# Create a web app with ASP.NET Core MVC on Windows with Visual Studio

Source: <https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/?view=aspnetcore-2.2>

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](https://docs.microsoft.com/en-us/aspnet/core/tutorials/razor-pages/razor-pages-start?view=aspnetcore-2.2) version of this tutorial, which provides an easier starting point.

The tutorial series includes the following:

1. [Get started](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/start-mvc?view=aspnetcore-2.2)
2. [Add a controller](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/adding-controller?view=aspnetcore-2.2)
3. [Add a view](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/adding-view?view=aspnetcore-2.2)
4. [Add a model](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/adding-model?view=aspnetcore-2.2)
5. [Work with SQL Server LocalDB](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/working-with-sql?view=aspnetcore-2.2)
6. [Controller methods and views](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/controller-methods-views?view=aspnetcore-2.2)
7. [Add search](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/search?view=aspnetcore-2.2)
8. [Add a new field](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/new-field?view=aspnetcore-2.2)
9. [Add validation](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/validation?view=aspnetcore-2.2)
10. [Examine the Details and Delete methods](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/details?view=aspnetcore-2.2)

# Get started with ASP.NET Core MVC

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](https://docs.microsoft.com/en-us/aspnet/core/tutorials/razor-pages/razor-pages-start?view=aspnetcore-2.2) version of this tutorial, which provides an easier starting point.

<https://docs.microsoft.com/en-us/visualstudio/ide/visual-studio-ide?view=vs-2017>

This tutorial teaches the basics of building an ASP.NET Core MVC web app.

The app manages a database of movie titles. You learn how to:

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

At the end, you have an app that can manage and display movie data.

[View or download sample code](https://github.com/aspnet/Docs/tree/master/aspnetcore/tutorials/first-mvc-app/start-mvc/sample) ([how to download](https://docs.microsoft.com/en-us/aspnet/core/index?view=aspnetcore-2.2#how-to-download-a-sample)).

**Note**

We’re testing the usability of a proposed new structure for the ASP.NET Core table of contents. If you have a few minutes to try an exercise of finding 7 different topics in the current or proposed table of contents, please [**click here to participate in the study**](https://dpk4xbh5.optimalworkshop.com/treejack/aa11wn82).

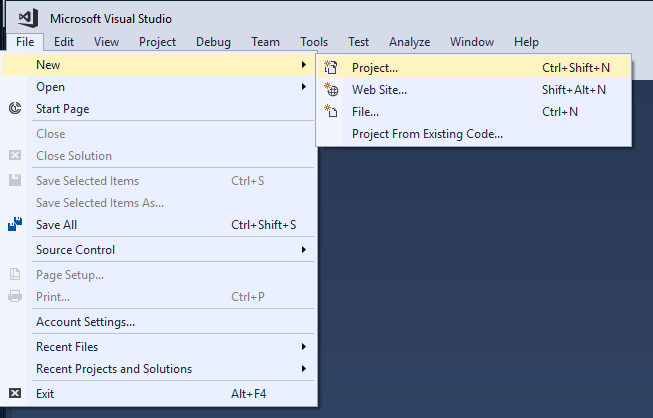
## Prerequisites

* [Visual Studio](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/start-mvc?view=aspnetcore-2.2&tabs=visual-studio#tabpanel_CeZOj-G++Q_visual-studio)
* [Visual Studio 2017 version 15.9 or later](https://visualstudio.microsoft.com/downloads/) with the **ASP.NET and web development** workload
* [.NET Core SDK 2.2 or later](https://www.microsoft.com/net/download/all)
* [Visual Studio Code](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/start-mvc?view=aspnetcore-2.2&tabs=visual-studio#tabpanel_CeZOj-G++Q_visual-studio-code)
* [Visual Studio Code](https://code.visualstudio.com/download)
* [.NET Core SDK 2.2 or later](https://www.microsoft.com/net/download/all)
* [C# for Visual Studio Code version 1.17.1 or later](https://marketplace.visualstudio.com/items?itemName=ms-vscode.csharp)
* [Visual Studio for Mac](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/start-mvc?view=aspnetcore-2.2&tabs=visual-studio#tabpanel_CeZOj-G++Q_visual-studio-mac)
* [Visual Studio for Mac version 7.7 or later](https://www.visualstudio.com/downloads/)
* [.NET Core SDK 2.2 or later](https://www.microsoft.com/net/download/all)

## Create a web app

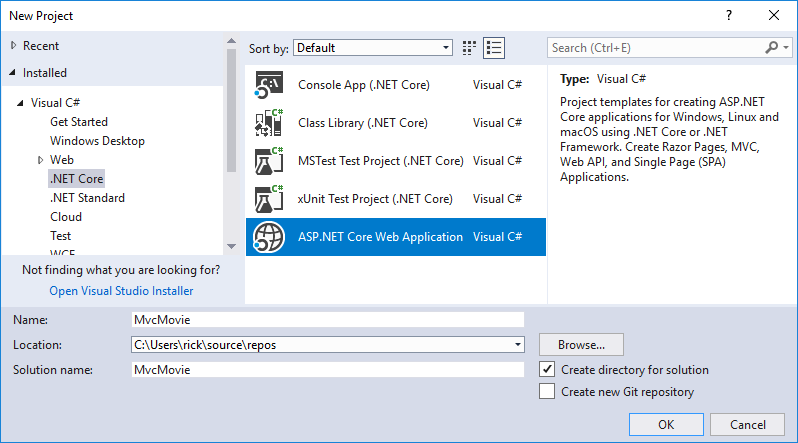
* [Visual Studio](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/start-mvc?view=aspnetcore-2.2&tabs=visual-studio#tabpanel_CeZOj-G++Q_visual-studio)

From Visual Studio, select **File > New > Project**.



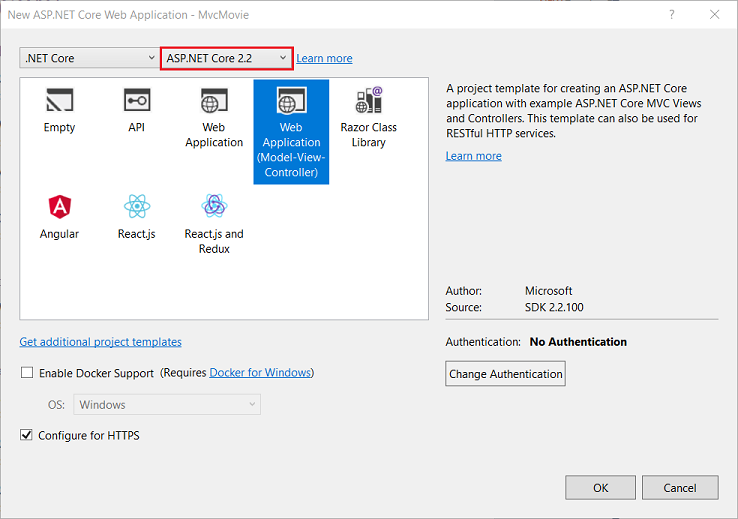
Complete the **New Project** dialog:

* In the left pane, select **.NET Core**
* In the center pane, select **ASP.NET Core Web Application (.NET Core)**
* Name the project "MvcMovie" (It's important to name the project "MvcMovie" so when you copy code, the namespace will match.)
* select **OK**



Complete the **New ASP.NET Core Web Application (.NET Core) - MvcMovie** dialog:

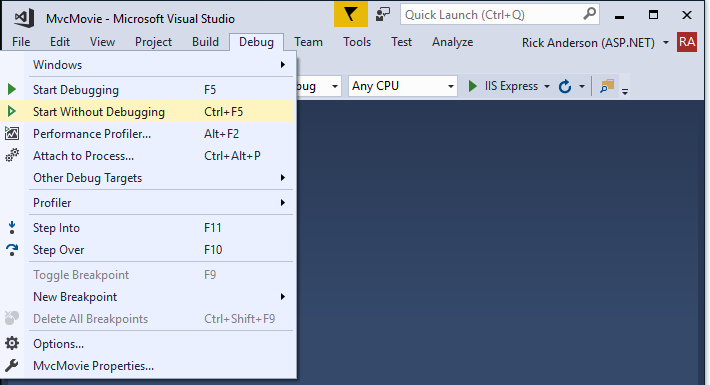
* In the version selector drop-down box select **ASP.NET Core 2.2**
* Select **Web Application (Model-View-Controller)**
* select **OK**.



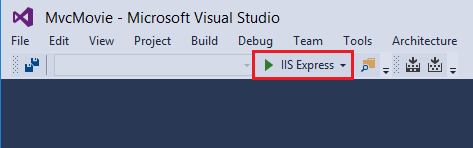
Visual Studio used a default template for the MVC project you just created. You have a working app right now by entering a project name and selecting a few options. This is a basic starter project, and it's a good place to start.

Select **Ctrl-F5** to run the app in non-debug mode.

* Visual Studio starts [IIS Express](https://docs.microsoft.com/iis/extensions/introduction-to-iis-express/iis-express-overview) and runs your app. Notice that the address bar shows localhost:port# and not something like example.com. That's because localhost is the standard hostname for your local computer. When Visual Studio creates a web project, a random port is used for the web server. In the image above, the port number is 5000. The URL in the browser shows localhost:5000. When you run the app, you'll see a different port number.
* Launching the app with **Ctrl+F5** (non-debug mode) allows you to make code changes, save the file, refresh the browser, and see the code changes. Many developers prefer to use non-debug mode to quickly launch the app and view changes.
* You can launch the app in debug or non-debug mode from the **Debug** menu item:



* You can debug the app by selecting the **IIS Express** button



* [Visual Studio Code](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/start-mvc?view=aspnetcore-2.2&tabs=visual-studio#tabpanel_CeZOj-G++Q_visual-studio-code)

The tutorial assumes familarity with VS Code. See [Getting started with VS Code](https://code.visualstudio.com/docs) and [Visual Studio Code help](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/start-mvc?view=aspnetcore-2.2&tabs=visual-studio-code#visual-studio-code-help) for more information.

* Open the [integrated terminal](https://code.visualstudio.com/docs/editor/integrated-terminal).
* Change directories (cd) to a folder which will contain the project.
* Run the following command:

consoleCopy

dotnet new mvc -o MvcMovie

code -r MvcMovie

* + A dialog box appears with **Required assets to build and debug are missing from 'MvcMovie'. Add them?** Select **Yes**
  + dotnet new mvc -o MvcMovie: creates a new ASP.NET Core MVC project in the MvcMovie folder.
  + code -r MvcMovie: Loads the MvcMovie.csproj project file in Visual Studio Code.

### Launch the app

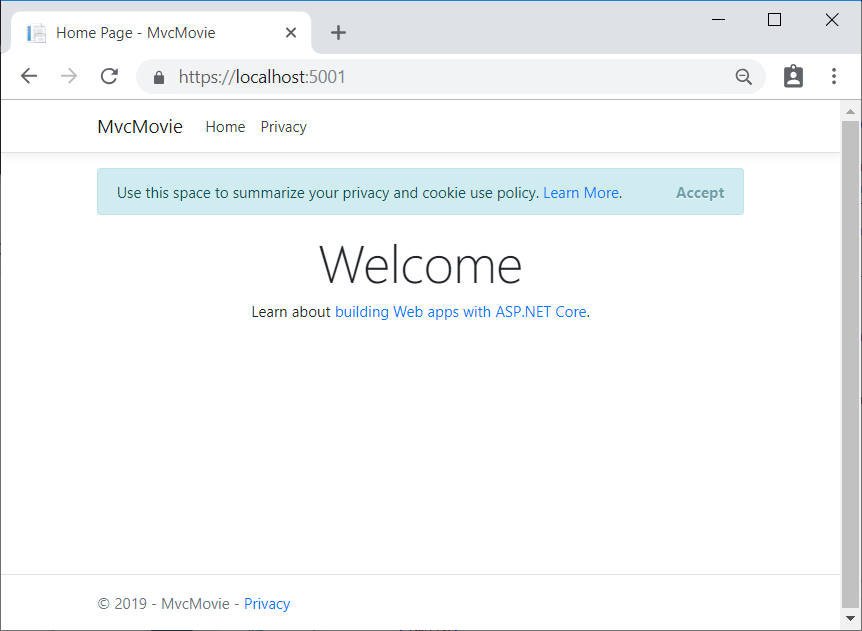
* Press **Ctrl-F5** to run without the debugger.

Visual Studio Code starts starts [Kestrel](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/servers/kestrel?view=aspnetcore-2.2), launches a browser, and navigates to http://localhost:5001. The address bar shows localhost:port:5001 and not something like example.com. That's because localhost is the standard hostname for local computer. Localhost only serves web requests from the local computer.

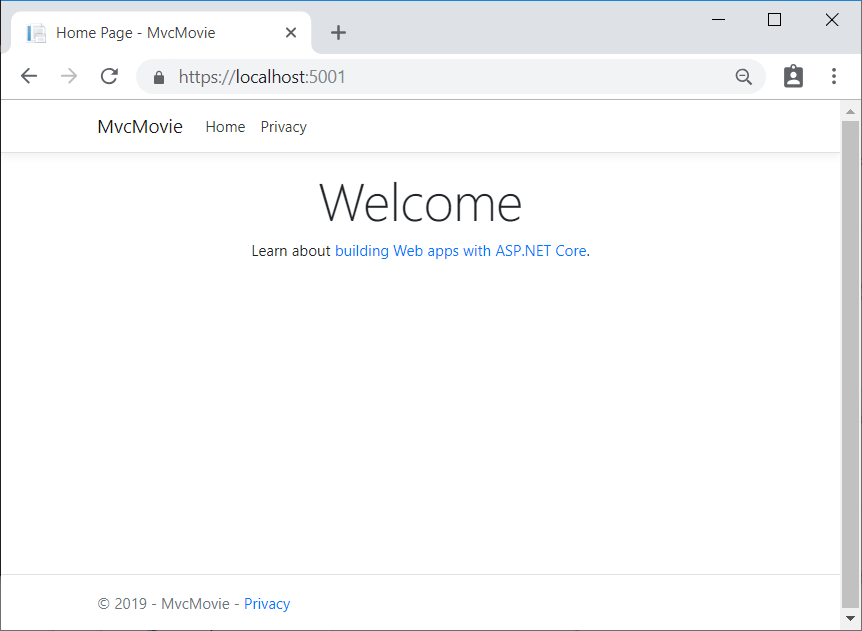
Launching the app with **Ctrl+F5** (non-debug mode) allows you to make code changes, save the file, refresh the browser, and see the code changes. Many developers prefer to use non-debug mode to refresh the page and view changes.

CONTINUE…

* Select **Accept** to consent to tracking. This app doesn't track personal information. The template generated code includes assets to help meet [General Data Protection Regulation (GDPR)](https://docs.microsoft.com/en-us/aspnet/core/security/gdpr?view=aspnetcore-2.2).



The following image shows the app after accepting tracking:



* [Visual Studio](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/start-mvc?view=aspnetcore-2.2&tabs=visual-studio#tabpanel_CeZOj-G++Q_visual-studio)

## Visual Studio help

* [Learn to debug C# code using Visual Studio](https://docs.microsoft.com/en-us/visualstudio/debugger/getting-started-with-the-debugger?view=vs-2017)
* [Introduction to the Visual Studio IDE](https://docs.microsoft.com/en-us/visualstudio/ide/visual-studio-ide?view=vs-2017)
* [Visual Studio Code](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/start-mvc?view=aspnetcore-2.2&tabs=visual-studio#tabpanel_CeZOj-G++Q_visual-studio-code)

## Visual Studio Code help

* [Getting started](https://code.visualstudio.com/docs)
* [Debugging](https://code.visualstudio.com/docs/editor/debugging)
* [Integrated terminal](https://code.visualstudio.com/docs/editor/integrated-terminal)
* [Keyboard shortcuts](https://code.visualstudio.com/docs/getstarted/keybindings#_keyboard-shortcuts-reference)
  + [macOS keyboard shortcuts](https://code.visualstudio.com/shortcuts/keyboard-shortcuts-macos.pdf)
  + [Linux keyboard shortcuts](https://code.visualstudio.com/shortcuts/keyboard-shortcuts-linux.pdf)
  + [Windows keyboard shortcuts](https://code.visualstudio.com/shortcuts/keyboard-shortcuts-windows.pdf)

CONTINUE…

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

# Add a controller to an ASP.NET Core MVC app

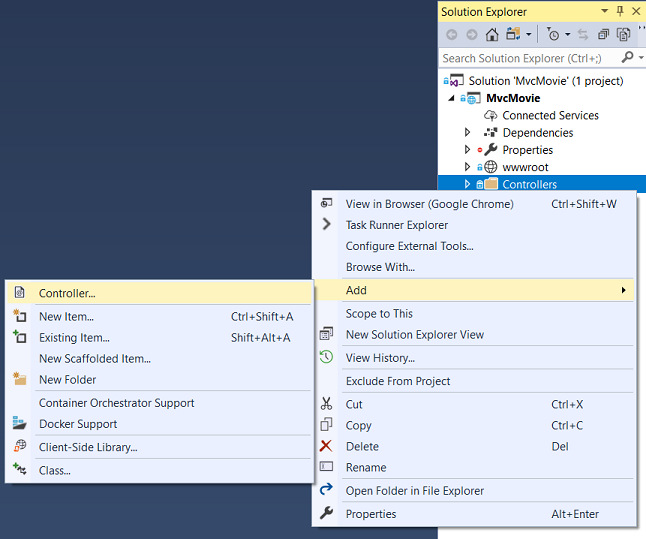
The Model-View-Controller (MVC) architectural pattern separates an app into three main components: **M**odel, **V**iew, and **C**ontroller. The MVC pattern helps you create apps that are more testable and easier to update than traditional monolithic apps. MVC-based apps contain:

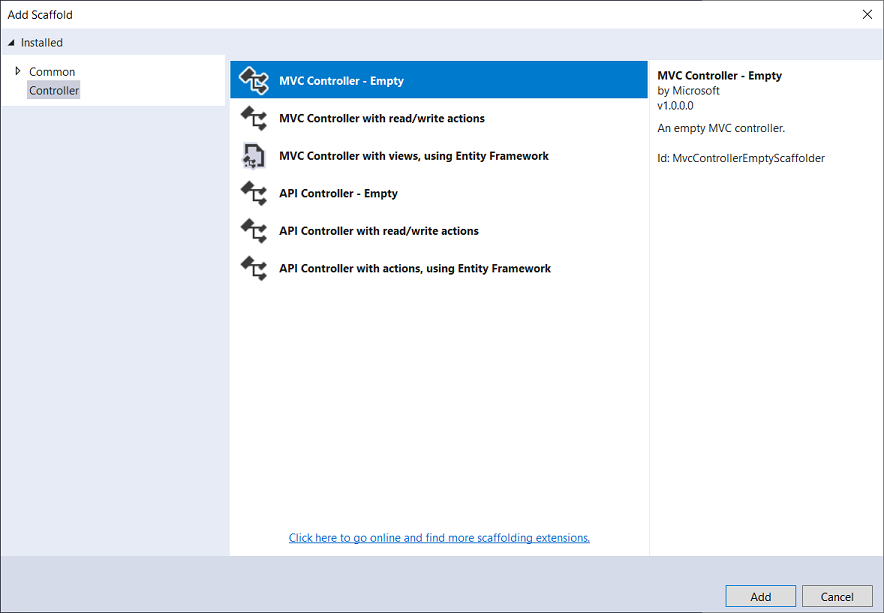
* **M**odels: Classes that represent the data of the app. The model classes use validation logic to enforce business rules for that data. Typically, model objects retrieve and store model state in a database. In this tutorial, a Movie model retrieves movie data from a database, provides it to the view 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. They retrieve model data and 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 route data 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:1234/Home/About has route data of Home (the controller) and About(the action method to call on the home controller). https://localhost:1234/Movies/Edit/5 is a request to edit the movie with ID=5 using the movie controller. Route data is explained later in the tutorial.

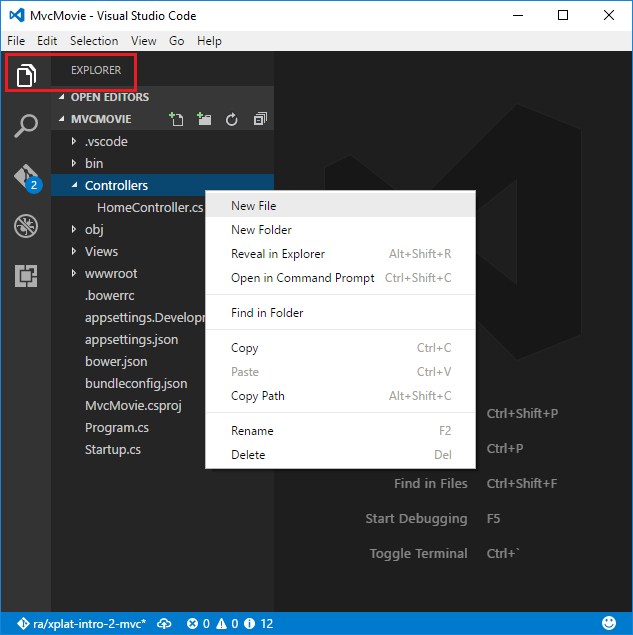
The MVC pattern helps you create apps that separate the different aspects of the app (input logic, business logic, and UI logic), while providing a loose coupling between these elements. The pattern specifies where each kind of logic should be located in the app. The UI logic belongs in the view. Input logic belongs in the controller. Business logic belongs in the model. This separation helps you manage complexity when you build an app, because it enables you to 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.

We cover these concepts in this tutorial series and show you how to use them to build a movie app. The MVC project contains folders for the Controllers and Views.

## Add a controller

* [Visual Studio](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/adding-controller?view=aspnetcore-2.2&tabs=visual-studio#tabpanel_CeZOj-G++Q_visual-studio)
* In **Solution Explorer**, right-click **Controllers > Add > Controller** 
* In the **Add Scaffold** dialog box, select **MVC Controller - Empty**



* In the **Add Empty MVC Controller dialog**, enter **HelloWorldController** and select **ADD**.
* [Visual Studio Code](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/adding-controller?view=aspnetcore-2.2&tabs=visual-studio#tabpanel_CeZOj-G++Q_visual-studio-code)
* Select the **EXPLORER** icon and then control-click (right-click) **Controllers > New File** and name the new file *HelloWorldController.cs*.
* 

CONTINUE…

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

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...";

}

}

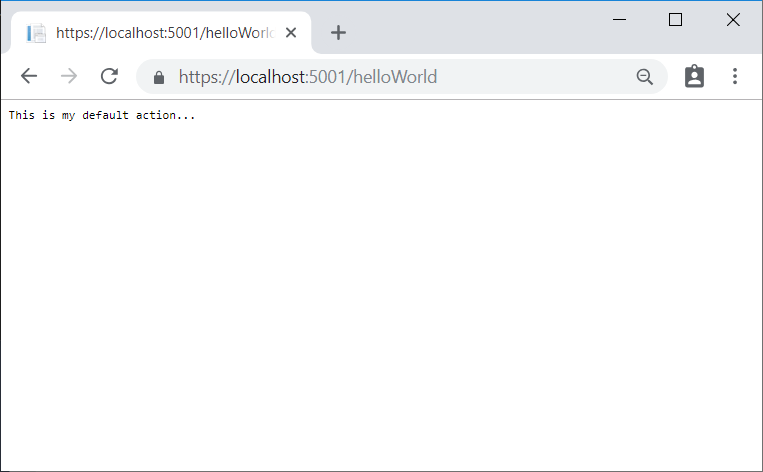
}

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, and combines the protocol used: HTTPS, the network location of the web server (including the TCP port): localhost:5001 and the target URI HelloWorld.

The first comment states this is an [HTTP GET](https://www.w3schools.com/tags/ref_httpmethods.asp) method that's invoked by appending /HelloWorld/ to the base URL. The second comment specifies an [HTTP GET](http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html) 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 in non-debug mode and append "HelloWorld" to the path in the address bar. The Indexmethod returns a string.



MVC invokes controller classes (and the action methods within them) depending on the incoming URL. The default [URL routing logic](https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/routing?view=aspnetcore-2.2) used by MVC uses a format like this to determine what code to invoke:

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

The routing format is set in the Configure method in Startup.cs file.

C#Copy

app.UseMvc(routes =>

{

routes.MapRoute(

name: "default",

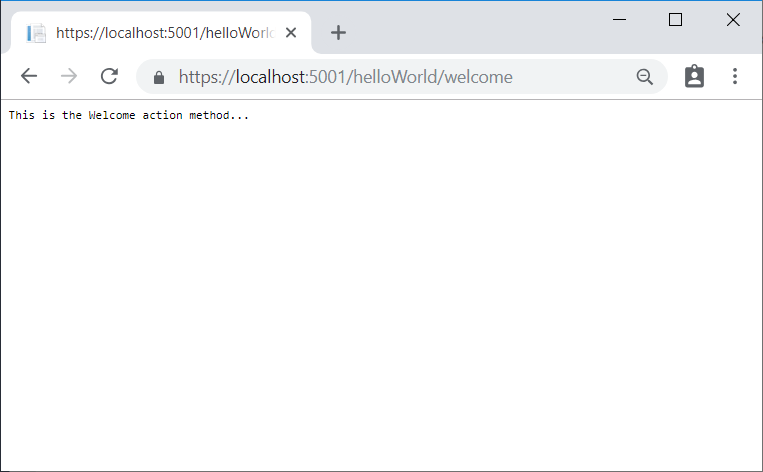
template: "{controller=Home}/{action=Index}/{id?}");

});

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.

The first URL segment determines the controller class to run. So localhost:xxxx/HelloWorld maps to the HelloWorldController class. The second part of the URL segment determines the action method on the class. So localhost:xxxx/HelloWorld/Index would cause the Index method of the HelloWorldController class to run. Notice that you only had to browse to localhost:xxxx/HelloWorld and the Index method was called by default. This is because 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:xxxx/HelloWorld/Welcome. 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.
* UsesHtmlEncoder.Default.Encode to protect the app from malicious input (namely JavaScript).
* Uses [Interpolated Strings](https://docs.microsoft.com/en-us/dotnet/articles/csharp/language-reference/keywords/interpolated-strings) in $"Hello {name}, NumTimes is: {numTimes}".

Run the app and browse to:

https://localhost:xxxx/HelloWorld/Welcome?name=Rick&numtimes=4

(Replace xxxx with your port number.) You can try different values for name and numtimes in the URL. The MVC [model binding](https://docs.microsoft.com/en-us/aspnet/core/mvc/models/model-binding?view=aspnetcore-2.2) system automatically maps the named parameters from the query string in the address bar to parameters in your method. See [Model Binding](https://docs.microsoft.com/en-us/aspnet/core/mvc/models/model-binding?view=aspnetcore-2.2) for more information.



In the image above, the URL segment (Parameters) isn't used, the name and numTimes parameters are passed as [query strings](https://wikipedia.org/wiki/Query_string). The ? (question mark) in the above URL is a separator, and the query strings follow. The & character separates query strings.

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:xxx/HelloWorld/Welcome/3?name=Rick

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

C#Copy

app.UseMvc(routes =>

{

routes.MapRoute(

name: "default",

template: "{controller=Home}/{action=Index}/{id?}");

});

In these examples the controller has been doing the "VC" portion of MVC - that is, the view and controller work. The controller is returning HTML directly. Generally you don't want controllers returning HTML directly, since that becomes very cumbersome to code and maintain. Instead you typically use a separate Razor view template file to help generate the HTML response. You do that in the next tutorial.

# Add a view to an ASP.NET Core MVC app

In this section you modify the HelloWorldController class to use [Razor](https://docs.microsoft.com/en-us/aspnet/core/mvc/views/razor?view=aspnetcore-2.2) view files to cleanly encapsulate the process of generating HTML responses to a client.

You create a view template file using Razor. Razor-based view templates have a .cshtml file extension. They provide an elegant way to create HTML output with C#.

Currently the Index method returns a string with a message that's hard-coded in the controller class. In the HelloWorldController class, replace the Index method with the following code:

C#Copy

public IActionResult Index()

{

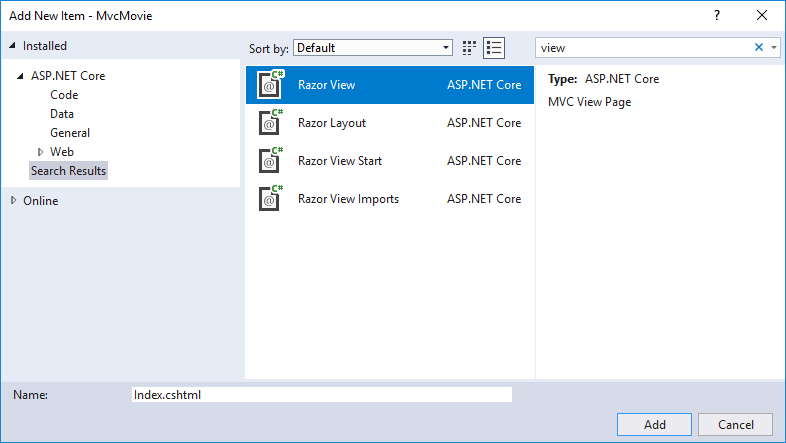
return View();

}

The preceding code returns a View object. It uses a view template to generate an HTML response to the browser. Controller methods (also known as action methods) such as the Index method above, generally return an [IActionResult](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.iactionresult) (or a class derived from ActionResult), not a type like string.

## Add a view

* [Visual Studio](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/adding-view?view=aspnetcore-2.2&tabs=visual-studio#tabpanel_CeZOj-G++Q_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 - MvcMovie** dialog
  + In the search box in the upper-right, enter view
  + Select **Razor View**
  + Keep the **Name** box value, Index.cshtml.
  + Select **Add**



* [Visual Studio Code](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/adding-view?view=aspnetcore-2.2&tabs=visual-studio#tabpanel_CeZOj-G++Q_visual-studio-code)

Add an Index view for the HelloWorldController.

* Add a new folder named *Views/HelloWorld*.
* Add a new file to the *Views/HelloWorld* folder name *Index.cshtml*.

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

HTMLCopy

@{

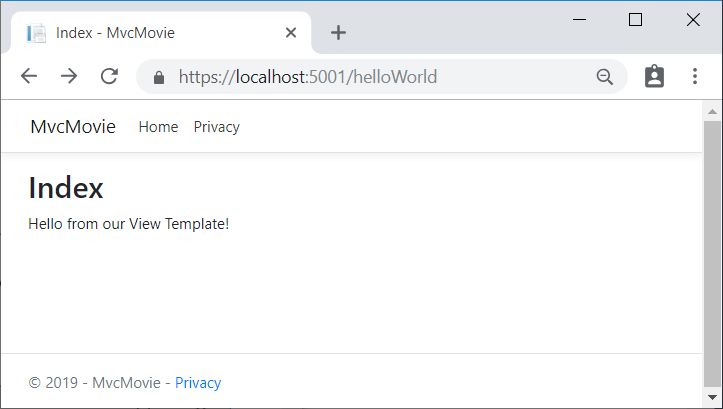
ViewData["Title"] = "Index";

}

<h2>Index</h2>

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

Navigate to https://localhost:xxxx/HelloWorld. The Index method in the HelloWorldController didn't do much; it ran the statement return View();, which specified that the method should use a view template file to render a response to the browser. Because you didn't explicitly specify the name of the view template file, MVC defaulted to using the Index.cshtml view file in the /Views/HelloWorld folder. The image below 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](https://docs.microsoft.com/en-us/aspnet/core/mvc/views/layout?view=aspnetcore-2.2) templates allow you to specify the HTML container layout of your site in one place and then apply it across multiple pages in your 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 and menu link in the layout file

* In the title element, change MvcMovie to Movie App.
* Change 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>.

The following markup shows the highlighted changes:

HTMLCopy

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8" />

<meta name="viewport" content="width=device-width, initial-scale=1.0" />

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

<environment include="Development">

<link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />

</environment>

<environment exclude="Development">

<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/twitter-bootstrap/4.1.3/css/bootstrap.min.css"

asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css"

asp-fallback-test-class="sr-only" asp-fallback-test-property="position" asp-fallback-test-value="absolute"

crossorigin="anonymous"

integrity="sha256-eSi1q2PG6J7g7ib17yAaWMcrr5GrtohYChqibrV7PBE="/>

</environment>

<link rel="stylesheet" href="~/css/site.css" />

</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">

<a class="navbar-brand" asp-controller="Movies" asp-action="Index">Movie App</a>

<button class="navbar-toggler" type="button" data-toggle="collapse" data-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 flex-sm-row-reverse">

<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">

<partial name="\_CookieConsentPartial" />

<main role="main" class="pb-3">

@RenderBody()

</main>

</div>

<footer class="border-top footer text-muted">

<div class="container">

&copy; 2019 - Movie App - <a asp-area="" asp-controller="Home" asp-action="Privacy">Privacy</a>

</div>

</footer>

<environment include="Development">

<script src="~/lib/jquery/dist/jquery.js"></script>

<script src="~/lib/bootstrap/dist/js/bootstrap.bundle.js"></script>

</environment>

<environment exclude="Development">

<script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.3.1/jquery.min.js"

asp-fallback-src="~/lib/jquery/dist/jquery.min.js"

asp-fallback-test="window.jQuery"

crossorigin="anonymous"

integrity="sha256-FgpCb/KJQlLNfOu91ta32o/NMZxltwRo8QtmkMRdAu8=">

</script>

<script src="https://cdnjs.cloudflare.com/ajax/libs/twitter-bootstrap/4.1.3/js/bootstrap.bundle.min.js"

asp-fallback-src="~/lib/bootstrap/dist/js/bootstrap.bundle.min.js"

asp-fallback-test="window.jQuery && window.jQuery.fn && window.jQuery.fn.modal"

crossorigin="anonymous"

integrity="sha256-E/V4cWE4qvAeO5MOhjtGtqDzPndRO1LBk8lJ/PR7CA4=">

</script>

</environment>

<script src="~/js/site.js" asp-append-version="true"></script>

@RenderSection("Scripts", required: false)

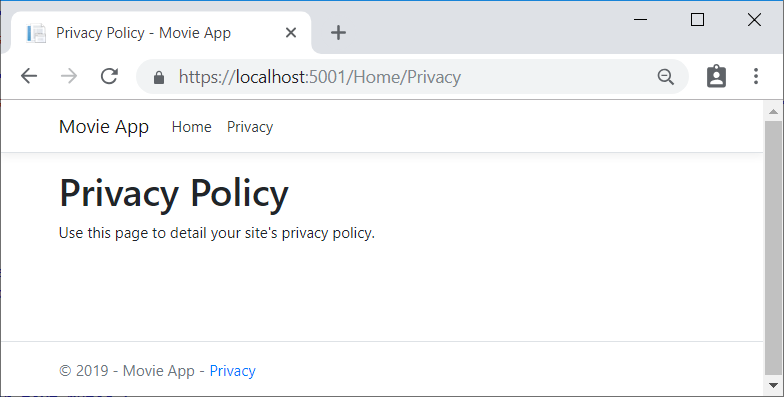
</body>

</html>

In the preceding markup, the asp-area [anchor Tag Helper attribute](https://docs.microsoft.com/en-us/aspnet/core/mvc/views/tag-helpers/built-in/anchor-tag-helper?view=aspnetcore-2.2) was omitted because this app is not using [Areas](https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/areas?view=aspnetcore-2.2).

**Note**: The Movies controller has not been implemented. At this point, the Movie App link is not functional.

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



Select the **Home** link and notice that the title and anchor text also display **Movie App**. We were able to make the change once in the layout template and have all pages on the site reflect the new link text and new title.

Examine the Views/\_ViewStart.cshtml file:

HTMLCopy

@{

Layout = "\_Layout";

}

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

Change the title and <h2> element of the Views/HelloWorld/Index.cshtml view file:

HTMLCopy

@{

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 you can see which bit of code changes the display.

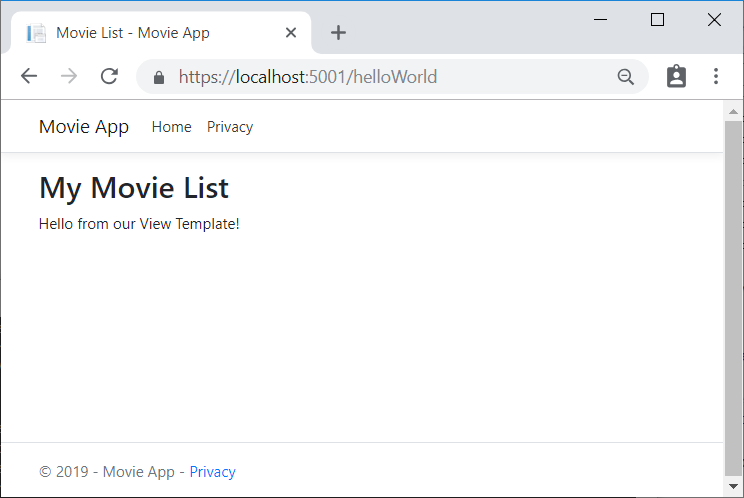
ViewData["Title"] = "Movie List"; in the code above sets the Title property of the ViewData dictionary to "Movie List". The Title property is used in the <title> HTML element in the layout page:

HTMLCopy

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

Save the change and navigate to https://localhost:xxxx/HelloWorld. Notice that the browser title, the primary heading, and the secondary headings have changed. (If you don't see changes in the browser, you might be viewing cached content. Press Ctrl+F5 in your 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.

Also notice how the content in the Index.cshtml view template was merged with the Views/Shared/\_Layout.cshtml view template and a single HTML response was sent to the browser. Layout templates make it really easy to make changes that apply across all of the pages in your application. To learn more see [Layout](https://docs.microsoft.com/en-us/aspnet/core/mvc/views/layout?view=aspnetcore-2.2).



Our little bit of "data" (in this case the "Hello from our View Template!" message) is hard-coded, though. The MVC application has a "V" (view) and you've got 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. A best practice: View templates should **not** perform business logic or interact with a database directly. Rather, 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, and maintainable.

Currently, the Welcome method in the HelloWorldController class takes a name and a 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 bits of data must be passed from the controller to the view in order to generate the response. Do this by having the controller put the dynamic data (parameters) that the view template needs in a ViewData dictionary that the view template can then access.

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 you put something inside it. The [MVC model binding system](https://docs.microsoft.com/en-us/aspnet/core/mvc/models/model-binding?view=aspnetcore-2.2)automatically maps the named parameters (name and numTimes) from the query string in the address bar to parameters in your method. The complete HelloWorldController.cs file looks like this:

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:

HTMLCopy

@{

ViewData["Title"] = "Welcome";

}

<h2>Welcome</h2>

<ul>

@for (int i = 0; i < (int)ViewData["NumTimes"]; i++)

{

<li>@ViewData["Message"]</li>

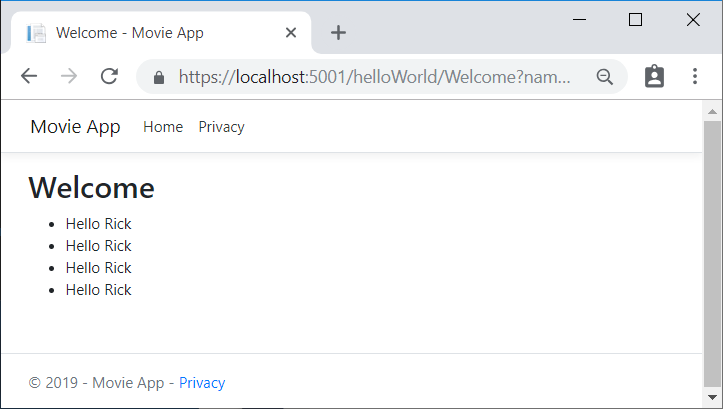
}

</ul>

Save your changes and browse to the following URL:

https://localhost:xxxx/HelloWorld/Welcome?name=Rick&numtimes=4

Data is taken from the URL and passed to the controller using the [MVC model binder](https://docs.microsoft.com/en-us/aspnet/core/mvc/models/model-binding?view=aspnetcore-2.2) . 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 sample above, 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 generally much preferred over the ViewData dictionary approach. See [When to use ViewBag, ViewData, or TempData](http://www.rachelappel.com/when-to-use-viewbag-viewdata-or-tempdata-in-asp-net-mvc-3-applications/)for more information.

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

# Add a model to an ASP.NET Core MVC app

In this section, you add classes for managing movies in a database. These classes will be the "**M**odel" part of the **M**VC app.

You use these classes with [Entity Framework Core](https://docs.microsoft.com/ef/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 you create are known as POCO classes (from **P**lain **O**ld **C**LR **O**bjects) because they don't have any dependency on EF Core. They just define the properties of the data that will be stored in the database.

In this tutorial, you write the model classes first, and EF Core creates the database. An alternate approach not covered here is to generate model classes from an existing database. For information about that approach, see [ASP.NET Core - Existing Database](https://docs.microsoft.com/ef/core/get-started/aspnetcore/existing-db).

## Add a data model class

* [Visual Studio](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/adding-model?view=aspnetcore-2.2&tabs=visual-studio#tabpanel_CeZOj-G++Q_visual-studio)

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

Add the following properties to the Movie class:

C#Copy

using System;

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:

* The Id field which is required by the database for the primary key.
* [DataType(DataType.Date)]: The [DataType](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.dataannotations.internal.datatypeattributeadapter) attribute specifies the type of the data (Date). With this attribute:
  + The user is not required to enter time information in the date field.
  + Only the date is displayed, not time information.

[DataAnnotations](https://docs.microsoft.com/en-us/dotnet/api/system.componentmodel.dataannotations) are covered in a later tutorial.

* [Visual Studio Code / Visual Studio for Mac](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/adding-model?view=aspnetcore-2.2&tabs=visual-studio#tabpanel_CeZOj-G++Q_visual-studio-code+visual-studio-mac)
* Add a class to the Models folder named Movie.cs.

Add the following properties to the Movie class:

C#Copy

using System;

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:

* The Id field which is required by the database for the primary key.
* [DataType(DataType.Date)]: The [DataType](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.dataannotations.internal.datatypeattributeadapter) attribute specifies the type of the data (Date). With this attribute:
  + The user is not required to enter time information in the date field.
  + Only the date is displayed, not time information.

[DataAnnotations](https://docs.microsoft.com/en-us/dotnet/api/system.componentmodel.dataannotations) are covered in a later tutorial.

Add the following MvcMovieContext class to the Models folder:

C#Copy

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Microsoft.EntityFrameworkCore;

namespace MvcMovie.Models

{

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 property for the entity set. In Entity Framework terminology, an entity set typically corresponds to a database table, and an entity corresponds to a row in the table.

### Add a database connection string

Add a connection string to the appsettings.json file:

JSONCopy

{

"Logging": {

"LogLevel": {

"Default": "Warning"

}

},

"AllowedHosts": "\*",

"ConnectionStrings": {

"MovieContext": "Data Source=MvcMovie.db"

}

}

### Add required NuGet packages

Run the following .NET Core CLI command to add SQLite and CodeGeneration.Design to the project:

consoleCopy

dotnet add package Microsoft.EntityFrameworkCore.SQLite

dotnet add package Microsoft.VisualStudio.Web.CodeGeneration.Design

The Microsoft.VisualStudio.Web.CodeGeneration.Design package is required for scaffolding.

### Register the database context

Add the following using statements at the top of Startup.cs:

C#Copy

using MvcMovie.Models;

using Microsoft.EntityFrameworkCore;

Register the database context with the [dependency injection](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-2.2) container in Startup.ConfigureServices.

C#Copy

public void ConfigureServices(IServiceCollection services)

{

services.Configure<CookiePolicyOptions>(options =>

{

// This lambda determines whether user consent for

// non-essential cookies is needed for a given request.

options.CheckConsentNeeded = context => true;

options.MinimumSameSitePolicy = SameSiteMode.None;

});

services.AddDbContext<MvcMovieContext>(options =>

options.UseSqlite(Configuration.GetConnectionString("MovieContext")));

services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version\_2\_2);

}

Build the project as a check for errors.

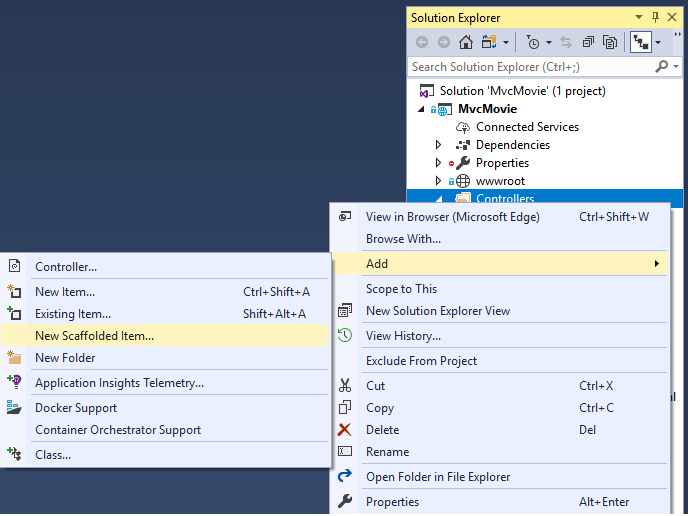
CONTINUE…

## Scaffold the movie model

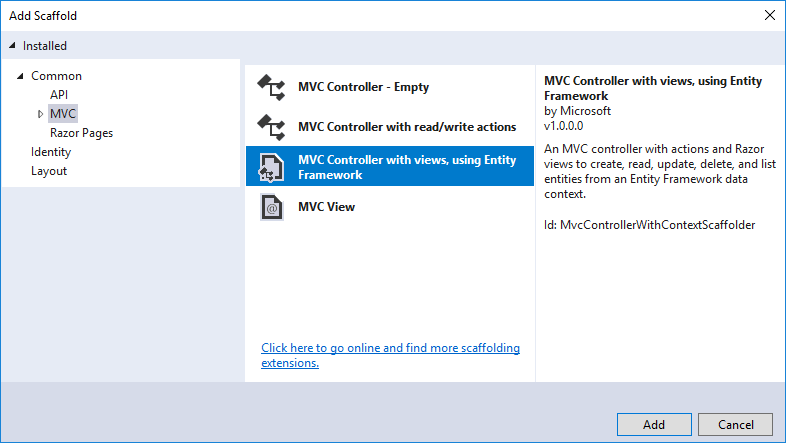
In this section, the movie model is scaffolded. That is, the scaffolding tool produces pages for Create, Read, Update, and Delete (CRUD) operations for the movie model.

* [Visual Studio](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/adding-model?view=aspnetcore-2.2&tabs=visual-studio#tabpanel_CeZOj-G++Q-1_visual-studio)

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

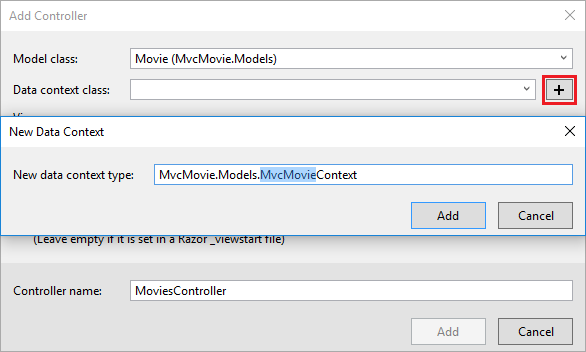


In the **Add Scaffold** dialog, select **MVC Controller with views, using Entity Framework > Add**.

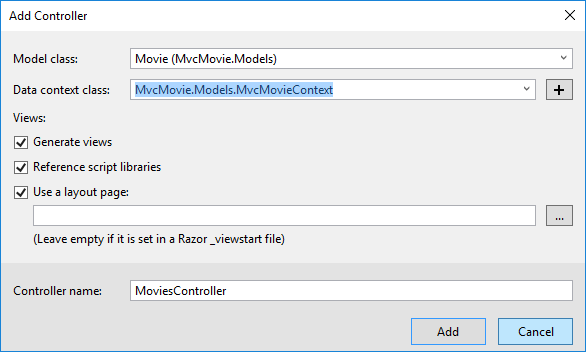


Complete the **Add Controller** dialog:

* **Model class:** Movie (MvcMovie.Models)
* **Data context class:** Select the **+** icon and add the default **MvcMovie.Models.MvcMovieContext**



* **Views:** Keep the default of each option checked
* **Controller name:** Keep the default MoviesController
* Select **Add**



Visual Studio creates:

* An Entity Framework Core [database context class](https://docs.microsoft.com/en-us/aspnet/core/data/ef-mvc/intro?view=aspnetcore-2.2#create-the-database-context) (Data/MvcMovieContext.cs)
* A movies controller (Controllers/MoviesController.cs)
* Razor view files for Create, Delete, Details, Edit, and Index pages (Views/Movies/\*.cshtml)

The automatic creation of the database context and [CRUD](https://wikipedia.org/wiki/Create,_read,_update_and_delete) (create, read, update, and delete) action methods and views is known as scaffolding.

* [Visual Studio Code](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/adding-model?view=aspnetcore-2.2&tabs=visual-studio#tabpanel_CeZOj-G++Q-1_visual-studio-code)
* Open a command window in the project directory (The directory that contains the *Program.cs*, *Startup.cs*, and *.csproj* files).
* Install the scaffolding tool:

consoleCopy

dotnet tool install --global dotnet-aspnet-codegenerator

* Run the following command:

consoleCopy

dotnet aspnet-codegenerator controller -name MoviesController -m Movie -dc MvcMovieContext --relativeFolderPath Controllers --useDefaultLayout --referenceScriptLibraries

The following table details the ASP.NET Core code generator parameters:

| **Parameter** | **Description** |
| --- | --- |
| -m | The name of the model. |
| -dc | The data context. |
| -udl | Use the default layout. |
| --relativeFolderPath | The relative output folder path to create the views. |
| --useDefaultLayout | The default layout should be used for the views. |
| --referenceScriptLibraries | Adds \_ValidationScriptsPartial to Edit and Create pages |

Use the h switch to get help on the aspnet-codegenerator controller command:

consoleCopy

dotnet aspnet-codegenerator controller -h

CONTINUE…

If you run the app and click on the **Mvc Movie** link, you get an error similar to the following:

errorCopy

An unhandled exception occurred while processing the request.

SqlException: Cannot open database "MvcMovieContext-<GUID removed>" requested by the login. The login failed.

Login failed for user 'Rick'.

System.Data.SqlClient.SqlInternalConnectionTds..ctor(DbConnectionPoolIdentity identity, SqlConnectionString

You need to create the database, and you use the EF Core [Migrations](https://docs.microsoft.com/en-us/aspnet/core/data/ef-mvc/migrations?view=aspnetcore-2.2) feature to do that. Migrations lets you create a database that matches your data model and update the database schema when your data model changes.

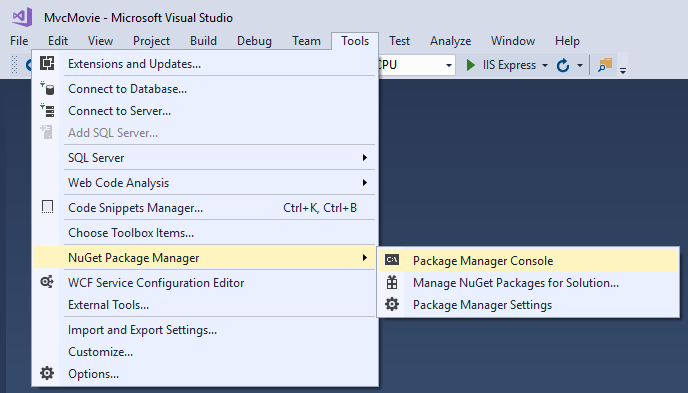
## Initial migration

* [Visual Studio](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/adding-model?view=aspnetcore-2.2&tabs=visual-studio#tabpanel_CeZOj-G++Q-2_visual-studio)

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

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

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



In the PMC, enter the following commands:

PMCCopy

Add-Migration Initial

Update-Database

The Add-Migration command generates code to create the initial database schema.

* [Visual Studio Code / Visual Studio for Mac](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/adding-model?view=aspnetcore-2.2&tabs=visual-studio#tabpanel_CeZOj-G++Q-2_visual-studio-code+visual-studio-mac)

Run the following .NET Core CLI commands:

consoleCopy

dotnet ef migrations add InitialCreate

dotnet ef database update

The ef migrations add InitialCreate command generates code to create the initial database schema.

CONTINUE…

The schema is based on the model specified in the DbContext (In the Models/MvcMovieContext.cs file). The InitialCreate argument is used to name the migrations. Any name can be used, but by convention a name is selected that describes the migration.

The ef database update command runs the Up method in the Migrations/<time-stamp>\_InitialCreate.cs file. The Up method creates the database.

* [Visual Studio](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/adding-model?view=aspnetcore-2.2&tabs=visual-studio#tabpanel_CeZOj-G++Q-3_visual-studio)

## Examine the context registered with dependency injection

ASP.NET Core is built with [dependency injection](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-2.2). Services (such as the EF Core DB context) are registered with dependency injection during application startup. Components that require these services (such as Razor Pages) are provided these services via constructor parameters. The constructor code that gets a DB context instance is shown later in the tutorial.

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

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

C#Copy

public void ConfigureServices(IServiceCollection services)

{

services.Configure<CookiePolicyOptions>(options =>

{

// This lambda determines whether user consent for non-essential cookies

// is needed for a given request.

options.CheckConsentNeeded = context => true;

options.MinimumSameSitePolicy = SameSiteMode.None;

});

services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version\_2\_2);

services.AddDbContext<MvcMovieContext>(options =>

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

}

The MvcMovieContext coordinates EF Core functionality (Create, Read, Update, Delete, etc.) for the Moviemodel. The data context (MvcMovieContext) is derived from [Microsoft.EntityFrameworkCore.DbContext](https://docs.microsoft.com/en-us/dotnet/api/microsoft.entityframeworkcore.dbcontext). The data context specifies which entities are included in the data model:

C#Copy

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Microsoft.EntityFrameworkCore;

namespace MvcMovie.Models

{

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>](https://docs.microsoft.com/en-us/dotnet/api/microsoft.entityframeworkcore.dbset-1) property for the entity set. In Entity Framework terminology, an entity set typically corresponds to a database table. An entity corresponds to a row in the table.

The name of the connection string is passed in to the context by calling a method on a [DbContextOptions](https://docs.microsoft.com/en-us/dotnet/api/microsoft.entityframeworkcore.dbcontextoptions)object. For local development, the [ASP.NET Core configuration system](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration/index?view=aspnetcore-2.2) reads the connection string from the appsettings.json file.

* [Visual Studio Code / Visual Studio for Mac](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/adding-model?view=aspnetcore-2.2&tabs=visual-studio#tabpanel_CeZOj-G++Q-3_visual-studio-code+visual-studio-mac)

ASP.NET Core is built with [dependency injection](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-2.2). Services (such as the EF Core DB context) are registered with dependency injection during application startup. Components that require these services (such as Razor Pages) are provided these services via constructor parameters. The constructor code that gets a DB context instance is shown later in the tutorial.

You created a DB context and registered it with the dependency injection container.

CONTINUE…

The schema is based on the model specified in the MvcMovieContext (In the Data/MvcMovieContext.cs file). The Initial argument is used to name the migrations. Any name can be used, but by convention a name that describes the migration is used. See [Introduction to migrations](https://docs.microsoft.com/en-us/aspnet/core/data/ef-mvc/migrations?view=aspnetcore-2.2#introduction-to-migrations) for more information.

The Update-Database command runs the Up method in the Migrations/{time-stamp}\_InitialCreate.cs file, which creates the database.

### Test the app

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

If you get a database exception similar to the following:

consoleCopy

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

Login failed for user 'User-name'.

You missed the [migrations step](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/adding-model?view=aspnetcore-2.2&tabs=visual-studio#pmc).

* Test the **Create** link.

**Note**

You may not be able to enter decimal commas in the Price field. To support [**jQuery validation**](https://jqueryvalidation.org/) 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**](https://github.com/aspnet/Docs/issues/4076#issuecomment-326590420).

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

Examine the Startup class:

C#Copy

public void ConfigureServices(IServiceCollection services)

{

services.Configure<CookiePolicyOptions>(options =>

{

// This lambda determines whether user consent for non-essential cookies

// is needed for a given request.

options.CheckConsentNeeded = context => true;

options.MinimumSameSitePolicy = SameSiteMode.None;

});

services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version\_2\_2);

services.AddDbContext<MvcMovieContext>(options =>

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

}

The preceding highlighted code shows the movie database context being added to the [Dependency Injection](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-2.2)container:

* services.AddDbContext<MvcMovieContext>(options => specifies the database to use and the connection string.
* => is a [lambda operator](https://docs.microsoft.com/en-us/dotnet/articles/csharp/language-reference/operators/lambda-operator)

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](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-2.2) to inject the database context (MvcMovieContext) into the controller. The database context is used in each of the [CRUD](https://wikipedia.org/wiki/Create,_read,_update_and_delete) methods in the controller.

## Strongly typed models and the @model keyword

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

MVC also provides the ability to pass strongly typed model objects to a view. This strongly typed approach enables better compile time checking of your code. The scaffolding mechanism used this approach (that is, passing a strongly typed model) with the MoviesController class and views when it created the methods 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/1sets:

* 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).

You can also pass in the id with a query string as follows:

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

The id parameter is defined as a [nullable type](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/nullable-types/index) (int?) in case an ID value isn't provided.

A [lambda expression](https://docs.microsoft.com/en-us/dotnet/articles/csharp/programming-guide/statements-expressions-operators/lambda-expressions) is passed in to FirstOrDefaultAsync 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:

HTMLCopy

@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>

By including a @model statement at the top of the view file, you can specify the type of object that the view expects. When you created the movie controller, the following @model statement was automatically included at the top of the Details.cshtml file:

HTMLCopy

@model MvcMovie.Models.Movie

This @model directive allows you to access the movie that the controller passed to the view by using a Modelobject that's 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());

}

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

HTMLCopy

@model IEnumerable<MvcMovie.Models.Movie>

The @model directive allows you to access 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:

HTMLCopy

@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, this means that you get compile time checking of the code:

## Additional resources

* [Tag Helpers](https://docs.microsoft.com/en-us/aspnet/core/mvc/views/tag-helpers/intro?view=aspnetcore-2.2)
* [Globalization and localization](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/localization?view=aspnetcore-2.2)

# Work with SQL in ASP.NET Core

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](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-2.2) container in the ConfigureServices method in the Startup.cs file:

* [Visual Studio](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/working-with-sql?view=aspnetcore-2.2&tabs=visual-studio#tabpanel_CeZOj-G++Q_visual-studio)

C#Copy

public void ConfigureServices(IServiceCollection services)

{

services.Configure<CookiePolicyOptions>(options =>

{

// This lambda determines whether user consent for non-essential cookies

// is needed for a given request.

options.CheckConsentNeeded = context => true;

options.MinimumSameSitePolicy = SameSiteMode.None;

});

services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version\_2\_2);

services.AddDbContext<MvcMovieContext>(options =>

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

}

The ASP.NET Core [Configuration](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration/index?view=aspnetcore-2.2) system reads the ConnectionString. For local development, it gets the connection string from the appsettings.json file:

JSONCopy

"ConnectionStrings": {

"MvcMovieContext": "Server=(localdb)\\mssqllocaldb;Database=MvcMovieContext-2;Trusted\_Connection=True;MultipleActiveResultSets=true"

}

* [Visual Studio Code / Visual Studio for Mac](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/working-with-sql?view=aspnetcore-2.2&tabs=visual-studio#tabpanel_CeZOj-G++Q_visual-studio-code+visual-studio-mac)

C#Copy

public void ConfigureServices(IServiceCollection services)

{

services.Configure<CookiePolicyOptions>(options =>

{

// This lambda determines whether user consent for

// non-essential cookies is needed for a given request.

options.CheckConsentNeeded = context => true;

options.MinimumSameSitePolicy = SameSiteMode.None;

});

services.AddDbContext<MvcMovieContext>(options =>

options.UseSqlite(Configuration.GetConnectionString("MovieContext")));

services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version\_2\_2);

}

The ASP.NET Core [Configuration](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration/index?view=aspnetcore-2.2) system reads the ConnectionString. For local development, it gets the connection string from the appsettings.json file:

JSONCopy

"ConnectionStrings": {

"MovieContext": "Data Source=MvcMovie.db"

}

CONTINUE…

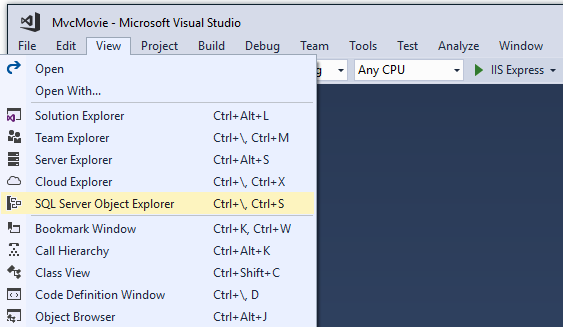
When you deploy the app to a test or production server, you can use an environment variable or another approach to set the connection string to a real SQL Server. See [Configuration](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration/index?view=aspnetcore-2.2) for more information.

* [Visual Studio](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/working-with-sql?view=aspnetcore-2.2&tabs=visual-studio#tabpanel_CeZOj-G++Q-1_visual-studio)

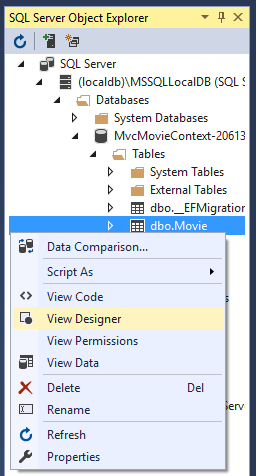
## SQL Server Express LocalDB

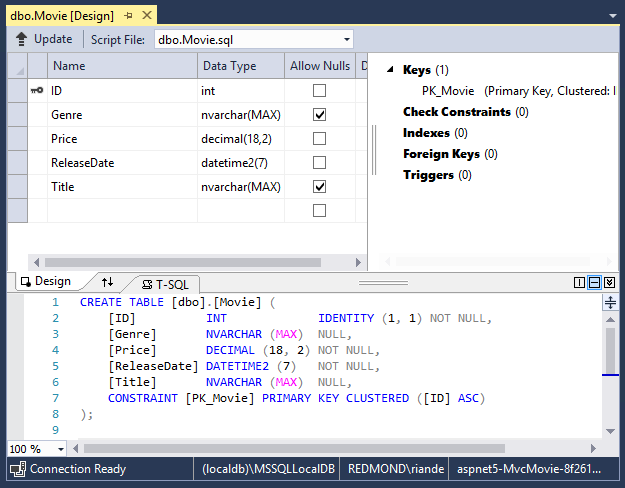
LocalDB is a lightweight version of the SQL Server Express Database Engine that's targeted for program development. LocalDB starts on demand and runs in user mode, so there's no complex configuration. By default, LocalDB database creates .mdf files in the C:/Users/{user} directory.

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



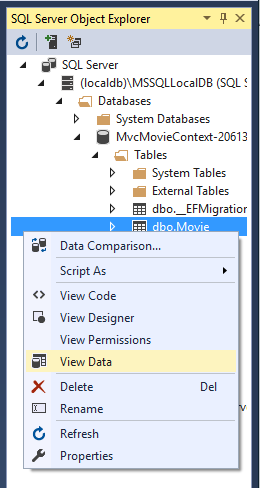
* Right click on the Movie table **> View Designer**

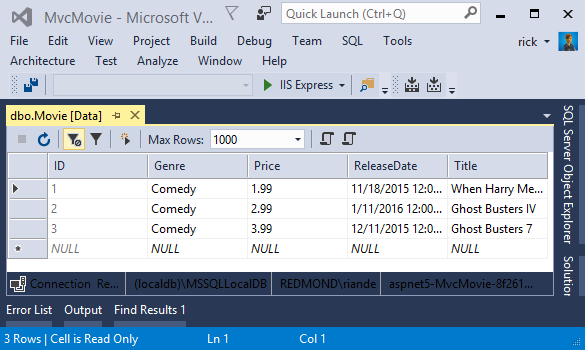




Note the key icon next to ID. By default, EF will make a property named ID the primary key.

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





* [Visual Studio Code / Visual Studio for Mac](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/working-with-sql?view=aspnetcore-2.2&tabs=visual-studio#tabpanel_CeZOj-G++Q-1_visual-studio-code+visual-studio-mac)

## SQLite

The [SQLite](https://www.sqlite.org/) website states:

SQLite is a self-contained, high-reliability, embedded, full-featured, public-domain, SQL database engine. SQLite is the most used database engine in the world.

There are many third party tools you can download to manage and view a SQLite database. The image below is from [DB Browser for SQLite](http://sqlitebrowser.org/).

CONTINUE…

## 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 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 DB, the seed initializer returns and no movies are added.

C#Copy

if (context.Movie.Any())

{

return; // DB has been seeded.

}

### Add the seed initializer

Replace the contents of Program.cs with the following code:

C#Copy

using Microsoft.AspNetCore;

using Microsoft.AspNetCore.Hosting;

using Microsoft.Extensions.DependencyInjection;

using Microsoft.Extensions.Logging;

using System;

using Microsoft.EntityFrameworkCore;

using MvcMovie.Models;

using MvcMovie;

namespace MvcMovie

{

public class Program

{

public static void Main(string[] args)

{

var host = CreateWebHostBuilder(args).Build();

using (var scope = host.Services.CreateScope())

{

var services = scope.ServiceProvider;

try

{

var context = services.GetRequiredService<MvcMovieContext>();

context.Database.Migrate();

SeedData.Initialize(services);

}

catch (Exception ex)

{

var logger = services.GetRequiredService<ILogger<Program>>();

logger.LogError(ex, "An error occurred seeding the DB.");

}

}

host.Run();

}

public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>

WebHost.CreateDefaultBuilder(args)

.UseStartup<Startup>();

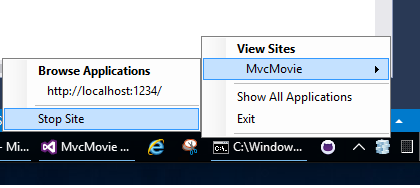
}

}

Test the app

* [Visual Studio](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/working-with-sql?view=aspnetcore-2.2&tabs=visual-studio#tabpanel_CeZOj-G++Q-2_visual-studio)
* Delete all the records in the DB. You can do this with the delete links in the browser or from SSOX.
* Force the app to initialize (call the methods in the Startup class) so the seed method runs. To force initialization, IIS Express must be stopped and restarted. You can do this with any of the following approaches:
  + Right click the IIS Express system tray icon in the notification area and tap **Exit** or **Stop Site**



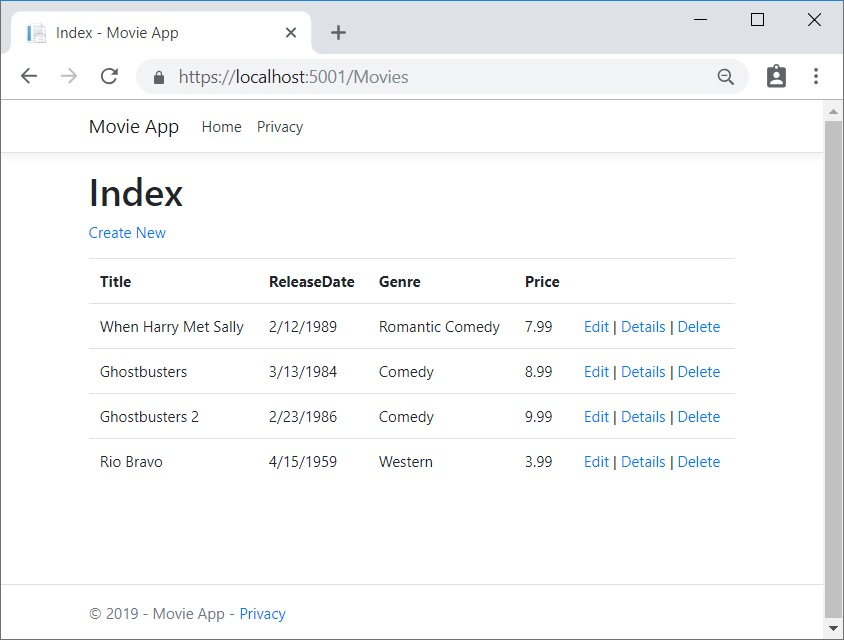


* + - If you were running VS in non-debug mode, press F5 to run in debug mode
    - If you were running VS in debug mode, stop the debugger and press F5
* [Visual Studio Code / Visual Studio for Mac](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/working-with-sql?view=aspnetcore-2.2&tabs=visual-studio#tabpanel_CeZOj-G++Q-2_visual-studio-code+visual-studio-mac)

Delete all the records in the DB (So the seed method will run). Stop and start the app to seed the database.

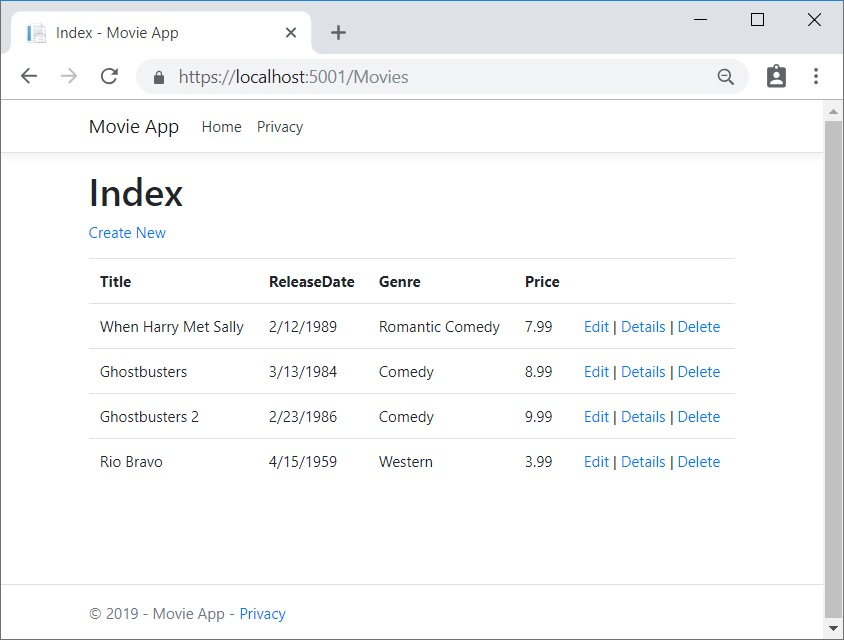
CONTINUE…

The app shows the seeded data.



# Controller methods and views in ASP.NET Core

We have a good start to the movie app, but the presentation isn't ideal. We don't want to see the time (12:00:00 AM in the image below) and **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; }

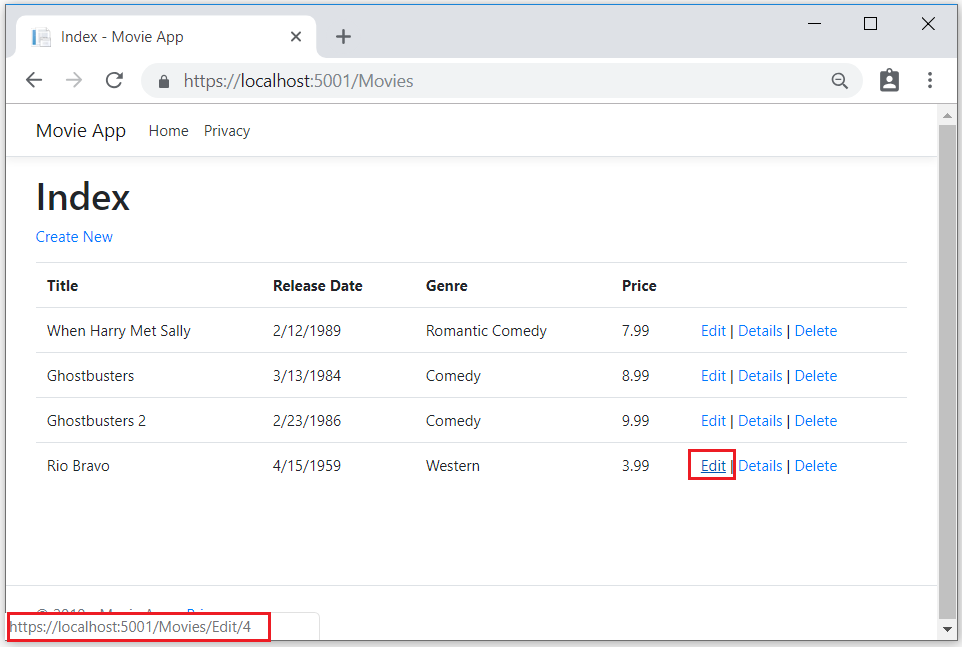
}

}

We cover [DataAnnotations](https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions/mvc-music-store/mvc-music-store-part-6) in the next tutorial. The [Display](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.modelbinding.metadata.displaymetadata) attribute specifies what to display for the name of a field (in this case "Release Date" instead of "ReleaseDate"). The [DataType](https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.dataannotations.internal.datatypeattributeadapter) 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](https://docs.microsoft.com/ef/core/modeling/relational/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.

HTMLCopy

<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](https://docs.microsoft.com/en-us/aspnet/core/mvc/views/tag-helpers/intro?view=aspnetcore-2.2) 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](https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/routing?view=aspnetcore-2.2) set in the Startup.cs file:

C#Copy

app.UseMvc(routes =>

{

routes.MapRoute(

name: "default",

template: "{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](https://docs.microsoft.com/en-us/aspnet/core/mvc/views/tag-helpers/intro?view=aspnetcore-2.2) are one of the most popular new features in ASP.NET Core. For more information, see [Additional resources](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/controller-methods-views?view=aspnetcore-2.2#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.cshtmlRazor 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, please 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")] 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("Index");

}

return View(movie);

}

The [Bind] attribute is one way to protect against [over-posting](https://docs.microsoft.com/en-us/aspnet/mvc/overview/getting-started/getting-started-with-ef-using-mvc/implementing-basic-crud-functionality-with-the-entity-framework-in-asp-net-mvc-application#overpost). You should only include properties in the [Bind] attribute that you want to change. For more information, see [Protect your controller from over-posting](https://docs.microsoft.com/en-us/aspnet/mvc/overview/getting-started/getting-started-with-ef-using-mvc/implementing-basic-crud-functionality-with-the-entity-framework-in-asp-net-mvc-application). [ViewModels](http://rachelappel.com/use-viewmodels-to-manage-data-amp-organize-code-in-asp-net-mvc-applications/) provide an alternative approach to prevent over-posting.

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

C#Copy

[HttpPost]

[ValidateAntiForgeryToken]

public async Task<IActionResult> Edit(int id, [Bind("ID,Title,ReleaseDate,Genre,Price")] 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](https://docs.microsoft.com/en-us/aspnet/core/security/anti-request-forgery?view=aspnetcore-2.2) 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](https://docs.microsoft.com/en-us/aspnet/core/mvc/views/working-with-forms?view=aspnetcore-2.2).

HTMLCopy

<form asp-action="Edit">

The [Form Tag Helper](https://docs.microsoft.com/en-us/aspnet/core/mvc/views/working-with-forms?view=aspnetcore-2.2) 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 [Anti-Request Forgery](https://docs.microsoft.com/en-us/aspnet/core/security/anti-request-forgery?view=aspnetcore-2.2).

The HttpGet Edit method takes the movie ID parameter, looks up the movie using the Entity Framework SingleOrDefaultAsync 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:

HTMLCopy

@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](https://docs.microsoft.com/en-us/aspnet/core/mvc/views/working-with-forms?view=aspnetcore-2.2)displays the name of the field ("Title", "ReleaseDate", "Genre", or "Price"). The [Input Tag Helper](https://docs.microsoft.com/en-us/aspnet/core/mvc/views/working-with-forms?view=aspnetcore-2.2) renders an HTML <input> element. The [Validation Tag Helper](https://docs.microsoft.com/en-us/aspnet/core/mvc/views/working-with-forms?view=aspnetcore-2.2) 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="CfDJ8Inyxgp63fRFqUePGvuI5jGZsloJu1L7X9le1gy7NCIlSduCRx9jDQClrV9pOTTmqUyXnJBXhmrjcUVDJyDUMm7-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](https://docs.microsoft.com/en-us/aspnet/core/security/anti-request-forgery?view=aspnetcore-2.2) token generated by the [Form Tag Helper](https://docs.microsoft.com/en-us/aspnet/core/mvc/views/working-with-forms?view=aspnetcore-2.2).

## Processing the POST Request

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

C#Copy

[HttpPost]

[ValidateAntiForgeryToken]

public async Task<IActionResult> Edit(int id, [Bind("ID,Title,ReleaseDate,Genre,Price")] 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));

}

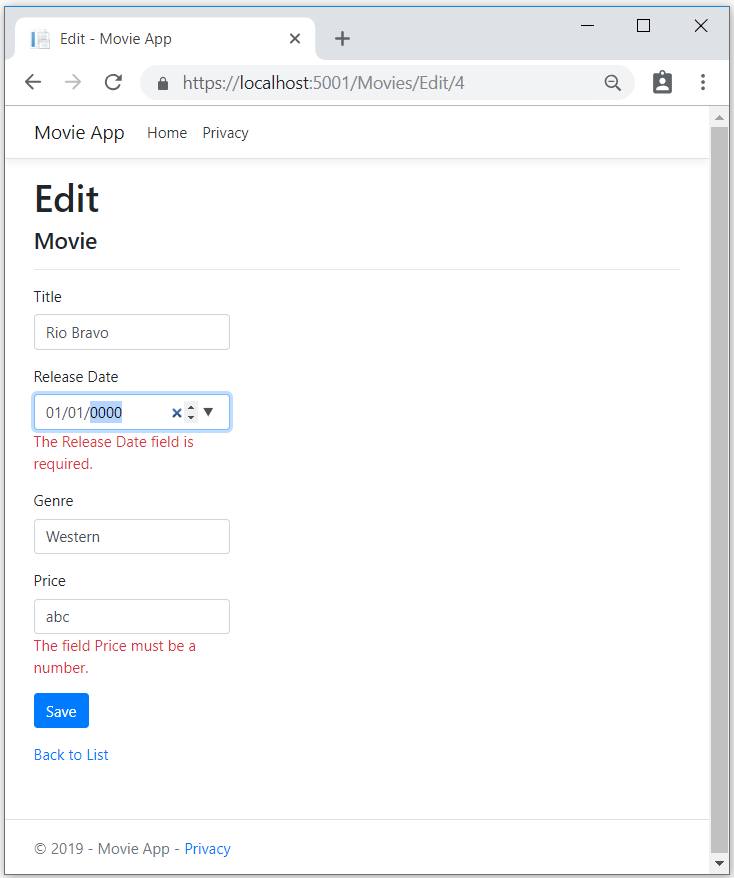
return View(movie);

}

The [ValidateAntiForgeryToken] attribute validates the hidden [XSRF](https://docs.microsoft.com/en-us/aspnet/core/security/anti-request-forgery?view=aspnetcore-2.2) token generated by the anti-forgery token generator in the [Form Tag Helper](https://docs.microsoft.com/en-us/aspnet/core/mvc/views/working-with-forms?view=aspnetcore-2.2)

The [model binding](https://docs.microsoft.com/en-us/aspnet/core/mvc/models/model-binding?view=aspnetcore-2.2) system takes the posted form values and creates a Movie object that's passed as the movie parameter. The ModelState.IsValid method 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](https://docs.microsoft.com/en-us/aspnet/core/mvc/models/validation?view=aspnetcore-2.2) in more detail. The [Validation Tag Helper](https://docs.microsoft.com/en-us/aspnet/core/mvc/views/working-with-forms?view=aspnetcore-2.2) 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](http://rest.elkstein.org/) 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](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/localization?view=aspnetcore-2.2)
* [Introduction to Tag Helpers](https://docs.microsoft.com/en-us/aspnet/core/mvc/views/tag-helpers/intro?view=aspnetcore-2.2)
* [Author Tag Helpers](https://docs.microsoft.com/en-us/aspnet/core/mvc/views/tag-helpers/authoring?view=aspnetcore-2.2)
* [Anti-Request Forgery](https://docs.microsoft.com/en-us/aspnet/core/security/anti-request-forgery?view=aspnetcore-2.2)
* Protect your controller from [over-posting](https://docs.microsoft.com/en-us/aspnet/mvc/overview/getting-started/getting-started-with-ef-using-mvc/implementing-basic-crud-functionality-with-the-entity-framework-in-asp-net-mvc-application)
* [ViewModels](http://rachelappel.com/use-viewmodels-to-manage-data-amp-organize-code-in-asp-net-mvc-applications/)
* [Form Tag Helper](https://docs.microsoft.com/en-us/aspnet/core/mvc/views/working-with-forms?view=aspnetcore-2.2)
* [Input Tag Helper](https://docs.microsoft.com/en-us/aspnet/core/mvc/views/working-with-forms?view=aspnetcore-2.2)
* [Label Tag Helper](https://docs.microsoft.com/en-us/aspnet/core/mvc/views/working-with-forms?view=aspnetcore-2.2)
* [Select Tag Helper](https://docs.microsoft.com/en-us/aspnet/core/mvc/views/working-with-forms?view=aspnetcore-2.2)
* [Validation Tag Helper](https://docs.microsoft.com/en-us/aspnet/core/mvc/views/working-with-forms?view=aspnetcore-2.2)

# Add search to an ASP.NET Core MVC app

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 with the following code:

C#Copy

public async Task<IActionResult> Index(string searchString)

{

var movies = from m in \_context.Movie

select m;

if (!String.IsNullOrEmpty(searchString))

{

movies = movies.Where(s => s.Title.Contains(searchString));

}

return View(await movies.ToListAsync());

}

The first line of the Index action method creates a [LINQ](https://docs.microsoft.com/en-us/dotnet/standard/using-linq) query to select the movies:

C#Copy

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#Copy

if (!String.IsNullOrEmpty(searchString))

{

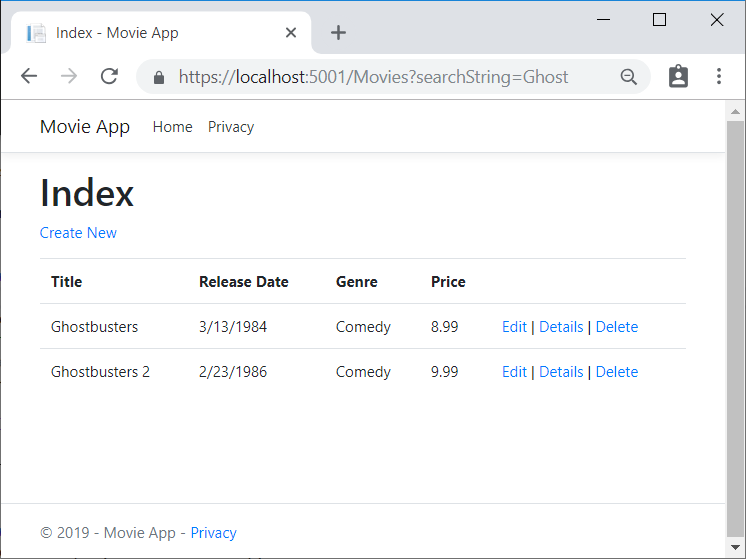
movies = movies.Where(s => s.Title.Contains(searchString));

}

The s => s.Title.Contains() code above is a [Lambda Expression](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/statements-expressions-operators/lambda-expressions). Lambdas are used in method-based [LINQ](https://docs.microsoft.com/en-us/dotnet/standard/using-linq)queries as arguments to standard query operator methods such as the [Where](https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.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 ToListAsyncmethod is called. For more information about deferred query execution, see [Query Execution](https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/ef/language-reference/query-execution).

Note: The [Contains](https://docs.microsoft.com/en-us/dotnet/api/system.data.objects.dataclasses.entitycollection-1.contains) method is run on the database, not in the c# code shown above. The case sensitivity on the query depends on the database and the collation. On SQL Server, [Contains](https://docs.microsoft.com/en-us/dotnet/api/system.data.objects.dataclasses.entitycollection-1.contains) maps to [SQL LIKE](https://docs.microsoft.com/sql/t-sql/language-elements/like-transact-sql), which is case insensitive. In SQLlite, with the default collation, it's case sensitive.

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



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 Startup.cs.

C#Copy

app.UseMvc(routes =>

{

routes.MapRoute(

name: "default",

template: "{controller=Home}/{action=Index}/{id?}");

});

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

The previous Index method:

C#Copy

public async Task<IActionResult> Index(string searchString)

{

var movies = from m in \_context.Movie

select m;

if (!String.IsNullOrEmpty(searchString))

{

movies = movies.Where(s => s.Title.Contains(searchString));

}

return View(await movies.ToListAsync());

}

The updated Index method with id parameter:

C#Copy

public async Task<IActionResult> Index(string id)

{

var movies = from m in \_context.Movie

select m;

if (!String.IsNullOrEmpty(id))

{

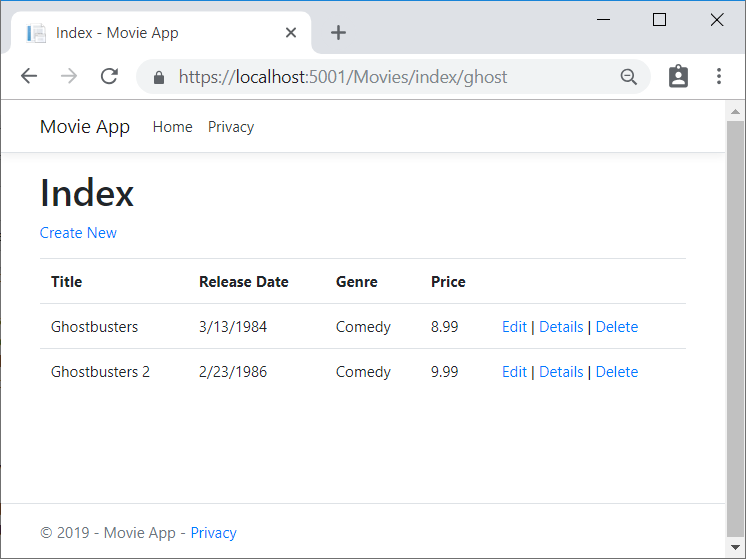
movies = movies.Where(s => s.Title.Contains(id));

}

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.



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#Copy

public async Task<IActionResult> Index(string searchString)

{

var movies = from m in \_context.Movie

select m;

if (!String.IsNullOrEmpty(searchString))

{

movies = movies.Where(s => s.Title.Contains(searchString));

}

return View(await movies.ToListAsync());

}

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

HTMLCopy

ViewData["Title"] = "Index";

}

<h2>Index</h2>

<p>

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

</p>

<form asp-controller="Movies" asp-action="Index">

<p>

Title: <input type="text" name="SearchString">

<input type="submit" value="Filter" />

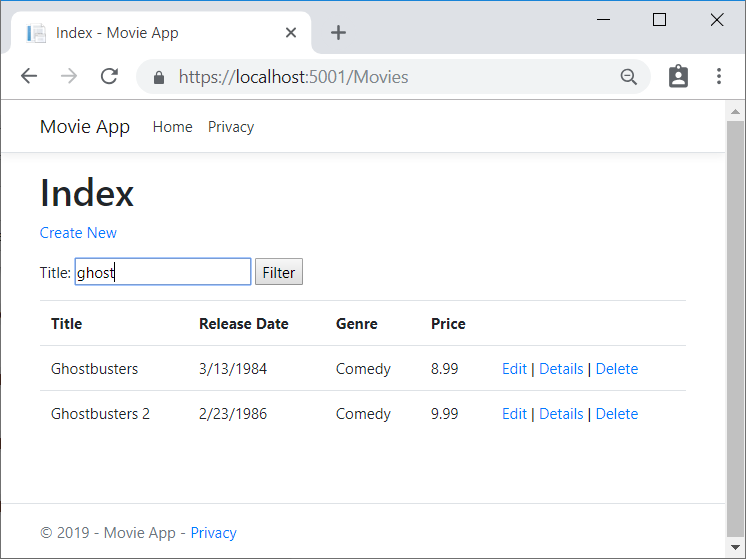
</p>

</form>

<table class="table">

<thead>

The HTML <form> tag uses the [Form Tag Helper](https://docs.microsoft.com/en-us/aspnet/core/mvc/views/working-with-forms?view=aspnetcore-2.2), 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.



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#Copy

[HttpPost]

public string Index(string searchString, bool notUsed)

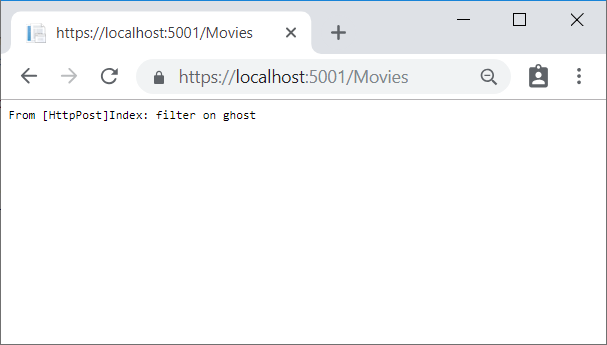
{

return "From [HttpPost]Index: filter on " + searchString;

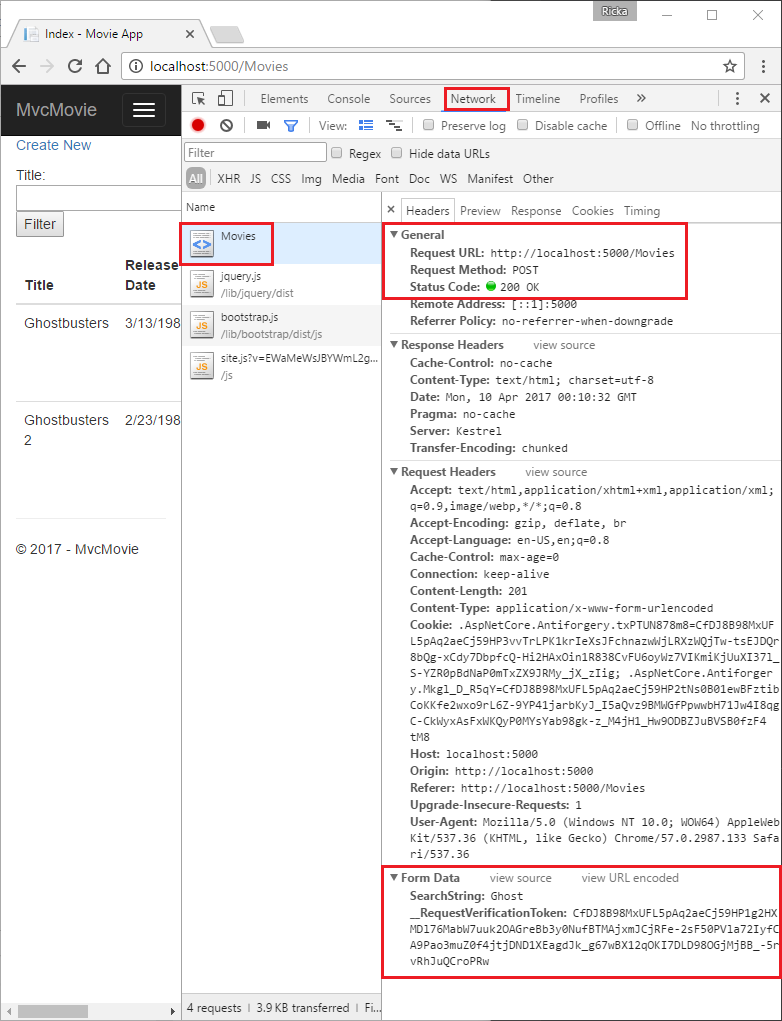
}

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.



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:xxxxx/Movies/Index) -- there's no search information in the URL. The search string information is sent to the server as a [form field value](https://developer.mozilla.org/docs/Learn/HTML/Forms/Sending_and_retrieving_form_data). You can verify that with the browser Developer tools or the excellent [Fiddler tool](http://www.telerik.com/fiddler). The image below shows the Chrome browser Developer tools:



You can see the search parameter and [XSRF](https://docs.microsoft.com/en-us/aspnet/core/security/anti-request-forgery?view=aspnetcore-2.2) token in the request body. Note, as mentioned in the previous tutorial, the [Form Tag Helper](https://docs.microsoft.com/en-us/aspnet/core/mvc/views/working-with-forms?view=aspnetcore-2.2) generates an [XSRF](https://docs.microsoft.com/en-us/aspnet/core/security/anti-request-forgery?view=aspnetcore-2.2) anti-forgery 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:

HTMLCopy

@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>

Title: <input type="text" name="SearchString">

<input type="submit" value="Filter" />

</p>

</form>

<table class="table">

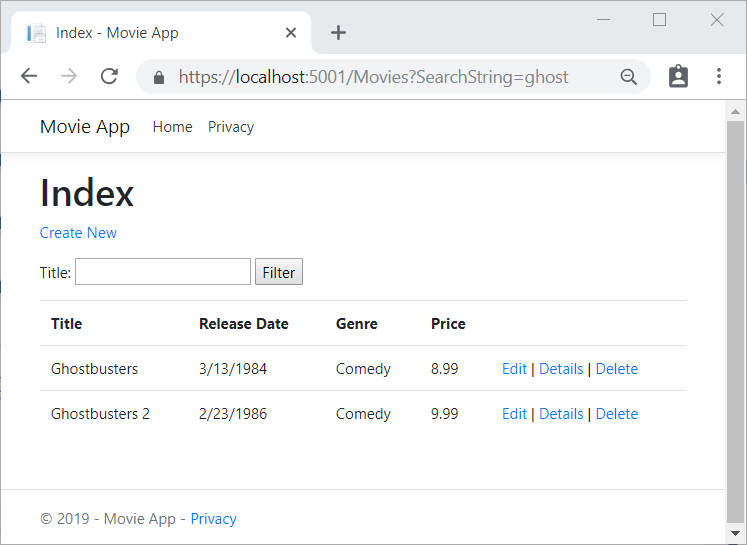
<thead>

<tr>

<th>

@Html.DisplayNameFor(model => model.Title)

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.



The following markup shows the change to the form tag:

HTMLCopy

<form asp-controller="Movies" asp-action="Index" method="get">

## Add Search by genre

Add the following MovieGenreViewModel class to the Models folder:

C#Copy

using Microsoft.AspNetCore.Mvc.Rendering;

using System.Collections.Generic;

namespace MvcMovie.Models

{

public class MovieGenreViewModel

{

public List<Movie> Movies;

public SelectList Genres;

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#Copy

// GET: Movies

public async Task<IActionResult> Index(string movieGenre, string searchString)

{

// 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.Contains(searchString));

}

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#Copy

// 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 as follows:

HTMLCopy

@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>

Title: <input type="text" asp-for="SearchString" />

<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>

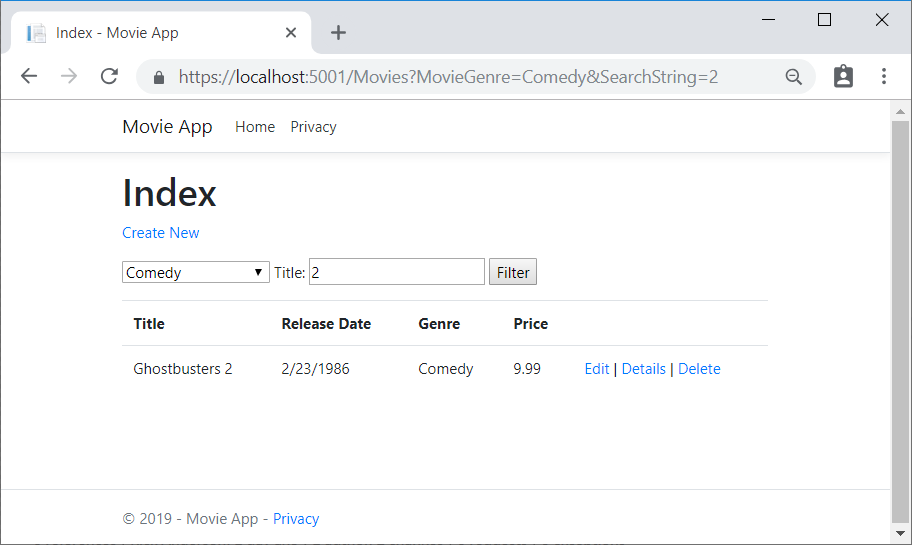
</table>

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 nullor empty. When the lambda expression is evaluated (for example, @Html.DisplayFor(modelItem => item.Title)), the model's property values are evaluated.

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



# Add a new field to an ASP.NET Core MVC app

In this section [Entity Framework](https://docs.microsoft.com/ef/core/get-started/aspnetcore/new-db) Code First Migrations is used to:

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

When EF Code First is used to automatically create a database, Code First:

* Adds a table to the database to track the schema of the database.
* Verify the database is 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#Copy

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 (Ctrl+Shift+B).

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

C#Copy

[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:

HTMLCopy

<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>

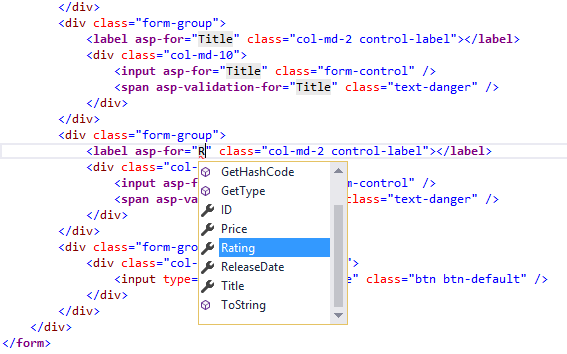
<td>

<a asp-action="Edit" asp-route-id="@item.Id">Edit</a> |

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

* [Visual Studio / Visual Studio for Mac](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/new-field?view=aspnetcore-2.2&tabs=visual-studio#tabpanel_CeZOj-G++Q_visual-studio+visual-studio-mac)

You can copy/paste the previous "form group" and let intelliSense help you update the fields. IntelliSense works with [Tag Helpers](https://docs.microsoft.com/en-us/aspnet/core/mvc/views/tag-helpers/intro?view=aspnetcore-2.2).



* [Visual Studio Code](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/new-field?view=aspnetcore-2.2&tabs=visual-studio#tabpanel_CeZOj-G++Q_visual-studio-code) (N/A)

CONTINUE…

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#Copy

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.)

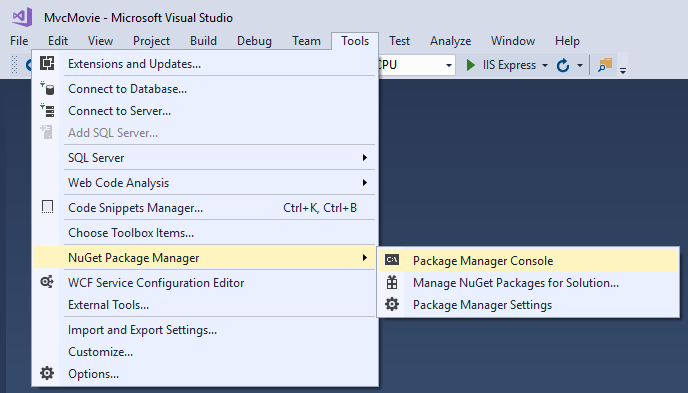
There are a few approaches to resolving the error:

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 Code First Migrations to update the database schema.

For this tutorial, Code First Migrations is used.

* [Visual Studio](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/new-field?view=aspnetcore-2.2&tabs=visual-studio#tabpanel_CeZOj-G++Q-1_visual-studio)

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



In the PMC, enter the following commands:

PowerShellCopy

Add-Migration Rating

Update-Database

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.

* [Visual Studio Code / Visual Studio for Mac](https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/new-field?view=aspnetcore-2.2&tabs=visual-studio#tabpanel_CeZOj-G++Q-1_visual-studio-code+visual-studio-mac)

**Note**

Many schema change operations are not supported by the EF Core SQLite provider. For example, adding a column is supported, but removing a column is not supported. If a migration is created to remove a column, the ef migrations add command succeeds but the ef database update command fails. Some of these limitations can be overcome by manually writing migrations code to perform a table rebuild. A table rebuild involves:

* Renaming the existing table.
* Creating a new table.
* Copying data from the old table to the new table.
* Dropping the old table.

For more information, see the following resources:

* [SQLite EF Core Database Provider Limitations](https://docs.microsoft.com/ef/core/providers/sqlite/limitations)
* [Customize migration code](https://docs.microsoft.com/ef/core/managing-schemas/migrations/#customize-migration-code)
* [Data seeding](https://docs.microsoft.com/ef/core/modeling/data-seeding)

Run the following command:

cliCopy

dotnet ef migrations add Rating

dotnet ef database update

CONTINUE…

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.

Run the app and verify you can create/edit/display movies with a Rating field. You should add the Ratingfield to the Edit, Details, and Delete view templates.

# Add validation to an ASP.NET Core MVC app

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](https://wikipedia.org/wiki/Don%27t_repeat_yourself) ("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 Code First 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.

## Adding validation rules to the movie model

Open the Movie.cs file. DataAnnotations provides a built-in set of validation attributes that you apply declaratively to any class or property. (It 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 Required, StringLength, RegularExpression, and Range validation attributes.

C#Copy

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:

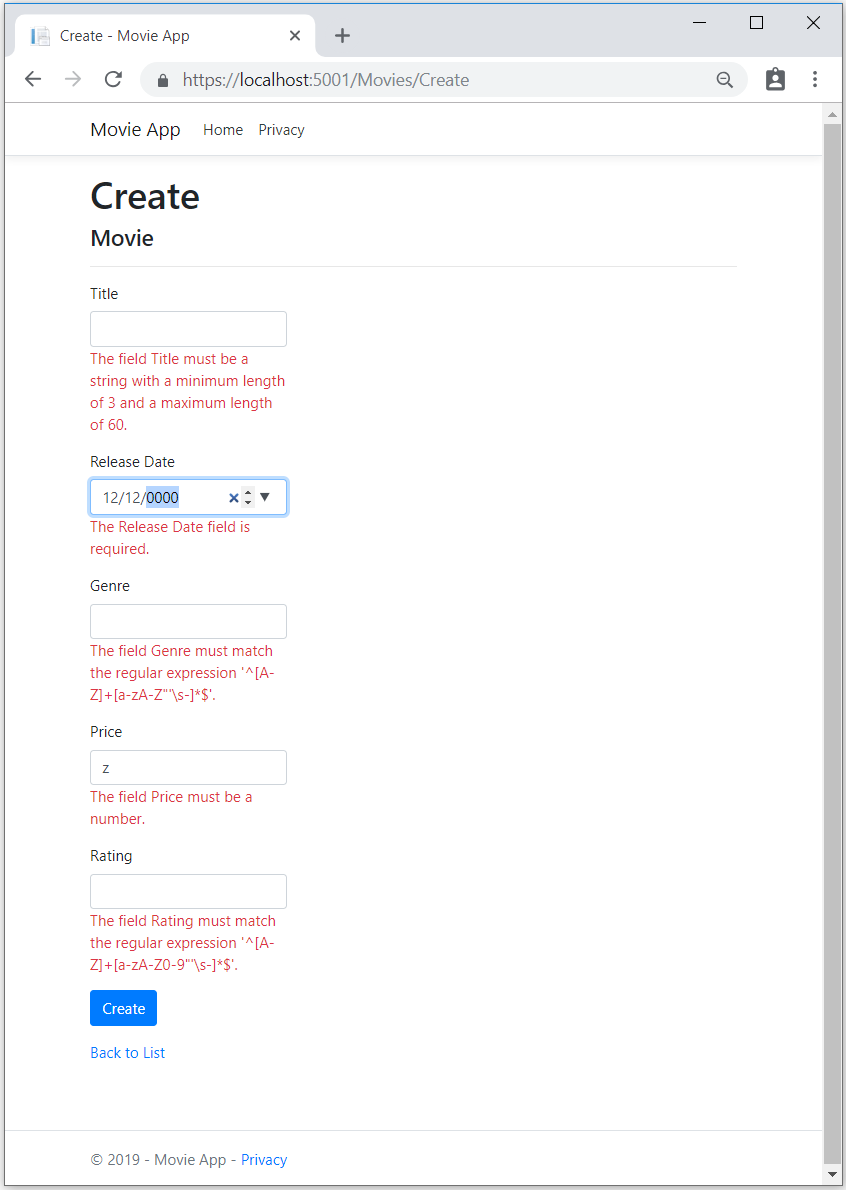
* The Required and MinimumLength attributes indicates that a property must have a value; but nothing prevents a user from entering white space to satisfy this validation.
* The RegularExpression attribute is used to limit what characters can be input. In the code above, Genre and Rating must use only letters (First letter uppercase, white space, numbers and special characters are not allowed).
* 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 in MVC

Run the app and navigate to the Movies controller.

Tap 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.



**Note**

You may not be able to enter decimal commas in decimal fields. To support [**jQuery validation**](https://jqueryvalidation.org/) 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. This [**GitHub issue 4076**](https://github.com/aspnet/Docs/issues/4076#issuecomment-326590420) 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](http://www.telerik.com/fiddler) , or the [F12 Developer tools](https://developer.microsoft.com/microsoft-edge/platform/documentation/f12-devtools-guide/).

## 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#Copy

// GET: Movies/Create

public IActionResult Create()

{

return View();

}

// POST: Movies/Create

[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("Index");

}

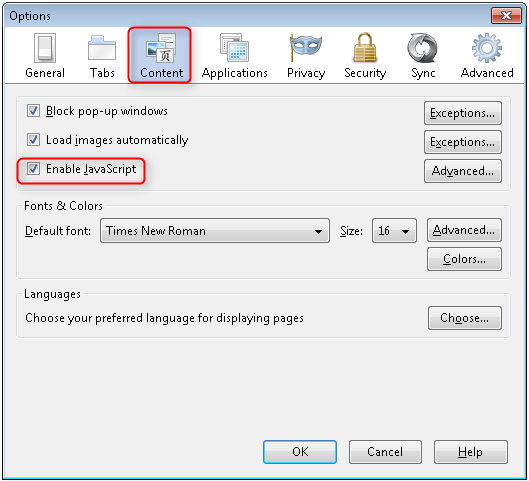
return View(movie);

}

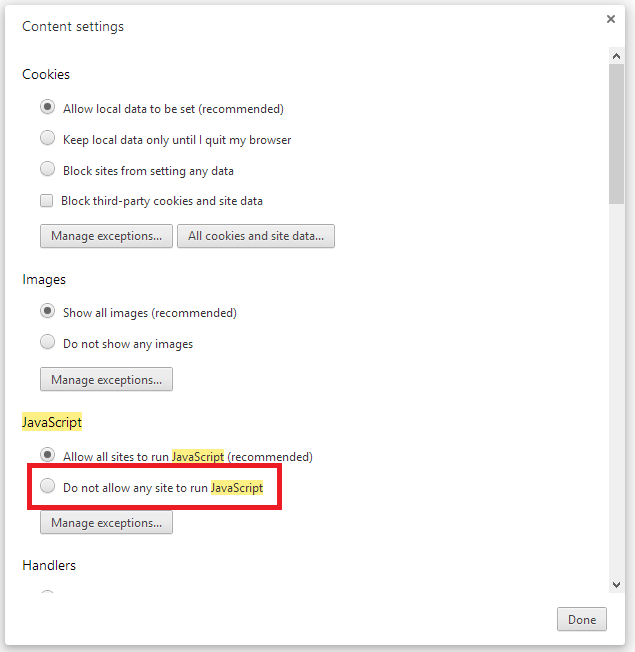
The first (HTTP GET) Create action method displays the initial Create form. The second ([HttpPost]) version handles the form post. The second Create method (The [HttpPost] version) calls ModelState.IsValid to check whether the movie has any validation errors. Calling this method evaluates any validation 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.

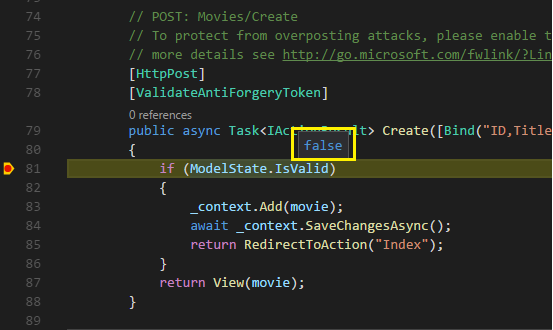
The following image shows how to disable JavaScript in the FireFox browser.



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



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



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

HTMLCopy

<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](https://docs.microsoft.com/en-us/aspnet/core/mvc/views/working-with-forms?view=aspnetcore-2.2) uses the [DataAnnotations](https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions/mvc-music-store/mvc-music-store-part-6) attributes and produces HTML attributes needed for jQuery Validation on the client side. The [Validation Tag Helper](https://docs.microsoft.com/en-us/aspnet/core/mvc/views/working-with-forms?view=aspnetcore-2.2#the-validation-tag-helpers) displays validation errors. See [Validation](https://docs.microsoft.com/en-us/aspnet/core/mvc/models/validation?view=aspnetcore-2.2) 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#Copy

[Display(Name = "Release Date")]

[DataType(DataType.Date)]

public DateTime ReleaseDate { get; set; }

[Range(1, 100)]

[DataType(DataType.Currency)]

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#Copy

[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#Copy

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](https://docs.microsoft.com/en-us/aspnet/core/mvc/views/working-with-forms?view=aspnetcore-2.2)
* [Globalization and localization](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/localization?view=aspnetcore-2.2)
* [Introduction to Tag Helpers](https://docs.microsoft.com/en-us/aspnet/core/mvc/views/tag-helpers/intro?view=aspnetcore-2.2)
* [Author Tag Helpers](https://docs.microsoft.com/en-us/aspnet/core/mvc/views/tag-helpers/authoring?view=aspnetcore-2.2)

# Examine the Details and Delete methods of an ASP.NET Core app

Open the Movie controller and examine the Details method:

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 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 Startup.cs.

C#Copy

app.UseMvc(routes =>

{

routes.MapRoute(

name: "default",

template: "{controller=Home}/{action=Index}/{id?}");

});

EF makes it easy to search for data using the SingleOrDefaultAsync 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:xxxx/Movies/Details/1 to something likehttp://localhost:xxxx/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#Copy

// 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);

\_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#Copy

// GET: Movies/Delete/5

public async Task<IActionResult> Delete(int? id)

{

C#Copy

// 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] Deletemethod:

C#Copy

// POST: Movies/Delete/6

[ValidateAntiForgeryToken]

public async Task<IActionResult> Delete(int id, bool notUsed)

### Publish to Azure

For information on deploying to Azure, see [Tutorial: Build a .NET Core and SQL Database web app in Azure App Service](https://docs.microsoft.com/en-us/azure/app-service/app-service-web-tutorial-dotnetcore-sqldb).