTokenTransformerConvention` is registered as an option:

```
C#

using Microsoft.AspNetCore.Mvc.ApplicationModels;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllersWithViews(options =>
{
    options.Conventions.Add(new RouteTokenTransformerConvention(
                                 new SlugifyParameterTransformer()));
});

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Home/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

app.MapControllerRoute(name: "default",
               pattern: "{controller=Home}/{action=Index}/{id?}");

app.Run();
```

See MDN web docs on Slug for the definition of Slug.

⚠ **Warning**

When using System.Text.RegularExpressions to process untrusted input, pass a timeout. A malicious user can provide input to `RegularExpressions` causing a Denial-of-Service attack. ASP.NET Core framework APIs that use `RegularExpressions` pass a timeout.

Multiple attribute routes

Attribute routing supports defining multiple routes that reach the same action. The most common usage of this is to mimic the behavior of the default conventional route as shown in the following example:

```C#
[Route("[controller]")]
public class Products13Controller : Controller
{
    [Route("")]      // Matches 'Products13'
    [Route("Index")] // Matches 'Products13/Index'
    public IActionResult Index()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }
```

Putting multiple route attributes on the controller means that each one combines with each of the route attributes on the action methods:

```C#
[Route("Store")]
[Route("[controller]")]
public class Products6Controller : Controller
{
    [HttpPost("Buy")]       // Matches 'Products6/Buy' and 'Store/Buy'
    [HttpPost("Checkout")]  // Matches 'Products6/Checkout' and 
'Store/Checkout'
    public IActionResult Buy()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }
}
```

All the HTTP verb route constraints implement `IActionConstraint`.

When multiple route attributes that implement IActionConstraint are placed on an action:

- Each action constraint combines with the route template applied to the controller.

```C#
[Route("api/[controller]")]
public class Products7Controller : ControllerBase
{
    [HttpPut("Buy")]        // Matches PUT 'api/Products7/Buy'
    [HttpPost("Checkout")]  // Matches POST 'api/Products7/Checkout'
    public IActionResult Buy()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }
}
```

Using multiple routes on actions might seem useful and powerful, it's better to keep your app's URL space basic and well defined. Use multiple routes on actions **only** where needed, for example, to support existing clients.

Specifying attribute route optional parameters, default values, and constraints

Attribute routes support the same inline syntax as conventional routes to specify optional parameters, default values, and constraints.

C#

```csharp
public class Products14Controller : Controller
{
    [HttpPost("product14/{id:int}")]
    public IActionResult ShowProduct(int id)
    {
        return ControllerContext.MyDisplayRouteInfo(id);
    }
}
```

In the preceding code, `[HttpPost("product14/{id:int}")]` applies a route constraint. The `Products14Controller.ShowProduct` action is matched only by URL paths like `/product14/3`. The route template portion `{id:int}` constrains that segment to only integers.

See Route Template Reference for a detailed description of route template syntax.

Custom route attributes using IRouteTemplateProvider

All of the route attributes implement IRouteTemplateProvider. The ASP.NET Core runtime:

- Looks for attributes on controller classes and action methods when the app starts.
- Uses the attributes that implement `IRouteTemplateProvider` to build the initial set of routes.

Implement `IRouteTemplateProvider` to define custom route attributes. Each `IRouteTemplateProvider` allows you to define a single route with a custom route template, order, and name:

C#

```
public class MyApiControllerAttribute : Attribute, IRouteTemplateProvider
{
    public string Template => "api/[controller]";
    public int? Order => 2;
    public string Name { get; set; } = string.Empty;
}

[MyApiController]
[ApiController]
public class MyTestApiController : ControllerBase
{
    // GET /api/MyTestApi
    [HttpGet]
    public IActionResult Get()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }
}
```

The preceding `Get` method returns `Order = 2, Template = api/MyTestApi`.

Use application model to customize attribute routes

The application model:

- Is an object model created at startup in `Program.cs`.
- Contains all of the metadata used by ASP.NET Core to route and execute the actions in an app.

The application model includes all of the data gathered from route attributes. The data from route attributes is provided by the `IRouteTemplateProvider` implementation. Conventions:

- Can be written to modify the application model to customize how routing behaves.
- Are read at app startup.

This section shows a basic example of customizing routing using application model. The following code makes routes roughly line up with the folder structure of the project.

C#

```
public class NamespaceRoutingConvention : Attribute,
IControllerModelConvention
{
    private readonly string _baseNamespace;

    public NamespaceRoutingConvention(string baseNamespace)
    {
        _baseNamespace = baseNamespace;
    }

    public void Apply(ControllerModel controller)
    {
        var hasRouteAttributes = controller.Selectors.Any(selector =>
                                                selector.AttributeRouteModel
!= null);
        if (hasRouteAttributes)
        {
            return;
        }

        var namespc = controller.ControllerType.Namespace;
        if (namespc == null)
            return;
        var template = new StringBuilder();
        template.Append(namespc, _baseNamespace.Length + 1,
                        namespc.Length - _baseNamespace.Length - 1);
        template.Replace('.', '/');
        template.Append("/[controller]/[action]/{id?}");

        foreach (var selector in controller.Selectors)
        {
            selector.AttributeRouteModel = new AttributeRouteModel()
            {
                Template = template.ToString()
            };
        }
    }
}
```

The following code prevents the `namespace` convention from being applied to controllers that are attribute routed:

C#

```csharp
public void Apply(ControllerModel controller)
{
    var hasRouteAttributes = controller.Selectors.Any(selector =>
                                            selector.AttributeRouteModel !=
null);
    if (hasRouteAttributes)
    {
        return;
    }
```

For example, the following controller doesn't use `NamespaceRoutingConvention`:

C#

```
[Route("[controller]/[action]/{id?}")]
public class ManagersController : Controller
{
    // /managers/index
    public IActionResult Index()
    {
        var template = 
ControllerContext.ActionDescriptor.AttributeRouteInfo?.Template;
        return Content($"Index- template:{template}");
    }

    public IActionResult List(int? id)
    {
        var path = Request.Path.Value;
        return Content($"List- Path:{path}");
    }
}
```

The `NamespaceRoutingConvention.Apply` method:

- Does nothing if the controller is attribute routed.
- Sets the controllers template based on the `namespace`, with the base `namespace` removed.

The `NamespaceRoutingConvention` can be applied in `Program.cs`:

C#

```csharp
using My.Application.Controllers;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllersWithViews(options =>
{
    options.Conventions.Add(
     new NamespaceRoutingConvention(typeof(HomeController).Namespace!));
});

var app = builder.Build();
```

For example, consider the following controller:

C#

```csharp
using Microsoft.AspNetCore.Mvc;

namespace My.Application.Admin.Controllers
{
    public class UsersController : Controller
    {
        // GET /admin/controllers/users/index
        public IActionResult Index()
        {
            var fullname = typeof(UsersController).FullName;
            var template = 
ControllerContext.ActionDescriptor.AttributeRouteInfo?.Template;
            var path = Request.Path.Value;

            return Content($"Path: {path} fullname: {fullname}  template:
{template}");
        }

        public IActionResult List(int? id)
        {
            var path = Request.Path.Value;
            return Content($"Path: {path} ID:{id}");
        }
    }
}
```

In the preceding code:

- The base `namespace` is `My.Application`.
- The full name of the preceding controller is `My.Application.Admin.Controllers.UsersController`.
- The `NamespaceRoutingConvention` sets the controllers template to `Admin/Controllers/Users/[action]/{id?`.

The `NamespaceRoutingConvention` can also be applied as an attribute on a controller:

```C#
[NamespaceRoutingConvention("My.Application")]
public class TestController : Controller
{
    // /admin/controllers/test/index
    public IActionResult Index()
    {
        var template = 
ControllerContext.ActionDescriptor.AttributeRouteInfo?.Template;
        var actionname = ControllerContext.ActionDescriptor.ActionName;
        return Content($"Action- {actionname} template:{template}");
    }

    public IActionResult List(int? id)
    {
        var path = Request.Path.Value;
        return Content($"List- Path:{path}");
    }
}
```

Mixed routing: Attribute routing vs conventional routing

ASP.NET Core apps can mix the use of conventional routing and attribute routing. It's typical to use conventional routes for controllers serving HTML pages for browsers, and attribute routing for controllers serving REST APIs.

Actions are either conventionally routed or attribute routed. Placing a route on the controller or the action makes it attribute routed. Actions that define attribute routes cannot be reached through the conventional routes and vice-versa. ***Any*** route attribute on the controller makes ***all*** actions in the controller attribute routed.

Attribute routing and conventional routing use the same routing engine.

Routing with special characters

Routing with special characters can lead to unexpected results. For example, consider a controller with the following action method:

C#

```csharp
[HttpGet("{id?}/name")]
public async Task<ActionResult<string>> GetName(string id)
{
    var todoItem = await _context.TodoItems.FindAsync(id);

    if (todoItem == null || todoItem.Name == null)
    {
        return NotFound();
    }

    return todoItem.Name;
}
```

When `string id` contains the following encoded values, unexpected results might occur:

Expand table

ASCII	Encoded
`/`	`%2F`
` `	`+`

Route parameters are not always URL decoded. This problem may be addressed in the future. For more information, see this GitHub issue;

URL Generation and ambient values

Apps can use routing URL generation features to generate URL links to actions. Generating URLs eliminates hard-coding URLs, making code more robust and maintainable. This section focuses on the URL generation features provided by MVC and only cover basics of how URL generation works. See Routing for a detailed description of URL generation.

The IUrlHelper interface is the underlying element of infrastructure between MVC and routing for URL generation. An instance of `IUrlHelper` is available through the `Url` property in controllers, views, and view components.

In the following example, the `IUrlHelper` interface is used through the `Controller.Url` property to generate a URL to another action.

```C#
public class UrlGenerationController : Controller
{
    public IActionResult Source()
    {
        // Generates /UrlGeneration/Destination
        var url = Url.Action("Destination");
        return ControllerContext.MyDisplayRouteInfo("", $" URL = {url}");
    }

    public IActionResult Destination()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }
}
```

If the app is using the default conventional route, the value of the `url` variable is the URL path string `/UrlGeneration/Destination`. This URL path is created by routing by combining:

- The route values from the current request, which are called **ambient values**.

- The values passed to `Url.Action` and substituting those values into the route template:

```text
ambient values: { controller = "UrlGeneration", action = "Source" }
values passed to Url.Action: { controller = "UrlGeneration", action =
"Destination" }
route template: {controller}/{action}/{id?}

result: /UrlGeneration/Destination
```

Each route parameter in the route template has its value substituted by matching names with the values and ambient values. A route parameter that doesn't have a value can:

- Use a default value if it has one.
- Be skipped if it's optional. For example, the `id` from the route template `{controller}/{action}/{id?}`.

URL generation fails if any required route parameter doesn't have a corresponding value. If URL generation fails for a route, the next route is tried until all routes have been tried or a match is found.

The preceding example of `Url.Action` assumes conventional routing. URL generation works similarly with attribute routing, though the concepts are different. With conventional routing:

- The route values are used to expand a template.
- The route values for `controller` and `action` usually appear in that template. This works because the URLs matched by routing adhere to a convention.

The following example uses attribute routing:

```C#
public class UrlGenerationAttrController : Controller
{
    [HttpGet("custom")]
    public IActionResult Source()
    {
        var url = Url.Action("Destination");
        return ControllerContext.MyDisplayRouteInfo("", $" URL = {url}");
    }

    [HttpGet("custom/url/to/destination")]
    public IActionResult Destination()
    {
       return ControllerContext.MyDisplayRouteInfo();
    }
}
```

The `Source` action in the preceding code generates `custom/url/to/destination`.

LinkGenerator was added in ASP.NET Core 3.0 as an alternative to `IUrlHelper`. `LinkGenerator` offers similar but more flexible functionality. Each method on `IUrlHelper` has a corresponding family of methods on `LinkGenerator` as well.

Generating URLs by action name

Url.Action, LinkGenerator.GetPathByAction, and all related overloads all are designed to generate the target endpoint by specifying a controller name and action name.

When using `Url.Action`, the current route values for `controller` and `action` are provided by the runtime:

- The value of `controller` and `action` are part of both ambient values and values. The method `Url.Action` always uses the current values of `action` and `controller` and generates a URL path that routes to the current action.

Routing attempts to use the values in ambient values to fill in information that wasn't provided when generating a URL. Consider a route like `{a}/{b}/{c}/{d}` with ambient values `{ a = Alice, b = Bob, c = Carol, d = David }`:

- Routing has enough information to generate a URL without any additional values.
- Routing has enough information because all route parameters have a value.

If the value `{ d = Donovan }` is added:

- The value `{ d = David }` is ignored.
- The generated URL path is `Alice/Bob/Carol/Donovan`.

Warning: URL paths are hierarchical. In the preceding example, if the value `{ c = Cheryl }` is added:

- Both of the values `{ c = Carol, d = David }` are ignored.
- There is no longer a value for `d` and URL generation fails.
- The desired values of `c` and `d` must be specified to generate a URL.

You might expect to hit this problem with the default route `{controller}/{action}/{id?}`. This problem is rare in practice because `Url.Action` always explicitly specifies a `controller` and `action` value.

Several overloads of Url.Action take a route values object to provide values for route parameters other than `controller` and `action`. The route values object is frequently used with `id`. For example, `Url.Action("Buy", "Products", new { id = 17 })`. The route values object:

- By convention is usually an object of anonymous type.
- Can be an `IDictionary<>` or a POCO).

Any additional route values that don't match route parameters are put in the query string.

```C#
public IActionResult Index()
{
    var url = Url.Action("Buy", "Products", new { id = 17, color = "red" });
    return Content(url!);
}
```

The preceding code generates `/Products/Buy/17?color=red`.

The following code generates an absolute URL:

```C#
public IActionResult Index2()
{
    var url = Url.Action("Buy", "Products", new { id = 17 }, protocol: 
Request.Scheme);
    // Returns https://localhost:5001/Products/Buy/17
    return Content(url!);
}
```

To create an absolute URL, use one of the following:

- An overload that accepts a `protocol`. For example, the preceding code.
- LinkGenerator.GetUriByAction, which generates absolute URIs by default.

Generate URLs by route

The preceding code demonstrated generating a URL by passing in the controller and action name. `IUrlHelper` also provides the Url.RouteUrl family of methods. These methods are similar to Url.Action, but they don't copy the current values of `action` and `controller` to the route values. The most common usage of `Url.RouteUrl`:

- Specifies a route name to generate the URL.
- Generally doesn't specify a controller or action name.

C#

```csharp
public class UrlGeneration2Controller : Controller
{
    [HttpGet("")]
    public IActionResult Source()
    {
        var url = Url.RouteUrl("Destination_Route");
        return ControllerContext.MyDisplayRouteInfo("", $" URL = {url}");
    }

    [HttpGet("custom/url/to/destination2", Name = "Destination_Route")]
    public IActionResult Destination()
    {
        return ControllerContext.MyDisplayRouteInfo();
    }
```

The following Razor file generates an HTML link to the `Destination_Route`:

CSHTML

```cshtml
<h1>Test Links</h1>

<ul>
    <li><a href="@Url.RouteUrl("Destination_Route")">Test
Destination_Route</a></li>
</ul>
```

Generate URLs in HTML and Razor

IHtmlHelper provides the HtmlHelper methods Html.BeginForm and Html.ActionLink to generate `<form>` and `<a>` elements respectively. These methods use the Url.Action method to generate a URL and they accept similar arguments. The `Url.RouteUrl` companions for `HtmlHelper` are `Html.BeginRouteForm` and `Html.RouteLink` which have similar functionality.

TagHelpers generate URLs through the `form` TagHelper and the `<a>` TagHelper. Both of these use `IUrlHelper` for their implementation. See Tag Helpers in forms for more information.

Inside views, the `IUrlHelper` is available through the `Url` property for any ad-hoc URL generation not covered by the above.

URL generation in Action Results

The preceding examples showed using `IUrlHelper` in a controller. The most common usage in a controller is to generate a URL as part of an action result.

The ControllerBase and Controller base classes provide convenience methods for action results that reference another action. One typical usage is to redirect after accepting user input:

```C#
[HttpPost]
[ValidateAntiForgeryToken]
public IActionResult Edit(int id, Customer customer)
{
    if (ModelState.IsValid)
    {
        // Update DB with new details.
        ViewData["Message"] = $"Successful edit of customer {id}";
        return RedirectToAction("Index");
    }
    return View(customer);
}
```

The action results factory methods such as RedirectToAction and CreatedAtAction follow a similar pattern to the methods on `IUrlHelper`.

Special case for dedicated conventional routes

Conventional routing can use a special kind of route definition called a dedicated conventional route. In the following example, the route named `blog` is a dedicated conventional route:

```C#
app.MapControllerRoute(name: "blog",
                pattern: "blog/{*article}",
                defaults: new { controller = "Blog", action = "Article" });
app.MapControllerRoute(name: "default",
               pattern: "{controller=Home}/{action=Index}/{id?}");
```

Using the preceding route definitions, `Url.Action("Index", "Home")` generates the URL path `/` using the `default` route, but why? You might guess the route values `{`

`controller = Home, action = Index }` would be enough to generate a URL using `blog`, and the result would be `/blog?action=Index&controller=Home`.

Dedicated conventional routes rely on a special behavior of default values that don't have a corresponding route parameter that prevents the route from being too greedy with URL generation. In this case the default values are `{ controller = Blog, action = Article }`, and neither `controller` nor `action` appears as a route parameter. When routing performs URL generation, the values provided must match the default values. URL generation using `blog` fails because the values `{ controller = Home, action = Index }` don't match `{ controller = Blog, action = Article }`. Routing then falls back to try `default`, which succeeds.

Areas

Areas are an MVC feature used to organize related functionality into a group as a separate:

- Routing namespace for controller actions.
- Folder structure for views.

Using areas allows an app to have multiple controllers with the same name, as long as they have different areas. Using areas creates a hierarchy for the purpose of routing by adding another route parameter, `area` to `controller` and `action`. This section discusses how routing interacts with areas. See Areas for details about how areas are used with views.

The following example configures MVC to use the default conventional route and an `area` route for an `area` named `Blog`:

C#

```csharp
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllersWithViews();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Home/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();
app.UseRouting();

app.UseAuthorization();

app.MapAreaControllerRoute("blog_route", "Blog",
        "Manage/{controller}/{action}/{id?}");
app.MapControllerRoute("default_route", "{controller}/{action}/{id?}");

app.Run();
```

In the preceding code, MapAreaControllerRoute is called to create the `"blog_route"`. The second parameter, `"Blog"`, is the area name.

When matching a URL path like `/Manage/Users/AddUser`, the `"blog_route"` route generates the route values `{ area = Blog, controller = Users, action = AddUser }`. The `area` route value is produced by a default value for `area`. The route created by `MapAreaControllerRoute` is equivalent to the following:

C#

```
app.MapControllerRoute("blog_route", "Manage/{controller}/{action}/{id?}",
        defaults: new { area = "Blog" }, constraints: new { area = "Blog"
});
app.MapControllerRoute("default_route", "{controller}/{action}/{id?}");
```

`MapAreaControllerRoute` creates a route using both a default value and constraint for `area` using the provided area name, in this case `Blog`. The default value ensures that the route always produces `{ area = Blog, ... }`, the constraint requires the value `{ area = Blog, ... }` for URL generation.

Conventional routing is order-dependent. In general, routes with areas should be placed earlier as they're more specific than routes without an area.

Using the preceding example, the route values `{ area = Blog, controller = Users, action = AddUser }` match the following action:

C#

```
using Microsoft.AspNetCore.Mvc;

namespace MyApp.Namespace1
{
    [Area("Blog")]
    public class UsersController : Controller
    {
        // GET /manage/users/adduser
        public IActionResult AddUser()
        {
            var area =
ControllerContext.ActionDescriptor.RouteValues["area"];
            var actionName = ControllerContext.ActionDescriptor.ActionName;
            var controllerName =
ControllerContext.ActionDescriptor.ControllerName;

            return Content($"area name:{area}" +
                $" controller:{controllerName}  action name: {actionName}");
        }
    }
}
```

The [Area] attribute is what denotes a controller as part of an area. This controller is in the `Blog` area. Controllers without an `[Area]` attribute are not members of any area, and do **not** match when the `area` route value is provided by routing. In the following example, only the first controller listed can match the route values `{ area = Blog, controller = Users, action = AddUser }`.

C#

```
using Microsoft.AspNetCore.Mvc;

namespace MyApp.Namespace1
{
    [Area("Blog")]
    public class UsersController : Controller
    {
        // GET /manage/users/adduser
        public IActionResult AddUser()
        {
            var area =
ControllerContext.ActionDescriptor.RouteValues["area"];
            var actionName = ControllerContext.ActionDescriptor.ActionName;
            var controllerName =
ControllerContext.ActionDescriptor.ControllerName;

            return Content($"area name:{area}" +
                $" controller:{controllerName}  action name: {actionName}");
        }
    }
}
```

C#

```
using Microsoft.AspNetCore.Mvc;

namespace MyApp.Namespace2
{
    // Matches { area = Zebra, controller = Users, action = AddUser }
    [Area("Zebra")]
    public class UsersController : Controller
    {
        // GET /zebra/users/adduser
        public IActionResult AddUser()
        {
            var area =
ControllerContext.ActionDescriptor.RouteValues["area"];
            var actionName = ControllerContext.ActionDescriptor.ActionName;
            var controllerName =
ControllerContext.ActionDescriptor.ControllerName;

            return Content($"area name:{area}" +
                $" controller:{controllerName}  action name: {actionName}");
        }
    }
}
```

C#

```csharp
using Microsoft.AspNetCore.Mvc;

namespace MyApp.Namespace3
{
    // Matches { area = string.Empty, controller = Users, action = AddUser }
    // Matches { area = null, controller = Users, action = AddUser }
    // Matches { controller = Users, action = AddUser }
    public class UsersController : Controller
    {
        // GET /users/adduser
        public IActionResult AddUser()
        {
            var area =
ControllerContext.ActionDescriptor.RouteValues["area"];
            var actionName = ControllerContext.ActionDescriptor.ActionName;
            var controllerName =
ControllerContext.ActionDescriptor.ControllerName;

            return Content($"area name:{area}" +
                $" controller:{controllerName}  action name: {actionName}");
        }
    }
}
```

The namespace of each controller is shown here for completeness. If the preceding controllers used the same namespace, a compiler error would be generated. Class namespaces have no effect on MVC's routing.

The first two controllers are members of areas, and only match when their respective area name is provided by the `area` route value. The third controller isn't a member of any area, and can only match when no value for `area` is provided by routing.

In terms of matching *no value*, the absence of the `area` value is the same as if the value for `area` were null or the empty string.

When executing an action inside an area, the route value for `area` is available as an ambient value for routing to use for URL generation. This means that by default areas act *sticky* for URL generation as demonstrated by the following sample.

```C#
app.MapAreaControllerRoute(name: "duck_route",
                                     areaName: "Duck",
                                     pattern:
"Manage/{controller}/{action}/{id?}");
app.MapControllerRoute(name: "default",
                             pattern:
"Manage/{controller=Home}/{action=Index}/{id?}");
```

```C#
using Microsoft.AspNetCore.Mvc;

namespace MyApp.Namespace4
{
    [Area("Duck")]
    public class UsersController : Controller
    {
        // GET /Manage/users/GenerateURLInArea
        public IActionResult GenerateURLInArea()
        {
            // Uses the 'ambient' value of area.
            var url = Url.Action("Index", "Home");
            // Returns /Manage/Home/Index
            return Content(url);
        }

        // GET /Manage/users/GenerateURLOutsideOfArea
        public IActionResult GenerateURLOutsideOfArea()
        {
            // Uses the empty value for area.
            var url = Url.Action("Index", "Home", new { area = "" });
            // Returns /Manage
            return Content(url);
        }
    }
}
```

The following code generates a URL to `/Zebra/Users/AddUser`:

C#

```csharp
public class HomeController : Controller
{
    public IActionResult About()
    {
        var url = Url.Action("AddUser", "Users", new { Area = "Zebra" });
        return Content($"URL: {url}");
    }
```

Action definition

Public methods on a controller, except those with the NonAction attribute, are actions.

Sample code

- MyDisplayRouteInfo is provided by the Rick.Docs.Samples.RouteInfo NuGet package and displays route information.
- View or download sample code (how to download)

Debug diagnostics

For detailed routing diagnostic output, set `Logging:LogLevel:Microsoft` to `Debug`. In the development environment, set the log level in `appsettings.Development.json`:

JSON

```json
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Debug",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  }
}
```

Dependency injection into controllers in ASP.NET Core

Article • 06/17/2024

By Shadi Alnamrouti and Rick Anderson

ASP.NET Core MVC controllers request dependencies explicitly via constructors. ASP.NET Core has built-in support for dependency injection (DI). DI makes apps easier to test and maintain.

View or download sample code (how to download)

Constructor injection

Services are added as a constructor parameter, and the runtime resolves the service from the service container. Services are typically defined using interfaces. For example, consider an app that requires the current time. The following interface exposes the `IDateTime` service:

C#

```
public interface IDateTime
{
    DateTime Now { get; }
}
```

The following code implements the `IDateTime` interface:

C#

```
public class SystemDateTime : IDateTime
{
    public DateTime Now
    {
        get { return DateTime.Now; }
    }
}
```

Add the service to the service container:

C#

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddSingleton<IDateTime, SystemDateTime>();

    services.AddControllersWithViews();
}
```

For more information on AddSingleton, see DI service lifetimes.

The following code displays a greeting to the user based on the time of day:

```C#
public class HomeController : Controller
{
    private readonly IDateTime _dateTime;

    public HomeController(IDateTime dateTime)
    {
        _dateTime = dateTime;
    }

    public IActionResult Index()
    {
        var serverTime = _dateTime.Now;
        if (serverTime.Hour < 12)
        {
            ViewData["Message"] = "It's morning here - Good Morning!";
        }
        else if (serverTime.Hour < 17)
        {
            ViewData["Message"] = "It's afternoon here - Good Afternoon!";
        }
        else
        {
            ViewData["Message"] = "It's evening here - Good Evening!";
        }
        return View();
    }
```

Run the app and a message is displayed based on the time.

Action injection with `FromServices`

The FromServicesAttribute enables injecting a service directly into an action method without using constructor injection:

C#

```
public IActionResult About([FromServices] IDateTime dateTime)
{
    return Content( $"Current server time: {dateTime.Now}");
}
```

Action injection with `FromKeyedServices`

The following code shows how to access keyed services from the DI container by using the [FromKeyedServices] attribute:

C#

```csharp
using Microsoft.AspNetCore.Mvc;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddKeyedSingleton<ICache, BigCache>("big");
builder.Services.AddKeyedSingleton<ICache, SmallCache>("small");
builder.Services.AddControllers();

var app = builder.Build();

app.MapControllers();

app.Run();

public interface ICache
{
    object Get(string key);
}
public class BigCache : ICache
{
    public object Get(string key) => $"Resolving {key} from big cache.";
}

public class SmallCache : ICache
{
    public object Get(string key) => $"Resolving {key} from small cache.";
}

[ApiController]
[Route("/cache")]
public class CustomServicesApiController : Controller
{
    [HttpGet("big")]
    public ActionResult<object> GetBigCache([FromKeyedServices("big")]
ICache cache)
    {
        return cache.Get("data-mvc");
    }
    [HttpGet("small")]
    public ActionResult<object> GetSmallCache([FromKeyedServices("small")]
ICache cache)
    {
        return cache.Get("data-mvc");
    }
}
```

Access settings from a controller

Accessing app or configuration settings from within a controller is a common pattern. The *options pattern* described in Options pattern in ASP.NET Core is the preferred approach to manage settings. Generally, don't directly inject IConfiguration into a controller.

Create a class that represents the options. For example:

```C#
public class SampleWebSettings
{
    public string Title { get; set; }
    public int Updates { get; set; }
}
```

Add the configuration class to the services collection:

```C#
public void ConfigureServices(IServiceCollection services)
{
    services.AddSingleton<IDateTime, SystemDateTime>();
    services.Configure<SampleWebSettings>(Configuration);

    services.AddControllersWithViews();
}
```

Configure the app to read the settings from a JSON-formatted file:

```C#
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureAppConfiguration((hostingContext, config) =>
            {
                config.AddJsonFile("samplewebsettings.json",
                    optional: false,
                    reloadOnChange: true);
            })
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

The following code requests the `IOptions<SampleWebSettings>` settings from the service container and uses them in the `Index` method:

C#

```
public class SettingsController : Controller
{
    private readonly SampleWebSettings _settings;

    public SettingsController(IOptions<SampleWebSettings> settingsOptions)
    {
        _settings = settingsOptions.Value;
    }

    public IActionResult Index()
    {
        ViewData["Title"] = _settings.Title;
        ViewData["Updates"] = _settings.Updates;
        return View();
    }
}
```

Additional resources

- See Test controller logic in ASP.NET Core to learn how to make code easier to test by explicitly requesting dependencies in controllers.
- Keyed service dependency injection container support
- Replace the default dependency injection container with a third party implementation.

Dependency injection into views in ASP.NET Core

Article • 06/17/2024

ASP.NET Core supports dependency injection into views. This can be useful for view-specific services, such as localization or data required only for populating view elements. Most of the data views display should be passed in from the controller.

View or download sample code (how to download)

Configuration injection

The values in settings files, such as `appsettings.json` and `appsettings.Development.json`, can be injected into a view. Consider the `appsettings.Development.json` from the sample code:

JSON

```json
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "MyRoot": {
    "MyParent": {
      "MyChildName": "Joe"
    }
  }
}
```

The following markup displays the configuration value in a Razor Pages view:

CSHTML

```cshtml
@page
@model PrivacyModel
@using Microsoft.Extensions.Configuration
@inject IConfiguration Configuration
@{
    ViewData["Title"] = "Privacy RP";
}
<h1>@ViewData["Title"]</h1>
```

```
<p>PR Privacy</p>

<h2>
   MyRoot:MyParent:MyChildName:
@Configuration["MyRoot:MyParent:MyChildName"]
</h2>
```

The following markup displays the configuration value in a MVC view:

CSHTML

```
@using Microsoft.Extensions.Configuration
@inject IConfiguration Configuration
@{
    ViewData["Title"] = "Privacy MVC";
}
<h1>@ViewData["Title"]</h1>

<p>MVC Use this page to detail your site's privacy policy.</p>

<h2>
   MyRoot:MyParent:MyChildName:
@Configuration["MyRoot:MyParent:MyChildName"]
</h2>
```

For more information, see Configuration in ASP.NET Core

Service injection

A service can be injected into a view using the `@inject` directive.

CSHTML

```
@using System.Threading.Tasks
@using ViewInjectSample.Model
@using ViewInjectSample.Model.Services
@model IEnumerable<ToDoItem>
@inject StatisticsService StatsService
<!DOCTYPE html>
<html>
<head>
    <title>To Do Items</title>
</head>
<body>
    <div>
        <h1>To Do Items</h1>
        <ul>
            <li>Total Items: @StatsService.GetCount()</li>
            <li>Completed: @StatsService.GetCompletedCount()</li>
            <li>Avg. Priority: @StatsService.GetAveragePriority()</li>
        </ul>
        <table>
            <tr>
                <th>Name</th>
                <th>Priority</th>
                <th>Is Done?</th>
            </tr>
            @foreach (var item in Model)
            {
                <tr>
                    <td>@item.Name</td>
                    <td>@item.Priority</td>
                    <td>@item.IsDone</td>
                </tr>
            }
        </table>
    </div>
</body>
</html>
```

This view displays a list of `ToDoItem` instances, along with a summary showing overall statistics. The summary is populated from the injected `StatisticsService`. This service is registered for dependency injection in `ConfigureServices` in `Program.cs`:

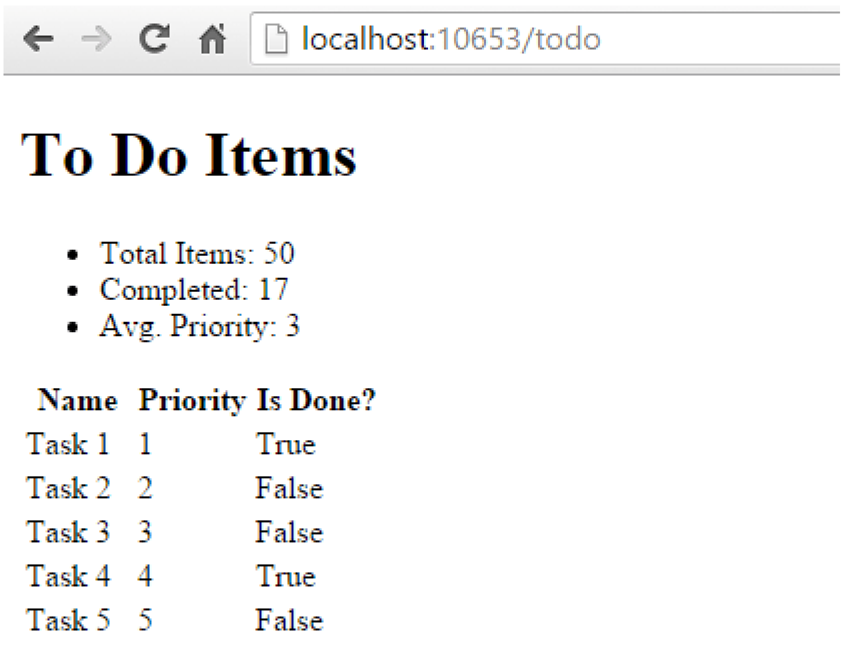
```C#
using ViewInjectSample.Helpers;
using ViewInjectSample.Infrastructure;
using ViewInjectSample.Interfaces;
using ViewInjectSample.Model.Services;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllersWithViews();
builder.Services.AddRazorPages();

builder.Services.AddTransient<IToDoItemRepository, ToDoItemRepository>();
builder.Services.AddTransient<StatisticsService>();
builder.Services.AddTransient<ProfileOptionsService>();
builder.Services.AddTransient<MyHtmlHelper>();

var app = builder.Build();

if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Home/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();
app.UseRouting();

app.MapRazorPages();

app.MapDefaultControllerRoute();


app.Run();
```

The `StatisticsService` performs some calculations on the set of `ToDoItem` instances, which it accesses via a repository:

```C#
using System.Linq;
using ViewInjectSample.Interfaces;

namespace ViewInjectSample.Model.Services
{
    public class StatisticsService
    {
        private readonly IToDoItemRepository _toDoItemRepository;

        public StatisticsService(IToDoItemRepository toDoItemRepository)
        {
            _toDoItemRepository = toDoItemRepository;
        }

        public int GetCount()
        {
            return _toDoItemRepository.List().Count();
        }

        public int GetCompletedCount()
        {
            return _toDoItemRepository.List().Count(x => x.IsDone);
        }

        public double GetAveragePriority()
        {
            if (_toDoItemRepository.List().Count() == 0)
            {
                return 0.0;
            }

            return _toDoItemRepository.List().Average(x => x.Priority);
        }
    }
}
```

The sample repository uses an in-memory collection. An in-memory implementation shouldn't be used for large, remotely accessed data sets.

The sample displays data from the model bound to the view and the service injected into the view:


localhost:10653/todo

To Do Items

- Total Items: 50
- Completed: 17
- Avg. Priority: 3

Name	Priority	Is Done?
Task 1	1	True
Task 2	2	False
Task 3	3	False
Task 4	4	True
Task 5	5	False

Populating Lookup Data

View injection can be useful to populate options in UI elements, such as dropdown lists. Consider a user profile form that includes options for specifying gender, state, and other preferences. Rendering such a form using a standard approach might require the controller or Razor Page to:

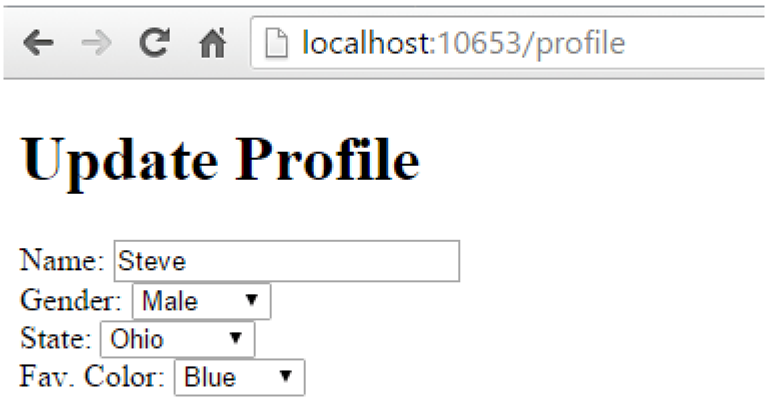
- Request data access services for each of the sets of options.
- Populate a model or `ViewBag` with each set of options to be bound.

An alternative approach injects services directly into the view to obtain the options. This minimizes the amount of code required by the controller or razor Page, moving this view element construction logic into the view itself. The controller action or Razor Page to display a profile editing form only needs to pass the form the profile instance:

C#

```csharp
using Microsoft.AspNetCore.Mvc;
using ViewInjectSample.Model;

namespace ViewInjectSample.Controllers;

public class ProfileController : Controller
{
    public IActionResult Index()
    {
        // A real app would up profile based on the user.
        var profile = new Profile()
        {
            Name = "Rick",
            FavColor = "Blue",
            Gender = "Male",
            State = new State("Ohio","OH")
        };
        return View(profile);
    }
}
```

The HTML form used to update the preferences includes dropdown lists for three of the properties:



Update Profile

Name: Steve
Gender: Male
State: Ohio
Fav. Color: Blue

These lists are populated by a service that has been injected into the view:

CSHTML

```
@using System.Threading.Tasks
@using ViewInjectSample.Model.Services
@model ViewInjectSample.Model.Profile
@inject ProfileOptionsService Options
<!DOCTYPE html>
<html>
<head>
    <title>Update Profile</title>
</head>
<body>
<div>
    <h1>Update Profile</h1>
    Name: @Html.TextBoxFor(m => m.Name)
    <br/>
    Gender: @Html.DropDownList("Gender",
           Options.ListGenders().Select(g =>
                new SelectListItem() { Text = g, Value = g }))
    <br/>
    State: @Html.DropDownListFor(m => m.State!.Code,
           Options.ListStates().Select(s =>
                new SelectListItem() { Text = s.Name, Value = s.Code}))
    <br />

    Fav. Color: @Html.DropDownList("FavColor",
           Options.ListColors().Select(c =>
                new SelectListItem() { Text = c, Value = c }))
    </div>
</body>
</html>
```

The `ProfileOptionsService` is a UI-level service designed to provide just the data needed for this form:

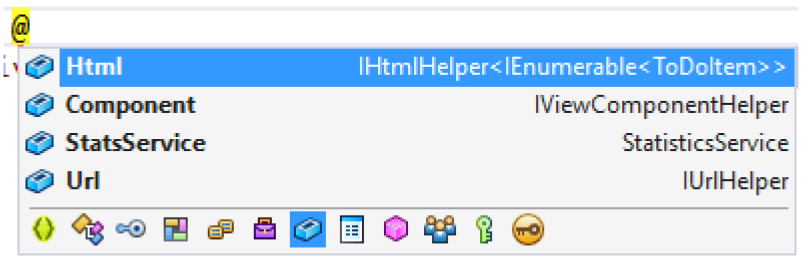
C#

```csharp
namespace ViewInjectSample.Model.Services;

public class ProfileOptionsService
{
    public List<string> ListGenders()
    {
        // Basic sample
        return new List<string>() {"Female", "Male"};
    }

    public List<State> ListStates()
    {
        // Add a few states
        return new List<State>()
        {
            new State("Alabama", "AL"),
            new State("Alaska", "AK"),
            new State("Ohio", "OH")
        };
    }

    public List<string> ListColors()
    {
        return new List<string>() { "Blue","Green","Red","Yellow" };
    }
}
```

Note an unregistered type throws an exception at runtime because the service provider is internally queried via GetRequiredService.

Overriding Services

In addition to injecting new services, this technique can be used to override previously injected services on a page. The figure below shows all of the fields available on the page used in the first example:



Html	IHtmlHelper<IEnumerable<ToDoItem>>
Component	IViewComponentHelper
StatsService	StatisticsService
Url	IUrlHelper

The default fields include `Html`, `Component`, and `Url`. To replace the default HTML Helpers with a custom version, use `@inject`:

CSHTML

```cshtml
@using System.Threading.Tasks
@using ViewInjectSample.Helpers
@inject MyHtmlHelper Html
<!DOCTYPE html>
<html>
<head>
    <title>My Helper</title>
</head>
<body>
    <div>
        Test: @Html.Value
    </div>
</body>
</html>
```

See Also

- Simon Timms Blog: Getting Lookup Data Into Your View

Unit test controller logic in ASP.NET Core

Article • 06/17/2024

By Steve Smith

Unit tests involve testing a part of an app in isolation from its infrastructure and dependencies. When unit testing controller logic, only the contents of a single action are tested, not the behavior of its dependencies or of the framework itself.

Unit testing controllers

Set up unit tests of controller actions to focus on the controller's behavior. A controller unit test avoids scenarios such as filters, routing, and model binding. Tests that cover the interactions among components that collectively respond to a request are handled by *integration tests*. For more information on integration tests, see Integration tests in ASP.NET Core.

If you're writing custom filters and routes, unit test them in isolation, not as part of tests on a particular controller action.

To demonstrate controller unit tests, review the following controller in the sample app.

View or download sample code (how to download)

The Home controller displays a list of brainstorming sessions and allows the creation of new brainstorming sessions with a POST request:

C#

```csharp
public class HomeController : Controller
{
    private readonly IBrainstormSessionRepository _sessionRepository;

    public HomeController(IBrainstormSessionRepository sessionRepository)
    {
        _sessionRepository = sessionRepository;
    }

    public async Task<IActionResult> Index()
    {
        var sessionList = await _sessionRepository.ListAsync();

        var model = sessionList.Select(session => new
StormSessionViewModel()
```

```
            {
                Id = session.Id,
                DateCreated = session.DateCreated,
                Name = session.Name,
                IdeaCount = session.Ideas.Count
            });

            return View(model);
        }

        public class NewSessionModel
        {
            [Required]
            public string SessionName { get; set; }
        }

        [HttpPost]
        public async Task<IActionResult> Index(NewSessionModel model)
        {
            if (!ModelState.IsValid)
            {
                return BadRequest(ModelState);
            }
            else
            {
                await _sessionRepository.AddAsync(new BrainstormSession()
                {
                    DateCreated = DateTimeOffset.Now,
                    Name = model.SessionName
                });
            }

            return RedirectToAction(actionName: nameof(Index));
        }
    }
```

The preceding controller:

- Follows the Explicit Dependencies Principle.
- Expects dependency injection (DI) to provide an instance of `IBrainstormSessionRepository`.
- Can be tested with a mocked `IBrainstormSessionRepository` service using a mock object framework, such as Moq. A *mocked object* is a fabricated object with a predetermined set of property and method behaviors used for testing. For more information, see Introduction to integration tests.

The `HTTP GET Index` method has no looping or branching and only calls one method. The unit test for this action:

- Mocks the `IBrainstormSessionRepository` service using the `GetTestSessions` method. `GetTestSessions` creates two mock brainstorm sessions with dates and session names.
- Executes the `Index` method.
- Makes assertions on the result returned by the method:
  - A ViewResult is returned.
  - The ViewDataDictionary.Model is a `StormSessionViewModel`.
  - There are two brainstorming sessions stored in the `ViewDataDictionary.Model`.

C#

```csharp
[Fact]
public async Task Index_ReturnsAViewResult_WithAListOfBrainstormSessions()
{
    // Arrange
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.ListAsync())
        .ReturnsAsync(GetTestSessions());
    var controller = new HomeController(mockRepo.Object);

    // Act
    var result = await controller.Index();

    // Assert
    var viewResult = Assert.IsType<ViewResult>(result);
    var model = Assert.IsAssignableFrom<IEnumerable<StormSessionViewModel>>(
        viewResult.ViewData.Model);
    Assert.Equal(2, model.Count());
}
```

C#

```csharp
private List<BrainstormSession> GetTestSessions()
{
    var sessions = new List<BrainstormSession>();
    sessions.Add(new BrainstormSession()
    {
        DateCreated = new DateTime(2016, 7, 2),
        Id = 1,
        Name = "Test One"
    });
    sessions.Add(new BrainstormSession()
    {
        DateCreated = new DateTime(2016, 7, 1),
        Id = 2,
        Name = "Test Two"
    });
    return sessions;
}
```

The Home controller's `HTTP POST Index` method tests verifies that:

- When ModelState.IsValid is `false`, the action method returns a *400 Bad Request* ViewResult with the appropriate data.
- When `ModelState.IsValid` is `true`:
    - The `Add` method on the repository is called.
    - A RedirectToActionResult is returned with the correct arguments.

An invalid model state is tested by adding errors using AddModelError as shown in the first test below:

```C#
[Fact]
public async Task
IndexPost_ReturnsBadRequestResult_WhenModelStateIsInvalid()
{
    // Arrange
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.ListAsync())
        .ReturnsAsync(GetTestSessions());
    var controller = new HomeController(mockRepo.Object);
    controller.ModelState.AddModelError("SessionName", "Required");
    var newSession = new HomeController.NewSessionModel();

    // Act
    var result = await controller.Index(newSession);

    // Assert
    var badRequestResult = Assert.IsType<BadRequestObjectResult>(result);
    Assert.IsType<SerializableError>(badRequestResult.Value);
}

[Fact]
public async Task
IndexPost_ReturnsARedirectAndAddsSession_WhenModelStateIsValid()
{
    // Arrange
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.AddAsync(It.IsAny<BrainstormSession>()))
        .Returns(Task.CompletedTask)
        .Verifiable();
    var controller = new HomeController(mockRepo.Object);
    var newSession = new HomeController.NewSessionModel()
    {
        SessionName = "Test Name"
    };

    // Act
    var result = await controller.Index(newSession);
```

```
        // Assert
        var redirectToActionResult = Assert.IsType<RedirectToActionResult>
    (result);
        Assert.Null(redirectToActionResult.ControllerName);
        Assert.Equal("Index", redirectToActionResult.ActionName);
        mockRepo.Verify();
    }
```

When ModelState isn't valid, the same `ViewResult` is returned as for a GET request. The test doesn't attempt to pass in an invalid model. Passing an invalid model isn't a valid approach, since model binding isn't running (although an integration test does use model binding). In this case, model binding isn't tested. These unit tests are only testing the code in the action method.

The second test verifies that when the `ModelState` is valid:

- A new `BrainstormSession` is added (via the repository).
- The method returns a `RedirectToActionResult` with the expected properties.

Mocked calls that aren't called are normally ignored, but calling `Verifiable` at the end of the setup call allows mock validation in the test. This is performed with the call to `mockRepo.Verify`, which fails the test if the expected method wasn't called.

> ⓘ **Note**
>
> The Moq library used in this sample makes it possible to mix verifiable, or "strict", mocks with non-verifiable mocks (also called "loose" mocks or stubs). Learn more about customizing Mock behavior with Moq.

SessionController in the sample app displays information related to a particular brainstorming session. The controller includes logic to deal with invalid `id` values (there are two `return` scenarios in the following example to cover these scenarios). The final `return` statement returns a new `StormSessionViewModel` to the view (`Controllers/SessionController.cs`):

```C#
public class SessionController : Controller
{
    private readonly IBrainstormSessionRepository _sessionRepository;

    public SessionController(IBrainstormSessionRepository sessionRepository)
    {
        _sessionRepository = sessionRepository;
    }
    public async Task<IActionResult> Index(int? id)
    {
        if (!id.HasValue)
        {
            return RedirectToAction(actionName: nameof(Index),
                controllerName: "Home");
        }

        var session = await _sessionRepository.GetByIdAsync(id.Value);
        if (session == null)
        {
            return Content("Session not found.");
        }

        var viewModel = new StormSessionViewModel()
        {
            DateCreated = session.DateCreated,
            Name = session.Name,
            Id = session.Id
        };

        return View(viewModel);
    }
}
```

The unit tests include one test for each `return` scenario in the Session controller `Index` action:

C#

```
[Fact]
public async Task IndexReturnsARedirectToIndexHomeWhenIdIsNull()
{
    // Arrange
    var controller = new SessionController(sessionRepository: null);

    // Act
    var result = await controller.Index(id: null);

    // Assert
    var redirectToActionResult =
        Assert.IsType<RedirectToActionResult>(result);
    Assert.Equal("Home", redirectToActionResult.ControllerName);
    Assert.Equal("Index", redirectToActionResult.ActionName);
}

[Fact]
public async Task
IndexReturnsContentWithSessionNotFoundWhenSessionNotFound()
{
    // Arrange
    int testSessionId = 1;
```

```
        var mockRepo = new Mock<IBrainstormSessionRepository>();
        mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
            .ReturnsAsync((BrainstormSession)null);
        var controller = new SessionController(mockRepo.Object);

        // Act
        var result = await controller.Index(testSessionId);

        // Assert
        var contentResult = Assert.IsType<ContentResult>(result);
        Assert.Equal("Session not found.", contentResult.Content);
    }

    [Fact]
    public async Task IndexReturnsViewResultWithStormSessionViewModel()
    {
        // Arrange
        int testSessionId = 1;
        var mockRepo = new Mock<IBrainstormSessionRepository>();
        mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
            .ReturnsAsync(GetTestSessions().FirstOrDefault(
                s => s.Id == testSessionId));
        var controller = new SessionController(mockRepo.Object);

        // Act
        var result = await controller.Index(testSessionId);

        // Assert
        var viewResult = Assert.IsType<ViewResult>(result);
        var model = Assert.IsType<StormSessionViewModel>(
            viewResult.ViewData.Model);
        Assert.Equal("Test One", model.Name);
        Assert.Equal(2, model.DateCreated.Day);
        Assert.Equal(testSessionId, model.Id);
    }
```

Moving to the Ideas controller, the app exposes functionality as a web API on the `api/ideas` route:

- A list of ideas (`IdeaDTO`) associated with a brainstorming session is returned by the `ForSession` method.
- The `Create` method adds new ideas to a session.

C#

```
    [HttpGet("forsession/{sessionId}")]
    public async Task<IActionResult> ForSession(int sessionId)
    {
        var session = await _sessionRepository.GetByIdAsync(sessionId);
        if (session == null)
        {
            return NotFound(sessionId);
        }

        var result = session.Ideas.Select(idea => new IdeaDTO()
        {
            Id = idea.Id,
            Name = idea.Name,
            Description = idea.Description,
            DateCreated = idea.DateCreated
        }).ToList();

        return Ok(result);
    }

    [HttpPost("create")]
    public async Task<IActionResult> Create([FromBody]NewIdeaModel model)
    {
        if (!ModelState.IsValid)
        {
            return BadRequest(ModelState);
        }

        var session = await _sessionRepository.GetByIdAsync(model.SessionId);
        if (session == null)
        {
            return NotFound(model.SessionId);
        }

        var idea = new Idea()
        {
            DateCreated = DateTimeOffset.Now,
            Description = model.Description,
            Name = model.Name
        };
        session.AddIdea(idea);

        await _sessionRepository.UpdateAsync(session);

        return Ok(session);
    }
```

Avoid returning business domain entities directly via API calls. Domain entities:

- Often include more data than the client requires.
- Unnecessarily couple the app's internal domain model with the publicly exposed API.

Mapping between domain entities and the types returned to the client can be performed:

- Manually with a LINQ `Select`, as the sample app uses. For more information, see LINQ (Language Integrated Query).

- Automatically with a library, such as AutoMapper.

Next, the sample app demonstrates unit tests for the `Create` and `ForSession` API methods of the Ideas controller.

The sample app contains two `ForSession` tests. The first test determines if `ForSession` returns a NotFoundObjectResult (HTTP Not Found) for an invalid session:

C#

```
[Fact]
public async Task ForSession_ReturnsHttpNotFound_ForInvalidSession()
{
    // Arrange
    int testSessionId = 123;
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync((BrainstormSession)null);
    var controller = new IdeasController(mockRepo.Object);

    // Act
    var result = await controller.ForSession(testSessionId);

    // Assert
    var notFoundObjectResult = Assert.IsType<NotFoundObjectResult>(result);
    Assert.Equal(testSessionId, notFoundObjectResult.Value);
}
```

The second `ForSession` test determines if `ForSession` returns a list of session ideas (`<List<IdeaDTO>>`) for a valid session. The checks also examine the first idea to confirm its `Name` property is correct:

C#

```
[Fact]
public async Task ForSession_ReturnsIdeasForSession()
{
    // Arrange
    int testSessionId = 123;
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync(GetTestSession());
    var controller = new IdeasController(mockRepo.Object);

    // Act
    var result = await controller.ForSession(testSessionId);

    // Assert
    var okResult = Assert.IsType<OkObjectResult>(result);
    var returnValue = Assert.IsType<List<IdeaDTO>>(okResult.Value);
    var idea = returnValue.FirstOrDefault();
    Assert.Equal("One", idea.Name);
}
```

To test the behavior of the `Create` method when the `ModelState` is invalid, the sample app adds a model error to the controller as part of the test. Don't try to test model validation or model binding in unit tests—just test the action method's behavior when confronted with an invalid `ModelState`:

C#

```
[Fact]
public async Task Create_ReturnsBadRequest_GivenInvalidModel()
{
    // Arrange & Act
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    var controller = new IdeasController(mockRepo.Object);
    controller.ModelState.AddModelError("error", "some error");

    // Act
    var result = await controller.Create(model: null);

    // Assert
    Assert.IsType<BadRequestObjectResult>(result);
}
```

The second test of `Create` depends on the repository returning `null`, so the mock repository is configured to return `null`. There's no need to create a test database (in memory or otherwise) and construct a query that returns this result. The test can be accomplished in a single statement, as the sample code illustrates:

C#

```
[Fact]
public async Task Create_ReturnsHttpNotFound_ForInvalidSession()
{
    // Arrange
    int testSessionId = 123;
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync((BrainstormSession)null);
    var controller = new IdeasController(mockRepo.Object);

    // Act
    var result = await controller.Create(new NewIdeaModel());

    // Assert
    Assert.IsType<NotFoundObjectResult>(result);
}
```

The third `Create` test, `Create_ReturnsNewlyCreatedIdeaForSession`, verifies that the repository's `UpdateAsync` method is called. The mock is called with `Verifiable`, and the mocked repository's `Verify` method is called to confirm the verifiable method is executed. It's not the unit test's responsibility to ensure that the `UpdateAsync` method saved the data—that can be performed with an integration test.

```C#
[Fact]
public async Task Create_ReturnsNewlyCreatedIdeaForSession()
{
    // Arrange
    int testSessionId = 123;
    string testName = "test name";
    string testDescription = "test description";
    var testSession = GetTestSession();
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync(testSession);
    var controller = new IdeasController(mockRepo.Object);

    var newIdea = new NewIdeaModel()
    {
        Description = testDescription,
        Name = testName,
        SessionId = testSessionId
    };
    mockRepo.Setup(repo => repo.UpdateAsync(testSession))
        .Returns(Task.CompletedTask)
        .Verifiable();

    // Act
    var result = await controller.Create(newIdea);

    // Assert
    var okResult = Assert.IsType<OkObjectResult>(result);
    var returnSession = Assert.IsType<BrainstormSession>(okResult.Value);
    mockRepo.Verify();
    Assert.Equal(2, returnSession.Ideas.Count());
    Assert.Equal(testName, returnSession.Ideas.LastOrDefault().Name);
    Assert.Equal(testDescription,
returnSession.Ideas.LastOrDefault().Description);
}
```

Test ActionResult<T>

ActionResult<T> (ActionResult<TValue>) can return a type deriving from `ActionResult` or return a specific type.

The sample app includes a method that returns a `List<IdeaDTO>` for a given session `id`. If the session `id` doesn't exist, the controller returns NotFound:

C#

```csharp
[HttpGet("forsessionactionresult/{sessionId}")]
[ProducesResponseType(200)]
[ProducesResponseType(404)]
public async Task<ActionResult<List<IdeaDTO>>> ForSessionActionResult(int
sessionId)
{
    var session = await _sessionRepository.GetByIdAsync(sessionId);

    if (session == null)
    {
        return NotFound(sessionId);
    }

    var result = session.Ideas.Select(idea => new IdeaDTO()
    {
        Id = idea.Id,
        Name = idea.Name,
        Description = idea.Description,
        DateCreated = idea.DateCreated
    }).ToList();

    return result;
}
```

Two tests of the `ForSessionActionResult` controller are included in the `ApiIdeasControllerTests`.

The first test confirms that the controller returns an `ActionResult` but not a nonexistent list of ideas for a nonexistent session `id`:

- The `ActionResult` type is `ActionResult<List<IdeaDTO>>`.
- The Result is a NotFoundObjectResult.

C#

```csharp
[Fact]
public async Task
ForSessionActionResult_ReturnsNotFoundObjectResultForNonexistentSession()
{
    // Arrange
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    var controller = new IdeasController(mockRepo.Object);
    var nonExistentSessionId = 999;

    // Act
    var result = await
controller.ForSessionActionResult(nonExistentSessionId);

    // Assert
    var actionResult = Assert.IsType<ActionResult<List<IdeaDTO>>>(result);
    Assert.IsType<NotFoundObjectResult>(actionResult.Result);
}
```

For a valid session `id`, the second test confirms that the method returns:

- An `ActionResult` with a `List<IdeaDTO>` type.
- The ActionResult<T>.Value is a `List<IdeaDTO>` type.
- The first item in the list is a valid idea matching the idea stored in the mock session (obtained by calling `GetTestSession`).

C#

```
[Fact]
public async Task ForSessionActionResult_ReturnsIdeasForSession()
{
    // Arrange
    int testSessionId = 123;
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync(GetTestSession());
    var controller = new IdeasController(mockRepo.Object);

    // Act
    var result = await controller.ForSessionActionResult(testSessionId);

    // Assert
    var actionResult = Assert.IsType<ActionResult<List<IdeaDTO>>>(result);
    var returnValue = Assert.IsType<List<IdeaDTO>>(actionResult.Value);
    var idea = returnValue.FirstOrDefault();
    Assert.Equal("One", idea.Name);
}
```

The sample app also includes a method to create a new `Idea` for a given session. The controller returns:

- BadRequest for an invalid model.
- NotFound if the session doesn't exist.
- CreatedAtAction when the session is updated with the new idea.

C#

```
[HttpPost("createactionresult")]
[ProducesResponseType(201)]
[ProducesResponseType(400)]
[ProducesResponseType(404)]
public async Task<ActionResult<BrainstormSession>>
CreateActionResult([FromBody]NewIdeaModel model)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }

    var session = await _sessionRepository.GetByIdAsync(model.SessionId);

    if (session == null)
    {
        return NotFound(model.SessionId);
    }

    var idea = new Idea()
    {
        DateCreated = DateTimeOffset.Now,
        Description = model.Description,
        Name = model.Name
    };
    session.AddIdea(idea);

    await _sessionRepository.UpdateAsync(session);

    return CreatedAtAction(nameof(CreateActionResult), new { id = session.Id
}, session);
}
```

Three tests of `CreateActionResult` are included in the `ApiIdeasControllerTests`.

The first text confirms that a BadRequest is returned for an invalid model.

```C#
[Fact]
public async Task CreateActionResult_ReturnsBadRequest_GivenInvalidModel()
{
    // Arrange & Act
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    var controller = new IdeasController(mockRepo.Object);
    controller.ModelState.AddModelError("error", "some error");

    // Act
    var result = await controller.CreateActionResult(model: null);

    // Assert
    var actionResult = Assert.IsType<ActionResult<BrainstormSession>>
(result);
    Assert.IsType<BadRequestObjectResult>(actionResult.Result);
}
```

The second test checks that a NotFound is returned if the session doesn't exist.

```C#
[Fact]
public async Task
CreateActionResult_ReturnsNotFoundObjectResultForNonexistentSession()
{
    // Arrange
    var nonExistentSessionId = 999;
    string testName = "test name";
    string testDescription = "test description";
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    var controller = new IdeasController(mockRepo.Object);

    var newIdea = new NewIdeaModel()
    {
        Description = testDescription,
        Name = testName,
        SessionId = nonExistentSessionId
    };

    // Act
    var result = await controller.CreateActionResult(newIdea);

    // Assert
    var actionResult = Assert.IsType<ActionResult<BrainstormSession>>
(result);
    Assert.IsType<NotFoundObjectResult>(actionResult.Result);
}
```

For a valid session `id`, the final test confirms that:

- The method returns an `ActionResult` with a `BrainstormSession` type.
- The ActionResult<T>.Result is a CreatedAtActionResult. `CreatedAtActionResult` is analogous to a *201 Created* response with a `Location` header.
- The ActionResult<T>.Value is a `BrainstormSession` type.
- The mock call to update the session, `UpdateAsync(testSession)`, was invoked. The `Verifiable` method call is checked by executing `mockRepo.Verify()` in the assertions.
- Two `Idea` objects are returned for the session.
- The last item (the `Idea` added by the mock call to `UpdateAsync`) matches the `newIdea` added to the session in the test.

C#

```csharp
[Fact]
public async Task CreateActionResult_ReturnsNewlyCreatedIdeaForSession()
{
    // Arrange
    int testSessionId = 123;
    string testName = "test name";
    string testDescription = "test description";
    var testSession = GetTestSession();
    var mockRepo = new Mock<IBrainstormSessionRepository>();
    mockRepo.Setup(repo => repo.GetByIdAsync(testSessionId))
        .ReturnsAsync(testSession);
    var controller = new IdeasController(mockRepo.Object);

    var newIdea = new NewIdeaModel()
    {
        Description = testDescription,
        Name = testName,
        SessionId = testSessionId
    };
    mockRepo.Setup(repo => repo.UpdateAsync(testSession))
        .Returns(Task.CompletedTask)
        .Verifiable();

    // Act
    var result = await controller.CreateActionResult(newIdea);

    // Assert
    var actionResult = Assert.IsType<ActionResult<BrainstormSession>>
(result);
    var createdAtActionResult = Assert.IsType<CreatedAtActionResult>
(actionResult.Result);
    var returnValue = Assert.IsType<BrainstormSession>
(createdAtActionResult.Value);
    mockRepo.Verify();
    Assert.Equal(2, returnValue.Ideas.Count());
    Assert.Equal(testName, returnValue.Ideas.LastOrDefault().Name);
    Assert.Equal(testDescription,
returnValue.Ideas.LastOrDefault().Description);
}
```

Additional resources

- Integration tests in ASP.NET Core
- Create and run unit tests with Visual Studio
- MyTested.AspNetCore.Mvc - Fluent Testing Library for ASP.NET Core MVC: Strongly-typed unit testing library, providing a fluent interface for testing MVC and web API apps. (*Not maintained or supported by Microsoft.*)

- JustMockLite : A mocking framework for .NET developers. (*Not maintained or supported by Microsoft.*)

ASP.NET Core Blazor

Article • 09/16/2024

> **Important**
>
> This information relates to a pre-release product that may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.
>
> For the current release, see the .NET 9 version of this article.

Welcome to Blazor!

Blazor is a .NET frontend web framework that supports both server-side rendering and client interactivity in a single programming model:

- Create rich interactive UIs using C#.
- Share server-side and client-side app logic written in .NET.
- Render the UI as HTML and CSS for wide browser support, including mobile browsers.
- Build hybrid desktop and mobile apps with .NET and Blazor.

Using .NET for client-side web development offers the following advantages:

- Write code in C#, which can improve productivity in app development and maintenance.
- Leverage the existing .NET ecosystem of .NET libraries.
- Benefit from .NET's performance, reliability, and security.
- Stay productive on Windows, Linux, or macOS with a development environment, such as Visual Studio or Visual Studio Code. Integrate with modern hosting platforms, such as Docker.
- Build on a common set of languages, frameworks, and tools that are stable, feature-rich, and easy to use.

> **Note**
>
> For a Blazor quick start tutorial, see Build your first Blazor app.

Components