

# Getting started with ASP.NET Core MVC and Visual Studio

This tutorial will teach you the basics of building an ASP.NET Core MVC web app using [Visual Studio 2017](#). This tutorial teaches ASP.NET MVC Core with controllers and views. Razor Pages is a new alternative in ASP.NET Core 2.0, a page-based programming model that makes building web UI easier and more productive. For a Razor version of this tutorial see [Razor Pages Tutorial](#).

There are 3 versions of this tutorial:

- macOS: [Create an ASP.NET Core MVC app with Visual Studio for Mac](#)
- Windows: [Create an ASP.NET Core MVC app with Visual Studio](#)
- macOS, Linux, and Windows: [Create an ASP.NET Core MVC app with Visual Studio Code](#)

For the Visual Studio 2015 version of this tutorial, see the [VS 2015 version of ASP.NET Core documentation in PDF format](#).

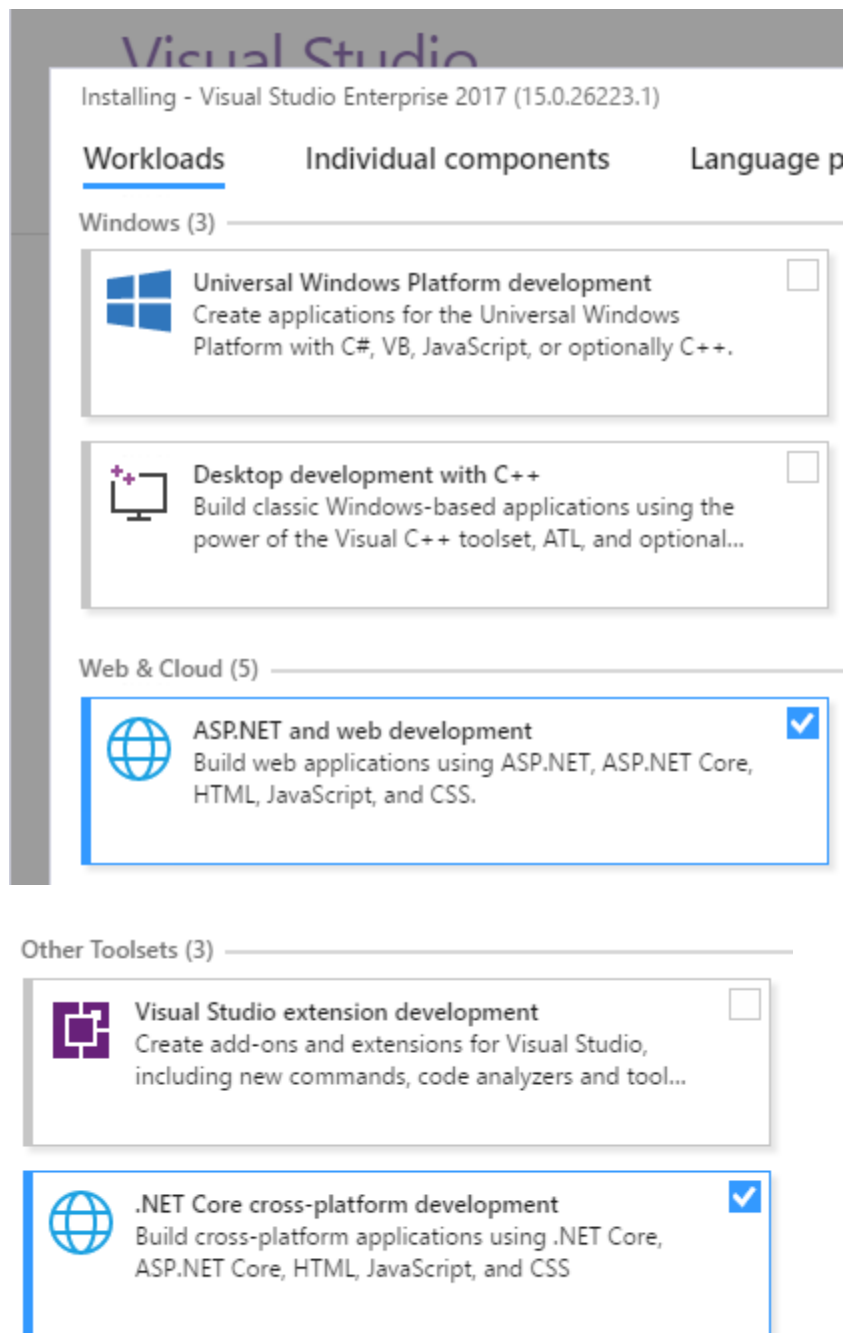
## Install Visual Studio and .NET Core

Install Visual Studio Community 2017. Select the Community download. Skip this step if you have Visual Studio 2017 installed.

- [Visual Studio 2017 Home page installer](#)

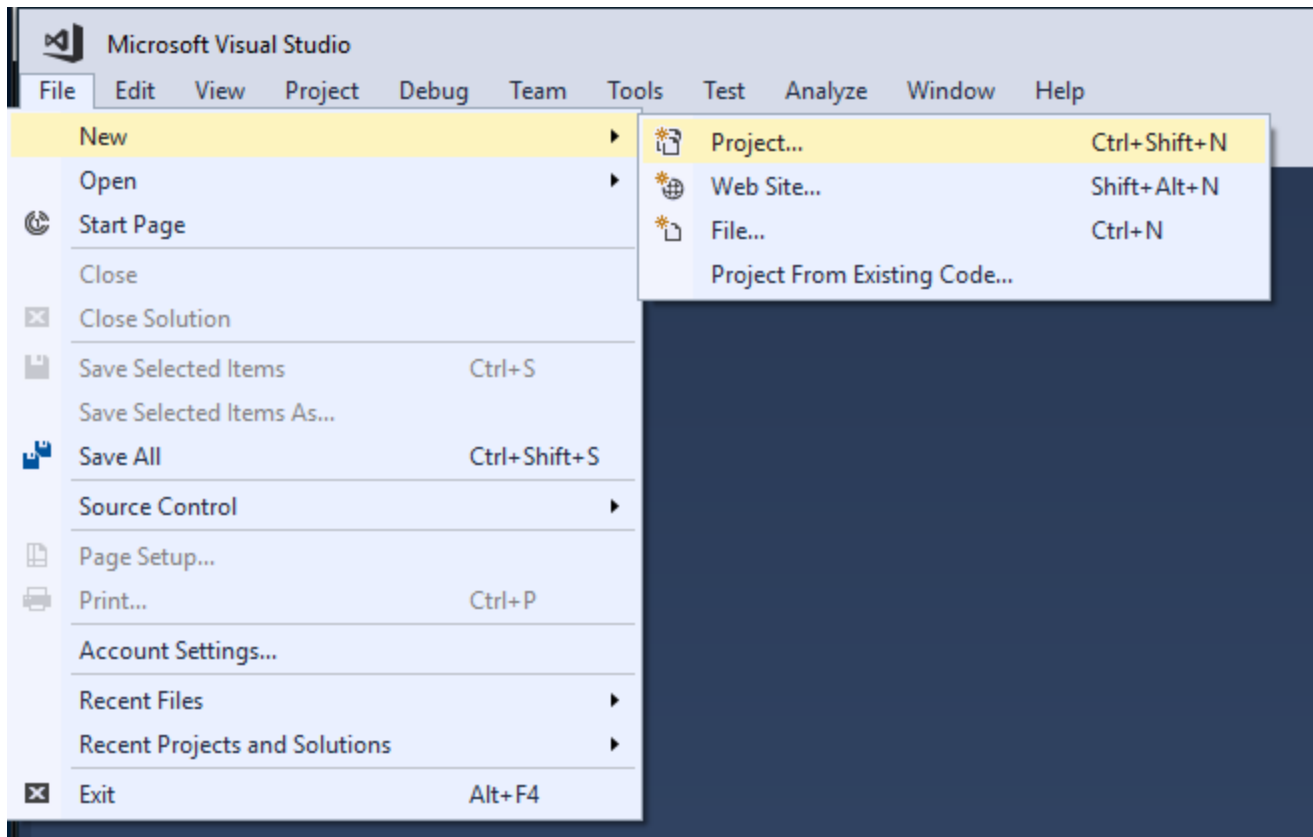
Run the installer and select the following workloads:

- ASP.NET and web development (under Web & Cloud)
- .NET Core cross-platform development (under Other Toolsets)



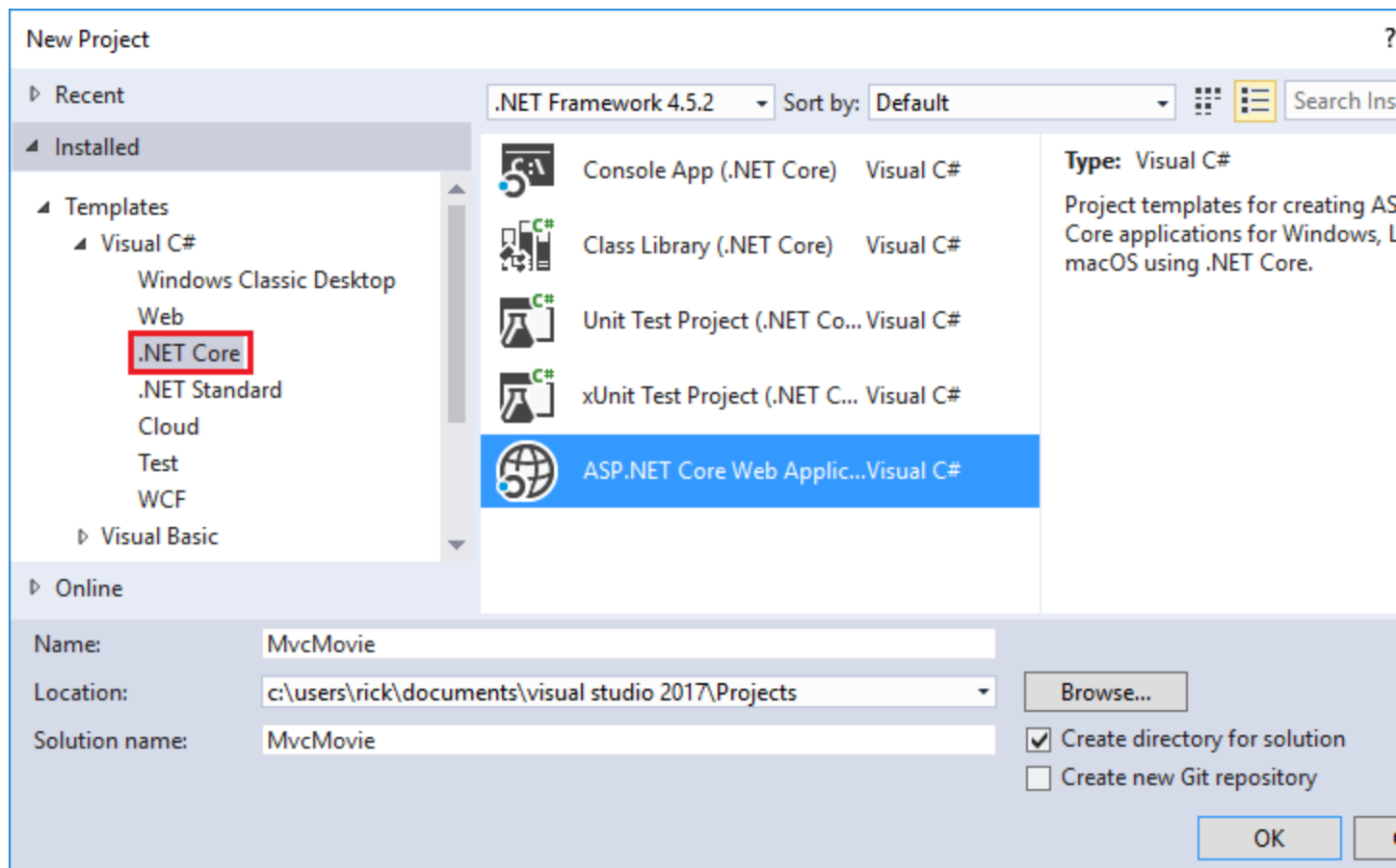
## Create a web app

From Visual Studio, select File > New > Project.



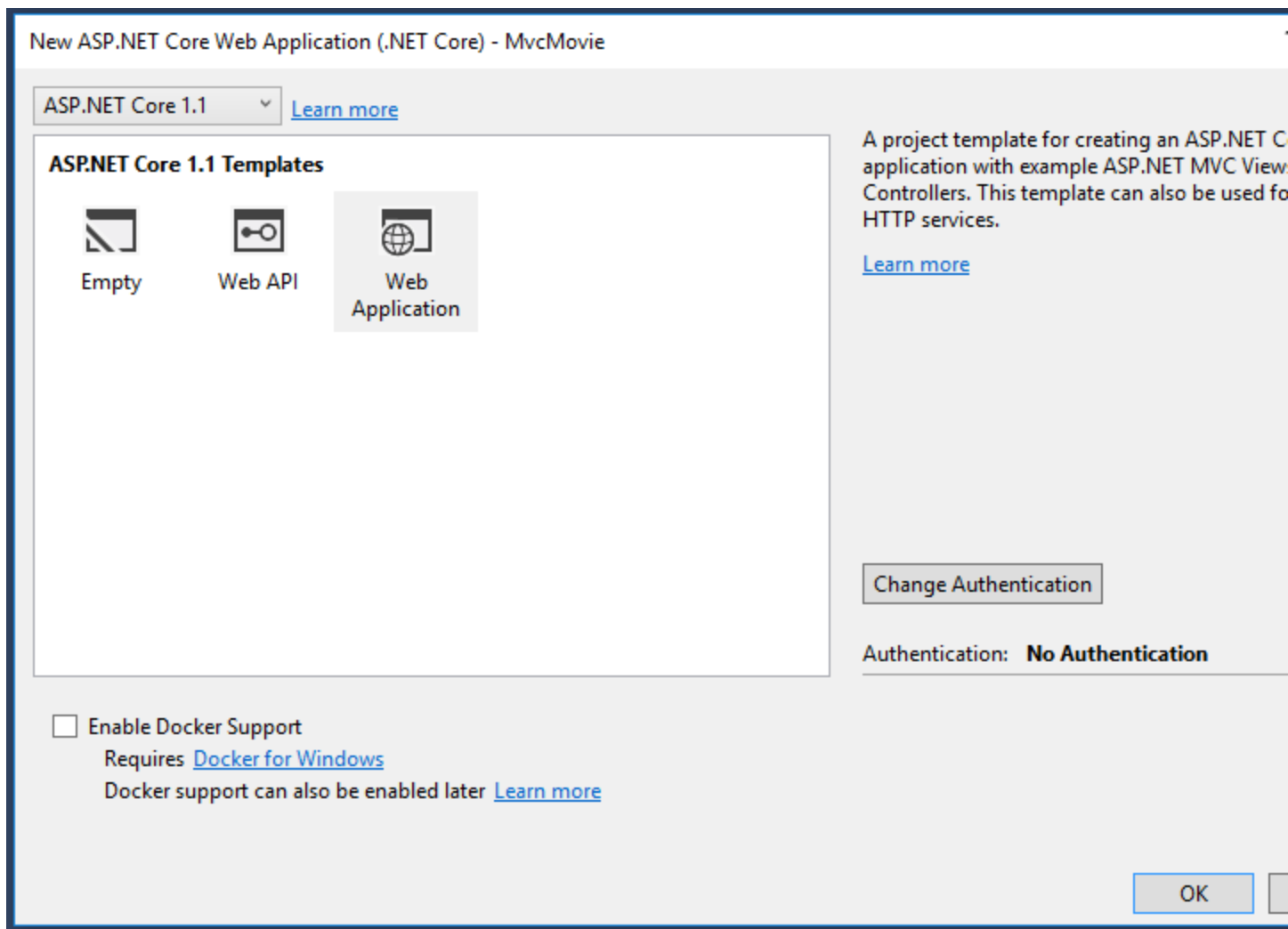
Complete the New Project dialog:

- In the left pane, tap .NET Core
- In the center pane, tap 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.)
- Tap OK



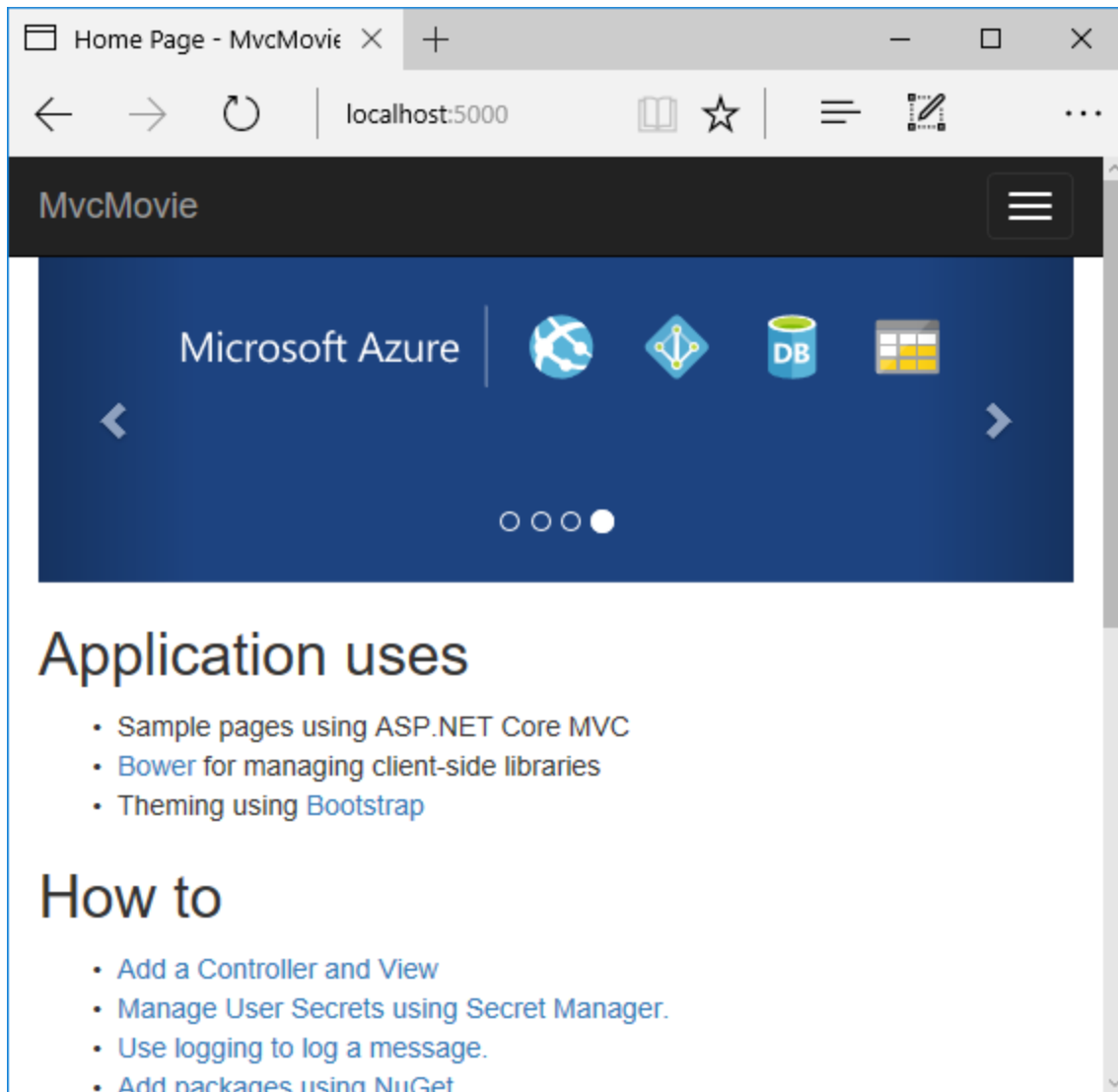
Complete the New ASP.NET Core Web Application (.NET Core) - MvcMovie dialog:

- In the version selector drop-down box tap ASP.NET Core 1.1
- Tap Web Application
- Keep the default No Authentication
- Tap OK.

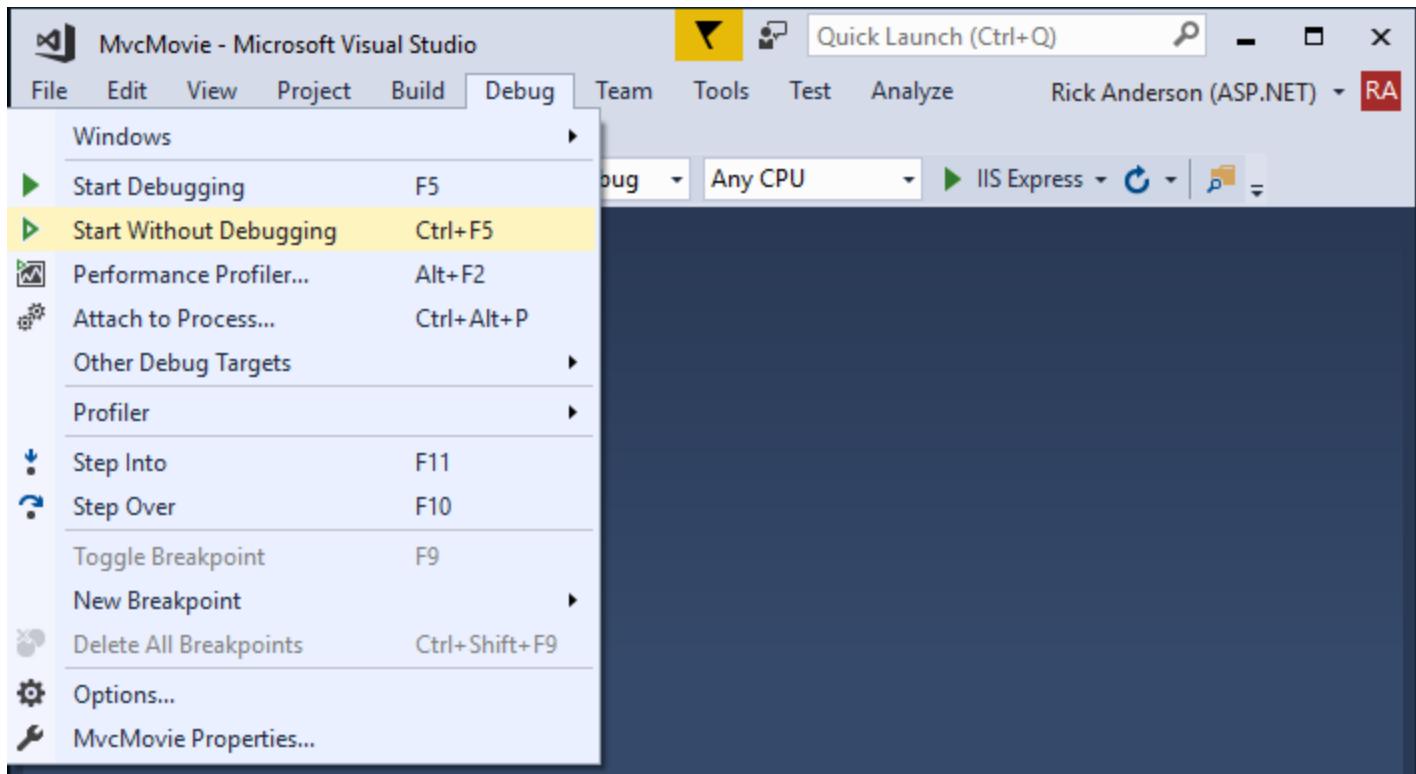


Visual Studio used a default template for the MVC project you just created. You have a working app right now by entering a project name and selecting a few options. This is a simple starter project, and it's a good place to start,<sup>1</sup>

Tap F5 to run the app in debug mode or Ctrl-F5 in non-debug mode.

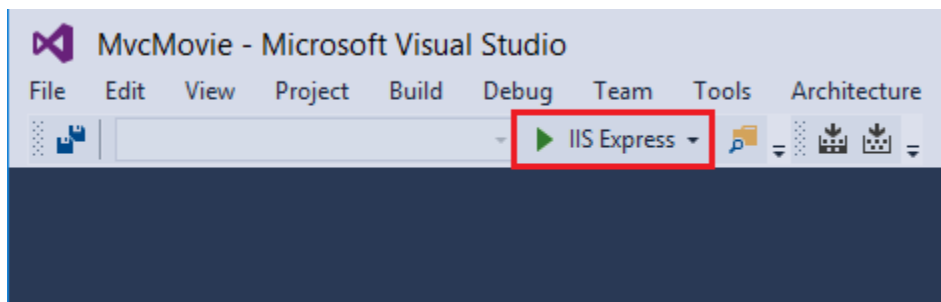


- Visual Studio starts [IIS Express](#) and runs your app. Notice that the address bar shows `localhost:port#` and not something like `example.com`. That's because `localhost` is the standard hostname for your local computer. When Visual Studio creates a web project, a random port is used for the web server. In the image above, the port number is 5000. 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:

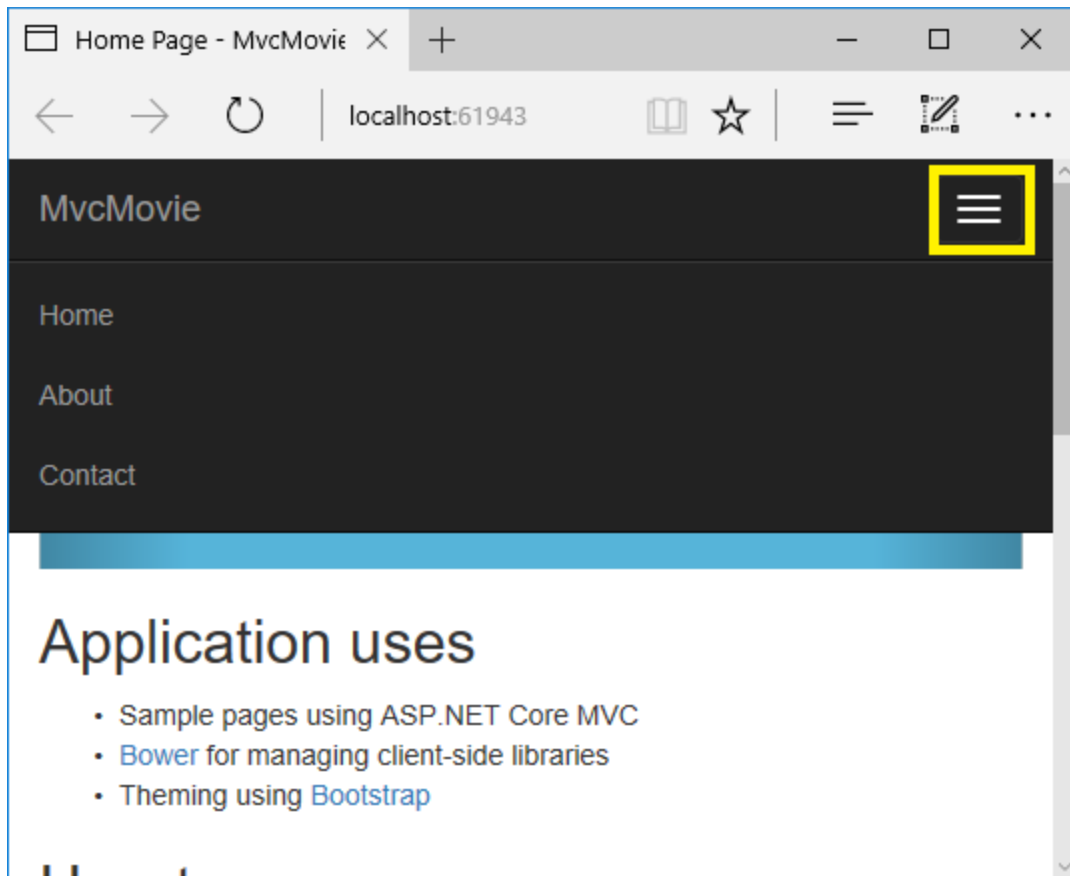


3

- You can debug the app by tapping the IIS Express button



The default template gives you working Home, About and Contact links. The browser image above doesn't show these links. Depending on the size of your browser, you might need to click the navigation icon to show them.



If you were running in debug mode, tap Shift-F5 to stop debugging.

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



# Adding a controller to a ASP.NET Core MVC app with Visual Studio

By [Rick Anderson](#)

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

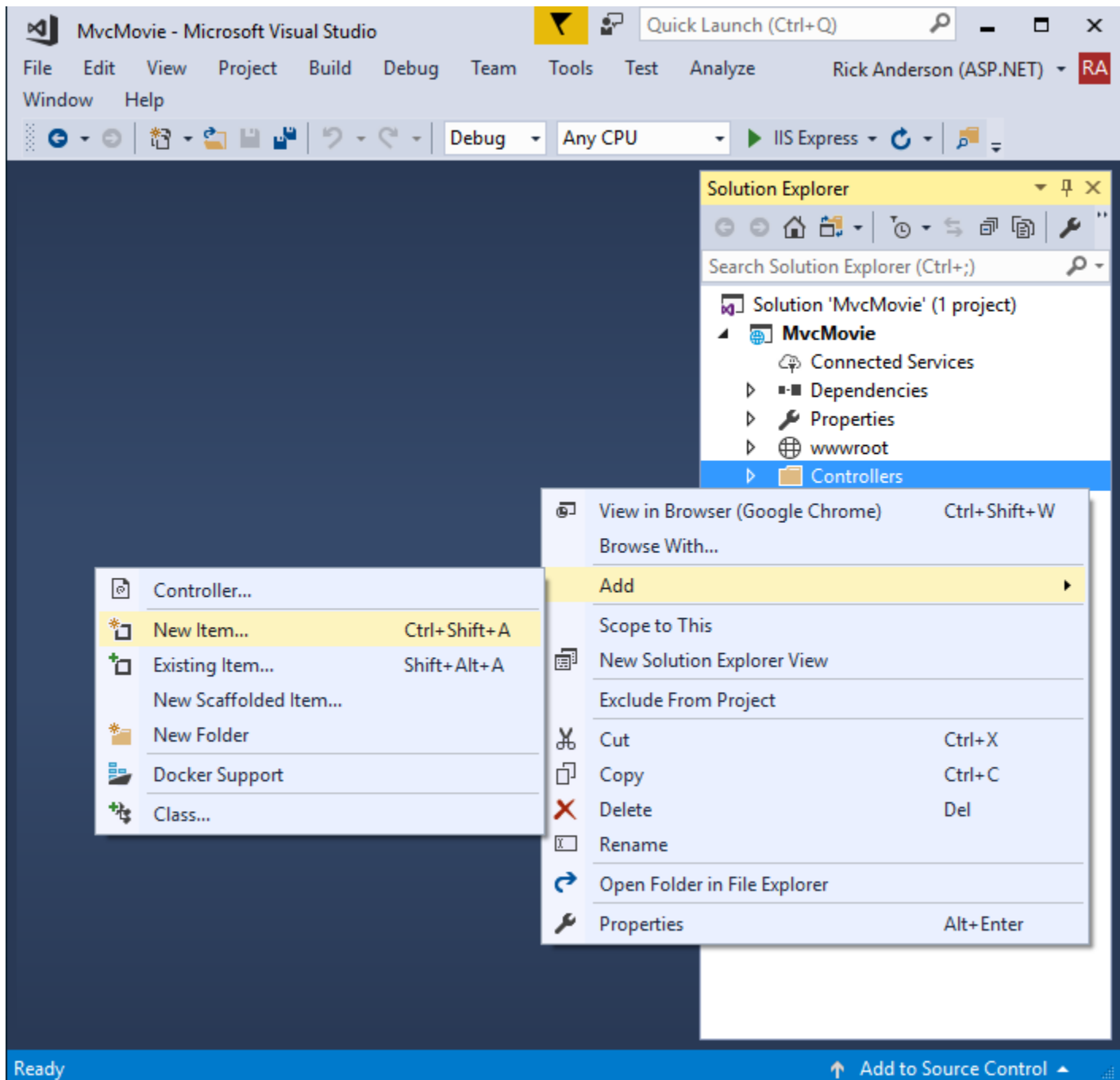
- **Models:** Classes that represent the data of the app. The model classes use validation logic to enforce business rules for that data. Typically, model objects retrieve and store model state in a database. In this tutorial, a `Movie` model retrieves movie data from a database, provides it to the view or updates it. Updated data is written to a database.
- **Views:** Views are the components that display the app's user interface (UI). Generally, this UI displays the model data.
- **Controllers:** 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, `http://localhost:1234/Home/About` has route data of `Home` (the controller) and `About` (the action method to call on the home controller). `http://localhost:1234/Movies/Edit/5` is a request to edit the movie with ID=5 using the movie controller. We'll talk about route data later in the tutorial.

2

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.<sup>7</sup>

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*. A *Models* folder will be added in a later step.2

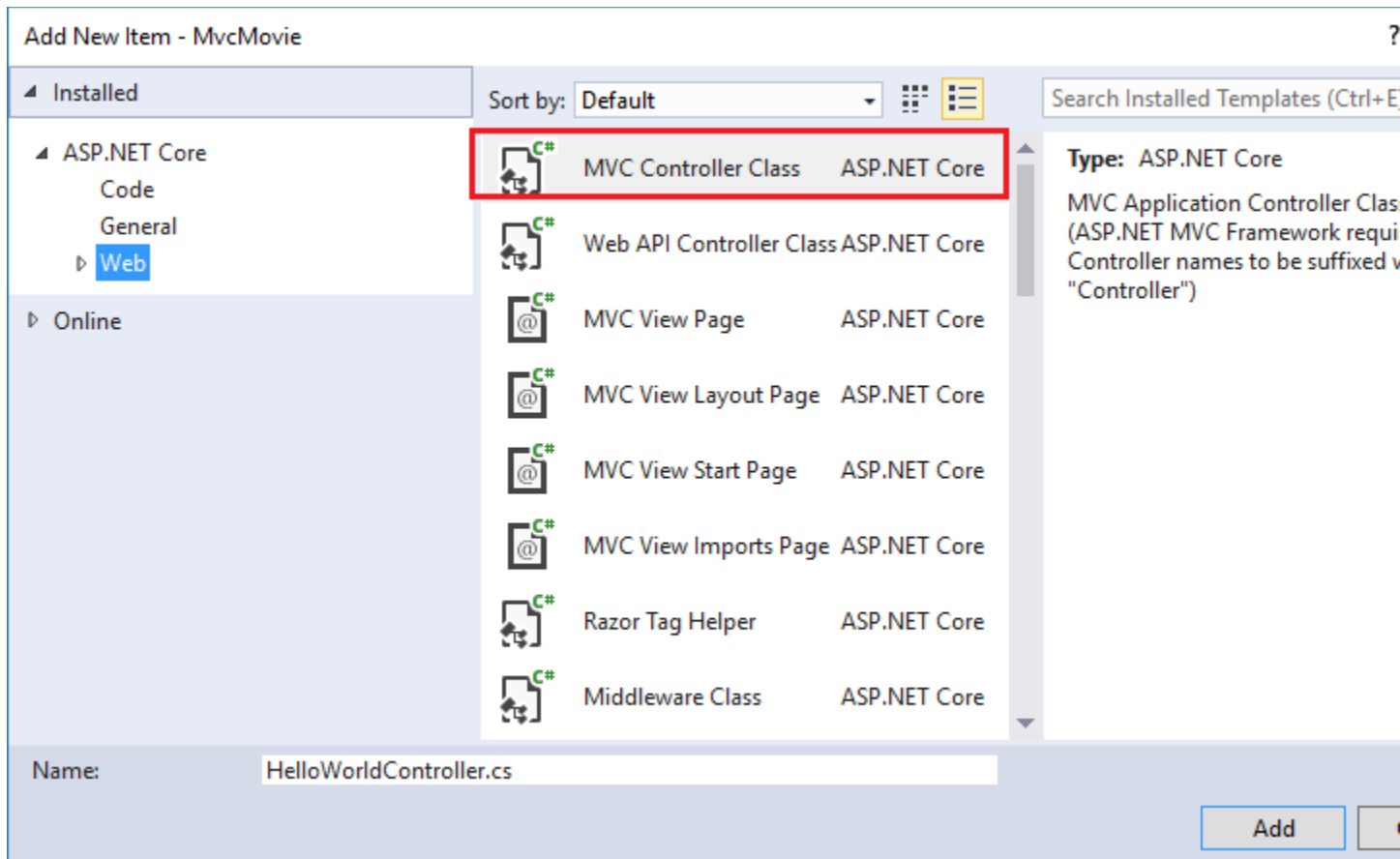
- In Solution Explorer, right-click Controllers > Add > New Item



2

- Select MVC Controller Class

- In the Add New Item dialog, enter HelloWorldController.



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/

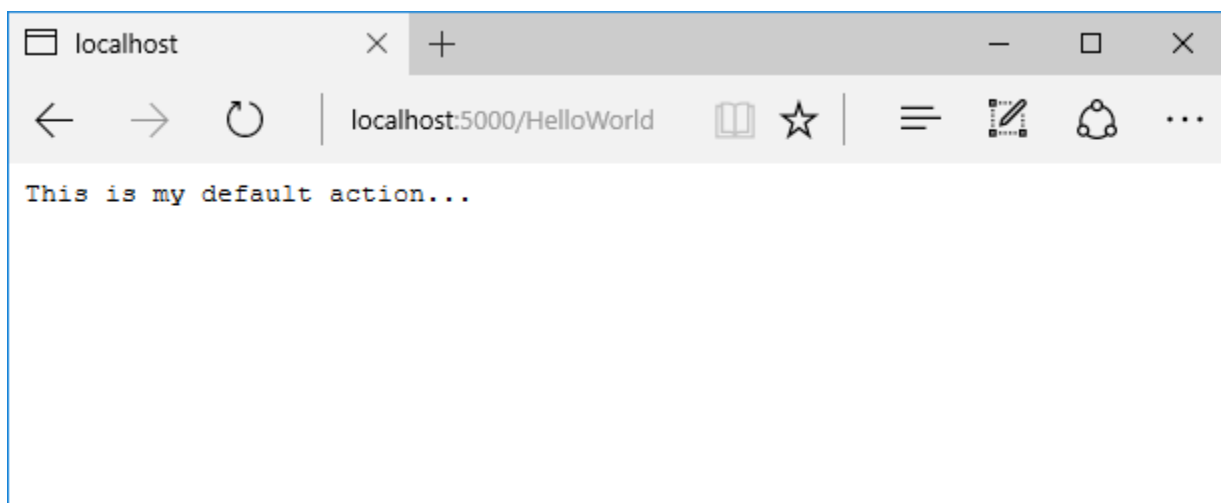
public string Welcome()
{
    return "This is the Welcome action method...";
}
}
```

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.<sup>5</sup>

An HTTP endpoint is a targetable URL in the web application, such as `http://localhost:1234/HelloWorld`, and combines the protocol used: `HTTP`, the network location of the web server (including the TCP port): `localhost:1234` and the target URI `HelloWorld`.

The first comment states this is an [HTTP GET](#) method that is invoked by appending `/HelloWorld/` to the base URL. The second comment specifies an [HTTP GET](#) method that is invoked by appending `/HelloWorld/Welcome/` to the URL. Later on in the tutorial you'll use the scaffolding engine to generate `HTTP POST` methods.

Run the app in non-debug mode and append "HelloWorld" to the path in the address bar. The `Index` method returns a string.



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

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

You set the format for routing in the *Startup.cs* file.

C#Copy

```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
});
```

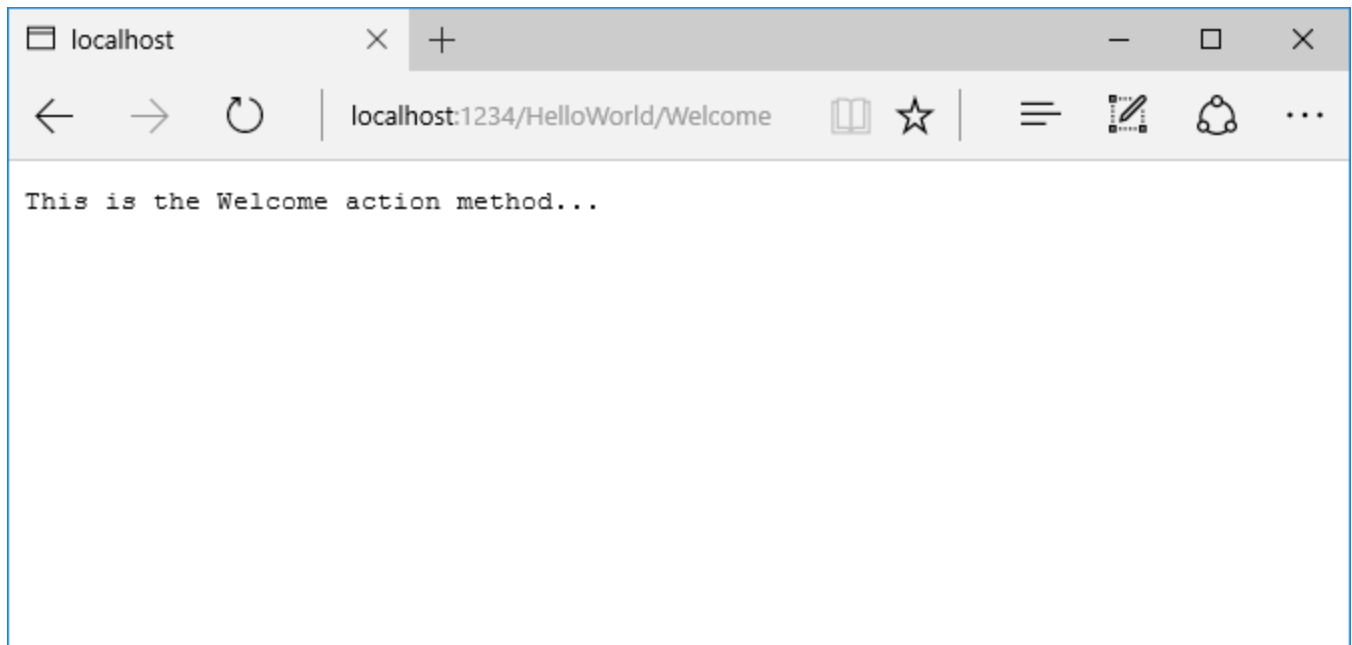
When you run the app and don't supply any URL segments, it defaults to the "Home" controller and the "Index" method specified in the template line highlighted above.

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 is not explicitly specified. The third part of the URL segment ( `id` ) is for route data. You'll see route data later on in this tutorial.<sup>3</sup>

Browse to `http://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.
- Uses `HtmlEncoder.Default.Encode` to protect the app from malicious input (namely JavaScript).
- Uses [Interpolated Strings](#).

1

Run your app and browse to:

```
http://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](#) system automatically maps the named parameters from the query string in the address bar to parameters in your method. See [Model Binding](#) for more information.



4

In the image above, the URL segment (`Parameters`) is not used, the `name` and `numTimes` parameters are passed as [query strings](#). The `?` (question mark) in the above URL is a separator, and the query strings follow. The `&` character separates query strings.

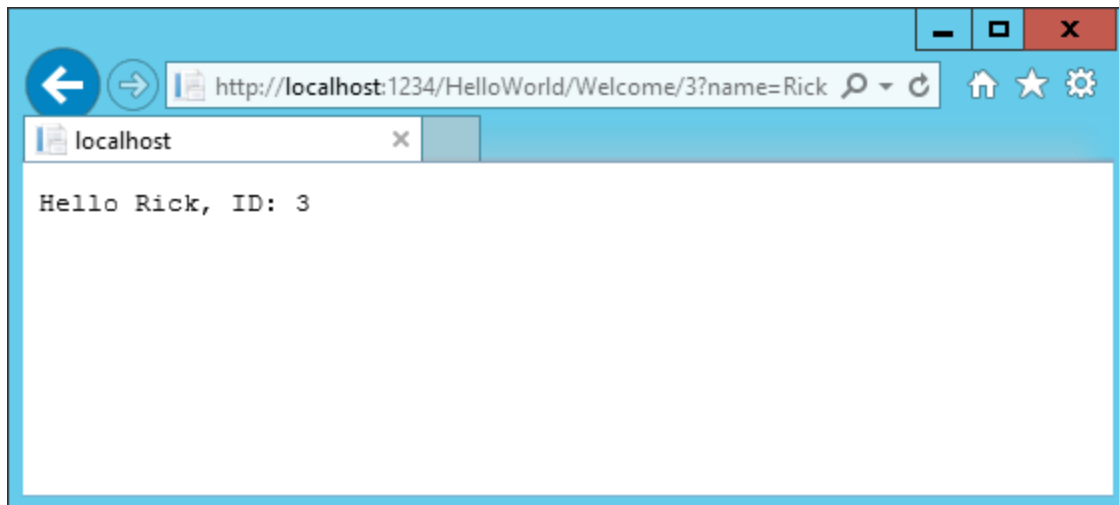
Replace the `Welcome` method with the following code:

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: `http://localhost:xxx/HelloWorld/Welcome/3?name=Rick` 5



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.<sup>1</sup>

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.<sup>2</sup>

In Visual Studio, in non-debug mode (Ctrl+F5), you don't need to build the app after changing code. Just save the file, refresh your browser and you can see the changes.



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

In this section you modify the `HelloWorldController` class to use Razor view template 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 using C#.

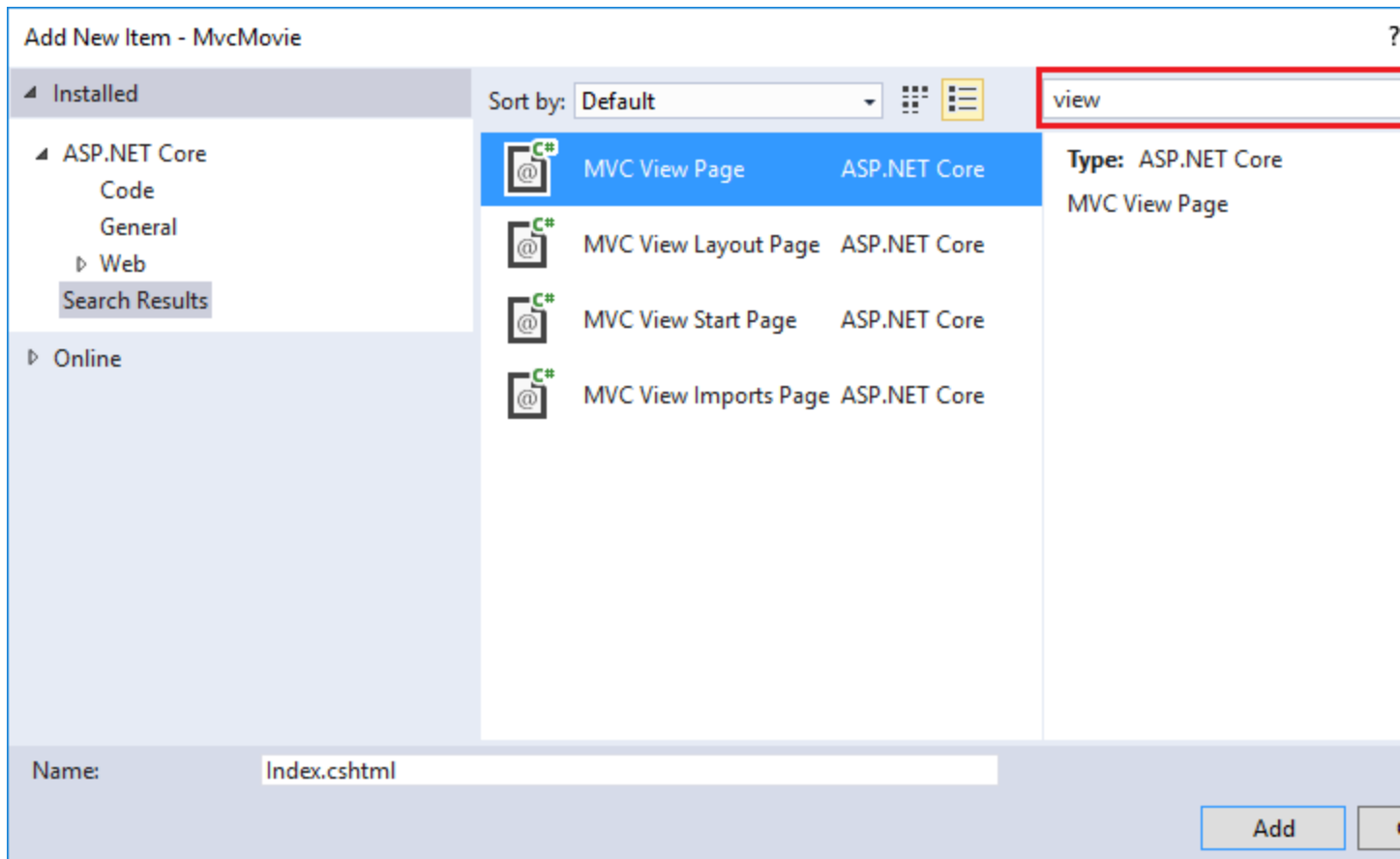
Currently the `Index` method returns a string with a message that is hard-coded in the controller class. In the `HelloWorldController` class, replace the `Index` method with the following code:

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 [ActionResult](#) (or a class derived from `ActionResult`), not primitive types like `string`.<sup>6</sup>

- 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*
  - Tap MVC View Page
  - In the Name box, change the name if necessary to *Index.cshtml*.
  - Tap Add



2

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

HTMLCopy

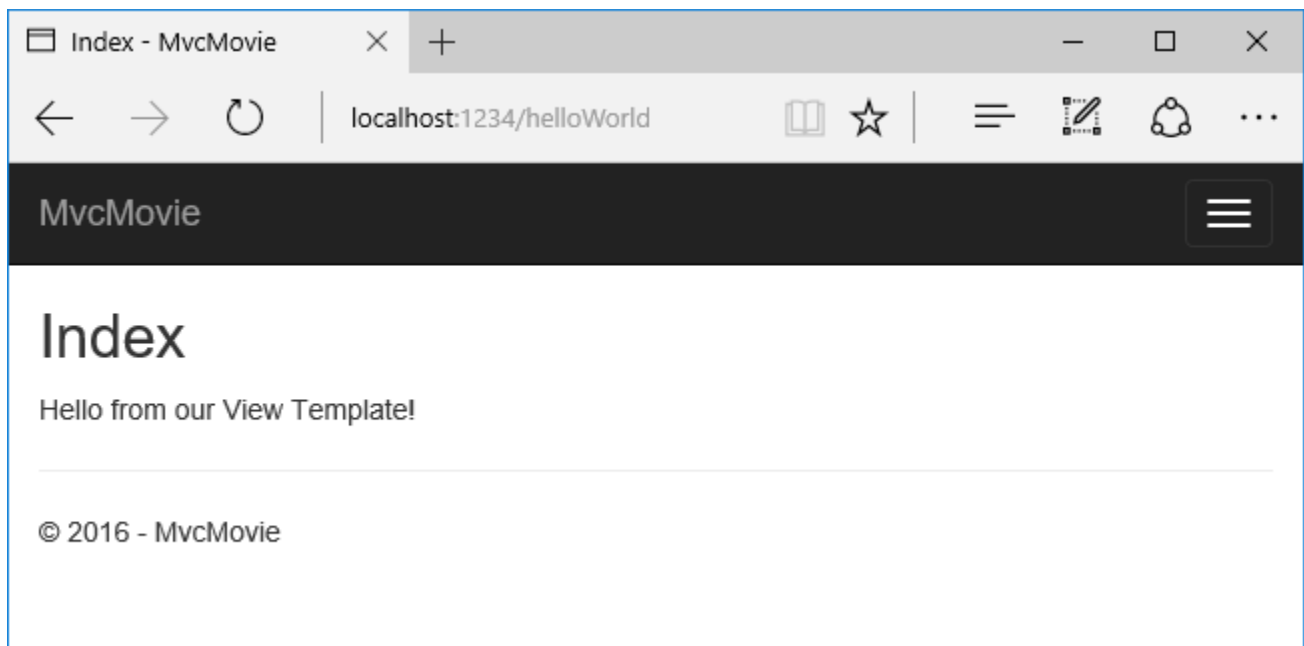
```
@{
    ViewData["Title"] = "Index";
}

<h2>Index</h2>

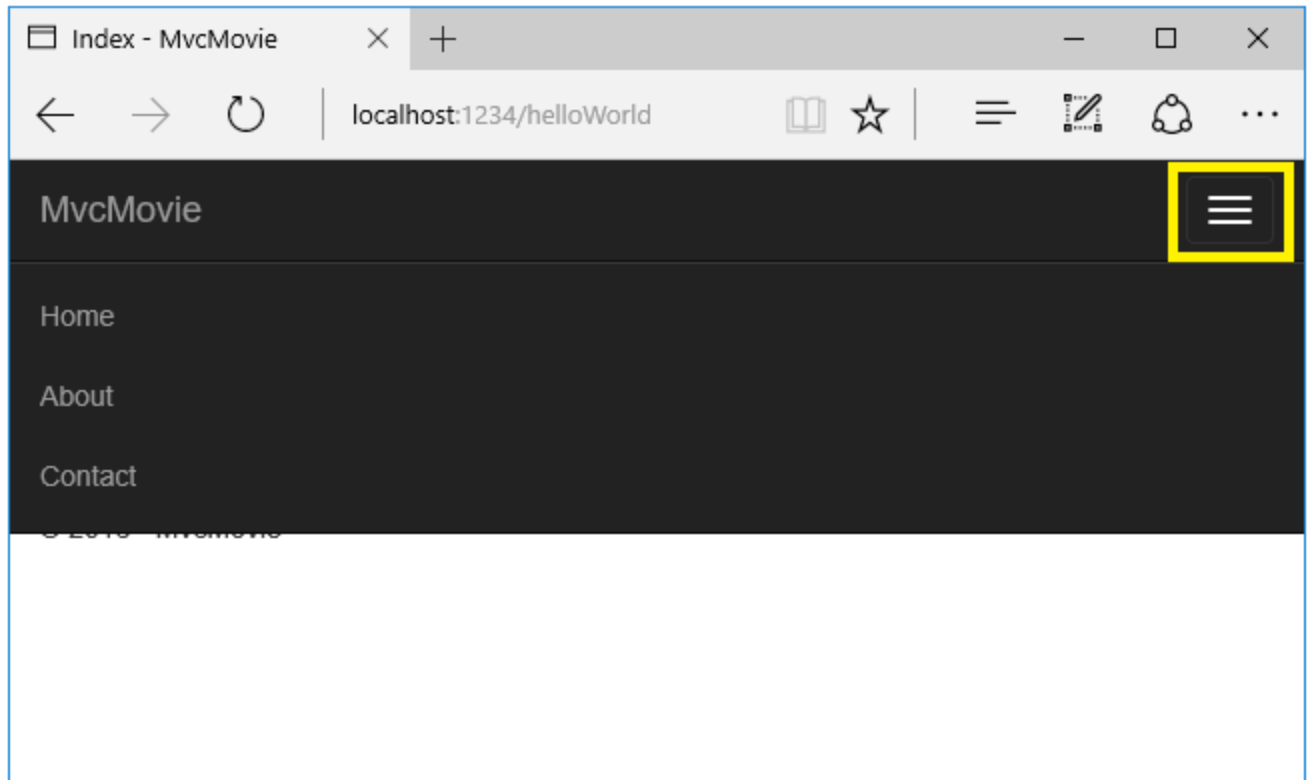
<p>Hello from our View Template!</p>
```

Navigate to `http://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.<sup>6</sup>



If your browser window is small (for example on a mobile device), you might need to toggle (tap) the [Bootstrap navigation button](#) in the upper right to see the Home, About, and Contact links.<sup>3</sup>



## Changing views and layout pages

Tap the menu links (MvcMovie, Home, About). Each page shows the same menu layout. The menu layout is implemented in the *Views/Shared/\_Layout.cshtml* file. Open the *Views/Shared/\_Layout.cshtml* file.

[Layout](#) templates allow 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 About link, the *Views/Home/About.cshtml* view is rendered inside the `RenderBody` method.<sup>3</sup>

## Change the title and menu link in the layout file

Change the contents of the title element. Change the anchor text in the layout template to "Movie App" and the controller from `Home` to `Movies` as highlighted below:<sup>6</sup>

htmlCopy

```

@inject Microsoft.ApplicationInsights.AspNetCore.JavaScriptSnippet JavaScriptSnippet
<!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 names="Development">
        <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
        <link rel="stylesheet" href="~/css/site.css" />
    </environment>
    <environment names="Staging,Production">
        <link rel="stylesheet"
href="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/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" />
        <link rel="stylesheet" href="~/css/site.min.css" asp-append-version="true" />
    </environment>
    @Html.Raw(JavaScriptSnippet.FullScript)
</head>
<body>
    <nav class="navbar navbar-inverse navbar-fixed-top">
        <div class="container">
            <div class="navbar-header">
                <button type="button" class="navbar-toggle" data-toggle="collapse"
data-target=".navbar-collapse">
                    <span class="sr-only">Toggle navigation</span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                </button>
                <a asp-area="" asp-controller="Movies" asp-action="Index"
class="navbar-brand">MvcMovie</a>
            </div>
            <div class="navbar-collapse collapse">
                <ul class="nav navbar-nav">
                    <li><a asp-area="" asp-controller="Home" asp-
action="Index">Home</a></li>
                    <li><a asp-area="" asp-controller="Home" asp-
action="About">About</a></li>
                    <li><a asp-area="" asp-controller="Home" asp-
action="Contact">Contact</a></li>

```

```

        </ul>
    </div>
</div>
</nav>
<div class="container body-content">
    @RenderBody()
    <hr />
    <footer>
        <p>&copy; 2017 - MvcMovie</p>
    </footer>
</div>

<environment names="Development">
    <script src="~/lib/jquery/dist/jquery.js"></script>
    <script src="~/lib/bootstrap/dist/js/bootstrap.js"></script>
    <script src="~/js/site.js" asp-append-version="true"></script>
</environment>
<environment names="Staging,Production">
    <script src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-2.2.0.min.js"
        asp-fallback-src="~/lib/jquery/dist/jquery.min.js"
        asp-fallback-test="window.jQuery"
        crossorigin="anonymous"
        integrity="sha384-
K+ctZQ+LL8q6tP7I94W+qzQsfRV2a+AfhIi9k8z8l9ggpc8X+Ytst4yBo/hH+8Fk">
    </script>
    <script
src="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.7/bootstrap.min.js"
        asp-fallback-src="~/lib/bootstrap/dist/js/bootstrap.min.js"
        asp-fallback-test="window.jQuery && window.jQuery.fn &&
window.jQuery.fn.modal"
        crossorigin="anonymous"
        integrity="sha384-
Tc5Iqib027qvyjSMfHjOMaLkfuWVxZxUPnCJA7l2mCWNIpG9mGCD8wGNICPD7Txa">
    </script>
    <script src="~/js/site.min.js" asp-append-version="true"></script>
</environment>

    @RenderSection("Scripts", required: false)
</body>
</html>

```

## Warning

We haven't implemented the `Movies` controller yet, so if you click on that link, you'll get a 404 (Not found) error.

Save your changes and tap the About link. Notice how the title on the browser tab now displays About - Movie App instead of About - Mvc Movie. Tap the Contact link and notice that it also displays 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.<sup>10</sup>

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. You can use the `Layout` property to set a different layout view, or set it to `null` so no layout file will be used.<sup>4</sup>

Change the title of the `Index` view.

Open *Views/HelloWorld/Index.cshtml*. There are two places to make a change:<sup>2</sup>

- The text that appears in the title of the browser.
- The secondary header (`<h2>` element).

You'll make them slightly different so you can see which bit of code changes which part of the app.

HTMLCopy

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

<h2>My Movie List</h2>

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

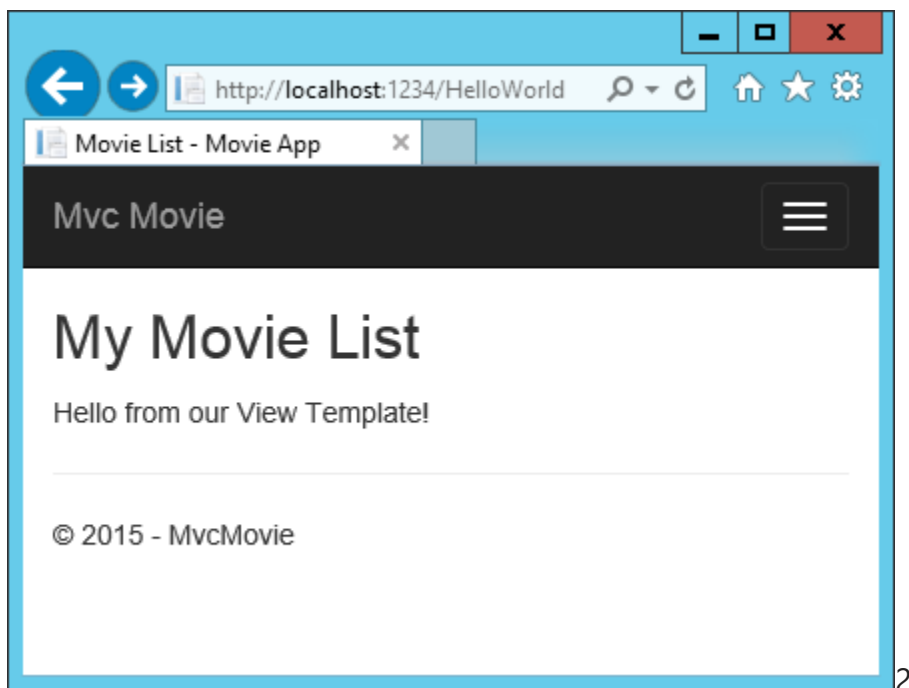
`ViewData["Title"] = "Movie List";` in the code above sets the `Title` property of the `ViewData` dictionary to "Movie List". The `Title` property is used in the `<title>` HTML element in the layout page:<sup>2</sup>

HTMLCopy

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

Save your change and navigate to `http://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](#).



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 you write the code 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.<sup>2</sup>

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 your 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, let's change the controller to use a view template instead. The view template will generate a dynamic response, which means that you need to pass appropriate bits of data from the controller to the view in order to generate the response. You can 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.

Return to the `HelloWorldController.cs` file and change the `Welcome` method to add a `Message` and `NumTimes` value to the `ViewData` dictionary. The `ViewData` dictionary is a dynamic object, which means you can put whatever you want in to it; the `ViewData` object has no defined properties until you put something inside it. The [MVC model binding system](#) 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:<sup>4</sup>

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.<sup>4</sup>

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>

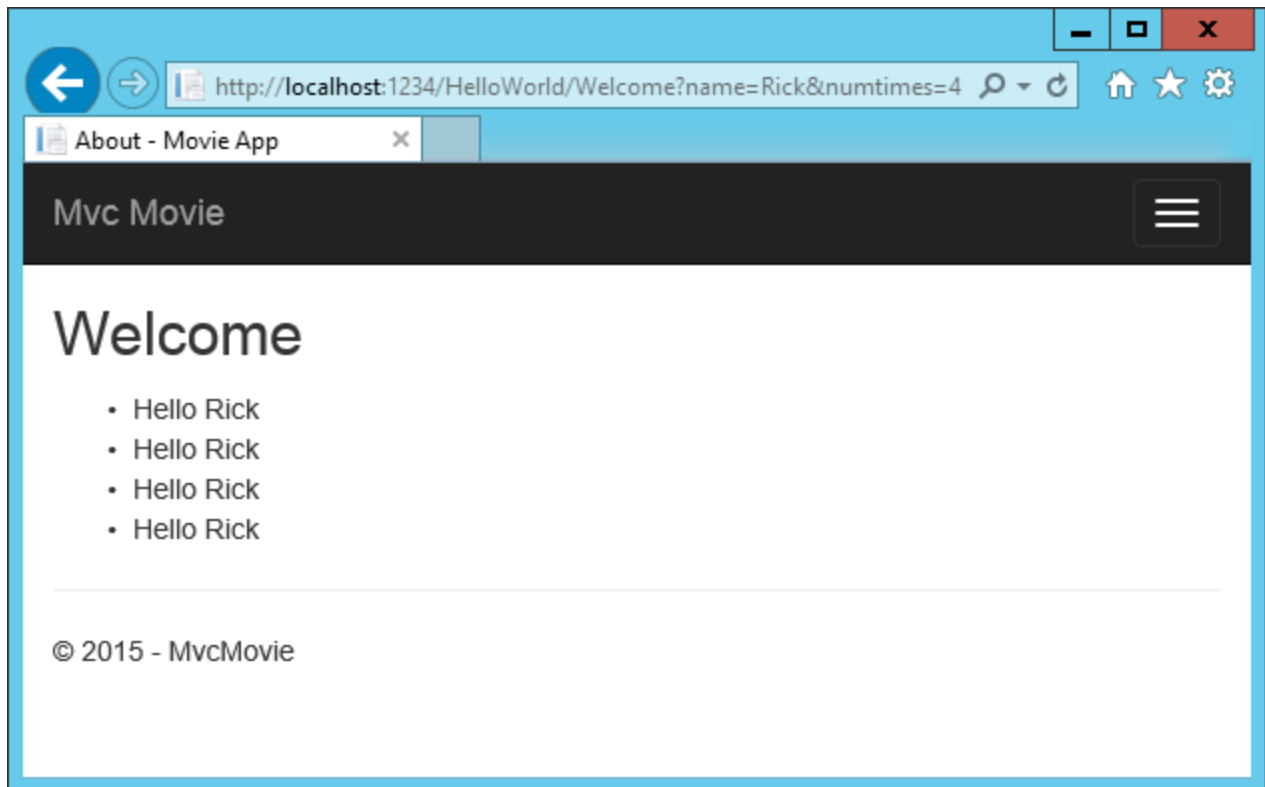
<ul>
    @for (int i = 0; i < (int)ViewData["NumTimes"]; i++)
    {
        <li>@ViewData["Message"]</li>
    }
</ul>

```

Save your changes and browse to the following URL:

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

Data is taken from the URL and passed to the controller using the [MVC model binder](#). The controller packages the data into a `ViewData` dictionary and passes that object to the view. The view then renders the data as HTML to the browser.<sup>2</sup>



5

In the sample above, we used the `ViewData` dictionary to pass data from the controller to a view. Later in the tutorial, we will use a view model 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 [ViewModel vs ViewData vs ViewBag vs TempData vs Session in MVC](#) for more information.

Well, that was a kind of an "M" for model, but not the database kind. Let's take what we've learned and create a database of movies.

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

By [Rick Anderson](#) and [Tom Dykstra](#)

In this section, you'll add some classes for managing movies in a database. These classes will be the "Model" part of the MVC app.

You use these classes with [Entity Framework Core](#) (EF Core) to work with a database. EF Core is an object-relational mapping (ORM) framework that simplifies the data access code that you have to write. [EF Core supports many database engines](#).<sup>4</sup>

The model classes you'll create are known as POCO classes (from "plain-old CLR objects") 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'll write the model classes first, and EF Core will create the database. An alternate approach not covered here is to generate model classes from an already-existing database. For information about that approach, see [ASP.NET Core - Existing Database](#).

## Add a data model class

In Solution Explorer, right click the MvcMovie project > Add > New Folder. Name the folder *Models*.<sup>4</sup>

Right click the *Models* folder > Add > Class. Name the class *Movie* and add the following properties:

C#Copy

```
using System;

namespace MvcMovie.Models
{
    public class Movie
    {
        public int ID { get; set; }
        public string Title { get; set; }
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; }
        public decimal Price { get; set; }
    }
}
```

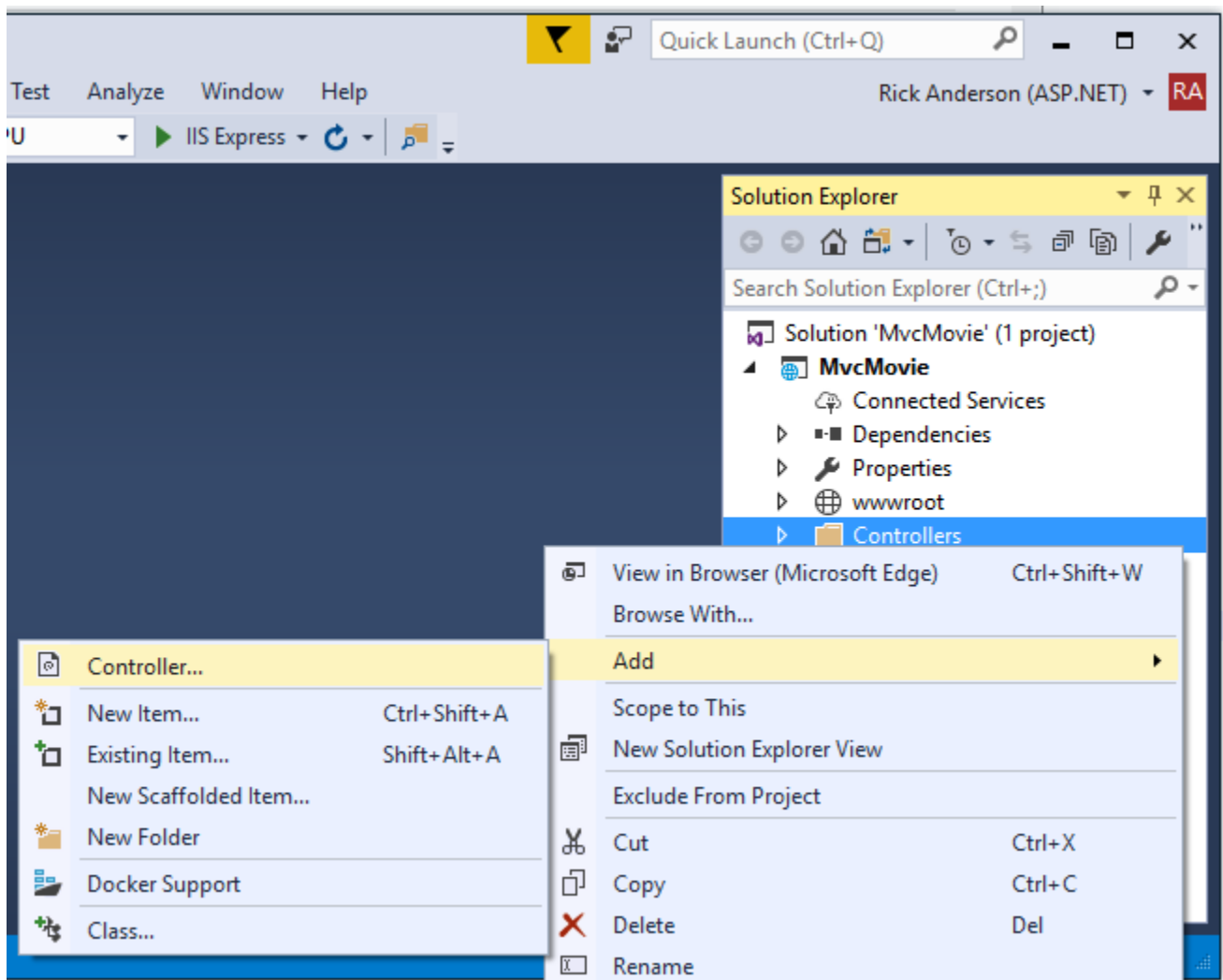
```
}  
}
```

The `ID` field is required by the database for the primary key.

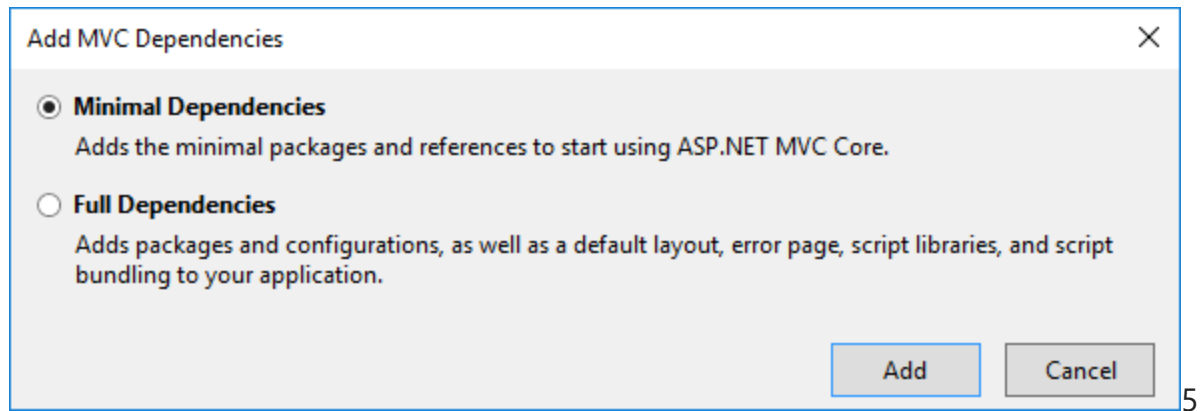
Build the project to verify you don't have any errors. You now have a Model in your MVC app.

## Scaffolding a controller

In Solution Explorer, right-click the *Controllers* folder > Add > Controller.5

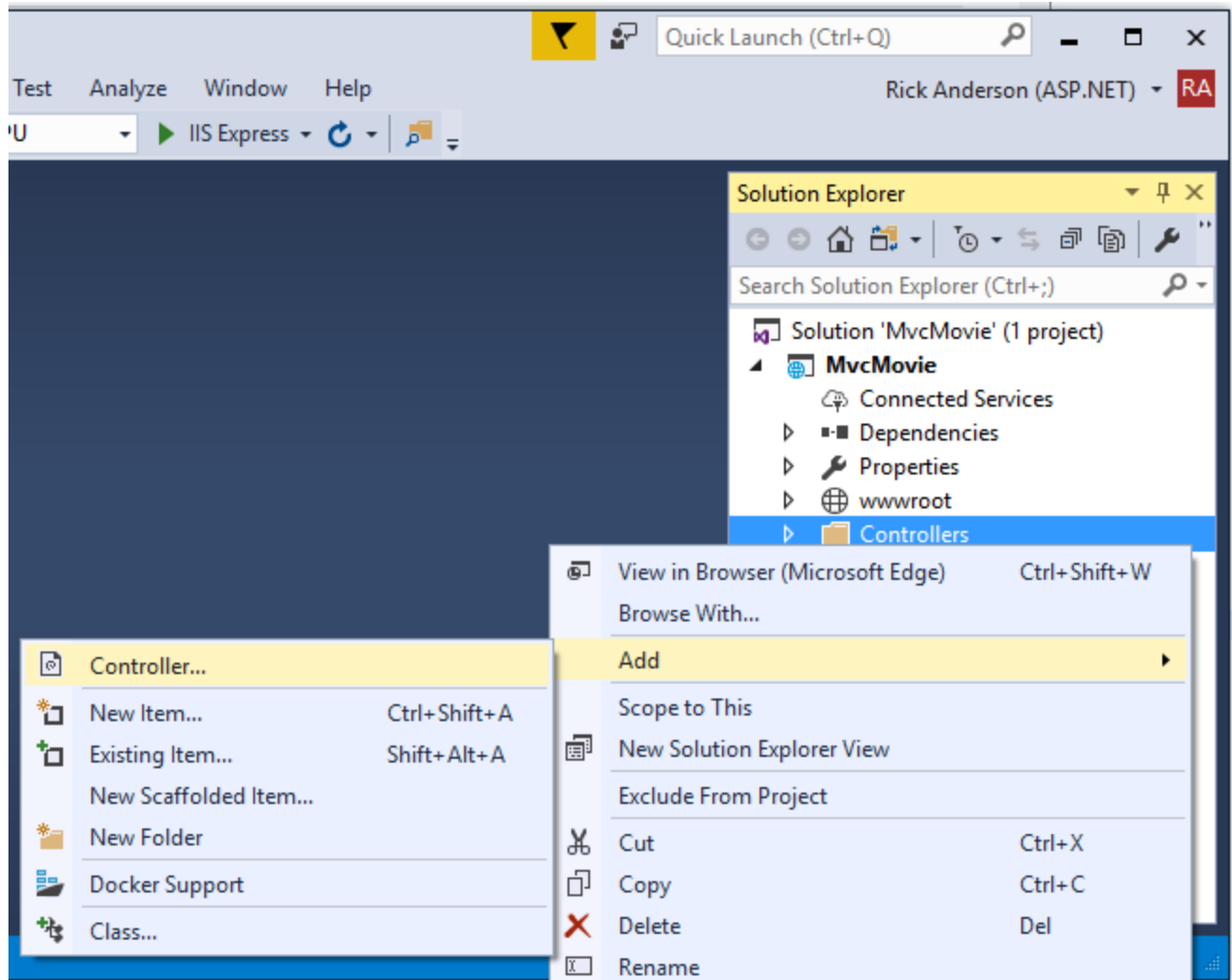


In the Add MVC Dependencies dialog, select Minimal Dependencies, and select Add.



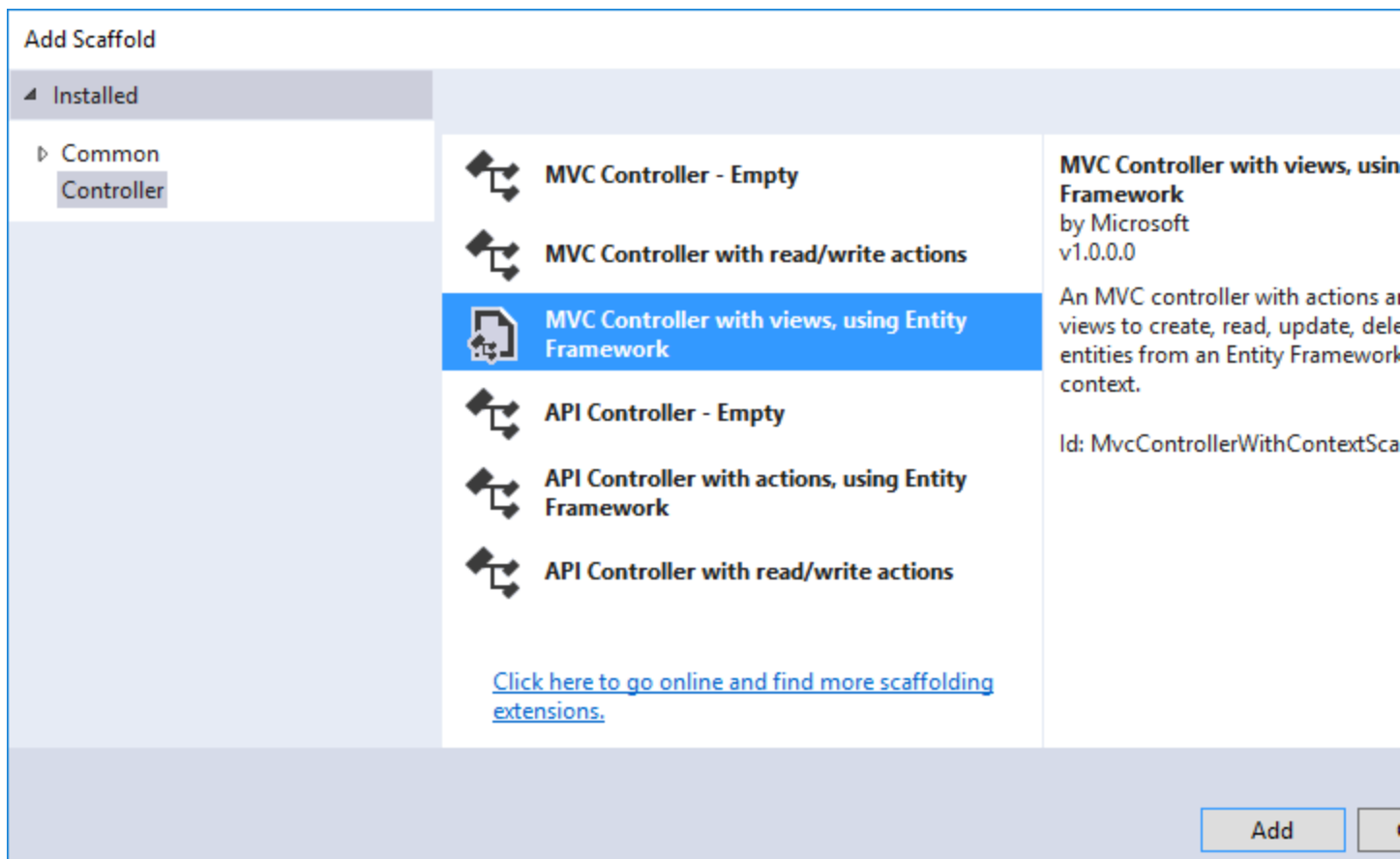
Visual Studio adds the dependencies needed to scaffold a controller, but the controller itself is not created. The next invoke of > Add > Controller creates the controller.

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



3

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



2

Complete the Add Controller dialog:

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

5



**Add Controller**

Model class:

Data context class:  **+**

**New Data Context**

New data context type:

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

Controller name:

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

**Add Controller**

Model class:

Data context class:  **+**

Views:

☒ Generate views

☒ Reference script libraries

☒ Use a layout page:

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

Controller name:

Visual Studio creates:

- An Entity Framework Core [database context class](#) (*Data/MvcMovieContext.cs*)
- A movies controller (*Controllers/MoviesController.cs*)
- Razor view files for Create, Delete, Details, Edit and Index pages (*Views/Movies/\*.cshtml*)

3

The automatic creation of the database context and [CRUD](#) (create, read, update, and delete) action methods and views is known as *scaffolding*. You'll soon have a fully functional web application that lets you manage a movie database.

If you run the app and click on the Mvc Movie link, you'll get an error similar to the following:10

Copy

```
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
```

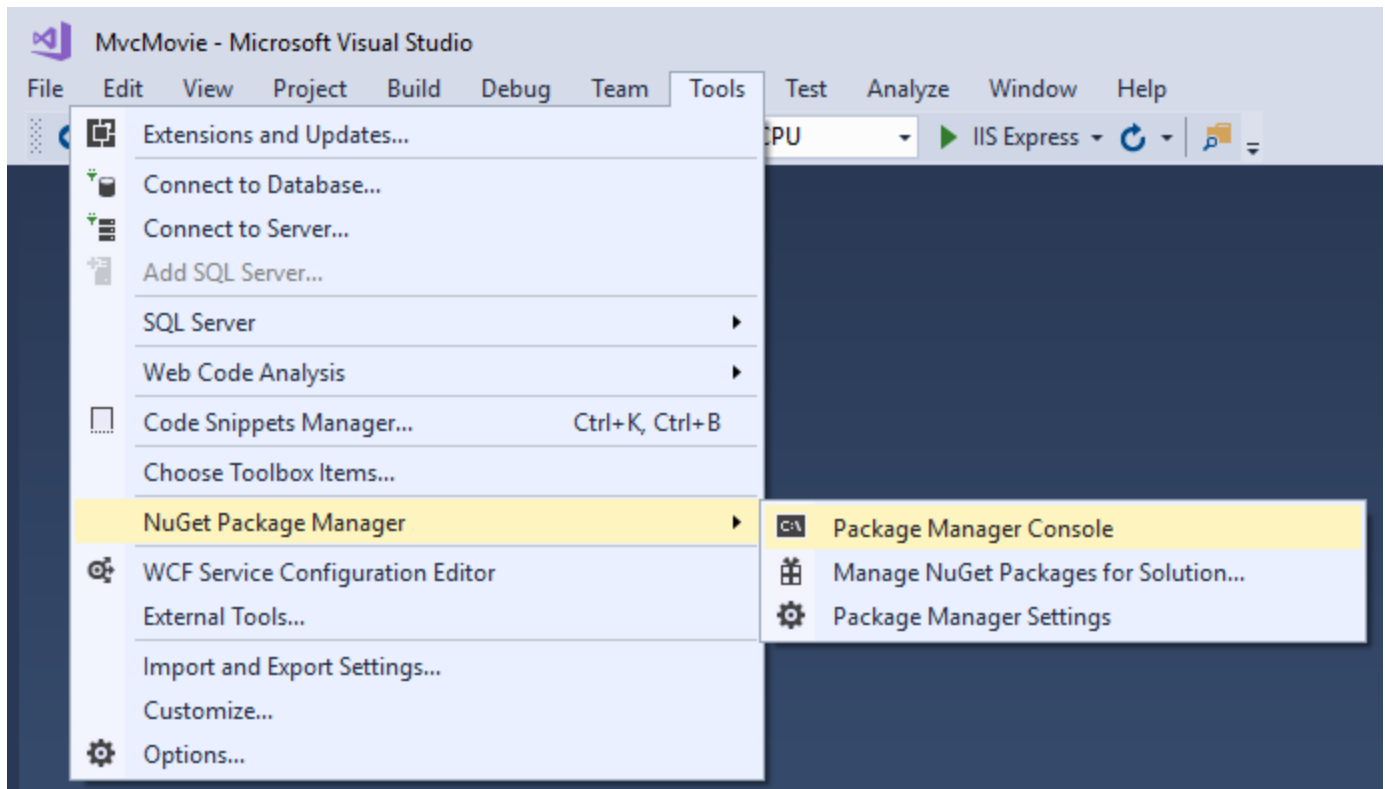
You need to create the database, and you'll use the EF Core [Migrations](#) feature to do that. Migrations lets you create a database that matches your data model and update the database schema when your data model changes.

## Add EF tooling and perform initial migration

In this section you'll use the Package Manager Console (PMC) to:

- Add the Entity Framework Core Tools package. This package is required to add migrations and update the database.
- 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
```

```
Install-Package Microsoft.EntityFrameworkCore.Tools
Add-Migration Initial
Update-Database
```

Note: See the [CLI approach](#) if you have problems with the PMC.

The `Add-Migration` command creates code to create the initial database schema. The schema is based on the model specified in the `DbContext` (In the `*Data/MvcMovieContext.cs` file). The `Initial` argument is used to name the migrations. You can use any name, but by convention you choose a name that describes the migration. See [Introduction to migrations](#) for more information.<sup>1</sup>

The `Update-Database` command runs the `Up` method in the `Migrations/<timestamp>_InitialCreate.cs` file, which creates the database.

You can perform the preceding steps using the command-line interface (CLI) rather than the PMC:

- Add [EF Core tooling](#) to the `.csproj` file.
- Run the following commands from the console (in the project directory):

```
consoleCopy
```

```
dotnet ef migrations add InitialCreate  
dotnet ef database update
```

43

## Test the app

- Run the app and tap the Mvc Movie link.
- Tap the Create New link and create a movie.

Create - Movie App

localhost:1234/Movies/

MvcMovie

## Create

### Movie

---

**Title**

**Release Date**

**Genre**

**Price**

[Back to List](#)

---

© 2017 - MvcMovie

- You may not be able to enter decimal points or commas in the **Price** field. To support [jQuery validation](#) for non-English locales that use a comma (",") for a decimal point, and non US-English date formats, you must take steps to globalize

your app. See [Additional resources](#) for more information. For now, just enter whole numbers like 10.

- In some locales you need to specify the date format. See the highlighted code below.

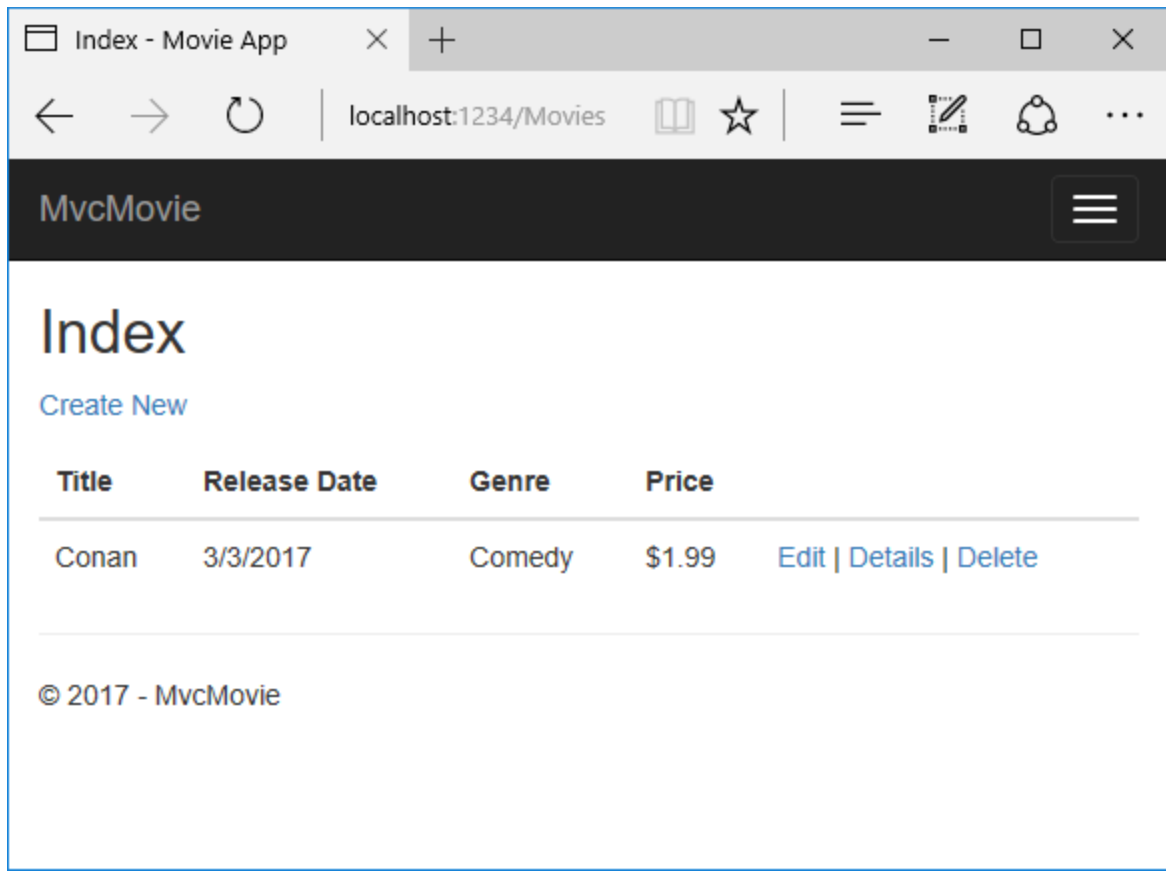
C#Copy

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

namespace MvcMovie.Models
{
    public class Movie
    {
        public int ID { get; set; }
        public string Title { get; set; }
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; }
        public decimal Price { get; set; }
    }
}
```

We'll talk about `DataAnnotations` later in the tutorial.

Tapping Create causes the form to be posted to the server, where the movie information is saved in a database. The app redirects to the `/Movies` URL, where the newly created movie information is displayed.<sup>3</sup>



2

Create a couple more movie entries. Try the Edit, Details, and Delete links, which are all functional.<sup>2</sup>

## Dependency Injection

Open the *Startup.cs* file and examine `ConfigureServices`:

C#Copy

```
public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddMvc();
}
```

```
services.AddDbContext<MvcMovieContext>(options =>
```

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

```
}
```

The highlighted code above shows the movie database context being added to the [Dependency Injection](#) container. The line following `services.AddDbContext<MvcMovieContext>(options =>` is not shown (see your code). It specifies the database to use and the connection string. `=>` is a [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](#) to inject the database context (`MvcMovieContext`) into the controller. The database context is used in each of the [CRUD](#) 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 `ViewData` dictionary. 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.<sup>3</sup>

Examine the generated `Details` method in the *Controllers/MoviesController.cs* file:<sup>6</sup>

C#Copy

```
// GET: Movies/Details/5
```



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

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

    return View(movie);
}
```

The `id` parameter is generally passed as route data. For example `http://localhost:5000/movies/details/1` sets:

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

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

```
http://localhost:1234/movies/details?id=1
```

The `id` parameter is defined as a [nullable type](#) (`int?`) in case an ID value is not provided.

A [lambda expression](#) is passed in to `SingleOrDefaultAsync` to select movie entities that match the route data or query string value.

C#Copy

```
var movie = await _context.Movie
    .SingleOrDefaultAsync(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";
}

<h2>Details</h2>

<div>
    <h4>Movie</h4>
    <hr />
    <dl class="dl-horizontal">
        <dt>
            @Html.DisplayNameFor(model => model.Title)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Title)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.ReleaseDate)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.ReleaseDate)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Genre)
        </dt>
        <dd>
            @Html.DisplayFor(model => model.Genre)
        </dd>
        <dt>
            @Html.DisplayNameFor(model => model.Price)
        </dt>
        <dd>
            @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, Visual Studio automatically included the following `@model` statement 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 `Model` object 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:5

htmlCopy

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

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

<h2>Index</h2>

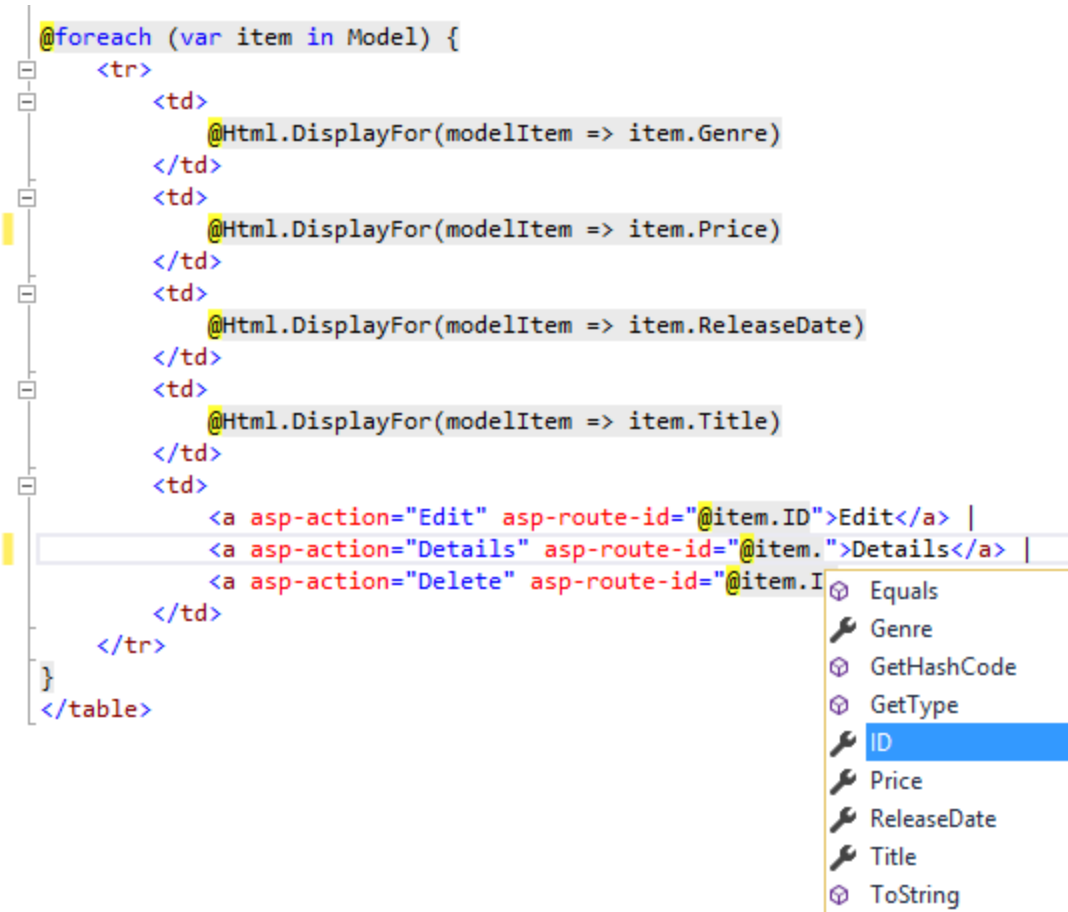
<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Title)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.ReleaseDate)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Genre)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Price)
            </th>
        </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](#)
- [Globalization and localization](#)

# Working with SQL Server LocalDB

By [Rick Anderson](#)

The `MvcMovieContext` object handles the task of connecting to the database and mapping `Movie` objects to database records. The database context is registered with the [Dependency Injection](#) container in the `ConfigureServices` method in the `Startup.cs` file:

C#Copy

```
public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddMvc();
```

```
    services.AddDbContext<MvcMovieContext>(options =>
```

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

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

JavaScriptCopy

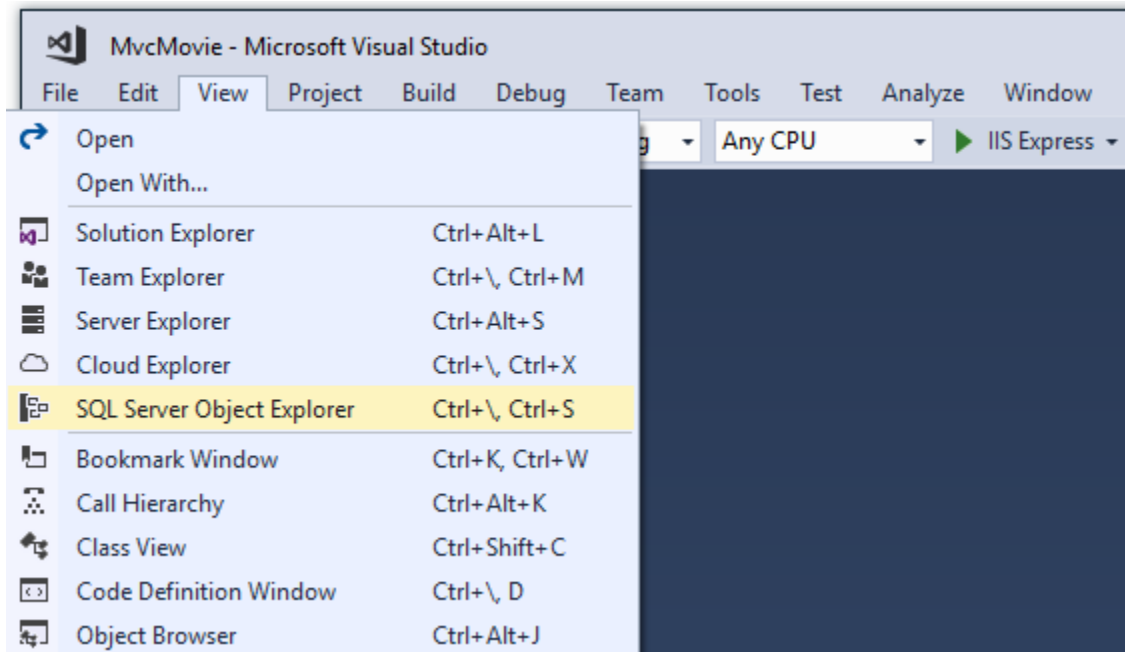
```
"ConnectionStrings": {
  "MvcMovieContext": "Server=(localdb)\\mssqllocaldb;Database=MvcMovieContext-20613a4b-deb5-4145-b6cc-a5fd19afda13;Trusted_Connection=True;MultipleActiveResultSets=true"
}
```

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](#) for more information.

## SQL Server Express LocalDB

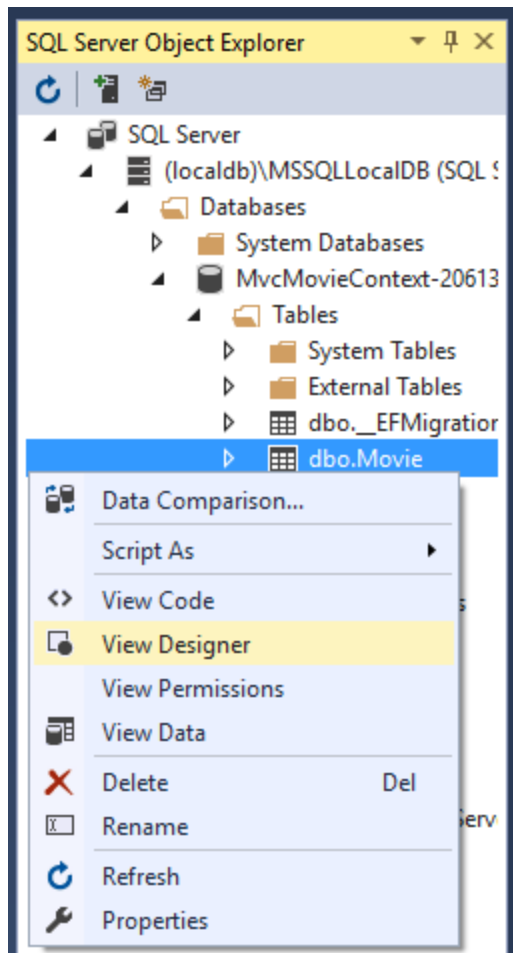
LocalDB is a lightweight version of the SQL Server Express Database Engine that is targeted for program development. LocalDB starts on demand and runs in user mode, so there is no complex configuration. By default, LocalDB database creates "\*.mdf" files in the *C:/Users/<user>* directory.

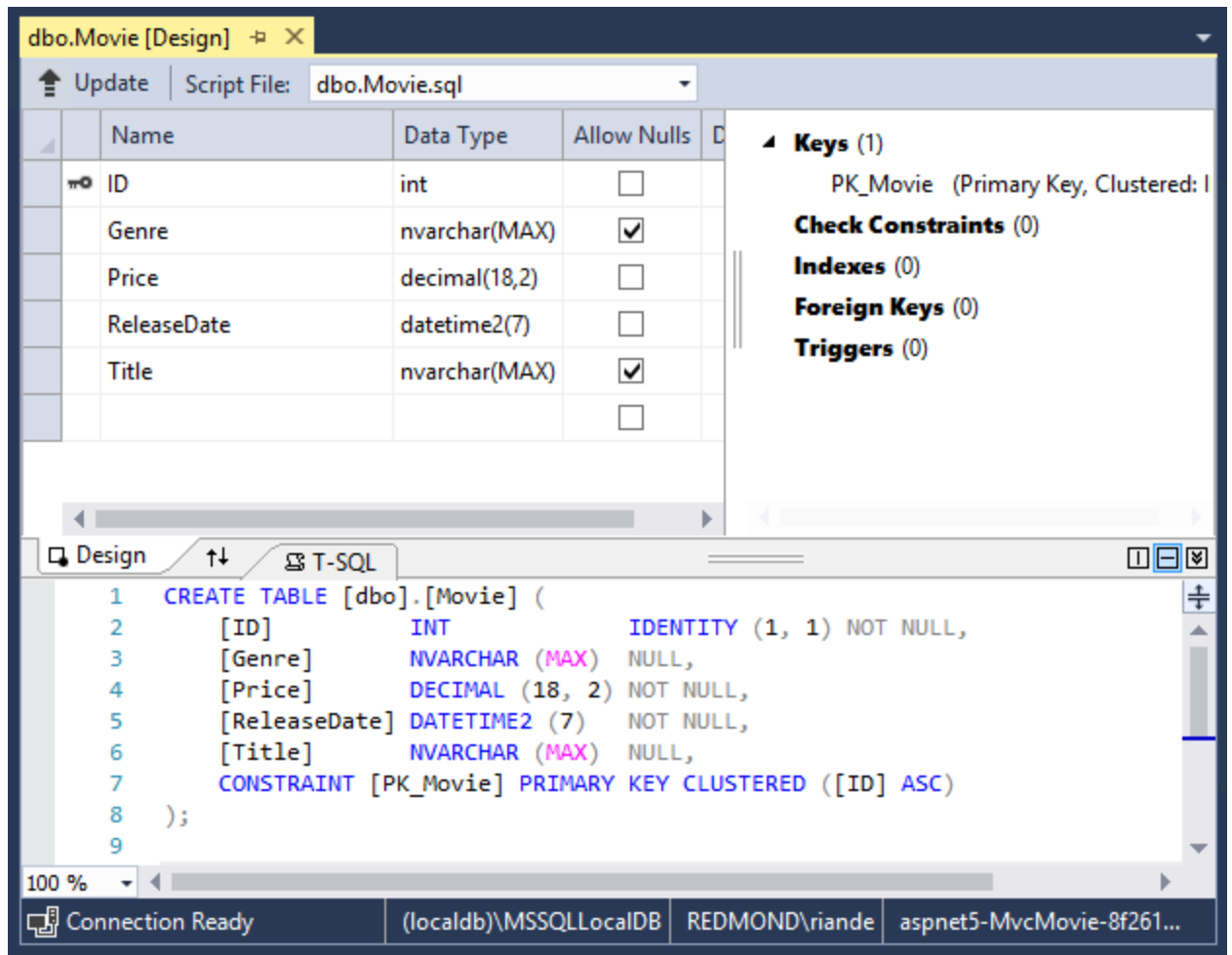
- From the View menu, open SQL Server Object Explorer (SSOX).



- Right click on the `Movie` table > View Designer



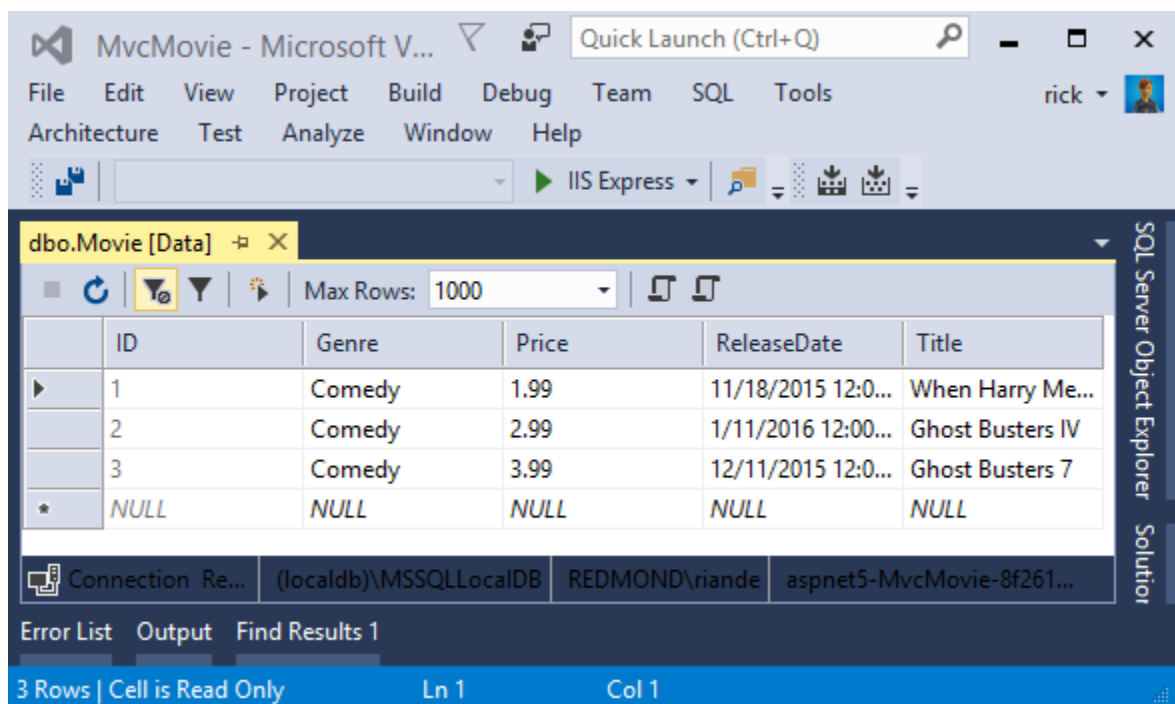
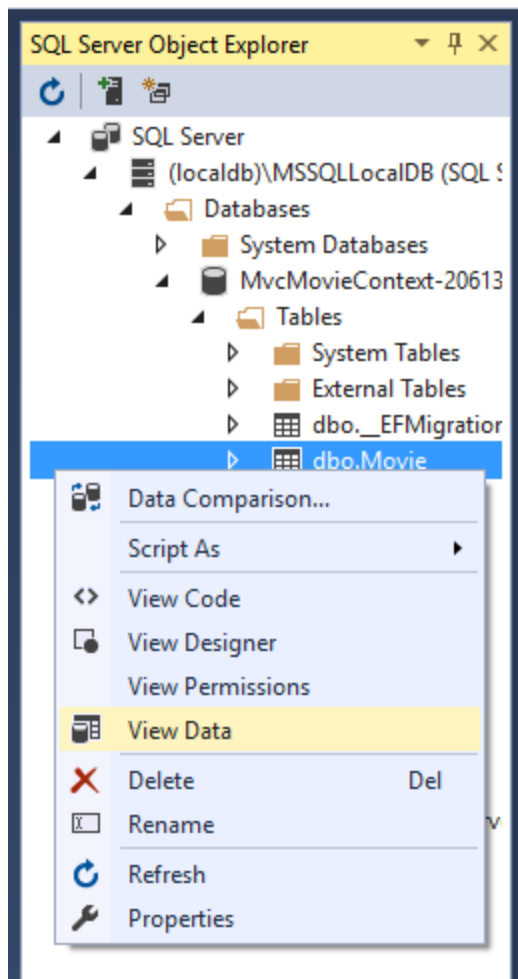




5

Note the key icon next to `ID`. By default, EF will make a property named `ID` the primary key.<sup>4</sup>

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



## Seed the database

Create a new class named `SeedData` in the *Models* folder. Replace the generated code with the following:4

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("1989-1-11"),
                        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.4

C#Copy

```

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

```

Add the seed initializer to the end of the `Configure` method in the *Startup.cs* file:3

C#Copy

```

app.UseStaticFiles();
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
});

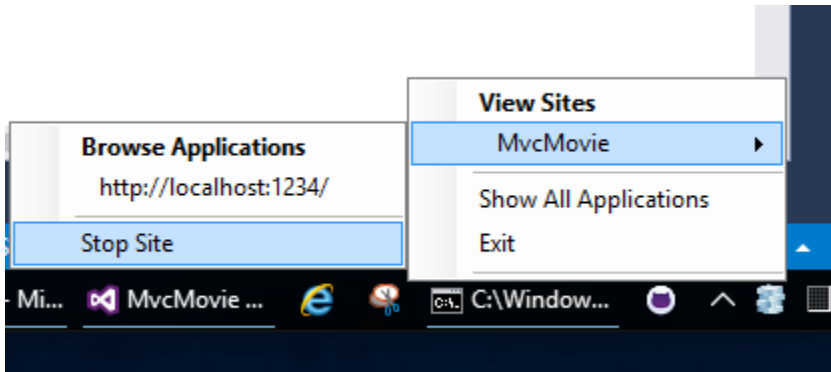
SeedData.Initialize(app.ApplicationServices);
}

```

```
}  
}
```

Test the app

- 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

10

The app shows the seeded data.

Movie App

localhost:5000/Movies

MvcMovie

Create New

Title	ReleaseDate	Genre	Price	
When Harry Met Sally	1/11/1989 12:00:00 AM	Romantic Comedy	7.99	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
Ghostbusters	3/13/1984 12:00:00 AM	Comedy	8.99	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
Ghostbusters 2	2/23/1986 12:00:00 AM	Comedy	9.99	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
Rio Bravo	4/15/1959 12:00:00 AM	Western	3.99	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>

© 2017 - MvcMovie

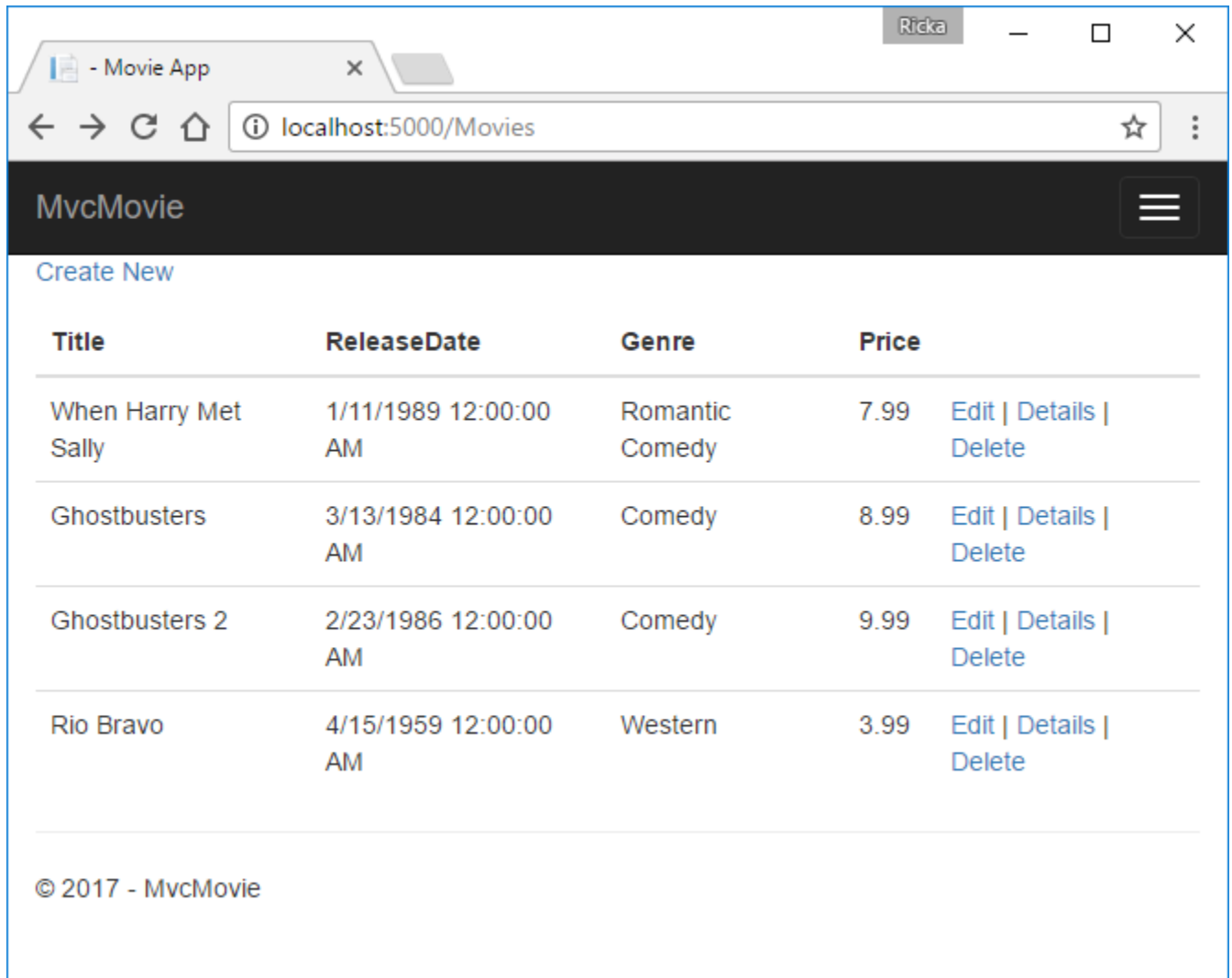




# Controller methods and views

By [Rick Anderson](#)

We have a good start to the movie app, but the presentation is not 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.Collections.Generic;  
using System.Linq;
```

```
using System.Threading.Tasks;

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; }
public decimal Price { get; set; }
    }
}
```

Right click on a red squiggly line > Quick Actions and Refactorings.

```
using System;
```

```
namespace MvcMovie.Models
```

```
{
```

```
    8 references | 0 changes | 0 authors, 0 changes
```

```
    public class Movie
```

```
    {
```

```
        7 references | 0 changes | 0 authors, 0 changes | 0 exceptions
```

```
        public int ID { get; set; }
```

```
        4 references | 0 changes | 0 authors, 0 changes | 0 exceptions
```

```
        public string Title { get; set; }
```

```
        [Display(Name = "Release Date")]
```

```
        [DataType(DataType.Date)]
```

```
        public D
```

```
        4 reference
```

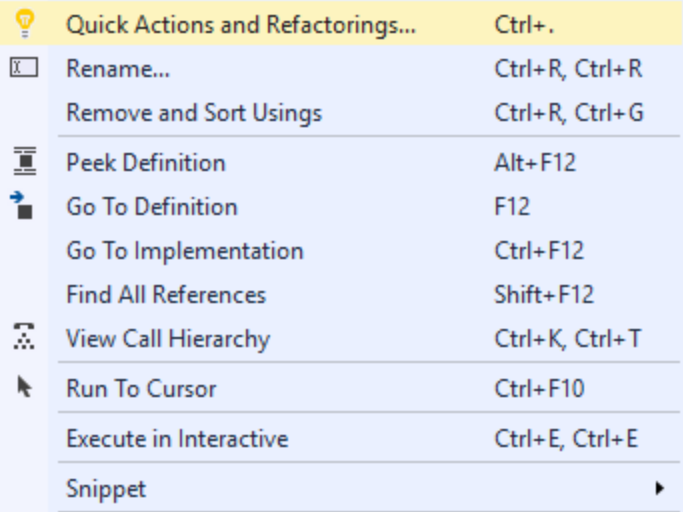
```
        public s
```

```
        4 reference
```

```
        public d
```

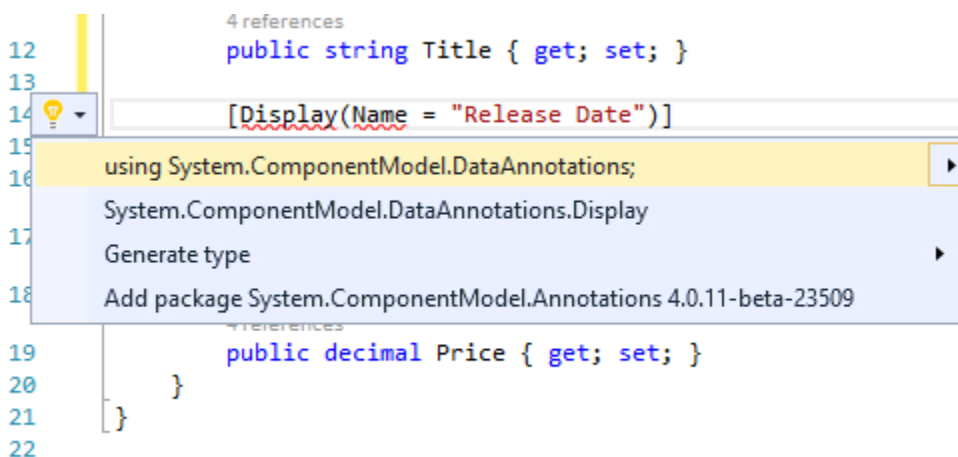
```
    }
```

```
}
```



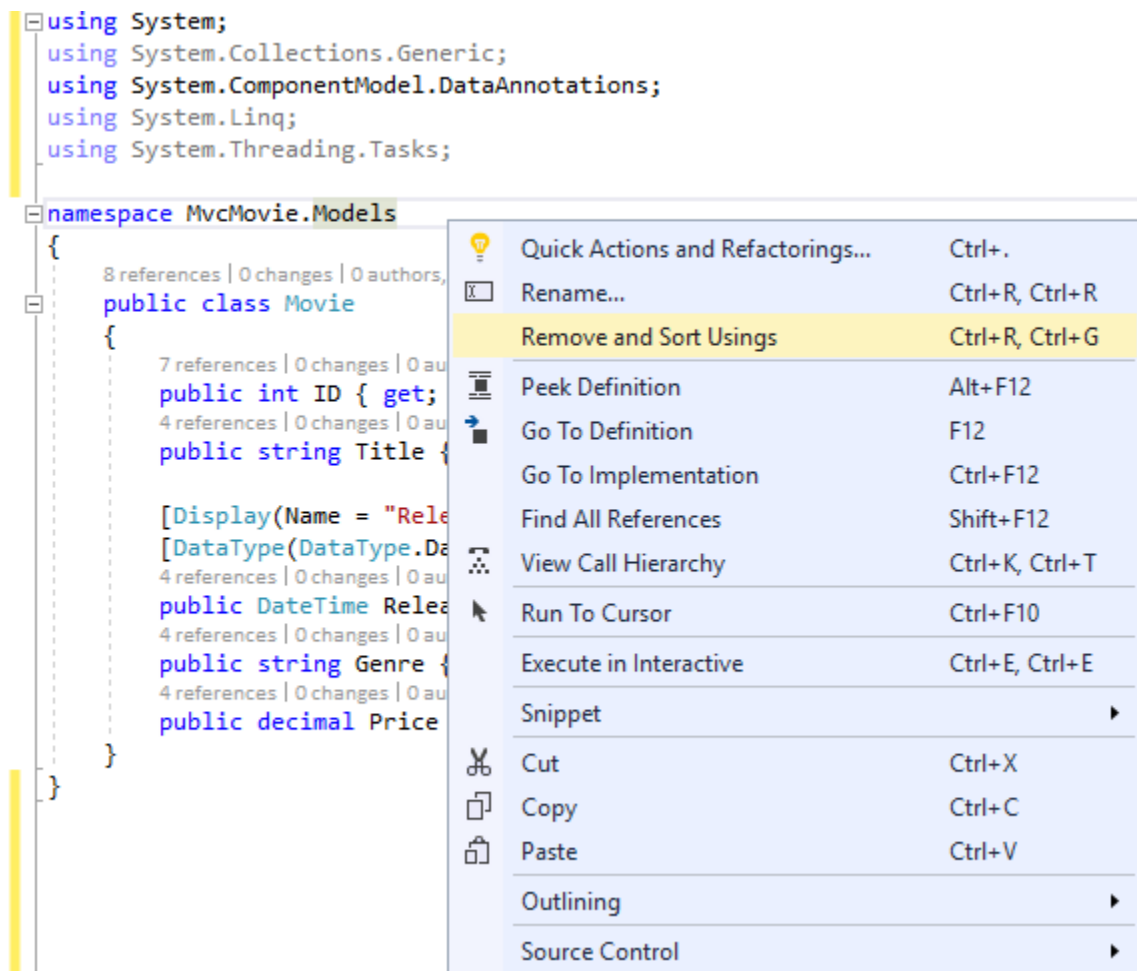
4

Tap `using System.ComponentModel.DataAnnotations;`



Visual studio adds `using System.ComponentModel.DataAnnotations;`.

Let's remove the `using` statements that are not needed. They show up by default in a light grey font. Right click anywhere in the *Movie.cs* file > Remove and Sort Usings.



The updated code:

C#Copy

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

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; }
        public decimal Price { get; set; }
    }
}

```

We'll cover [DataAnnotations](#) in the next tutorial. The [Display](#) attribute specifies what to display for the name of a field (in this case "Release Date" instead of "ReleaseDate"). The [DataType](#) attribute specifies the type of the data (Date), so the time information stored in the field is not displayed.

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

Index - Movie App

localhost:1234/Movies

MvcMovie

## Index

[Create New](#)

Genre	Price	Release Date	Title	
Romantic Comedy	7.99	1/11/1989	When Harry Met Sally	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
Comedy	8.99	3/13/1984	Ghostbusters	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
Comedy	9.99	2/23/1986	Ghostbusters 2	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
Western	3.99	4/15/1959	Rio Bravo	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>

© 2016 - MvcMovie

<http://localhost:1234/Movies/Edit/5>

The Edit, Details, and Delete links are generated by the MVC Core 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](#) enable server-side code to participate in creating and rendering HTML elements in Razor files. In the code above, the `AnchorTagHelper` dynamically generates the HTML `href` attribute value from the controller action method and route id. You use View Source from your favorite browser or use the developer tools to examine the generated markup. A portion of the generated HTML is shown below:

htmlCopy

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

Recall the format for [routing](#) set in the *Startup.cs* file:

C#Copy

```
app.UseMvc(routes =>  
{  
    routes.MapRoute(  
        name: "default",  
        template: "{controller=Home}/{action=Index}/{id?}");  
});
```

ASP.NET Core translates `http://localhost:1234/Movies/Edit/4` into a request to the `Edit` action method of the `Movies` controller with the parameter `Id` of 4. (Controller methods are also known as action methods.)

[Tag Helpers](#) are one of the most popular new features in ASP.NET Core. See [Additional resources](#) for more information.

Open the `Movies` controller and examine the two `Edit` action methods. The following code shows the `HTTP GET Edit` method, which fetches the movie and populates the edit form generated by the `Edit.cshtml` Razor file.<sup>4</sup>

C#Copy

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

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

The following code shows the `HTTP POST Edit` method, which processes the posted movie values:<sup>2</sup>

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](#). You should only include properties in the `[Bind]` attribute that you want to change. See [Protect your controller from over-posting](#) for more information. [ViewModels](#) provide an alternative approach to prevent over-posting.

Notice the second `Edit` action method is preceded by the `[HttpPost]` attribute.<sup>2</sup>

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 `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.<sup>1</sup>

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

HTMLCopy

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

The [Form Tag Helper](#) generates a hidden anti-forgery token that must match the `[ValidateAntiForgeryToken]` generated anti-forgery token in the `Edit` method of the `Movies` controller. For more information, see [Anti-Request Forgery](#).

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.<sup>4</sup>

C#Copy

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

    var movie = await _context.Movie.SingleOrDefaultAsync(m => m.ID == 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:<sup>2</sup>

HTMLCopy

```
@model MvcMovie.Models.Movie

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

<h2>Edit</h2>

<form asp-action="Edit">
    <div class="form-horizontal">
        <h4>Movie</h4>
        <hr />
        <div asp-validation-summary="ModelOnly" class="text-danger"></div>
        <input type="hidden" asp-for="ID" />
        <div class="form-group">
```

```

        <label asp-for="Title" class="col-md-2 control-label"></label>
        <div class="col-md-10">
            <input asp-for="Title" class="form-control" />
            <span asp-validation-for="Title" class="text-danger"></span>
        </div>
    </div>
    <div class="form-group">
        <label asp-for="ReleaseDate" class="col-md-2 control-label"></label>
        <div class="col-md-10">
            <input asp-for="ReleaseDate" class="form-control" />
            <span asp-validation-for="ReleaseDate" class="text-danger"></span>
        </div>
    </div>
    <div class="form-group">
        <label asp-for="Genre" class="col-md-2 control-label"></label>
        <div class="col-md-10">
            <input asp-for="Genre" class="form-control" />
            <span asp-validation-for="Genre" class="text-danger"></span>
        </div>
    </div>
    <div class="form-group">
        <label asp-for="Price" class="col-md-2 control-label"></label>
        <div class="col-md-10">
            <input asp-for="Price" class="form-control" />
            <span asp-validation-for="Price" class="text-danger"></span>
        </div>
    </div>
    <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>
</form>

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

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

```

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

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

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

HTMLCopy

```
<form action="/Movies/Edit/7" method="post">
  <div class="form-horizontal">
    <h4>Movie</h4>
    <hr />
    <div class="text-danger" />
    <input type="hidden" data-val="true" data-val-required="The ID field is
required." id="ID" name="ID" value="7" />
    <div class="form-group">
      <label class="control-label col-md-2" for="Genre" />
      <div class="col-md-10">
        <input class="form-control" type="text" id="Genre" name="Genre"
value="Western" />
        <span class="text-danger field-validation-valid" data-valmsg-
for="Genre" data-valmsg-replace="true"></span>
      </div>
    </div>
    <div class="form-group">
      <label class="control-label col-md-2" for="Price" />
      <div class="col-md-10">
        <input class="form-control" type="text" data-val="true" data-val-
number="The field Price must be a number." data-val-required="The Price field is
required." id="Price" name="Price" value="3.99" />
        <span class="text-danger field-validation-valid" data-valmsg-
for="Price" data-valmsg-replace="true"></span>
      </div>
    </div>
    <!-- Markup removed for brevity -->
    <div class="form-group">
      <div class="col-md-offset-2 col-md-10">
```

```

        <input type="submit" value="Save" class="btn btn-default" />
    </div>
</div>
</div>
<input name="__RequestVerificationToken" type="hidden"
value="CfDJ8Inyxgp63fRFqUePGvuI5jGZsloJu1L7X9le1gy7NCILSduCRx9jDQClrV9pOTTmqUyXnJBXhm
rjcUVDJyDUMm7-MF_9rK8aAZdRdlOri7FmKVkRe_2v5LIHGKFcTjPrWPYnc9AdSbomkiOSaTEg7RU" />
</form>

```

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

## Processing the POST Request

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

C#Copy

```

// POST: Movies/Edit/5
// To protect from overposting attacks, 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 `[ValidateAntiForgeryToken]` attribute validates the hidden [XSRF](#) token generated by the anti-forgery token generator in the [Form Tag Helper](#)

The [model binding](#) system takes the posted form values and creates a `Movie` object that's passed as the `movie` parameter. The `ModelState.IsValid` 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 is not posted. If JavaScript is disabled, you won't have client side validation but the server will detect the posted values that are not valid, and the form values will be redisplayed with error messages. Later in the tutorial we examine [Model Validation](#) in more detail. The [Validation Tag Helper](#) in the `Views/Movies/Edit.cshtml` view template takes care of displaying appropriate error messages.

http://localhost:1235/Movies/Edit/

Edit - Movie App

Mvc Movie

## Edit

Movie

**Genre**

Western

**Price**

abc

The field Price must be a number.

**Release Date**

xyz

Please enter a valid date.

**Title**

Rio Bravo

Save

[Back to List](#)

© 2015 - MvcMovie

All the `HttpGet` methods in the movie controller follow a similar pattern. They get a movie object (or list of objects, in the case of `Index`), and pass the object (model) to the view. The `Create` method passes an empty movie object to the `Create` view. All the

methods that create, edit, delete, or otherwise modify data do so in the `[HttpPost]` overload of the method. Modifying data in an `HTTP GET` method is a security risk. Modifying data in an `HTTP GET` method also violates HTTP best practices and the architectural [REST](#) pattern, which specifies that GET requests should not 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.<sup>2</sup>

## Additional resources

- [Globalization and localization](#)
- [Introduction to Tag Helpers](#)
- [Authoring Tag Helpers](#)
- [Anti-Request Forgery](#)
- Protect your controller from [over-posting](#)
- [ViewModels](#)
- [Form Tag Helper](#)
- [Input Tag Helper](#)
- [Label Tag Helper](#)
- [Select Tag Helper](#)
- [Validation Tag Helper](#)



# Adding Search to an ASP.NET Core MVC app

By [Rick Anderson](#)

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

Update the `Index` method with the following code:12

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](#) query to select the movies:7

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:2

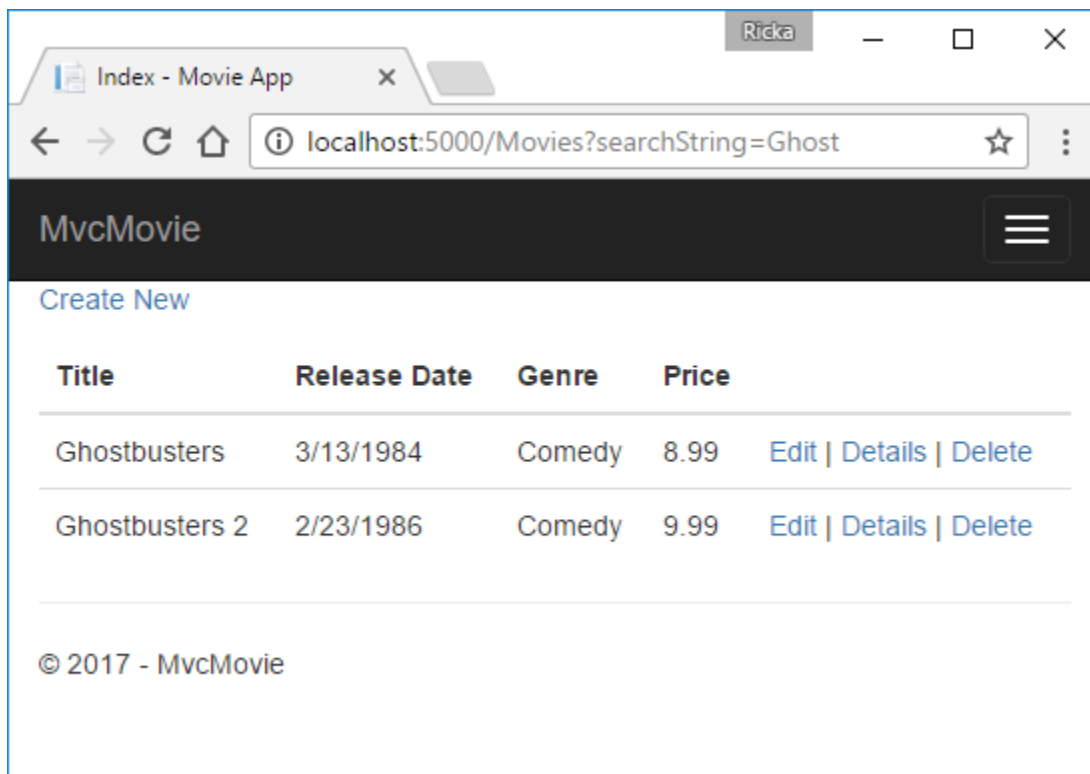
C#Copy

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

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

Note: The [Contains](#) method is run on the database, not in the c# code shown above. The case sensitivity on the query depends on the database and the collation. On SQL Server, [Contains](#) maps to [SQL LIKE](#), which is case insensitive. In SQLite, 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.

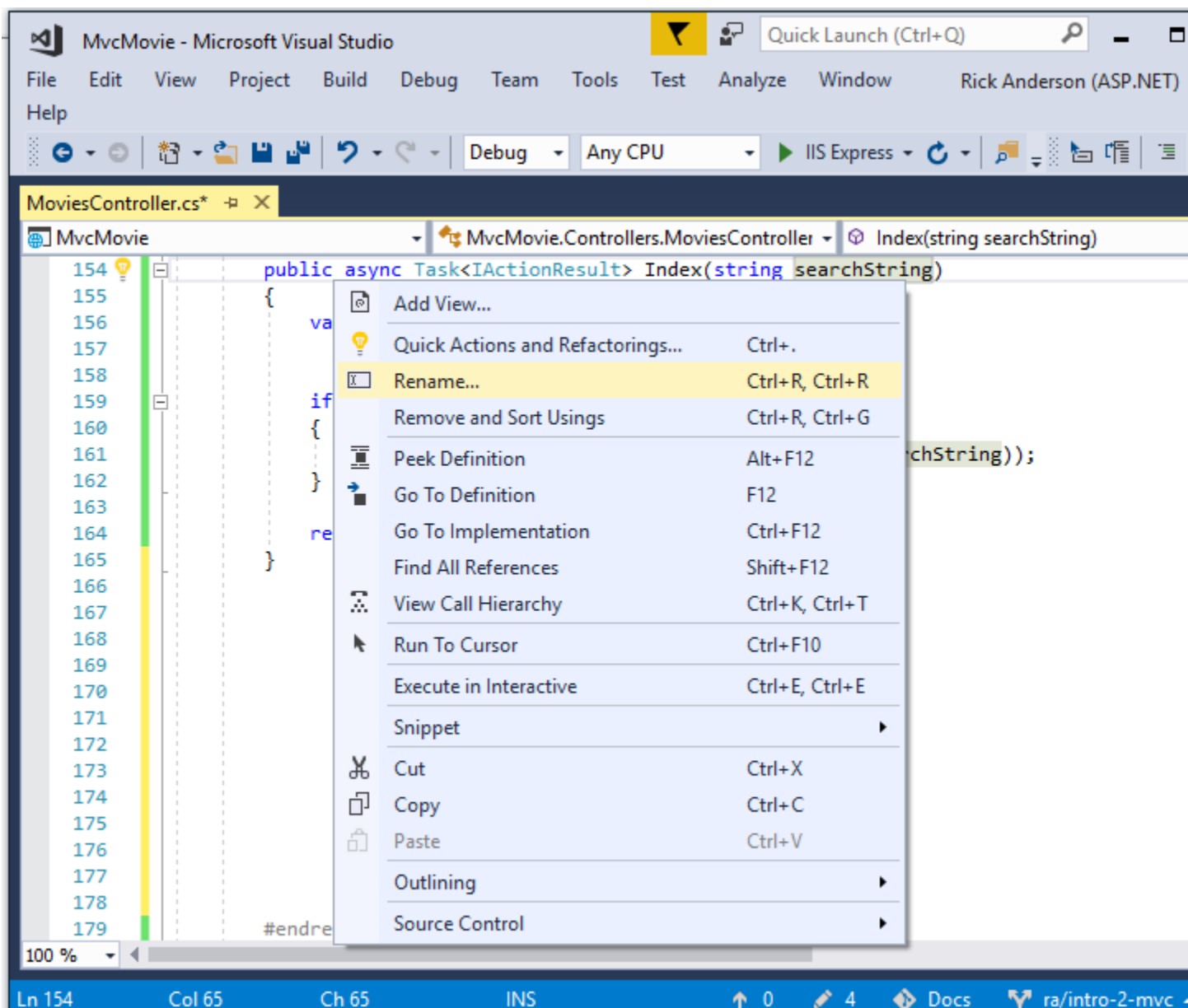


If you change the signature of the `Index` method to have a parameter named `id`, the `id` parameter will match the optional `{id}` placeholder for the default routes set in `Startup.cs.7`

## C#Copy

```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "{controller=Home}/{action=Index}/{id?}");
});
```

You can quickly rename the `searchString` parameter to `id` with the rename command. Right click on `searchString` > Rename.



The rename targets are highlighted.

```
public ActionResult 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(movies);
}
```

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

```
public ActionResult 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(movies);
}
```

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

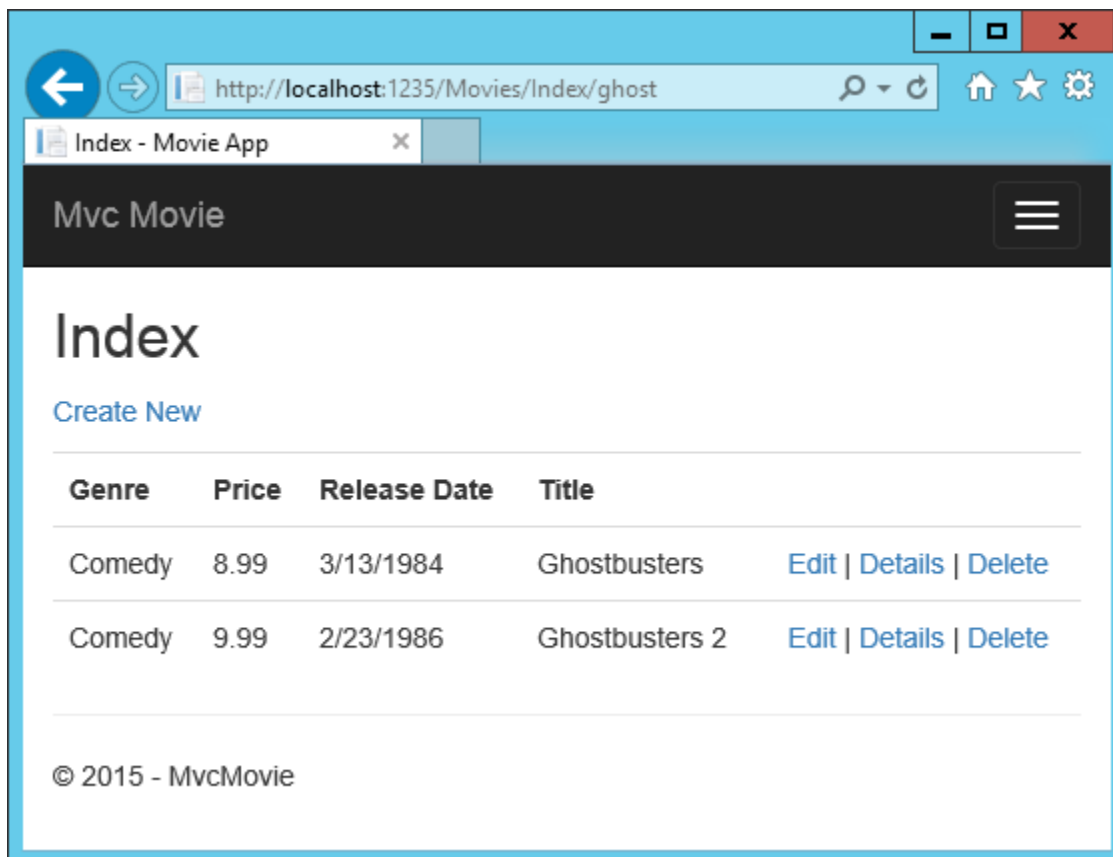
```
public async Task<IActionResult> Index(string id)

{
    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 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](#), so when you submit the form, the filter string is posted to the `Index` action of the movies controller. Save your changes and then test the filter.

Index - Movie App

localhost:1899/Movies

MvcMovie Window Snip

Create New

Title:

Genre	Price	Release Date	Title	
Romantic Comedy	7.99	1/11/1989	When Harry Met Sally	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
Comedy	8.99	3/13/1984	Ghostbusters	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
Comedy	9.99	2/23/1986	Ghostbusters 2	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>
Western	3.99	4/15/1959	Rio Bravo	<a href="#">Edit</a>   <a href="#">Details</a>   <a href="#">Delete</a>

© 2016 - MvcMovie

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.

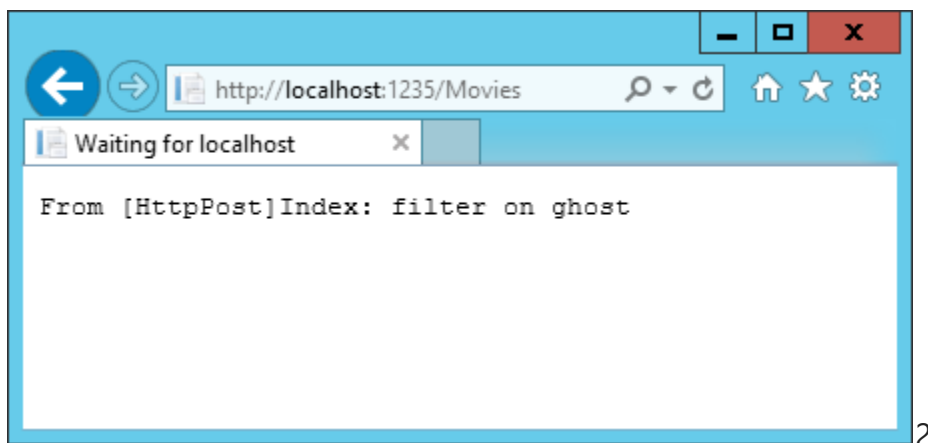
C#Copy

```
[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](#). You can verify that with the browser Developer tools or the excellent [Fiddler tool](#). The image below shows the Chrome browser Developer tools:



Index - Movie App

localhost:5000/Movies

MvcMovie

Create New

Title:

Filter

Title	Release Date
Ghostbusters	3/13/198
Ghostbusters	2/23/1982

© 2017 - MvcMovie

Elements

Console

Sources

Network

Timeline

Profiles

Filter

Regex

Hide data URLs

All

XHR

JS

CSS

Img

Media

Font

Doc

WS

Manifest

Other

Name

Headers

Preview

Response

Cookies

Timing

Movies

jquery.js

/lib/jquery/dist

bootstrap.js

/lib/bootstrap/dist/js

site.js?v=EWaMeWsJBYWmL2g...

/js

General

Request URL: http://localhost:5000/Movies

Request Method: POST

Status Code: 200 OK

Remote Address: [::1]:5000

Referrer Policy: no-referrer-when-downgrade

Response Headers

view source

Cache-Control: no-cache

Content-Type: text/html; charset=utf-8

Date: Mon, 10 Apr 2017 00:10:32 GMT

Pragma: no-cache

Server: Kestrel

Transfer-Encoding: chunked

Request Headers

view source

Accept: text/html,application/xhtml+xml,application/javascript;q=0.9,image/webp,\*/\*;q=0.8

Accept-Encoding: gzip, deflate, br

Accept-Language: en-US,en;q=0.8

Cache-Control: max-age=0

Connection: keep-alive

Content-Length: 201

Content-Type: application/x-www-form-urlencoded

Cookie: .AspNetCore.Antiforgery.txPTUN878m8=CfDJL5pAq2aeCj59HP3vvTrLPK1krIeXsJFchnazwWjLRXzWQJT8bQg-xCdy7DbpfcQ-Hi2HAX0in1R838CvFU6oyWz7VIKmiKS-YZR0pBdNaP0mTxZX9JRMj\_xX\_zIig; .AspNetCore.Antiforgery.Mkg1\_D\_R5qY=CfDJ8B98MxUFL5pAq2aeCj59HP2tNs0B0CoKKfe2wXo9rL6Z-9YP41jarbKyJ\_I5aQvz9BMWGFpPwwbHC-CkWyXAsFwKQyP0MYsYab98gk-z\_M4jH1\_Hw90DBZJuBVtm8

Host: localhost:5000

Origin: http://localhost:5000

Referer: http://localhost:5000/Movies

Upgrade-Insecure-Requests: 1

User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/57.0.2980.130 Safari/537.36

Form Data

view source

view URL encoded

SearchString: Ghost

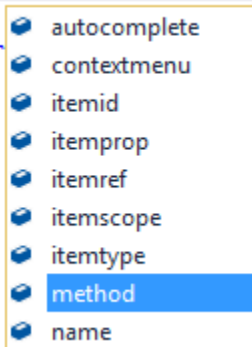
\_RequestVerificationToken: CfDJ8B98MxUFL5pAq2aeCj59HP2tNs0B0CoKKfe2wXo9rL6Z-9YP41jarbKyJ\_I5aQvz9BMWGFpPwwbHC-CkWyXAsFwKQyP0MYsYab98gk-z\_M4jH1\_Hw90DBZJuBVtm8

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

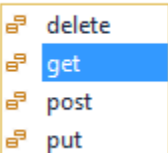
Because the search parameter is in the request body and not the URL, you can't capture that search information to bookmark or share with others. We'll fix this by specifying the request should be `HTTP GET`.

Notice how IntelliSense helps us update the markup.

```
<form asp-controller="Movies" asp-action="Index" m>
  <p>
    Title: <input type="text" name="SearchStr
    <input type="submit" value="Filter" />
  </p>
</form>
```



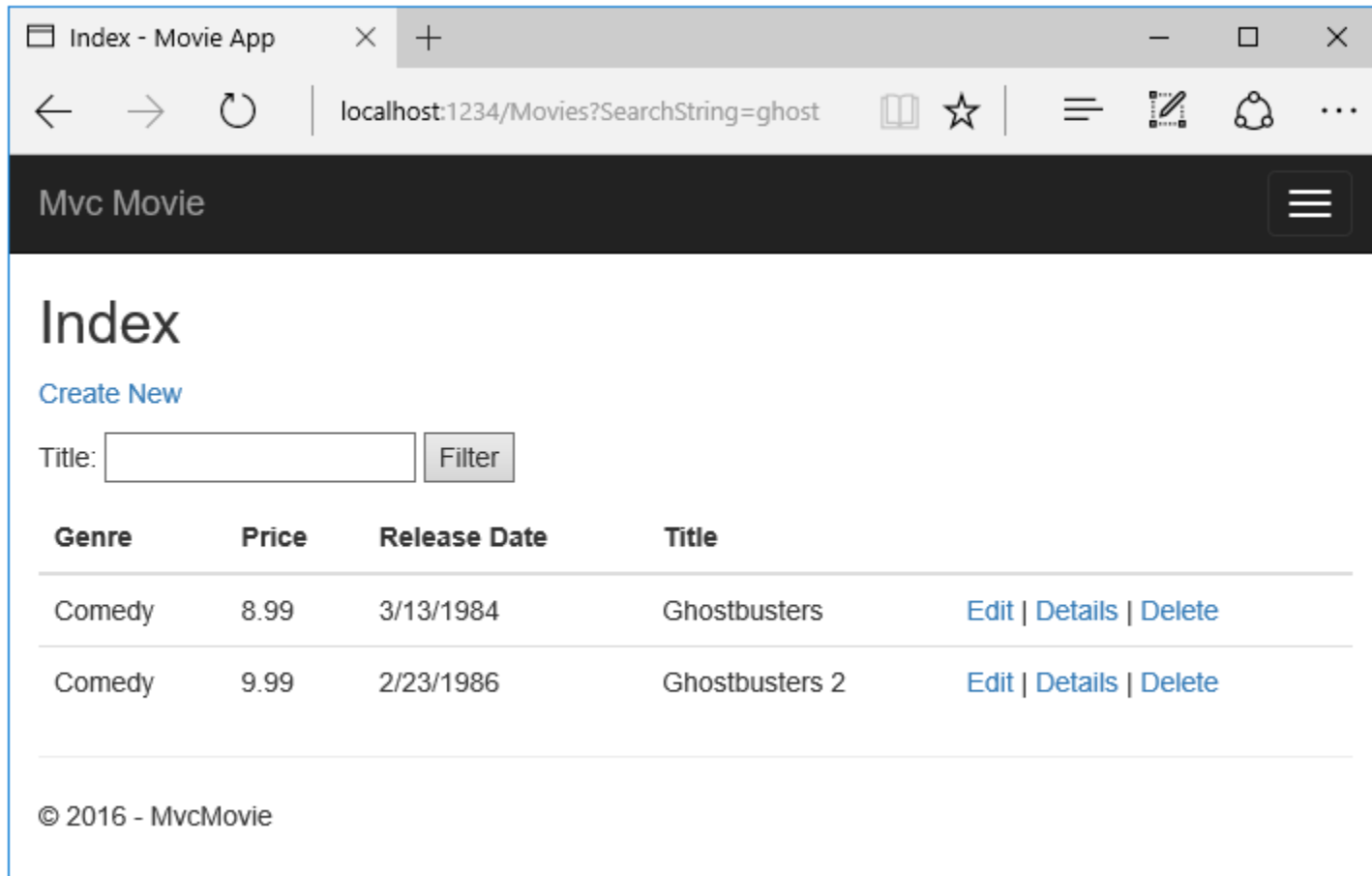
```
<form asp-controller="Movies" asp-action="Index" method="">>
  <p>
    Title: <input type="text" name="SearchString">
    <input type="submit" value="Filter" />
  </p>
</form>
```



Notice the distinctive font in the `<form>` tag. That distinctive font indicates the tag is supported by [Tag Helpers](#).<sup>11</sup>

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

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:4

htmlCopy

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

## Adding 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; }
    }
}

```

The movie-genre view model will contain:

- A list of movies.
- A `SelectList` containing the list of genres. This will allow the user to select a genre from the list.
- `movieGenre`, which contains the selected genre.

Replace the `Index` method in `MoviesController.cs` with the following code:12

C#Copy

```

// Requires using Microsoft.AspNetCore.Mvc.Rendering;
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();
}

```

```

        movieGenreVM.genres = new SelectList(await genreQuery.Distinct().ToListAsync());
        movieGenreVM.movies = await movies.ToListAsync();

        return View(movieGenreVM);
    }

```

The following code is a `LINQ` query that retrieves all the genres from the database.2

C#Copy

```

// Use LINQ to get list of genres.
IEnumerable<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).

C#Copy

```

movieGenreVM.genres = new SelectList(await genreQuery.Distinct().ToListAsync())

```

## Adding search by genre to the Index view

Update `Index.cshtml` as follows:11

HTMLCopy

```

@model MvcMovie.Models.MovieGenreViewModel

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

<h2>Index</h2>

<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" name="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>
        </tr>
    </thead>
    <tbody>
        @foreach (var item in Model.movies)
        {
            <tr>
                <td>
                    @Html.DisplayFor(modelItem => item.Title)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.ReleaseDate)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Genre)
                </td>
                <td>
                    @Html.DisplayFor(modelItem => item.Price)
                </td>
                <td>
                    <a asp-action="Edit" asp-route-id="@item.ID">Edit</a> |
                    <a asp-action="Details" asp-route-id="@item.ID">Details</a> |

```

```
        <a asp-action="Delete" asp-route-id="@item.ID">Delete</a>
      </td>
    </tr>
  }
</tbody>
</table>
```

Examine the lambda expression used in the following HTML Helper:

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

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

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

# Adding a New Field

By [Rick Anderson](#)

In this section you'll use [Entity Framework](#) Code First Migrations to add a new field to the model and migrate that change to the database.

When you use EF Code First to automatically create a database, Code First adds a table to the database to help track whether the schema of 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.

## Adding a Rating Property to the Movie Model

Open the *Models/Movie.cs* file and add a `Rating` property:

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; }
    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 also 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 `Edit` action methods to include the `Rating` property:

C#Copy



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

You also need to update the view templates in order to display, create and edit the new `Rating` property in the browser view.<sup>2</sup>

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

HTMLCopy

```
<table class="table">
  <thead>
    <tr>
      <th>
        @Html.DisplayNameFor(model => model.movies[0].Title)
      </th>
      <th>
        @Html.DisplayNameFor(model => model.movies[0].ReleaseDate)
      </th>
      <th>
        @Html.DisplayNameFor(model => model.movies[0].Genre)
      </th>
      <th>
        @Html.DisplayNameFor(model => model.movies[0].Price)
      </th>
      <th>
        @Html.DisplayNameFor(model => model.movies[0].Rating)
      </th>
    </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>


```

Update the `/Views/Movies/Create.cshtml` with a `Rating` field. You can copy/paste the previous "form group" and let IntelliSense help you update the fields. IntelliSense works with [Tag Helpers](#). Note: In the RTM version of Visual Studio 2017 you need to install the [Razor Language Services](#) for Razor IntelliSense. This will be fixed in the next release.

```

</div>
<div class="form-group">
    <label asp-for="Title" class="col-md-2 control-label"></label>
    <div class="col-md-10">
        <input asp-for="Title" class="form-control" />
        <span asp-validation-for="Title" class="text-danger" />
    </div>
</div>
<div class="form-group">
    <label asp-for="R" class="col-md-2 control-label"></label>
    <div class="col-
        <input asp-f
        <span asp-va
    </div>
</div>
<div class="form-gro
    <div class="col-
        <input type=
    </div>
</div>
</div>
</form>

```



The app won't work until we update the DB to include the new field. If you run it now, you'll get the following `SQLException`:

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

You're seeing this error 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 are 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.
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.

2

For this tutorial, we'll use Code First Migrations.

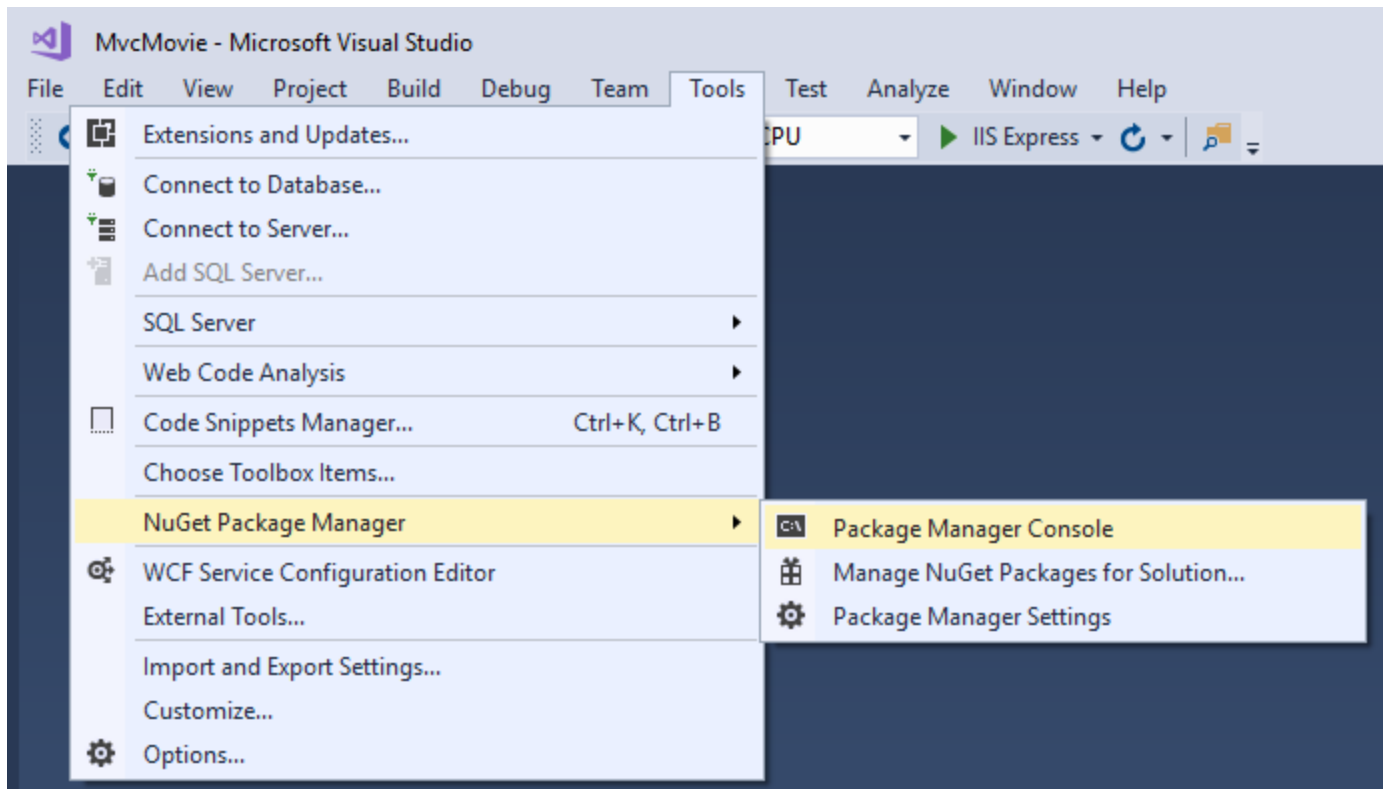
Update the `SeedData` class so that it provides a value for the new column. A sample change is shown below, but you'll want to make this change for each `new Movie`.<sup>2</sup>

C#Copy

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

Build the solution.

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



In the PMC, enter the following commands:<sup>16</sup>

```
PMCCopy
```

```
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. 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 you delete all the records in the DB, the initialize will seed the DB and include the `Rating` field. You can do this with the delete links in the browser or from SSOX.<sup>10</sup>

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

# Adding validation

By [Rick Anderson](#)

In this section you'll add validation logic to the `Movie` model, and you'll ensure that the validation rules are enforced any time a user creates or edits a movie.

## Keeping things DRY

One of the design tenets of MVC is [DRY](#) ("Don't Repeat Yourself"). ASP.NET 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.<sup>2</sup>

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; }

    [DisplayName = "Release Date"]
    [DataType(DataType.Date)]
```

```

public DateTime ReleaseDate { get; set; }

[Range(1, 100)]
[DataType(DataType.Currency)]
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-Z' '-'\s]*$")]
[StringLength(5)]
[Required]
public string Rating { get; set; }
}

```

The validation attributes specify behavior that you want to enforce on the model properties they are 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 (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.<sup>5</sup>

Having validation rules automatically enforced by ASP.NET 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.<sup>2</sup>

Create - Movie App

localhost:5000/Movies/Cre

MvcMovie

# Create

## Movie

Title

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

Release Date

mm/dd/yyyy

Genre

The Genre field is required.

Price

z

The field Price must be a number.

Rating

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

Create

[Back to List](#)

© 2017 - MvcMovie

## Note

You may not be able to enter decimal points or commas in the `Price` field. To support [jQuery validation](#) for non-English locales that use a comma (",") for a decimal point, and non US-English date formats, you must take steps to globalize your app. See [Additional resources](#) for more information. For now, just enter whole numbers like 10.

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 is not sent to the server until there are no client side validation errors. You can verify this by putting a break point in the `HTTP Post` method, by using the [Fiddler tool](#) , or the [F12 Developer tools](#).

## How validation works

You might wonder how the validation UI was generated without any updates to the code in the controller or views. The following code shows the two `Create` methods.<sup>1</sup>

C#Copy

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

// POST: Movies/Create
[HttpPost]
[ValidateAntiForgeryToken]
public async Task<IActionResult> Create(
```



```

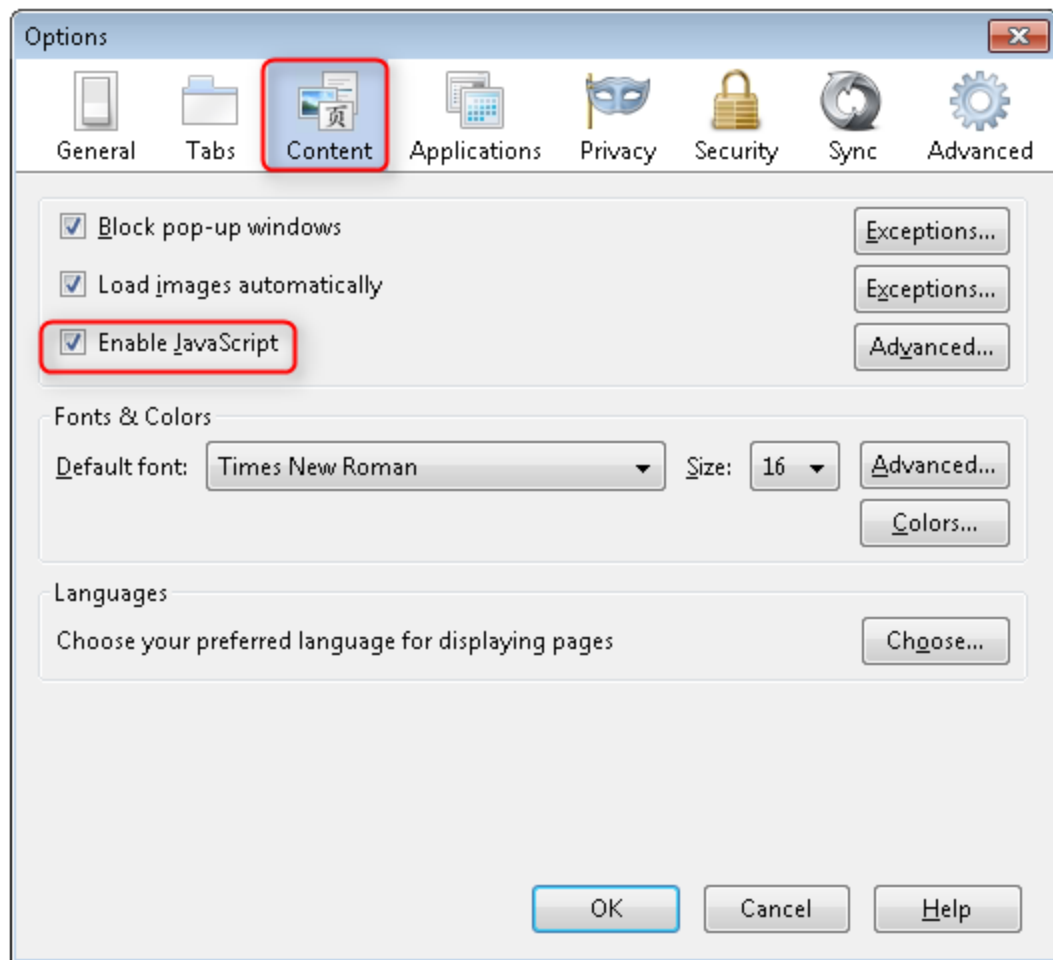
[Bind("ID,Title,ReleaseDate,Genre,Price, Rating")] Movie movie)
{
    if (ModelState.IsValid)
    {
        _context.Add(movie);
        await _context.SaveChangesAsync();
        return RedirectToAction("Index");
    }
    return View(movie);
}

```

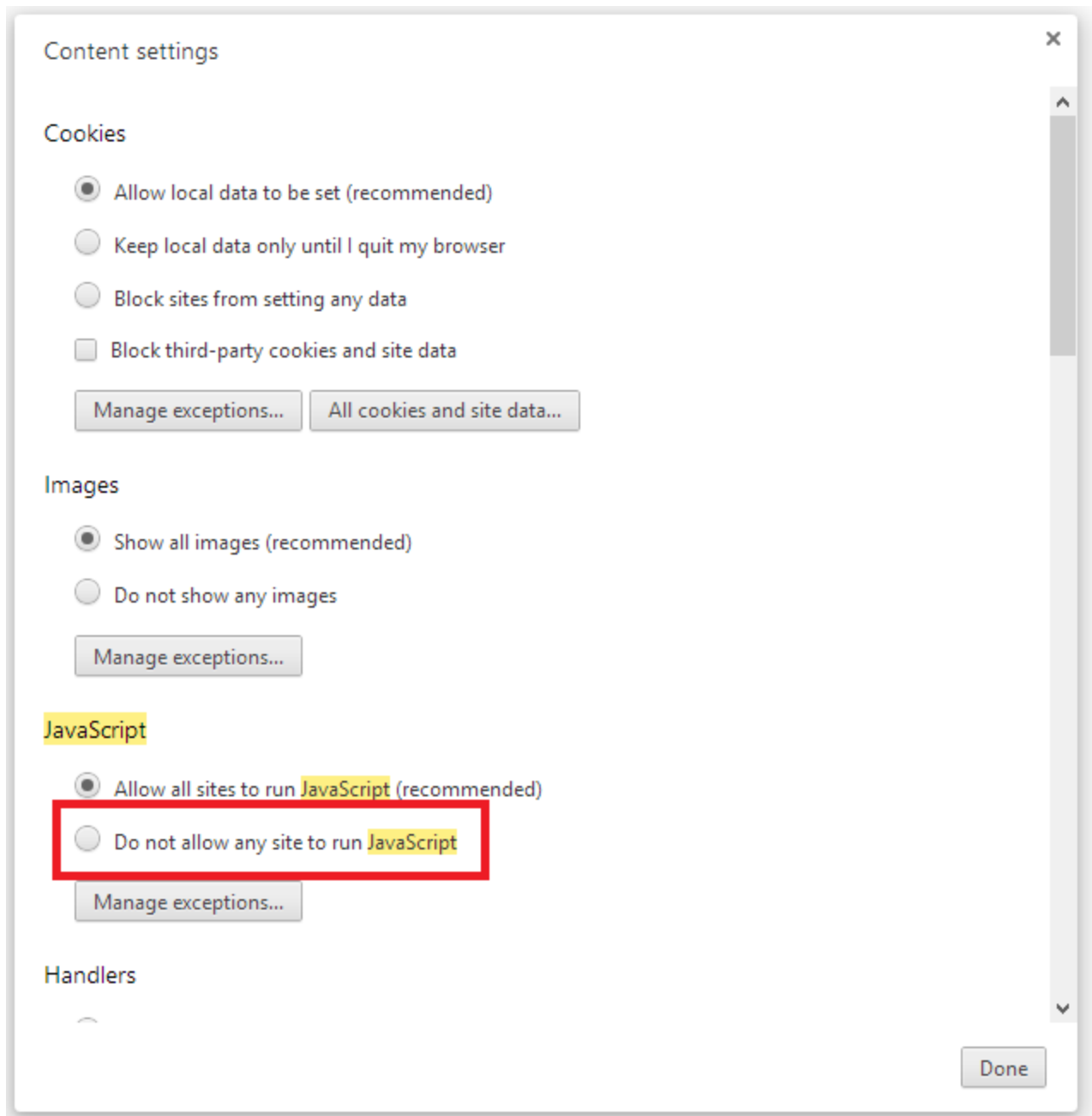
The first (HTTP GET) `Create` action method displays the initial Create form. The second ([HttpPost]) version handles the form post. The second `Create` method (The [HttpPost] version) calls `ModelState.IsValid` to check whether the movie has any validation errors. Calling this method evaluates any validation 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 is not 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 will not 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.

```

74      // POST: Movies/Create
75      // To protect from overposting attacks, please enable t
76      // more details see http://go.microsoft.com/fwlink/?Lin
77      [HttpPost]
78      [ValidateAntiForgeryToken]
79      0 references
80      public async Task<IActionResult> Create([Bind("ID,Title
81      {
82          if (ModelState.IsValid)
83          {
84              _context.Add(movie);
85              await _context.SaveChangesAsync();
86              return RedirectToAction("Index");
87          }
88          return View(movie);
89      }

```

Below is portion of the *Create.cshtml* view template that you scaffolded earlier in the tutorial. It's used by the action methods shown above both to display the initial form and to redisplay it in the event of an error.

HTMLCopy

```

<form asp-action="Create">
    <div class="form-horizontal">
        <h4>Movie</h4>
        <hr />

        <div asp-validation-summary="ModelOnly" class="text-danger"></div>
        <div class="form-group">
            <label asp-for="Title" class="col-md-2 control-label"></label>
            <div class="col-md-10">
                <input asp-for="Title" class="form-control" />
                <span asp-validation-for="Title" class="text-danger"></span>
            </div>
        </div>

        @*Markup removed for brevity.*@
    </div>
</form>

```

The [Input Tag Helper](#) uses the [DataAnnotations](#) attributes and produces HTML attributes needed for jQuery Validation on the client side. The [Validation Tag Helper](#) displays validation errors. See [Validation](#) for more information.<sup>2</sup>

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 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 supply 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 is more specific than the database intrinsic type, they are 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 emits HTML 5 `data-` (pronounced data dash) attributes that HTML 5 browsers can understand. The `DataType` attributes do not provide any validation.

`DataType.Date` does not specify the format of the date that is 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 do not 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 does not 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:

C#Copy

```
[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.<sup>4</sup>

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)]
    public decimal Price { get; set; }

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

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

## Additional resources

- [Working with Forms](#)
- [Globalization and localization](#)
- [Introduction to Tag Helpers](#)
- [Authoring Tag Helpers](#)



# Examining the Details and Delete methods

By [Rick Anderson](#)

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
        .SingleOrDefaultAsync(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 like `http://localhost:xxxx/Movies/Details/12345` (or some other value that doesn't represent an actual movie). If you did not check for a null movie, the app would throw an exception.<sup>2</sup>

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
        .SingleOrDefaultAsync(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.SingleOrDefaultAsync(m => m.ID == id);
    _context.Movie.Remove(movie);
    await _context.SaveChangesAsync();
    return RedirectToAction("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] Delete` method:

C#Copy

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

Thanks for completing this introduction to ASP.NET Core MVC. We appreciate any comments you leave. [Getting started with MVC and EF Core](#) is an excellent follow up to this tutorial.