Get started with ASP.NET Core MVC

This article is taken from the Microsoft Documentation for ASP.NET Core MVC
https://learn.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/start-mvc?view=aspnetcore-8.0&tabs=visual-studio

This tutorial teaches ASP.NET Core MVC web development with controllers and views. If you're new to ASP.NET Core web development, consider the Razor Pages version of this tutorial, which provides an easier starting point. See Choose an ASP.NET Core UI, which compares Razor Pages, MVC, and Blazor for UI development.

This is the first tutorial of a series that teaches ASP.NET Core MVC web development with controllers and views.

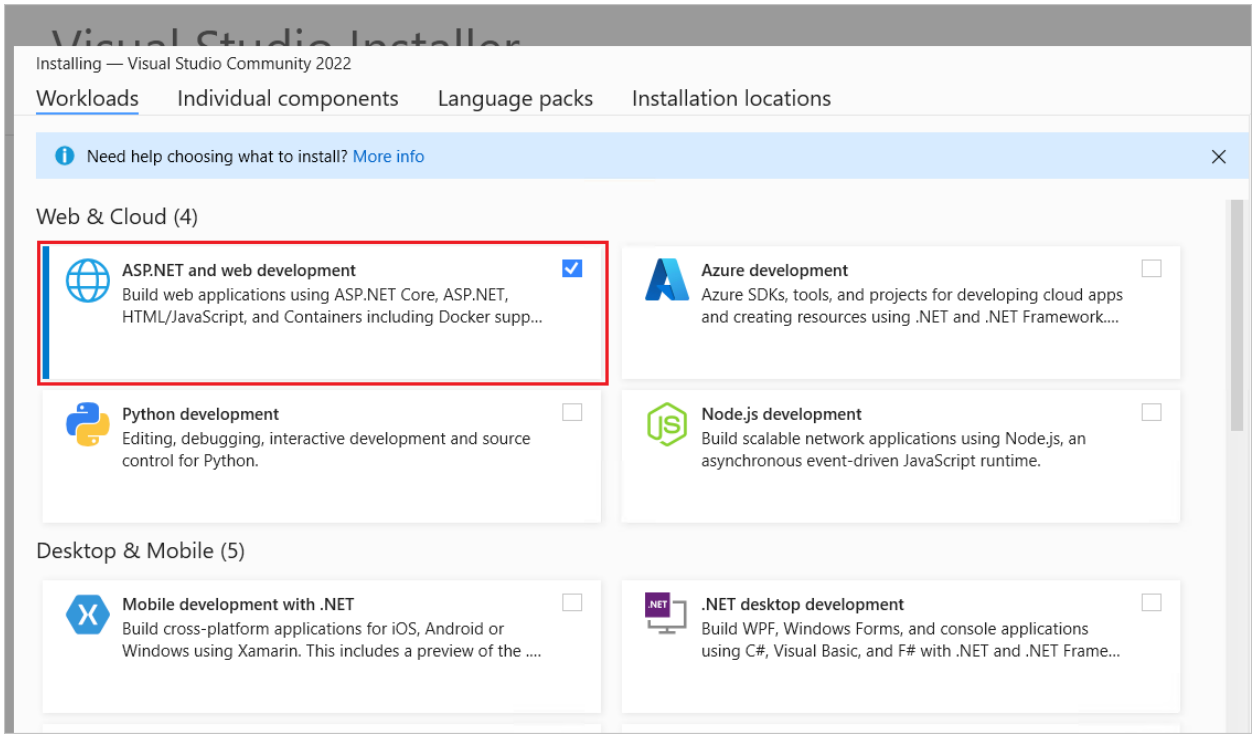
At the end of the series, you'll have an app that manages and displays movie data. You learn how to:

- Create a web app.
- Add and scaffold a model.
- Work with a database.
- Add search and validation.

View or download sample code (how to download).

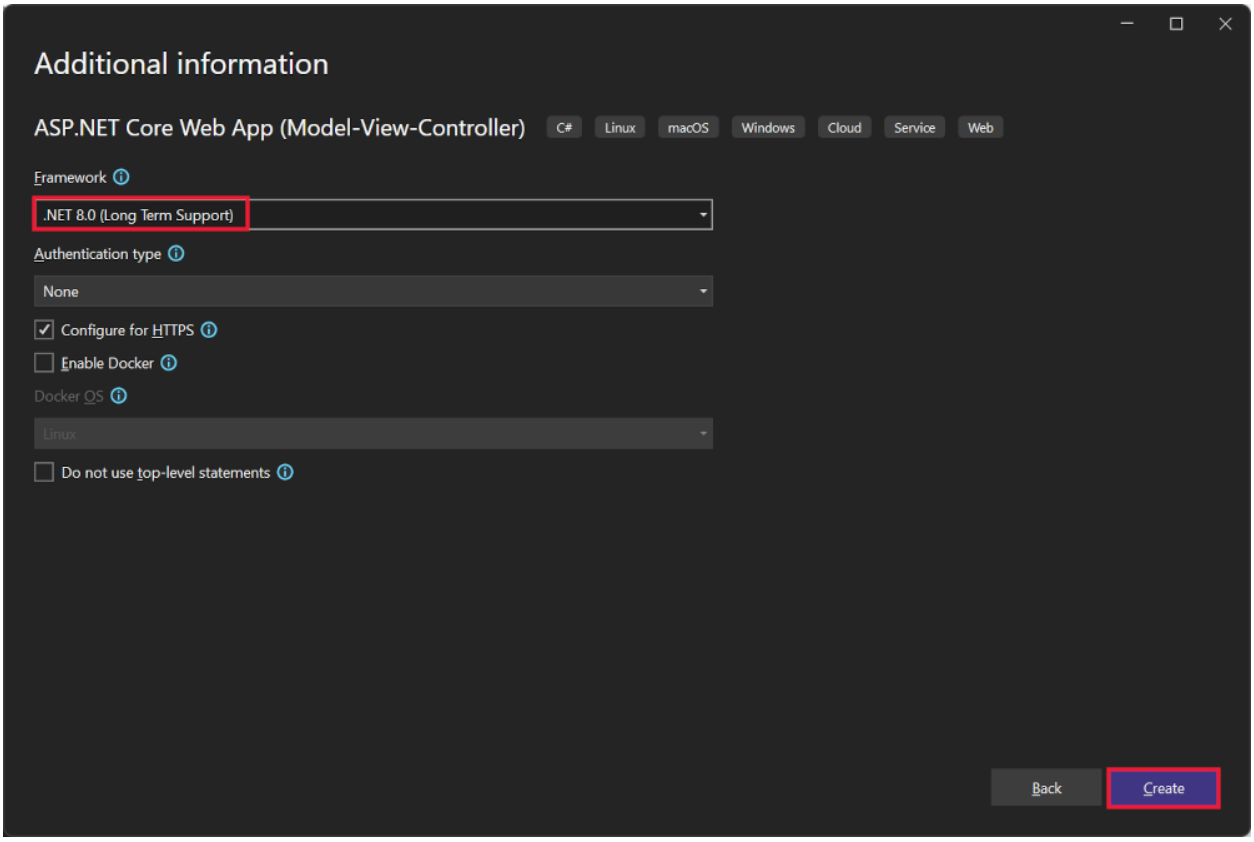
Prerequisites

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac
    - Visual Studio 2022 Preview with the **ASP.NET and web development** workload.



Create a web app

- Visual Studio
- Visual Studio Code
- Visual Studio for Mac
    - Start Visual Studio and select **Create a new project**.
    - In the **Create a new project** dialog, select **ASP.NET Core Web App (Model-View-Controller)** > **Next**.
    - In the **Configure your new project** dialog, enter `MvcMovie` for **Project name**. It's important to name the project *MvcMovie*. Capitalization needs to match each `namespace` when code is copied.
    - Select **Next**.
    - In the **Additional information** dialog:
        - Select **.NET 8.0 (Long Term Support)**.
        - Verify that **Do not use top-level statements** is unchecked.
    - Select **Create**.



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:

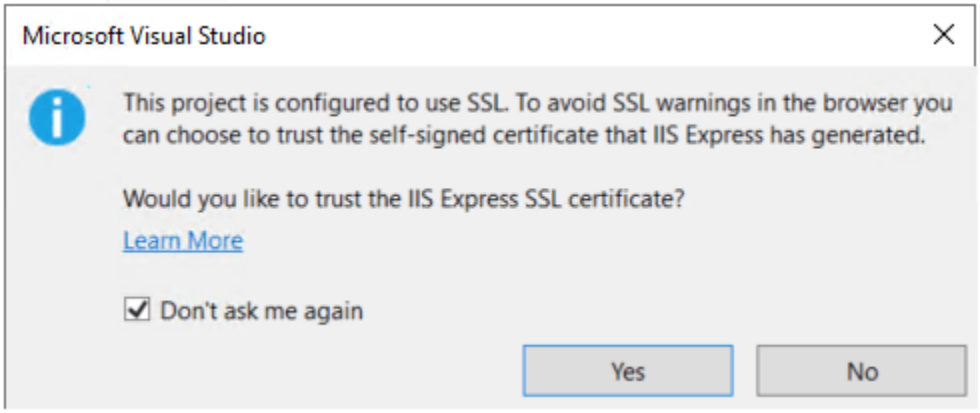
- Is a working app.
- Is a basic starter project.

Run the app

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

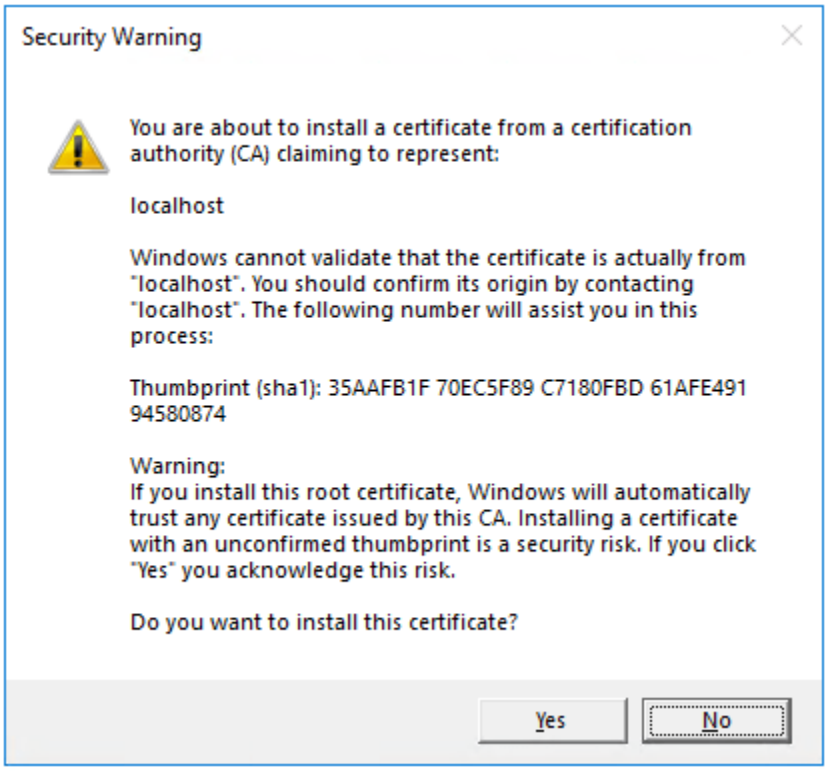
  - Select Ctrl+F5 to run the app without the debugger.

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



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

The following dialog is displayed:



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.

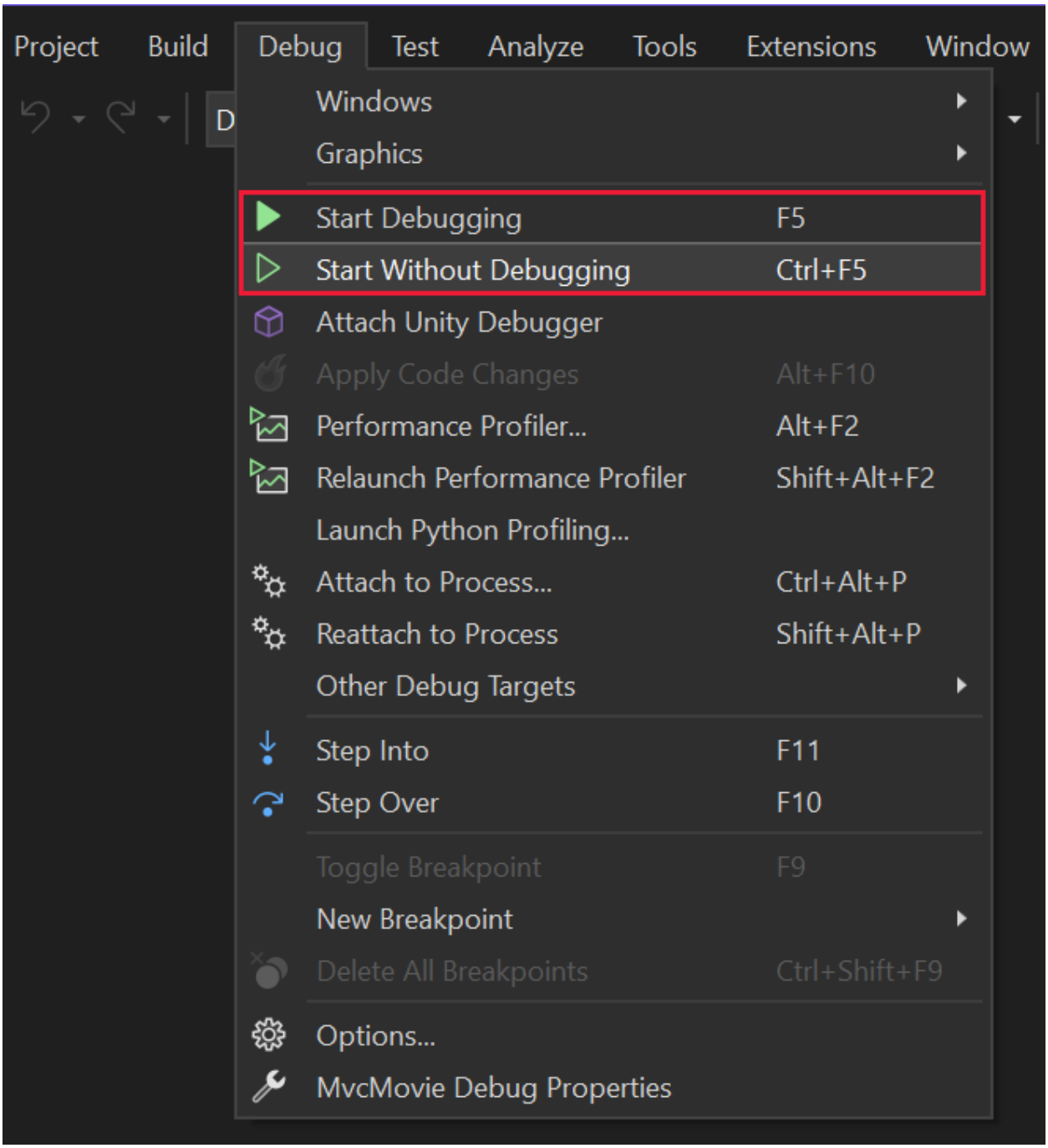
Visual Studio runs the app and opens the default browser.

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.

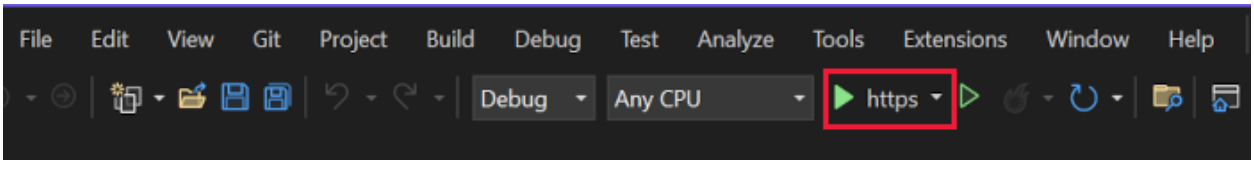
Launching the app without debugging by selecting Ctrl+F5 allows you to:

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

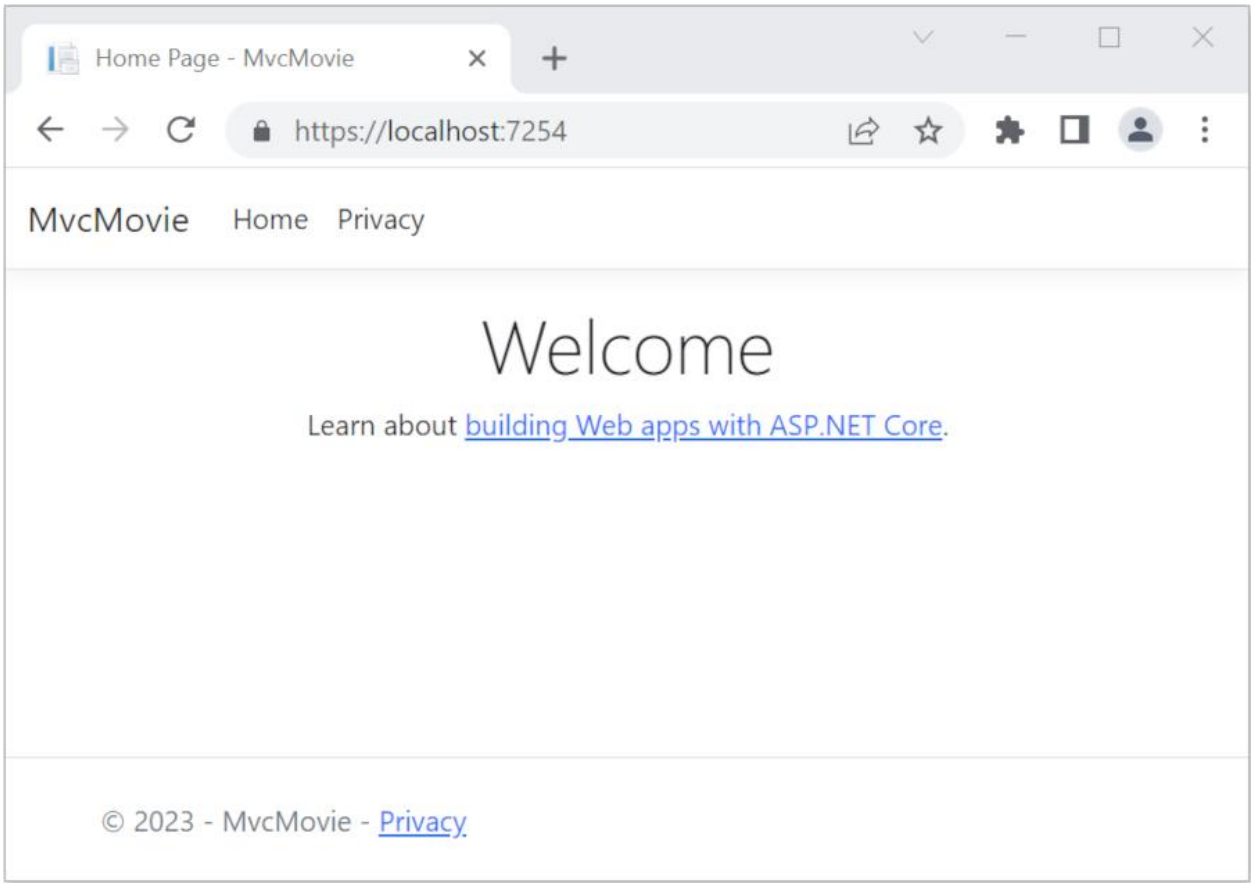
You can launch the app in debug or non-debug mode from the **Debug** menu:



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



The following image shows the app:



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

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.

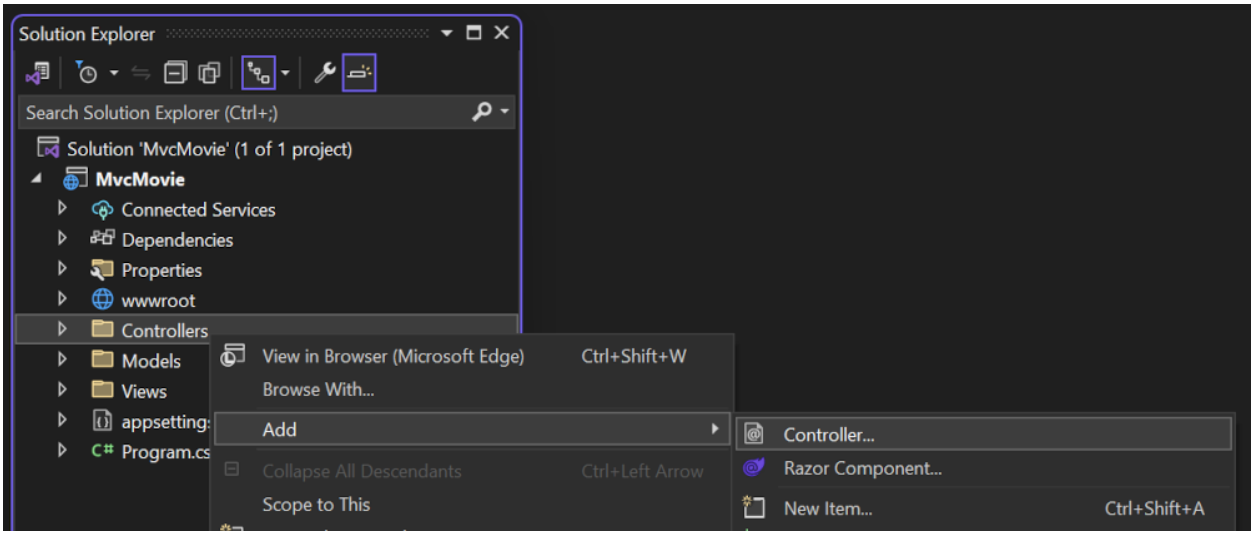
The MVC architectural pattern separates an app into three main groups of components: Models, Views, and Controllers. This pattern helps to achieve separation of concerns: The UI logic belongs in the view. Input logic belongs in the controller. Business logic belongs in the model. This separation helps 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*.

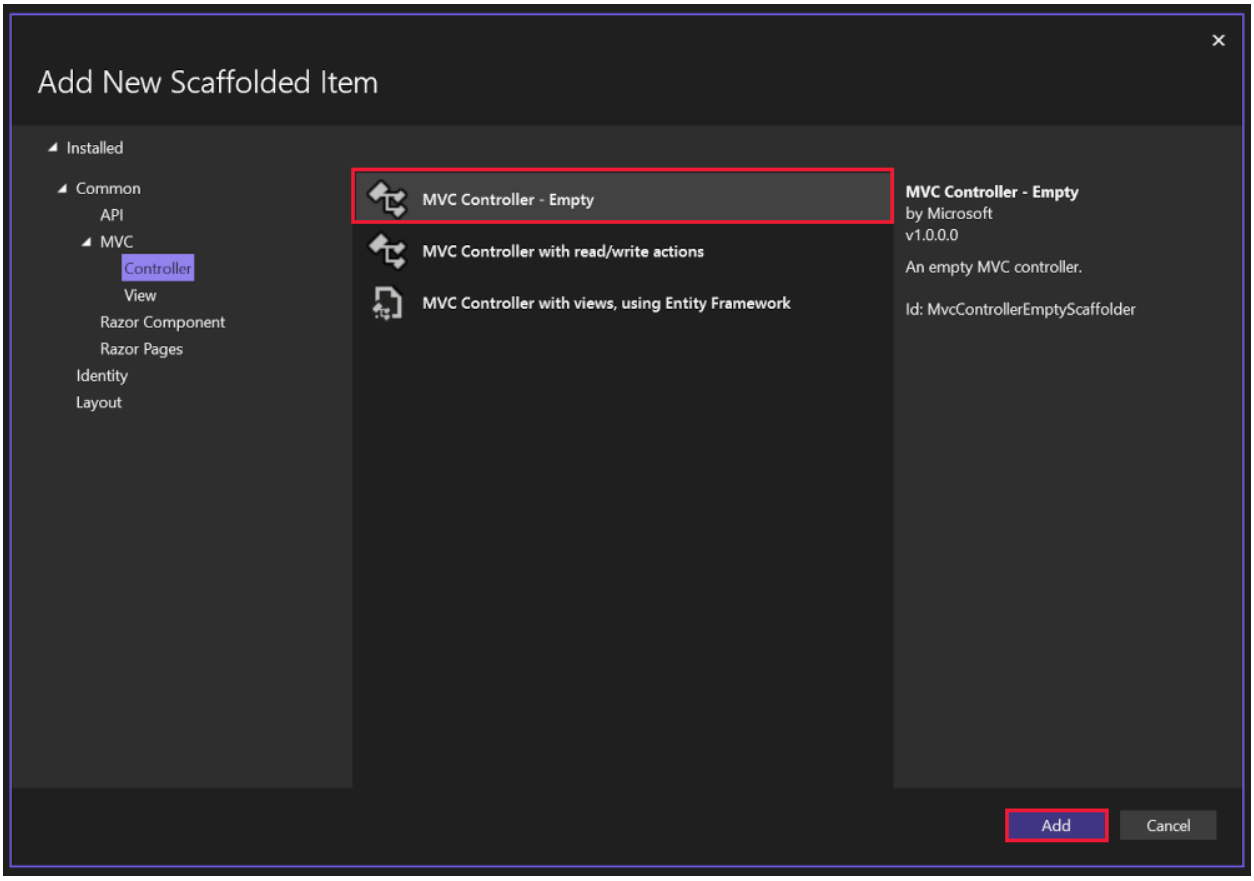
Add a controller

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

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



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



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#Copy

```
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:

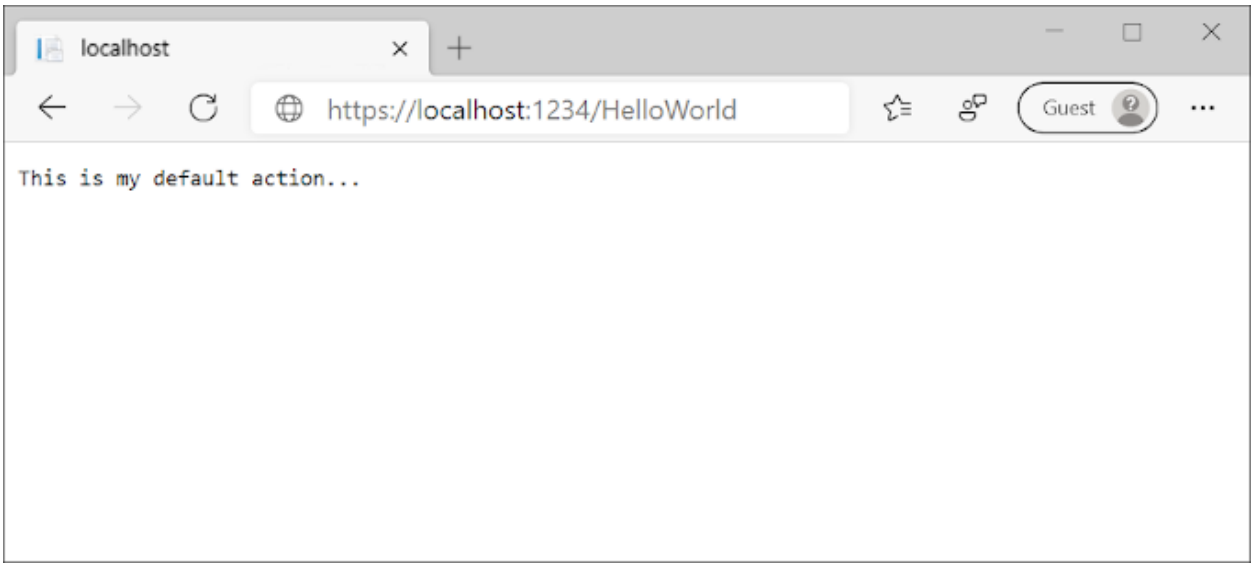
- Is a targetable URL in the web application, such as `https://localhost:5001/HelloWorld`.
- Combines:
    - The protocol used: `HTTPS`.
    - The network location of the web server, including the TCP port: `localhost:5001`.
    - The target URI: `HelloWorld`.

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.

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



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#Copy

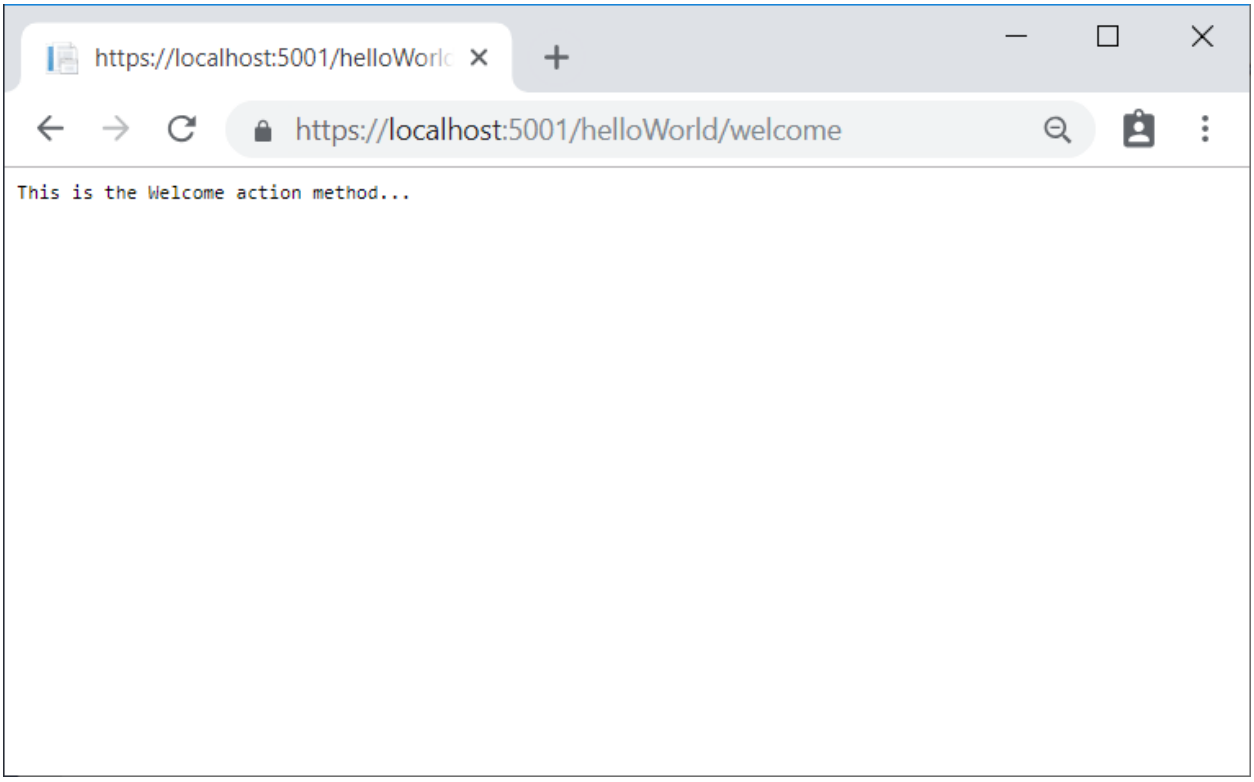
```
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
```

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.



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#Copy

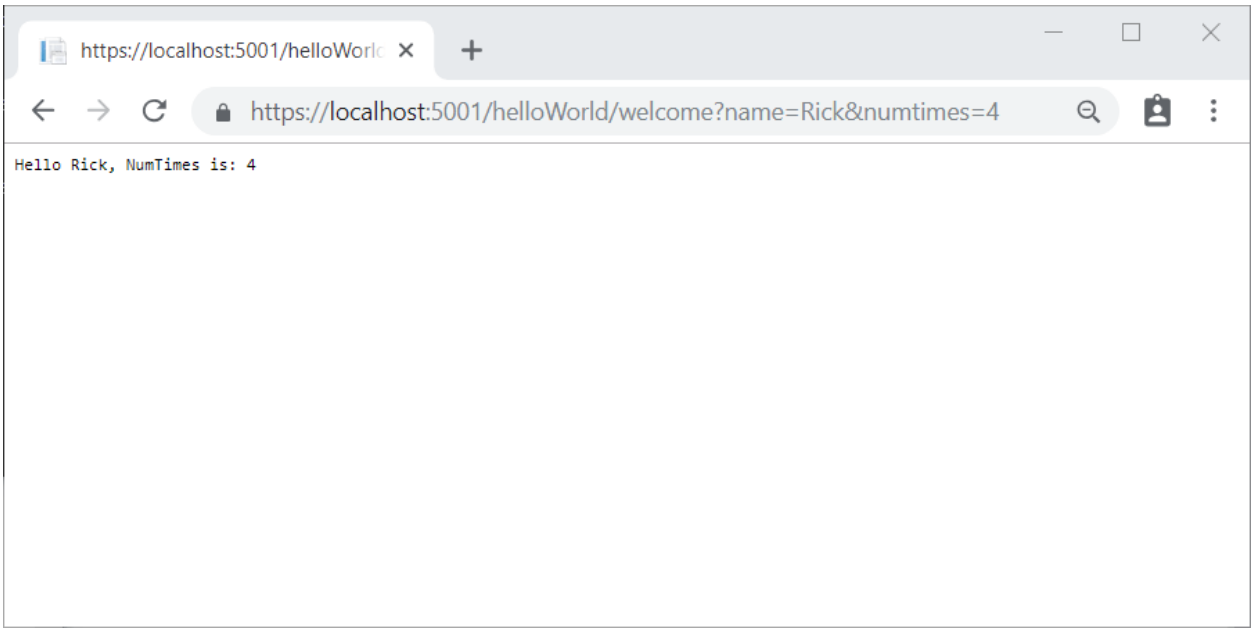
```
// GET: /HelloWorld/Welcome/
// Requires using System.Text.Encodings.Web;
public string Welcome(string name, int numTimes = 1)
{
    return HtmlEncoder.Default.Encode($"Hello {name}, NumTimes is: {numTimes}");
}
```

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.



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#Copy

```
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#Copy

```
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.

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

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#Copy

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

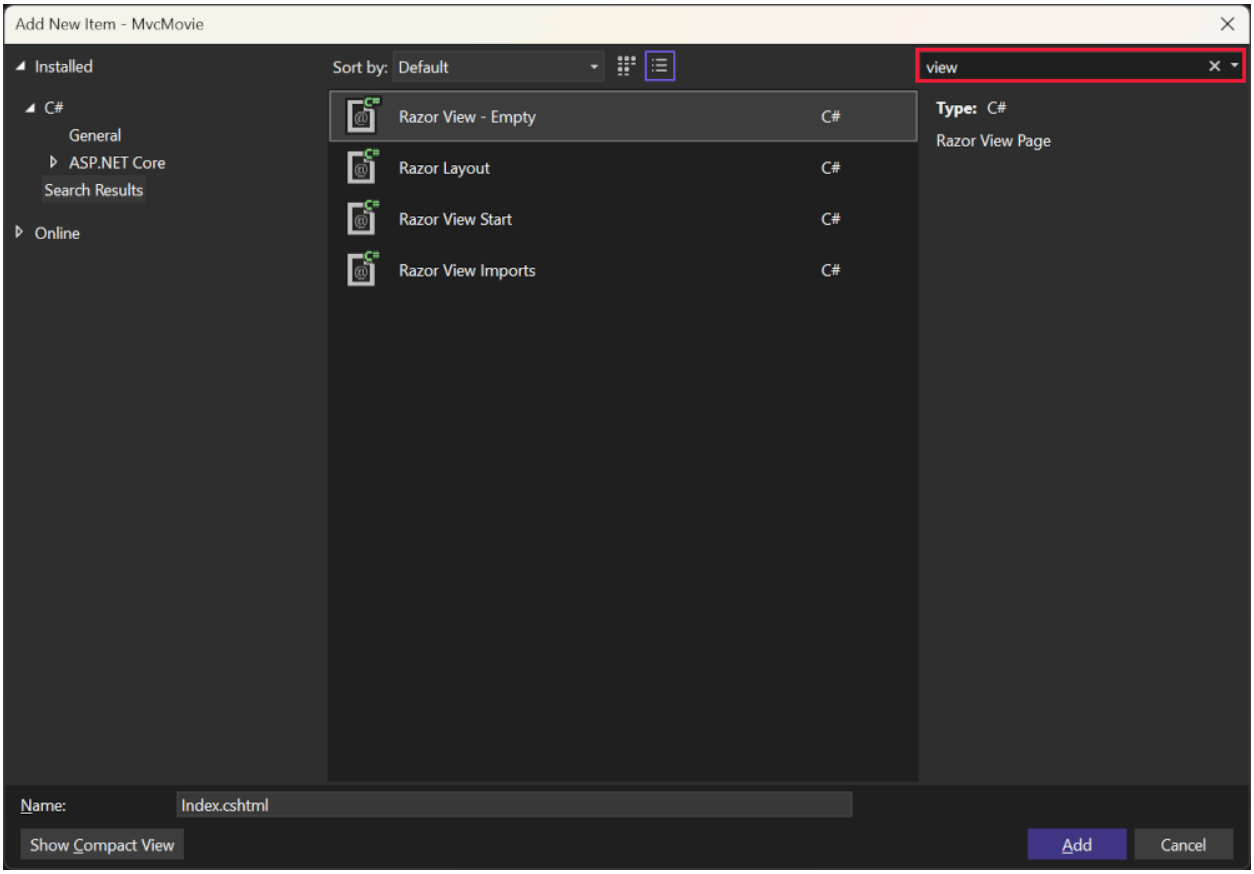
Right-click on the *Views* folder, and then **Add > New Folder** and name the folder *HelloWorld*.

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**



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

CSHTMLCopy

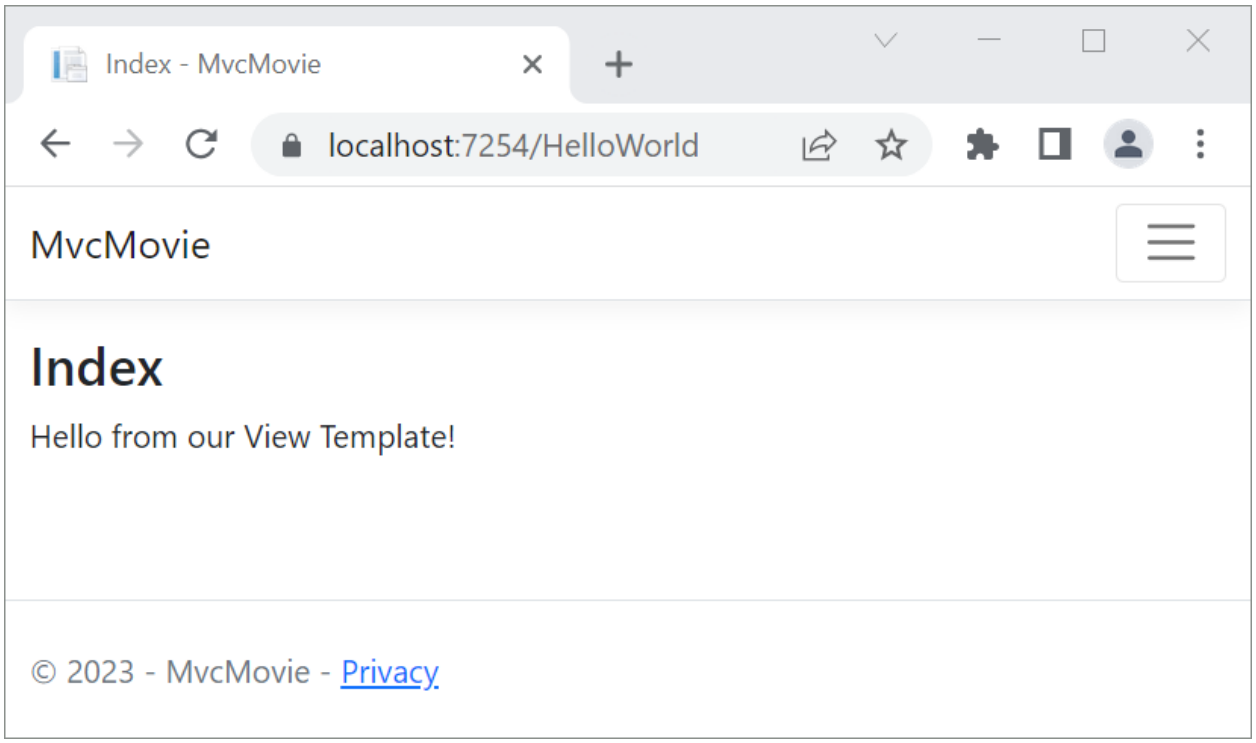
```
@{
    ViewData["Title"] = "Index";
}

<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:



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:

CSHTMLCopy

```
<!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" />
</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; 2023 - 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>
```

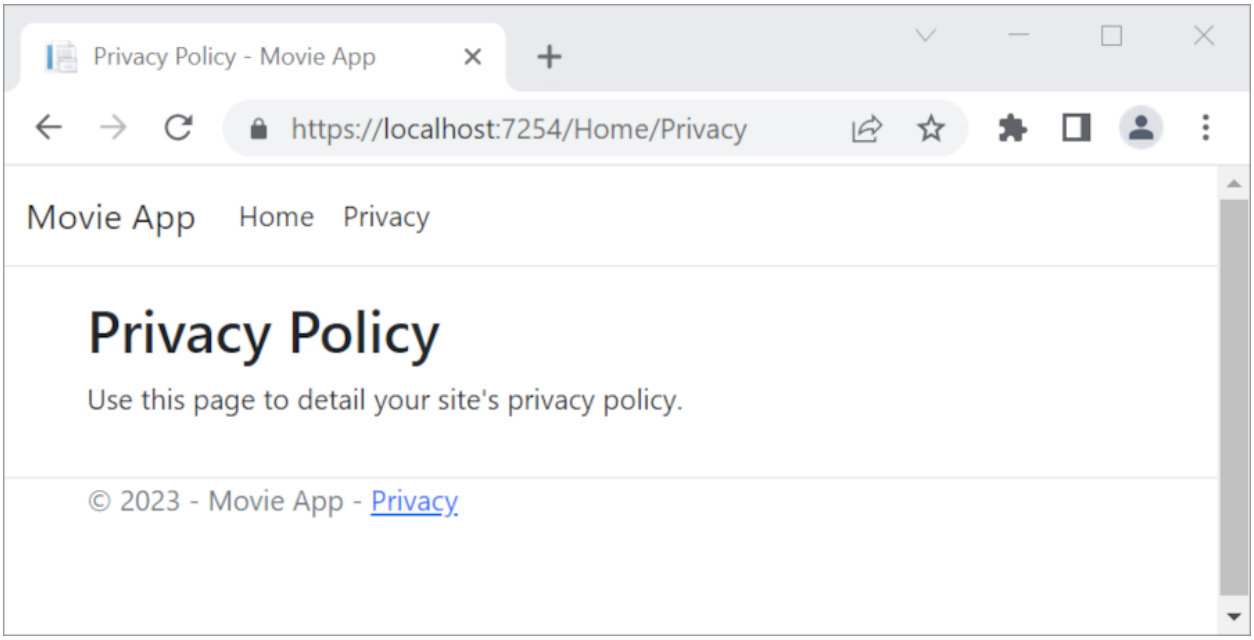
The preceding markup made the following changes:

- 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**



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:

CSHTMLCopy

```
@{
    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:

CSHTMLCopy

```
@{
    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:

CSHTMLCopy

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

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

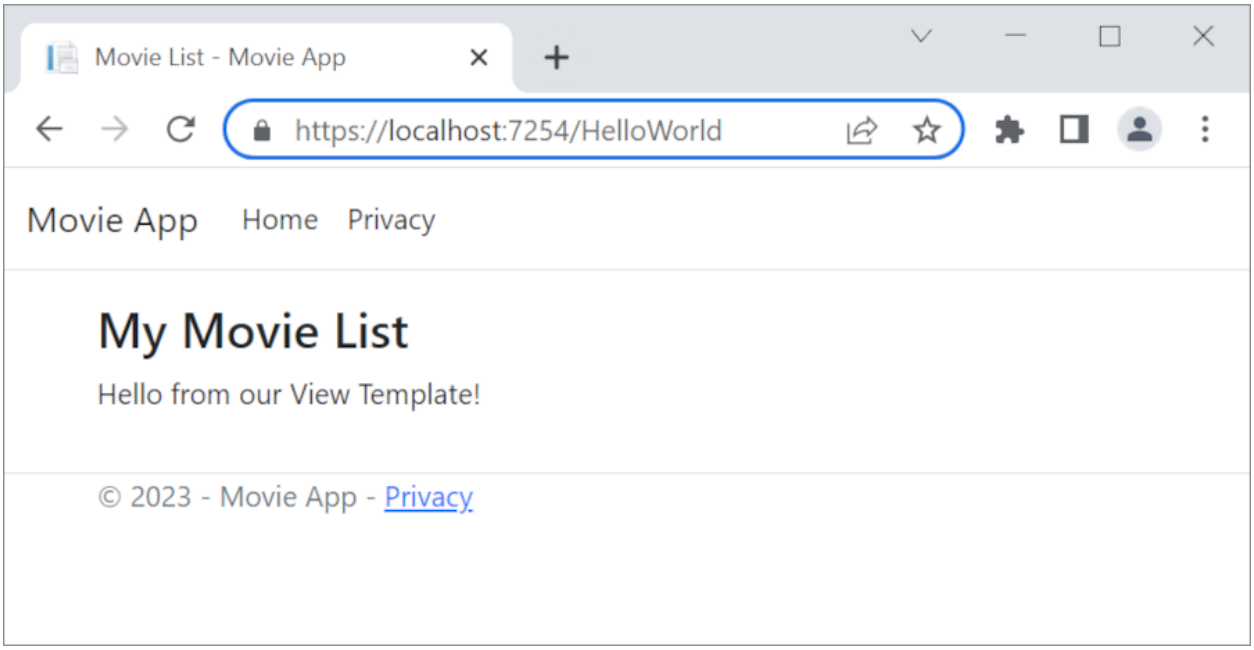
Notice that the following have changed:

- 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.



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#Copy

```
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:

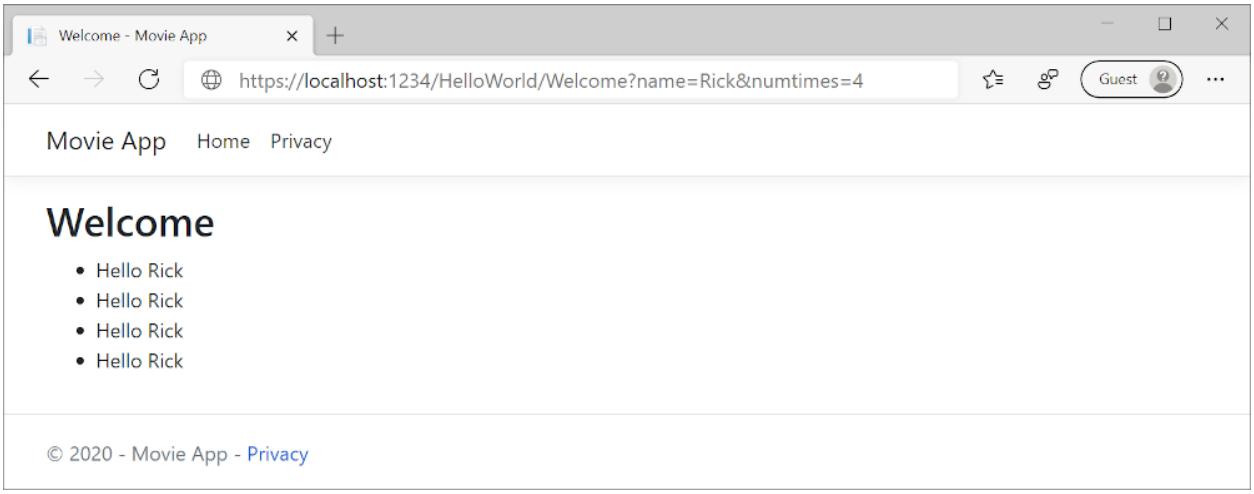
CSHTMLCopy

```
@{
    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.



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.

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

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

- Visual Studio Code
- Visual Studio for Mac

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

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

C#Copy

```
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 Code
- Visual Studio for Mac

Visual Studio automatically installs the required packages.

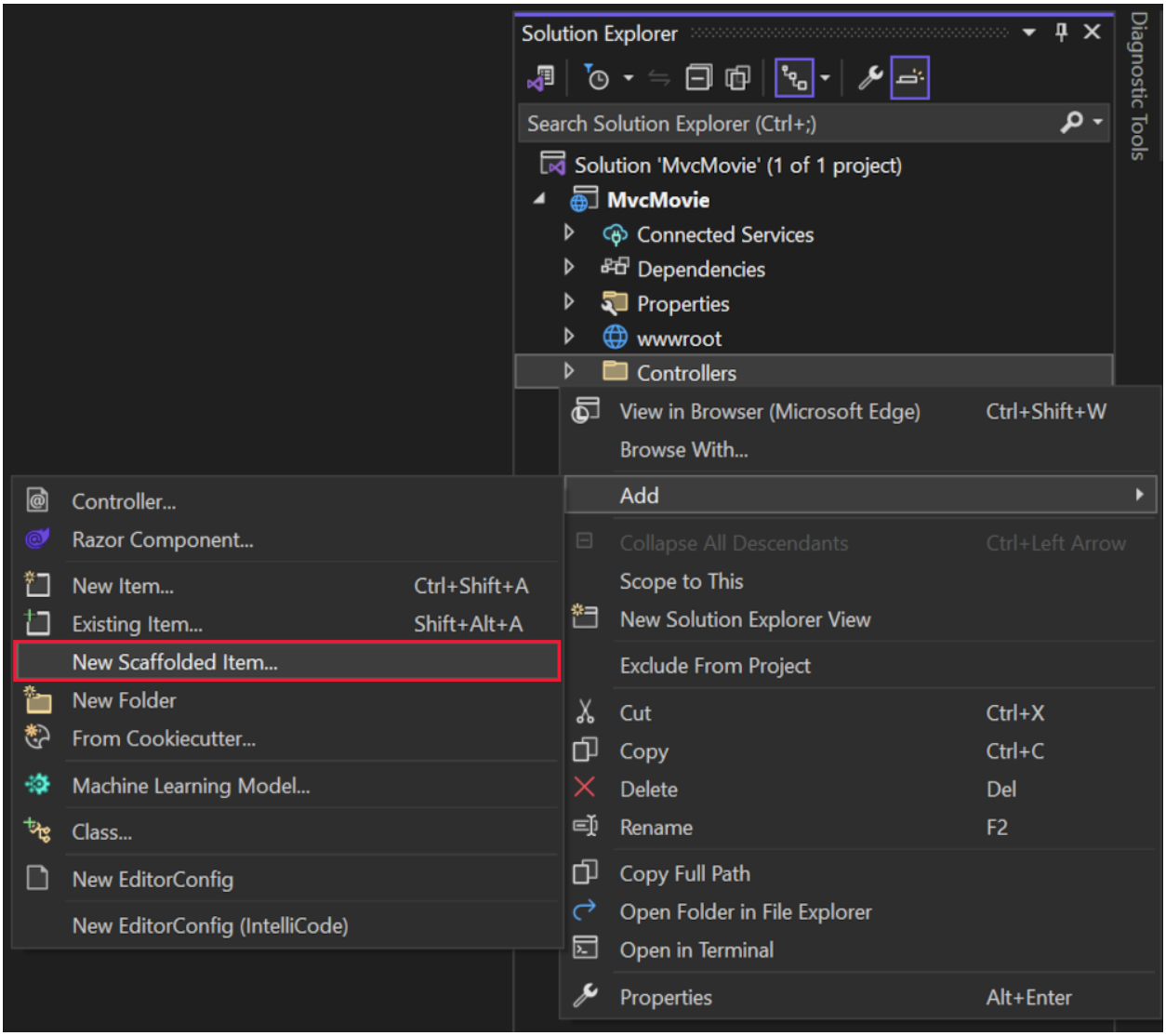
Build the project as a check for compiler errors.

Scaffold movie pages

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

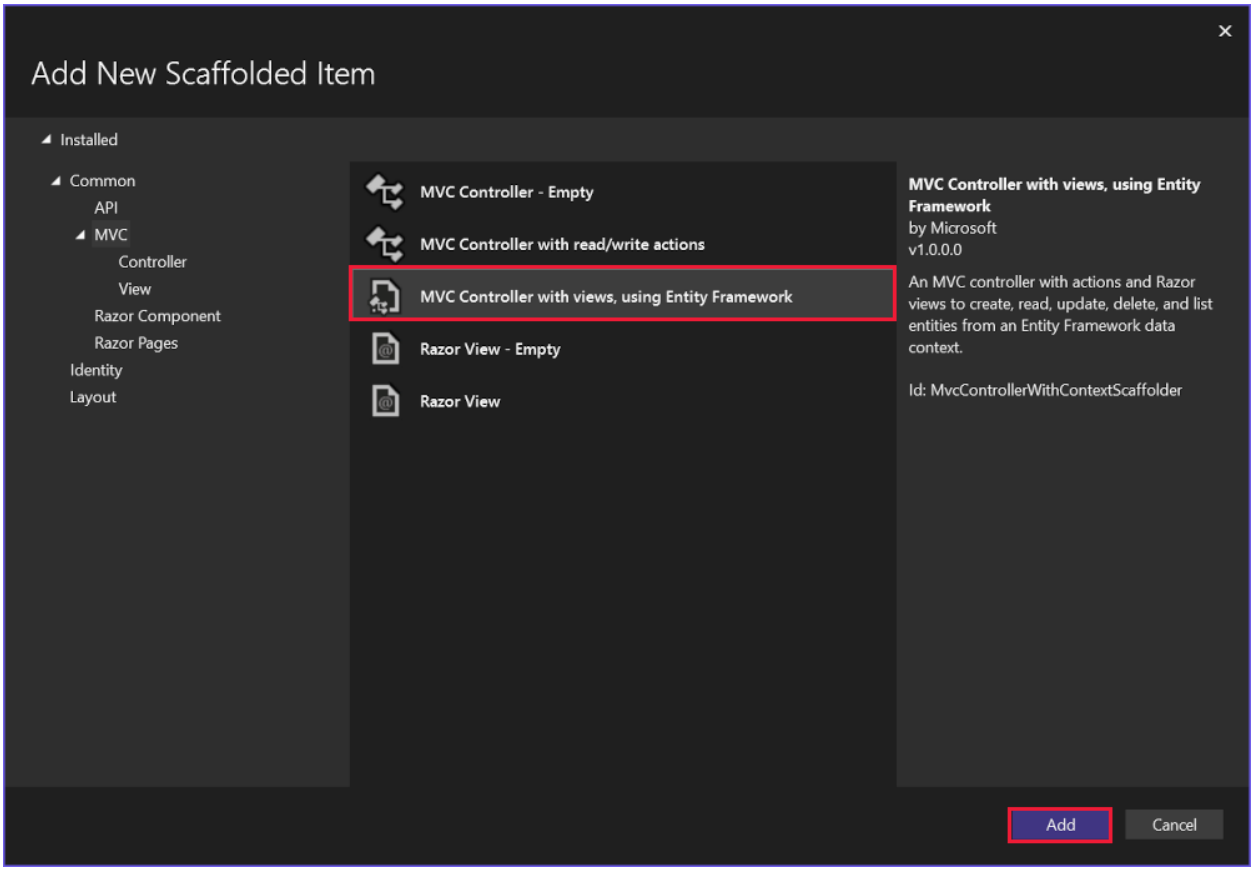
- Visual Studio
- Visual Studio Code
- Visual Studio for Mac

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



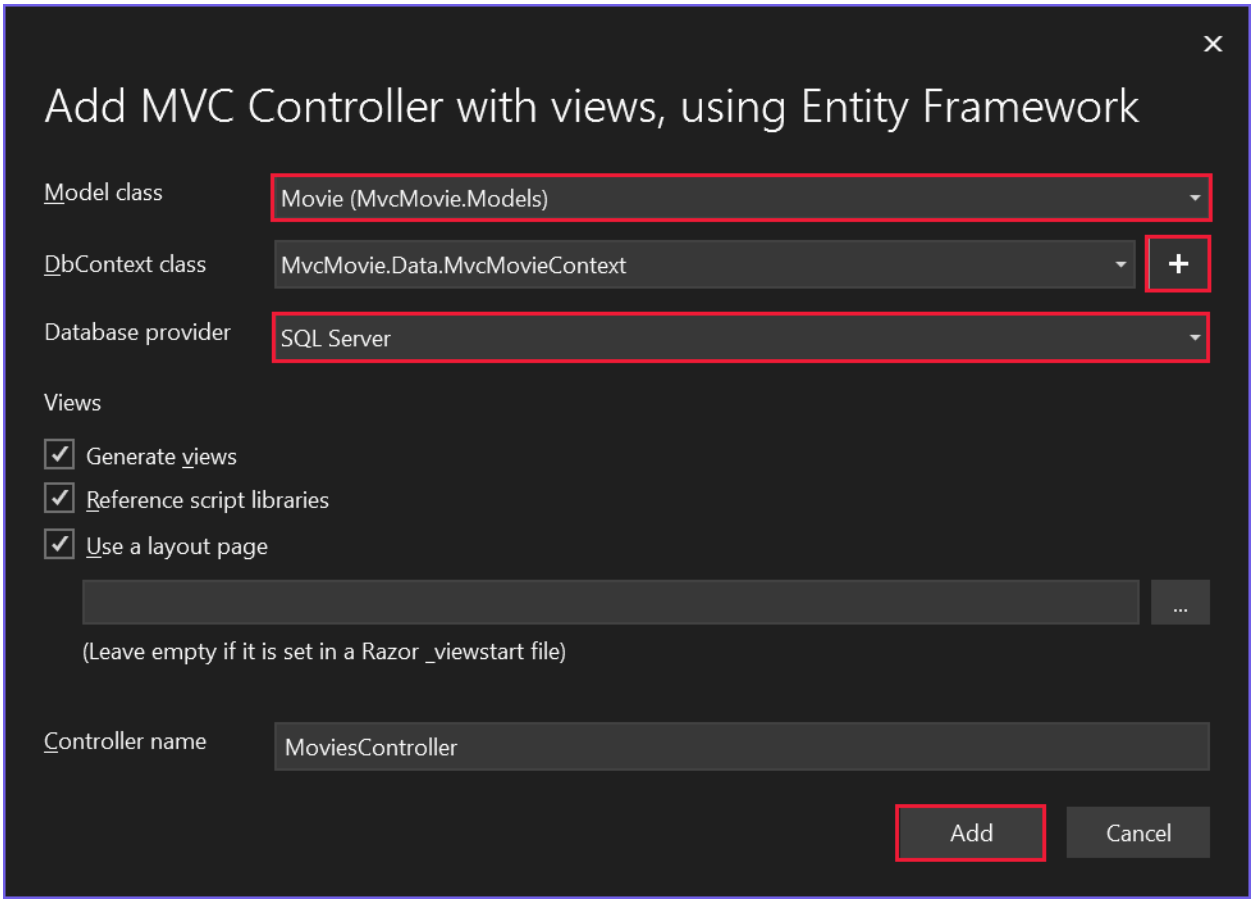
In the **Add New Scaffolded Item** dialog:

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



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**.



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
- Visual Studio Code / Visual Studio for Mac

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

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

PowerShellCopy

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

- `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.
- `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.

The `Update-Database` command generates the following warning:

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'.

Ignore the preceding warning, it's fixed in a later tutorial.

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

Test the app

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

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:

ConsoleCopy

```
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#Copy

```
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; }
    }
}
```

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
- Visual Studio Code / Visual Studio for Mac

C#Copy

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

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

options.UseSqlServer(builder.Configuration.GetConnectionString("MvcMovieContext")));
```

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
- Visual Studio Code / Visual Studio for Mac

JSONCopy

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "MvcMovieContext": "Data Source=MvcMovieContext-ea7a4069-f366-4742-bd1c-
3f753a804ce1.db"
  }
}
```

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#Copy

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

#nullable disable

namespace MvcMovie.Migrations
{
    public partial class InitialCreate : Migration
    {
        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);
                });
        }

        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#Copy

```
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#Copy

```
// 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#Copy

```
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#Copy

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

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

CSHTMLCopy

```
@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:

CSHTMLCopy

```
@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#Copy

```
// 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:

CSHTMLCopy

```
@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:

CSHTMLCopy

```
@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.

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

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
- Visual Studio Code / Visual Studio for Mac

C#Copy

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

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

options.UseSqlServer(builder.Configuration.GetConnectionString("MvcMovieContext")));
```

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

JSONCopy

```
"ConnectionStrings": {
  "MvcMovieContext": "Data Source=MvcMovieContext-ea7a4069-f366-4742-bd1c-
3f753a804ce1.db"
}
```

When the app is deployed to a test or production server, an environment variable can be used to set the connection string to a production SQL Server. For more information, see Configuration.

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

SQL Server Express LocalDB

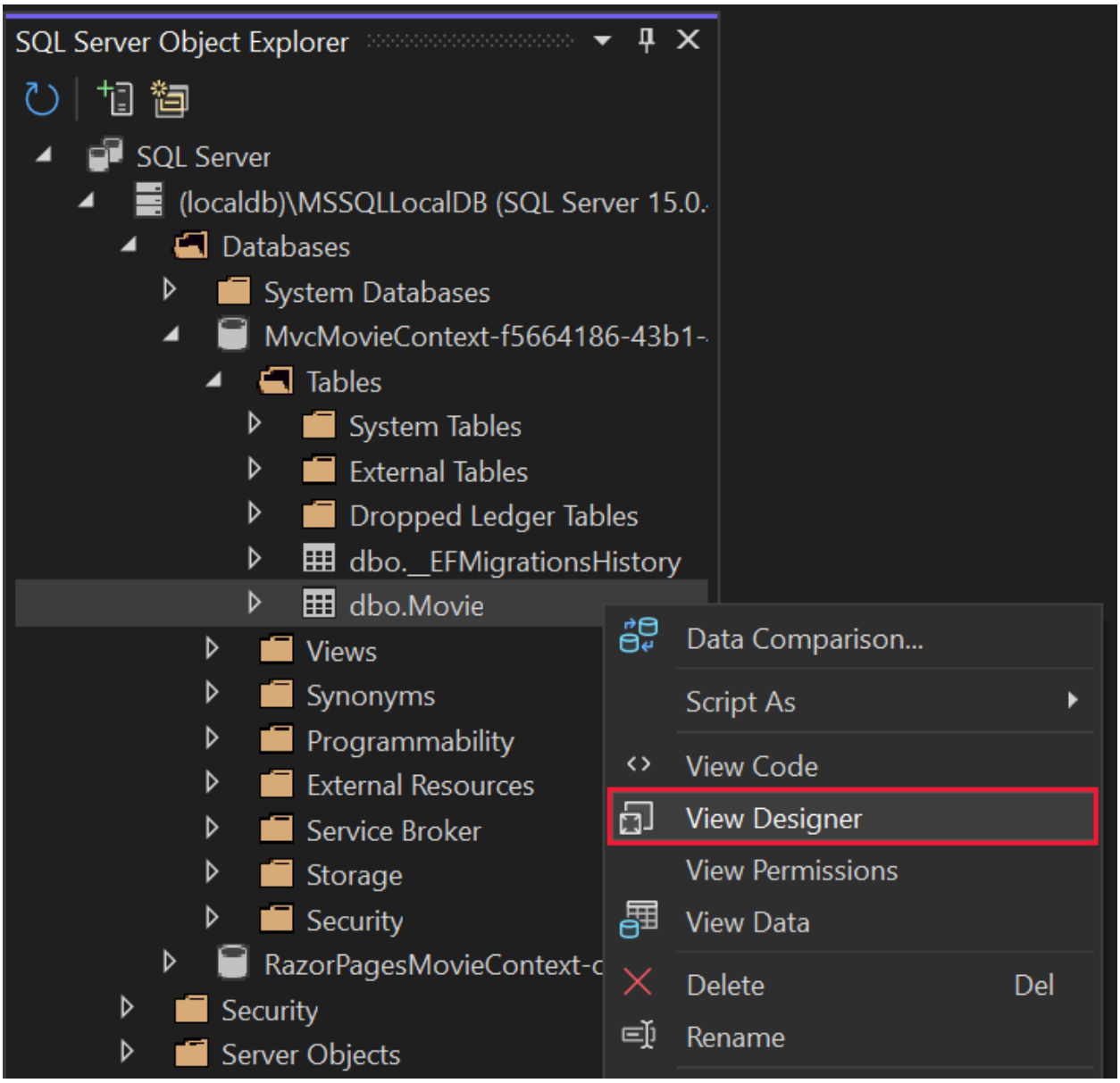
LocalDB:

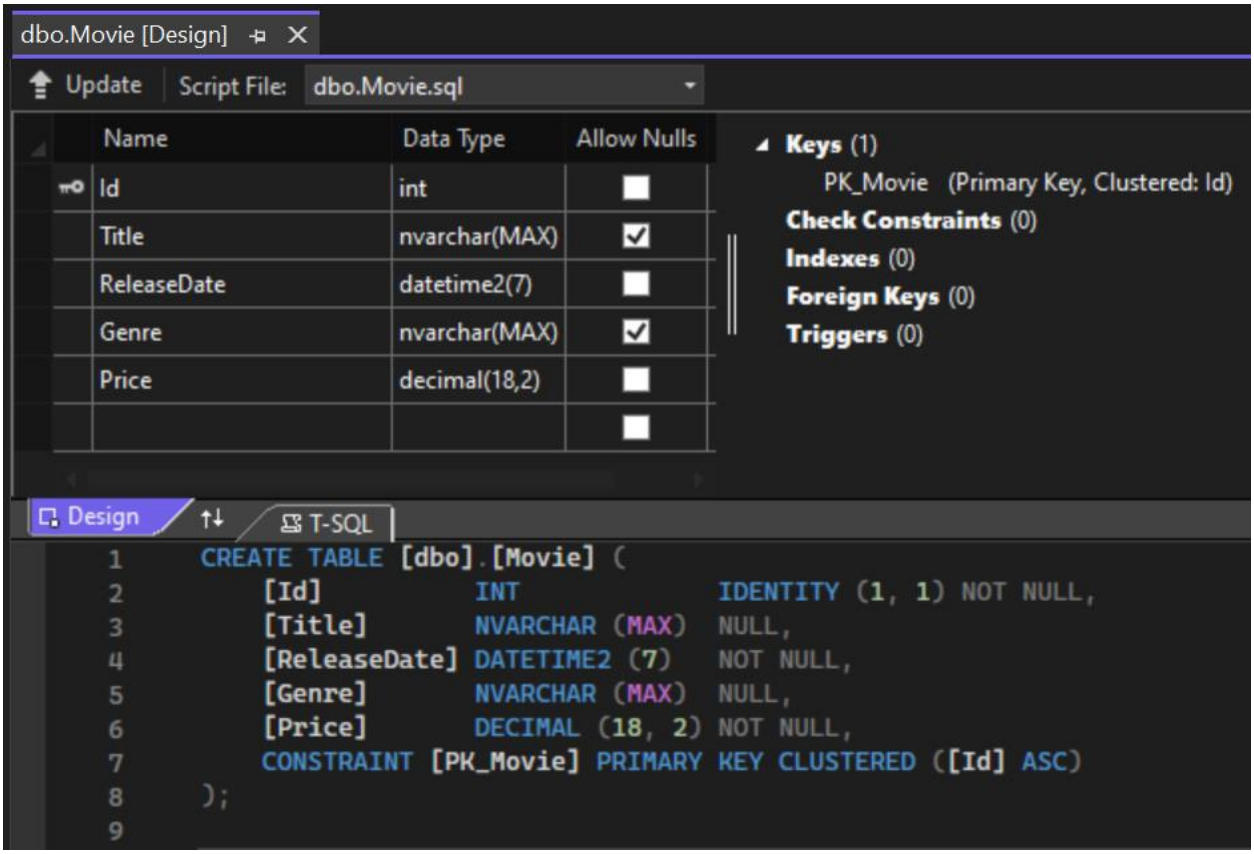
- Is a lightweight version of the SQL Server Express Database Engine, installed by default with Visual Studio.
- Starts on demand by using a connection string.
- Is targeted for program development. It runs in user mode, so there's no complex configuration.
- By default creates *.mdf* files in the *C:/Users/{user}*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

Examine the database

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

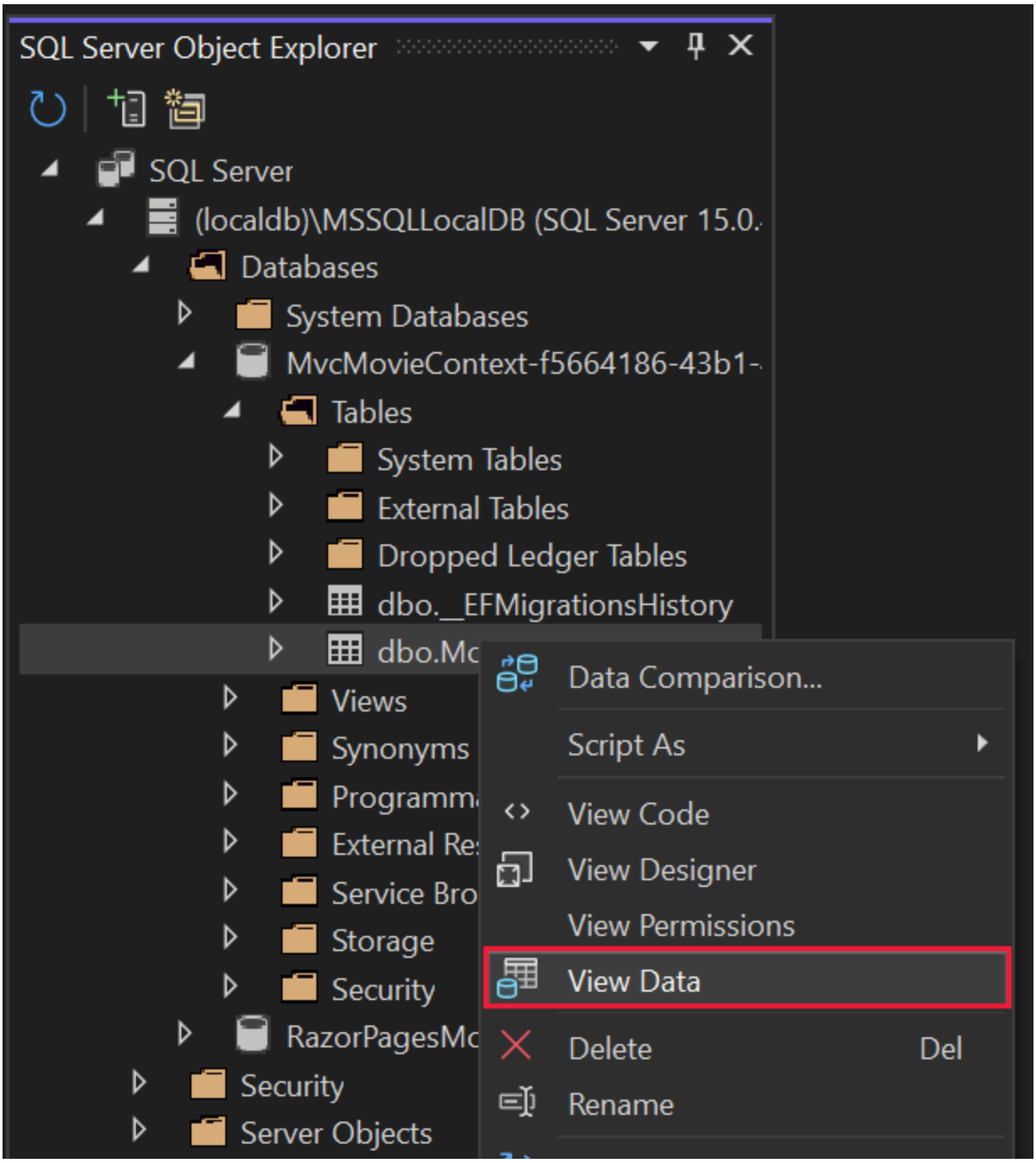
Right-click on the `Movie` table (`dbo.Movie`) **> View Designer**

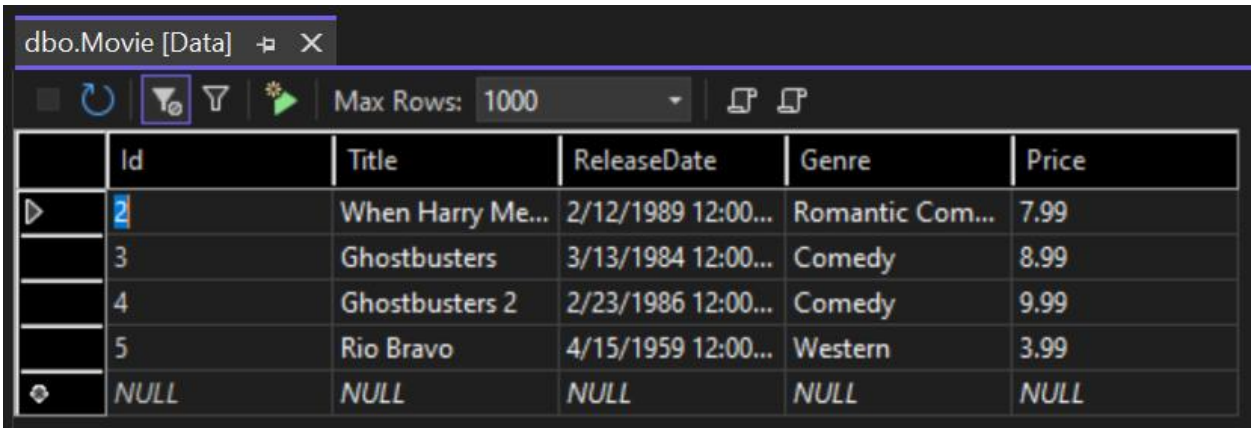

SQL Server Object Explorer
SQL Server
(localdb)\MSSQLLocalDB (SQL Server 15.0.
Databases
System Databases
MvcMovieContext-f5664186-43b1-
Tables
System Tables
External Tables
Dropped Ledger Tables
dbo.__EFMigrationsHistory
dbo.Movie
Views
Synonyms
Programmability
External Resources
Service Broker
Storage
Security
RazorPagesMovieContext-c
Security
Server Objects
Data Comparison...
Script As
View Code
View Designer
View Permissions
View Data
Delete
Del
Rename



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**


SQL Server Object Explorer
SQL Server
(localdb)\MSSQLLocalDB (SQL Server 15.0.
Databases
System Databases
MvcMovieContext-f5664186-43b1-
Tables
System Tables
External Tables
Dropped Ledger Tables
dbo.__EFMigrationsHistory
dbo.Mc
Views
Synonyms
Programm
External Re
Service Bro
Storage
Security
RazorPagesMc
Security
Server Objects
Data Comparison...
Script As
View Code
View Designer
View Permissions
View Data
Delete
Del
Rename



Seed the database

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

C#Copy

```
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#Copy

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

Add the seed initializer

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

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

C#Copy

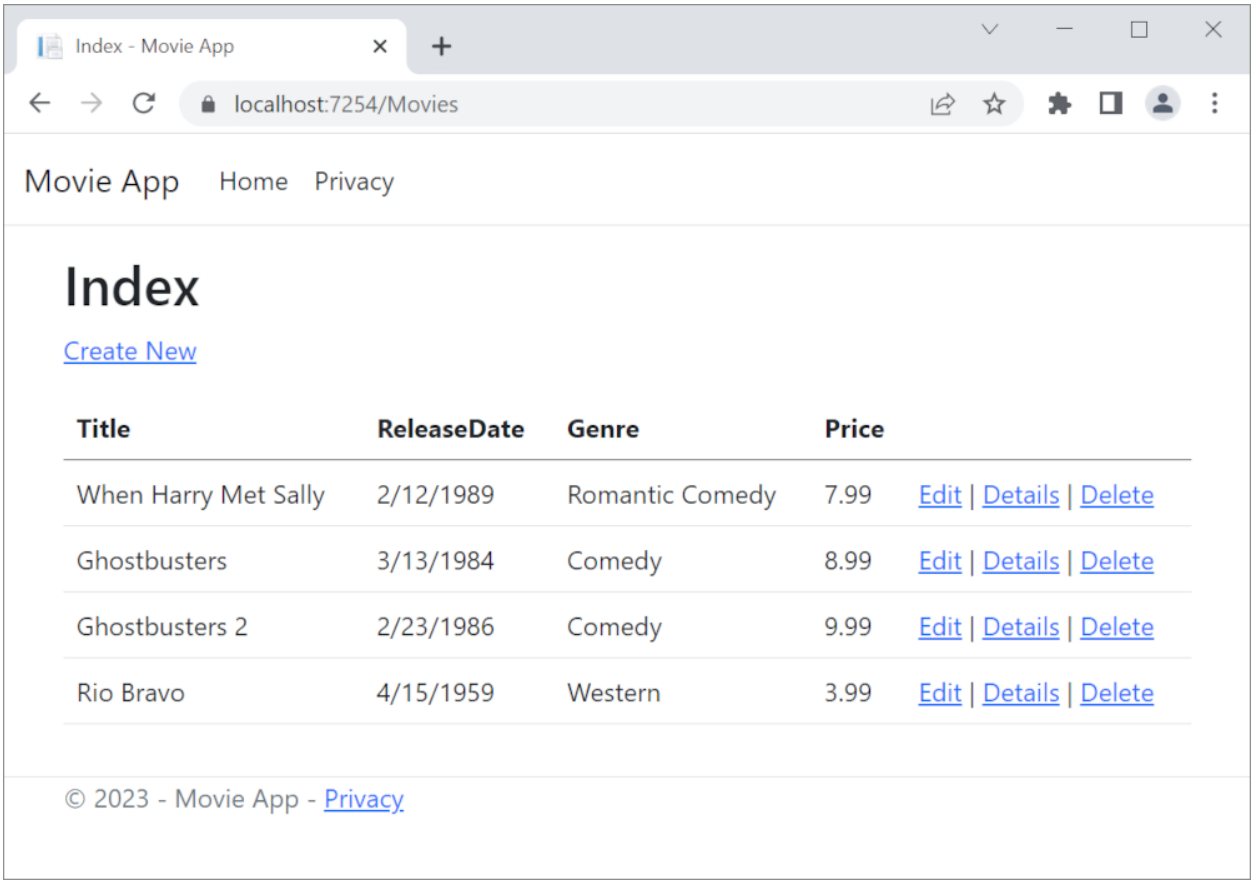
```
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using MvcMovie.Data;
using MvcMovie.Models;
```

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

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

options.UseSqlServer(builder.Configuration.GetConnectionString("MvcMovieContext")));

// 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.UseStaticFiles();

app.UseRouting();

app.UseAuthorization();

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.

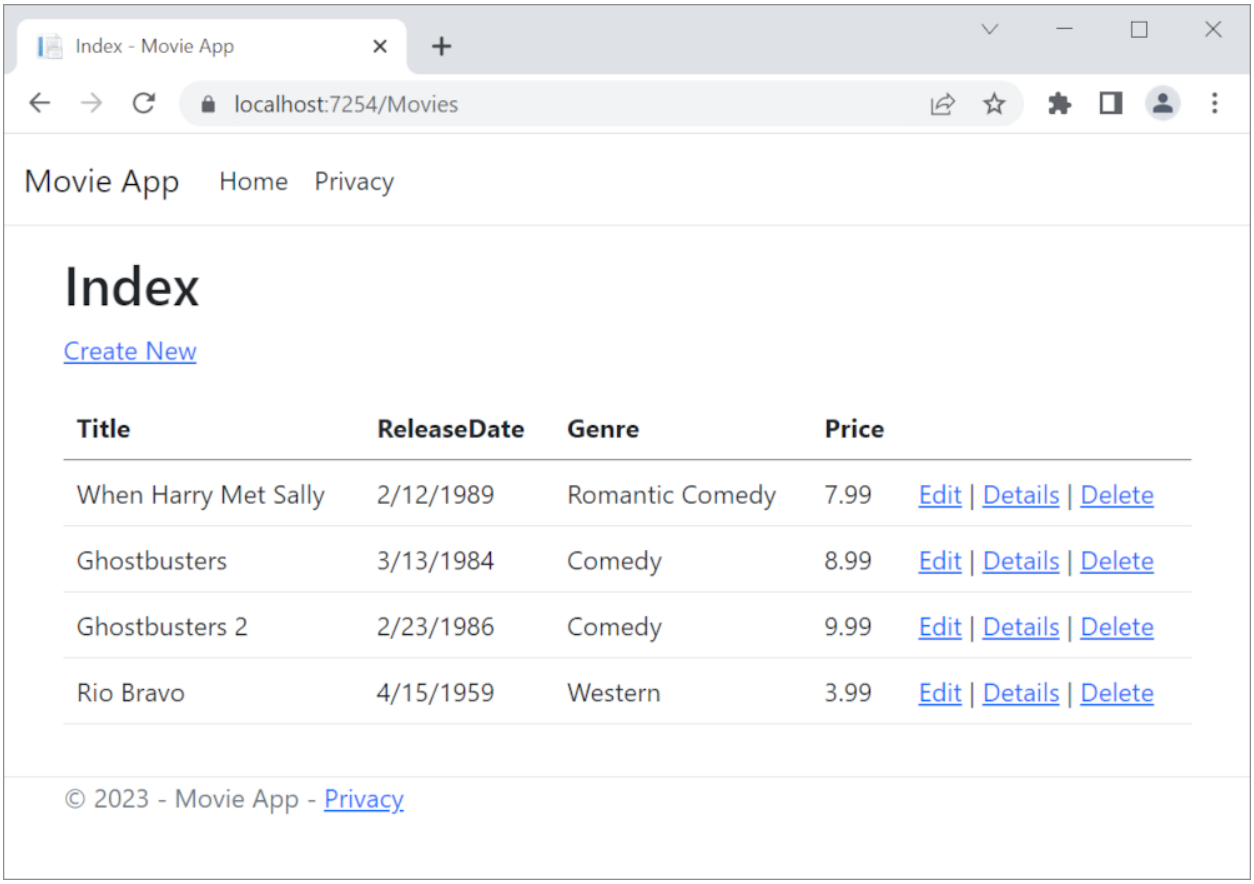
Test the app. Force the app to initialize, calling the code in the `Program.cs` file, so the seed method runs. To force initialization, close the command prompt window that Visual Studio opened, and restart by pressing Ctrl+F5.

The app shows the seeded data.



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

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



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

C#Copy

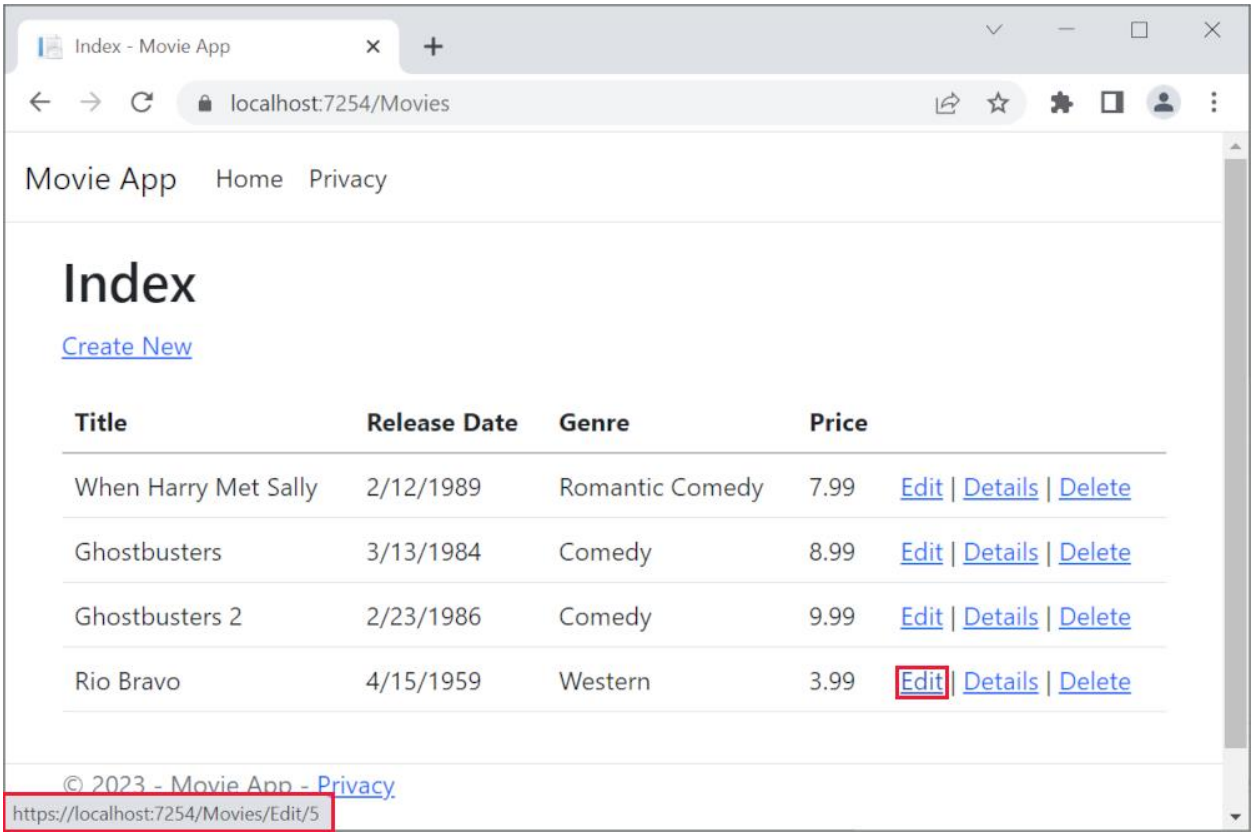
```
using System;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

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.



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

CSHTMLCopy

```
        <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:

HTMLCopy

```
 <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#Copy

```
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#Copy

```
// 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#Copy

```
// 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#Copy

```
// 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 anti-forgery token generated in the edit view file (`Views/Movies/Edit.cshtml`). The edit view file generates the anti-forgery token with the Form Tag Helper.

CSHTMLCopy

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

The Form Tag Helper generates a hidden anti-forgery token that must match the `[ValidateAntiForgeryToken]` generated anti-forgery 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#Copy

```
// 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:

CSHTMLCopy

```
@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.

HTMLCopy

```
<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="CfDJ8Inyxgp63fRFqUePGvuI5jGZsloJu1L7X9le1gy7NCIlSduCRx9jDQClrV9pOTTmqUyXnJBXhm
rjcUVDJyDUMm7-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#Copy

```
// 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]
```

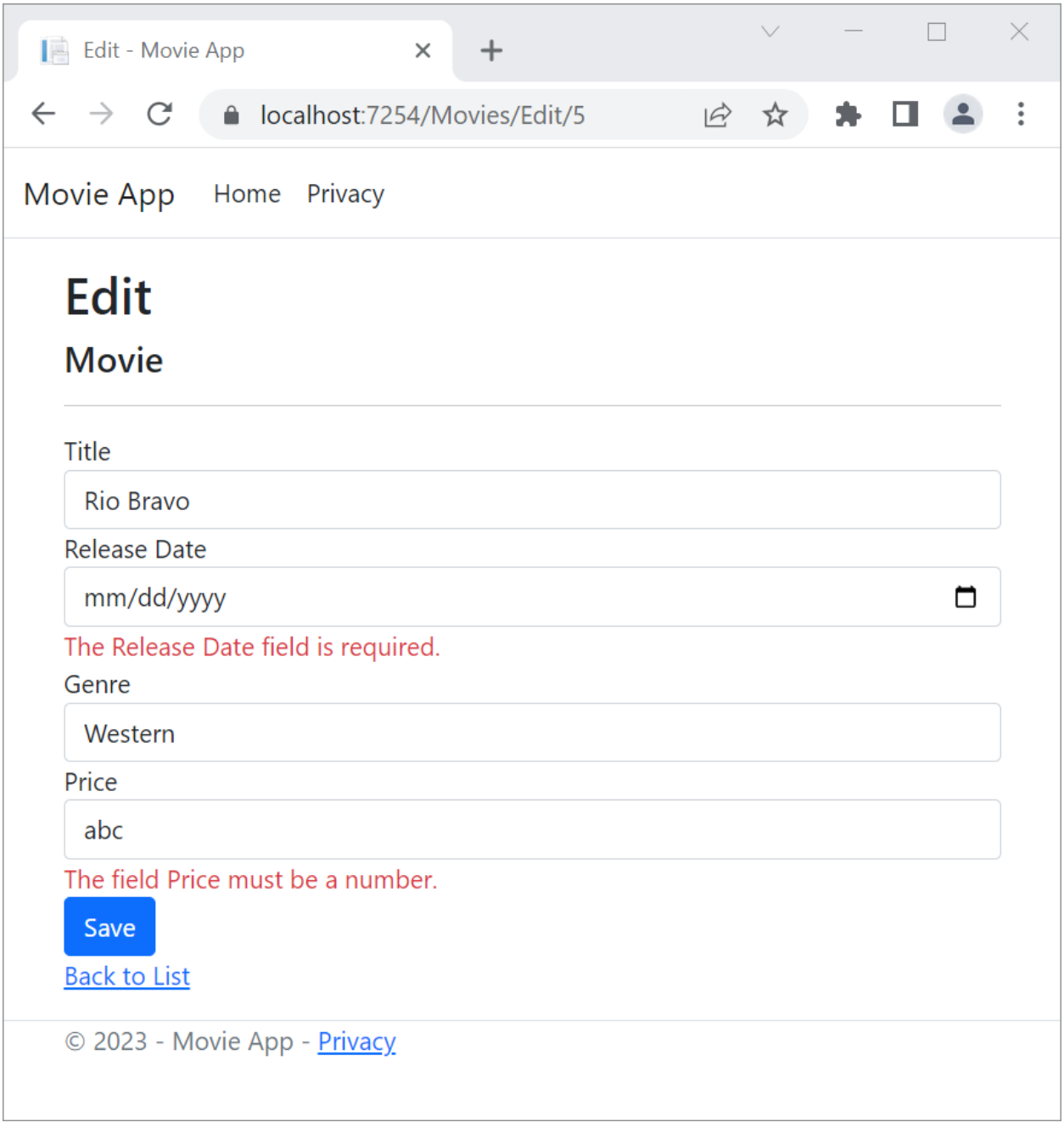
```
public async Task<IActionResult> Edit(int id,
[Bind("Id,Title,ReleaseDate,Genre,Price,Rating")] Movie movie)
{
    if (id != movie.Id)
    {
        return NotFound();
    }

    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 anti-forgery 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.



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.

For other parts use the given below link

https://learn.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/search?view=aspnetcore-8.0