**Building your first ASP.NET Core MVC app with Visual Studio 2015**

**Lab Exercise**

Table of Contents

[1 Getting started with ASP.NET Core MVC and Visual Studio 3](#_Toc464495948)

[1.1 Prerequisites 3](#_Toc464495949)

[1.2 Create a web app 3](#_Toc464495950)

[2 Adding a controller 8](#_Toc464495951)

[3 Adding a view 14](#_Toc464495952)

[3.1 Changing views and layout pages 16](#_Toc464495953)

[3.1.1 Change the title and menu link in the layout file 16](#_Toc464495954)

[3.2 Passing Data from the Controller to the View 18](#_Toc464495955)

[4 Adding a model 21](#_Toc464495956)

[4.1 Create a new project with individual user accounts 21](#_Toc464495957)

[4.1.1 Adding data model classes 24](#_Toc464495958)

[4.1.2 Scaffolding a controller 24](#_Toc464495959)

[4.1.3 Update the database 28](#_Toc464495960)

[4.2 To Stop IIS Express: 28](#_Toc464495961)

[4.3 dotnet ef commands 30](#_Toc464495962)

[4.3.1 Test the app 30](#_Toc464495963)

[4.3.2 Examining the Generated Code 32](#_Toc464495964)

[4.4 Strongly typed models and the @model keyword 33](#_Toc464495965)

[5 Working with SQL Server LocalDB 38](#_Toc464495966)

[5.1 SQL Server Express LocalDB 38](#_Toc464495967)

[5.2 Seed the database 42](#_Toc464495968)

[5.3 Test the app 44](#_Toc464495969)

[6 Controller methods and views 47](#_Toc464495970)

[6.1 Processing the POST Request 57](#_Toc464495971)

[7 Adding Search 60](#_Toc464495972)

[7.1 Adding Search by Genre 69](#_Toc464495973)

[7.2 Adding search by genre to the Index view 70](#_Toc464495974)

[8 Adding a New Field 72](#_Toc464495975)

[8.1 Adding a Rating Property to the Movie Model 72](#_Toc464495976)

[9 Adding Validation 76](#_Toc464495977)

[9.1 Keeping things DRY 76](#_Toc464495978)

[9.2 Adding validation rules to the movie model 76](#_Toc464495979)

[9.3 Validation Error UI in MVC 77](#_Toc464495980)

[9.4 How Validation Occurs in the Create View and Create Action Method 79](#_Toc464495981)

[9.5 Using DataType Attributes 85](#_Toc464495982)

[10 Examining the Details and Delete methods 87](#_Toc464495983)

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

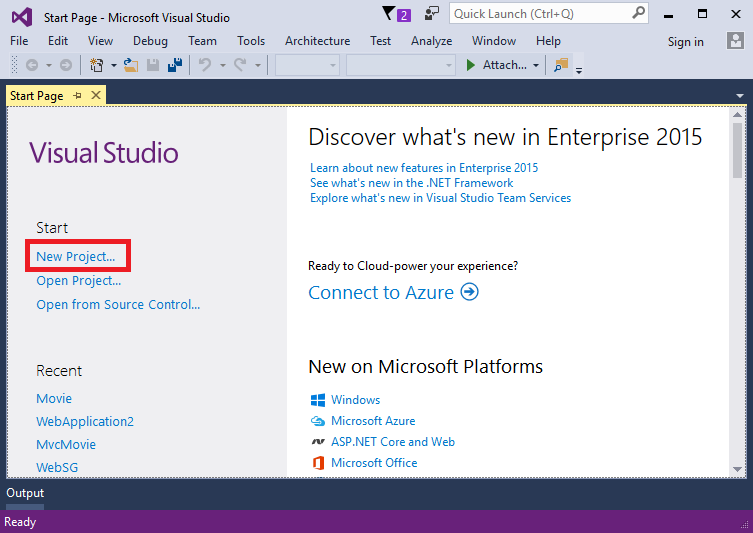
.

## Prerequisites

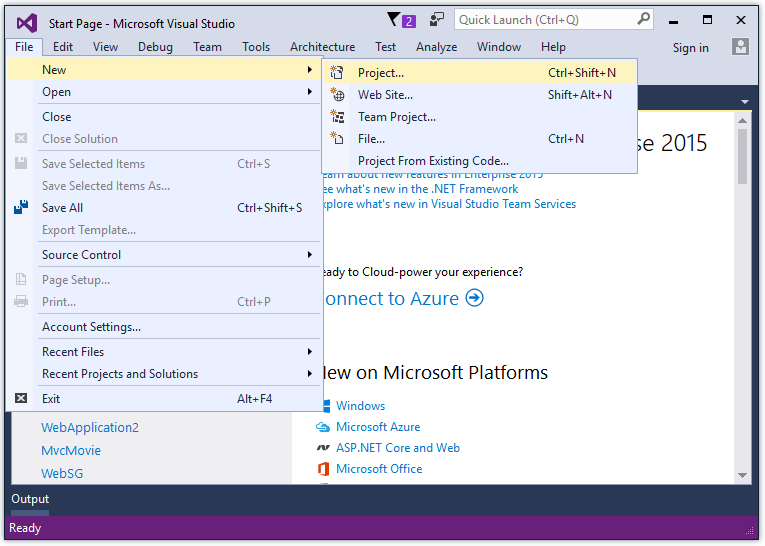
* Install Visual Studio Community 2015. Select the Community download and the default installation. Skip this step if you have Visual Studio 2015 installed.
  + Visual Studio 2015 Home page installer
* Install .NET Core + Visual Studio tooling

## Create a web app

From the Visual Studio **Start** page, tap **New Project**.

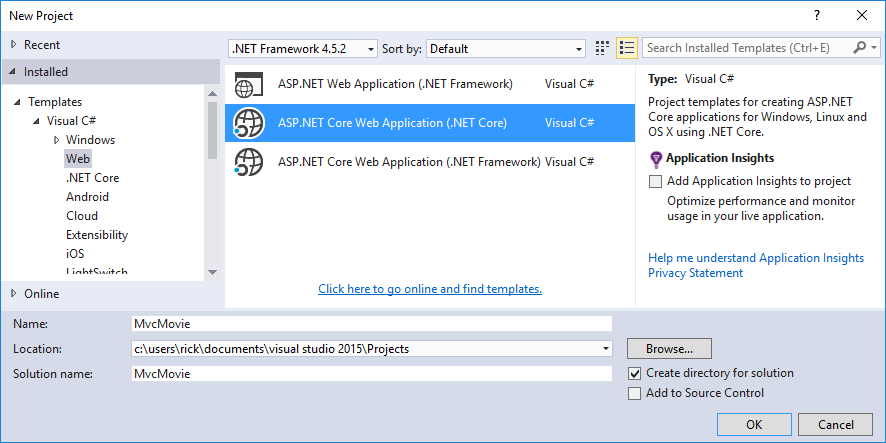


Alternatively, you can use the menus to create a new project. Tap **File > New > Project**.



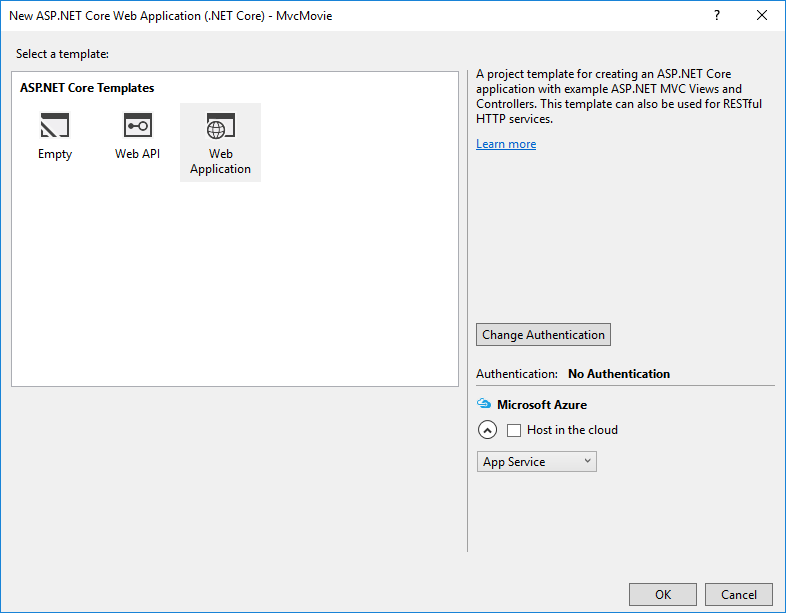
Complete the **New Project** dialog:

* In the left pane, tap **Web**
* 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**



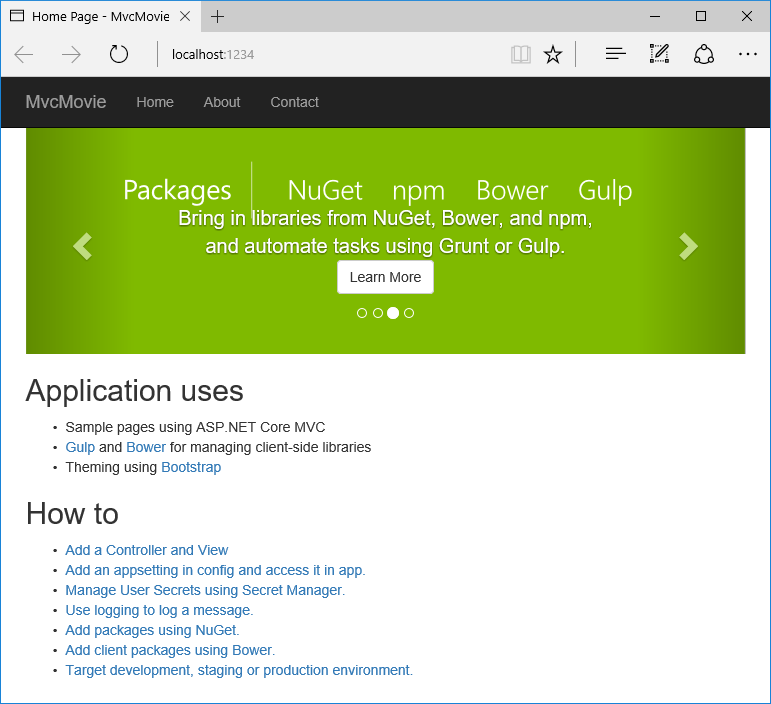
Complete the **New ASP.NET Core Web Application - MvcMovie** dialog:

* Tap **Web Application**
* Clear **Host in the cloud**
* Tap **OK**.

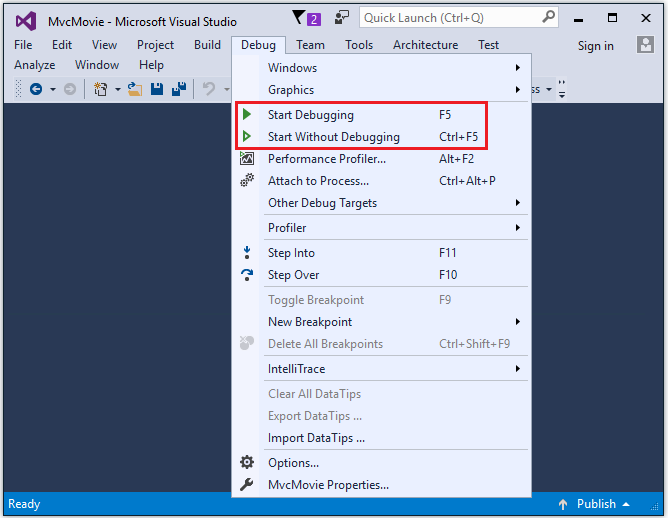


Visual Studio used a default template for the MVC project you just created, so you have a working app right now by entering a project name and selecting a few options. This is a simple “Hello World!” project, and it’s a good place to start,

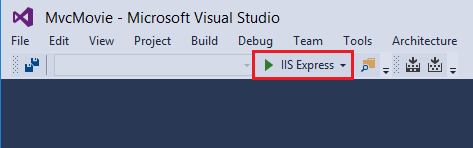
Tap **F5** to run the app in debug mode or **Ctl-F5** in non-debug mode.



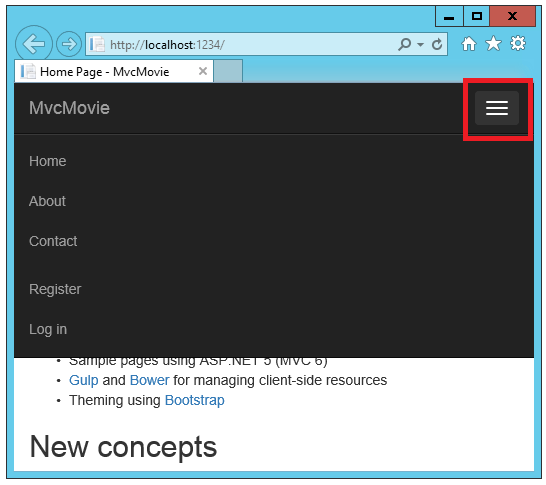
* Visual Studio starts IIS Express and runs your app. Notice that the address bar showslocalhost:port# and not something like example.com. That’s because localhost always points to your own local computer, which in this case is running the app you just created. When Visual Studio creates a web project, a random port is used for the web server. In the image above, the port number is 1234. When you run the app, you’ll see a different port number.
* Launching the app with **Ctrl+F5** (non-debug mode) allows you to make code changes, save the file, refresh the browser, and see the code changes. Many developers prefer to use non-debug mode to quickly launch the app and view changes.
* You can launch the app in debug or non-debug mode from the **Debug** menu item:



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



The default template gives you working **Home, Contact, About, Register** and **Log in** 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.



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

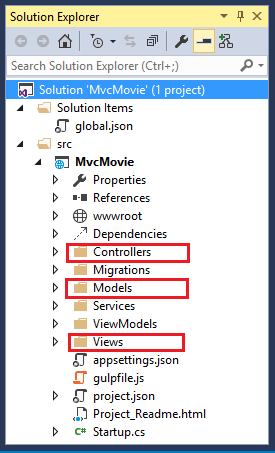
# Adding a controller

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

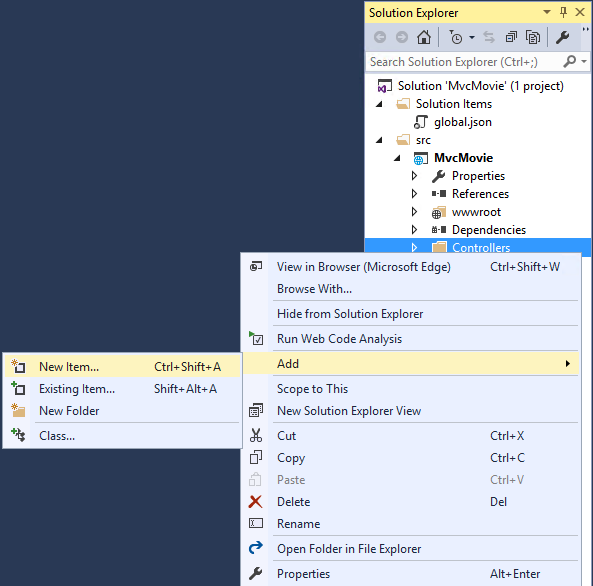
* **M**odels: Classes that represent the data of the app and that use validation logic to enforce business rules for that data. Typically, model objects retrieve and store model state in a database. In this tutorial, a Movie model retrieves movie data from a database, provides it to the view or updates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

The MVC pattern helps you create apps that separate the different aspects of the app (input logic, business logic, and UI logic), while providing a loose coupling between these elements. The pattern specifies where each kind of logic should be located in the app. The UI logic belongs in the view. Input logic belongs in the controller. Business logic belongs in the model. This separation helps you manage complexity when you build an app, because it enables you to work on one aspect of the implementation at a time without impacting the code of another. For example, you can work on the view code without depending on the business logic code.

We’ll be covering all these concepts in this tutorial series and show you how to use them to build a simple movie app. The following image shows the *Models*, *Views* and *Controllers* folders in the MVC project.

[](https://docs.asp.net/en/latest/_images/mvc1.png)

* In **Solution Explorer**, right-click **Controllers > Add > New Item... > MVC Controller Class**



* In the **Add New Item** dialog, enter **HelloWorldController**.

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

**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...";

}

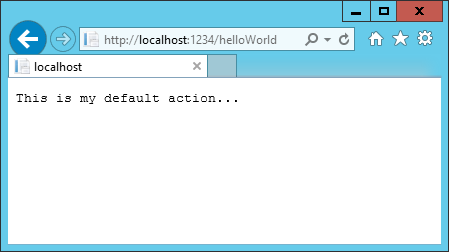
}

}

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.

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 we’ll use the scaffolding engine to generate HTTP POST methods.

Run the app in non-debug mode (press Ctrl+F5) and append “HelloWorld” to the path in the address bar. (In the image below, http://localhost:1234/HelloWorld is used, but you’ll have to replace *1234* with the port number of your app.) The Index method returns a string. You told the system to return some HTML, and it did!



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.

app.UseMvc(routes =>

{

routes.MapRoute(

name: "default",

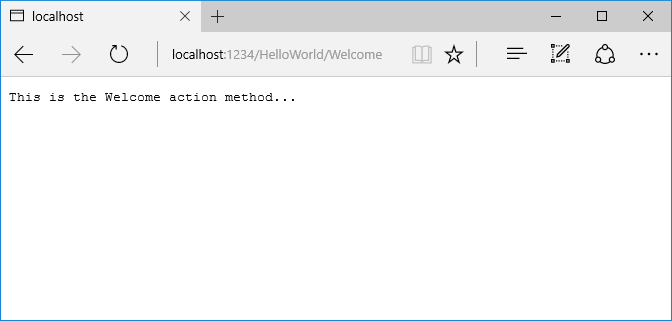
template: "{controller=Home}/{action=Index}/{id?}");

});

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 theHelloWorldController class to run. Notice that we only had to browse to localhost:xxxx/HelloWorldand 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. We’ll see route data later on in this tutorial.

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 andWelcome is the action method. We haven’t used the [Parameters] part of the URL yet.



Let’s modify the example slightly so that you can pass some parameter information from the URL to the controller (for example, /HelloWorld/Welcome?name=Scott&numtimes=4). Change the Welcomemethod to include two parameters as shown below. Note that the code uses the C# optional-parameter feature to indicate that the numTimes parameter defaults to 1 if no value is passed for that parameter.

public string Welcome(string name, int numTimes = 1)

{

return HtmlEncoder.Default.Encode($"Hello {name}, numTimes: {numTimes}");

}

**Note**

The code above uses HtmlEncoder.Default.Encode to protect the app from malicious input (namely JavaScript). It also uses Interpolated Strings.

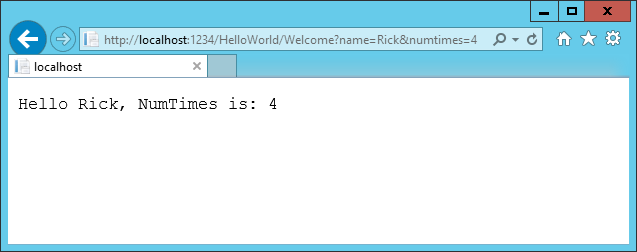
**Note**

In Visual Studio 2015, when you are running in IIS Express without debugging (Ctl+F5), you don’t need to build the app after changing the code. Just save the file, refresh your browser and you can see the changes.

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.



In the sample above, the URL segment (Parameters) is not used, the name and numTimesparameters 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:

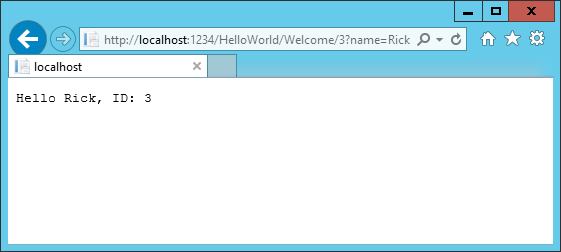
public string Welcome(string name, int ID = 1)

{

return HtmlEncoder.Default.Encode($"Hello {name}, ID: {ID}");

}

Run the app and enter the following URL: http://localhost:xxx/HelloWorld/Welcome/3?name=Rick



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.

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 we’ll typically use a separate Razor view template file to help generate the HTML response. We’ll do that in the next tutorial.

# Adding a view

In this section you’re going to modify the HelloWorldController class to use Razor view template files to cleanly encapsulate the process of generating HTML responses to a client.

You’ll create a view template file using Razor. Razor-based view templates have a .cshtml file extension, and provide an elegant way to create HTML output using C#. Razor seamlessly blends C# and HTML, minimizing the number of characters and keystrokes required when writing a view template, and enables a fast, fluid coding workflow.

Currently the Index method returns a string with a message that is hard-coded in the controller class. Change the Index method to return a View object, as shown in the following code:

**public** IActionResult **Index**()

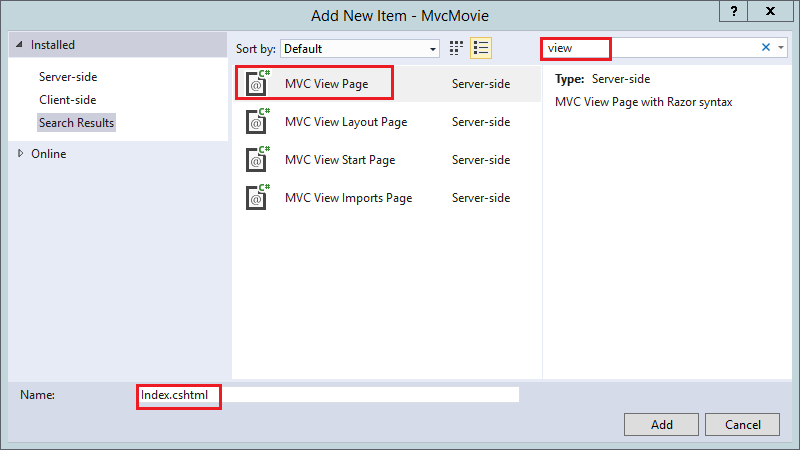
{

**return** **View**();

}

The Index method above uses a view template to generate an HTML response to the browser. Controller methods (also known as action methods) such as the Index method above, generally return an IActionResult (or a class derived from ActionResult), not primitive types like string.

* 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, keep the default Index.cshtml
  + Tap **Add**



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

@{

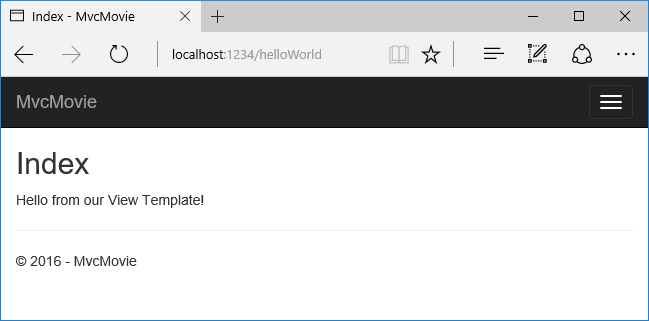
ViewData["Title"] = "Index";

}

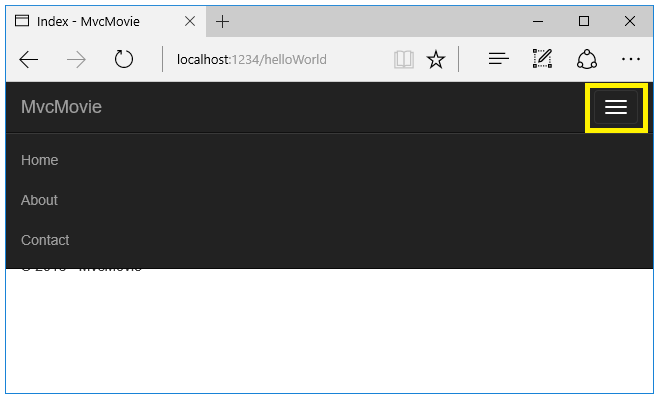
<h2>Index</h2>

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

Navigate to http://localhost:xxxx/HelloWorld. The Index method in the HelloWorldController didn’t do much work; it simply 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 to use, 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.



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 to the **Home**, **About**, and **Contact**links.



## Changing views and layout pages

Tap on 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 theViews/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.

### 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 “MVC Movie” and the controller from Home to Movies as highlighted below:

**<!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/3.3.6/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>

</head>

<body>

<div 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>

</div>

<div class**=**"container body-content">

@RenderBody()

<hr />

<footer>

<p>&copy; 2016 - MvcMovie</p>

</footer>

</div>

**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 each page displays the **Mvc Movie** link. We were able to make the change once in the layout template and have all pages on the site reflect the new link text and new title.

Examine the Views/\_ViewStart.cshtml file:

@{

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.

Now, let’s change the title of the Index view.

Open Views/HelloWorld/Index.cshtml. There are two places to make a change:

* 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.

@{

ViewData["Title"] = "Movie List";

}

<h2>My Movie List</h2>

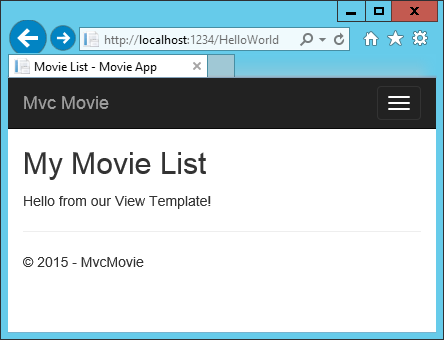
<p>Hello from our View Template!</p>

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

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

Save your change and refresh the page. 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 theViews/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. Shortly, we’ll walk through how to create a database and retrieve model data from it.

## Passing Data from the Controller to the View

Before we go to a database and talk about models, though, let’s first talk about passing information from the controller to a 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, retrieves data from a database, and ultimately decides what type of response to send back to the browser. View templates can then be used from a controller to generate and format an HTML response to the browser.

Controllers are responsible for providing whatever data or objects are required in order for a view template to render a response to the browser. A best practice: A view template should never perform business logic or interact with a database directly. Instead, 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 more 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 ViewDatadictionary that the view template can then access.

Return to the HelloWorldController.cs file and change the Welcome method to add a Message andNumTimes 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:

**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. Next, you need a Welcome view template.

* 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, enter Welcome.cshtml
  + Tap **Add**

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:

@{

ViewData["Title"] = "About";

}

<h2>Welcome</h2>

<ul>

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

{

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

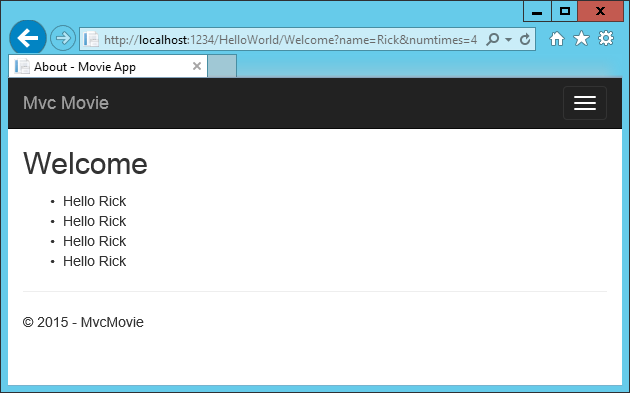
}

</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.



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.

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

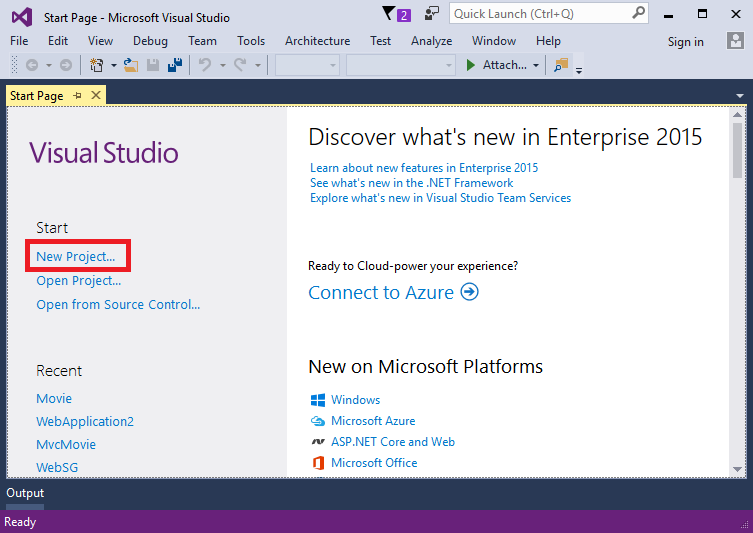
In this section you’ll add some classes for managing movies in a database. These classes will be the “**M**odel” part of the **M**VC app.

You’ll use a .NET Framework data-access technology known as the Entity Framework Core to define and work with these data model classes. Entity Framework Core (often referred to as **EF**Core) features a development paradigm called Code First. You write the code first, and the database tables are created from this code. Code First allows you to create data model objects by writing simple classes. (These are also known as POCO classes, from “plain-old CLR objects.”) The database is created from your classes. If you are required to create the database first, you can still follow this tutorial to learn about MVC and EF app development.

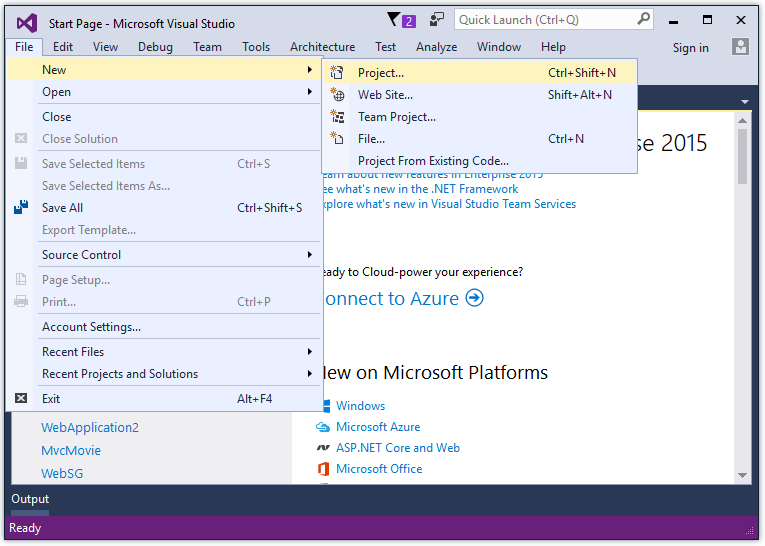
## Create a new project with individual user accounts

In the current version of the ASP.NET Core MVC tools for Visual Studio, scaffolding a model is only supported when you create a new project with individual user accounts. We hope to have this fixed in the next tooling update. Until that’s fixed, you’ll need to create a new project with the same name. Because the project has the same name, you’ll need to create it in another directory.

From the Visual Studio **Start** page, tap **New Project**.

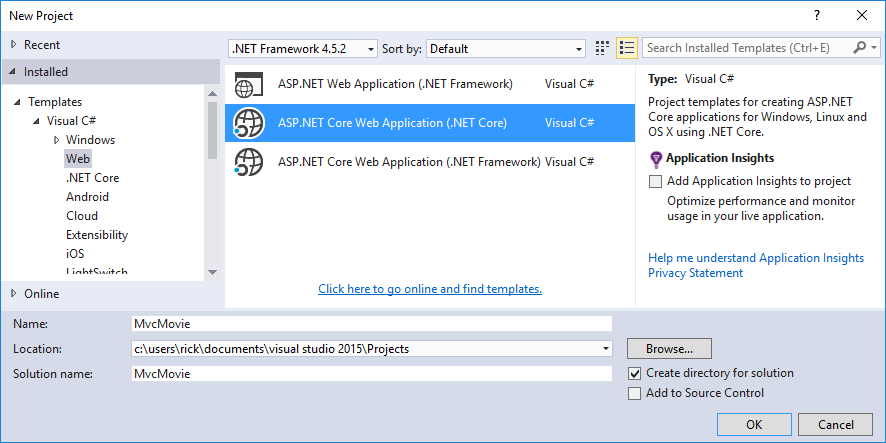


Alternatively, you can use the menus to create a new project. Tap **File > New > Project**.



Complete the **New Project** dialog:

* In the left pane, tap **Web**
* In the center pane, tap **ASP.NET Core Web Application (.NET Core)**
* Change the location to a different directory from the previous project you created or you’ll get an error
* Name the project “MvcMovie” (It’s important to name the project “MvcMovie” so when you copy code, the namespace will match.)
* Tap **OK**

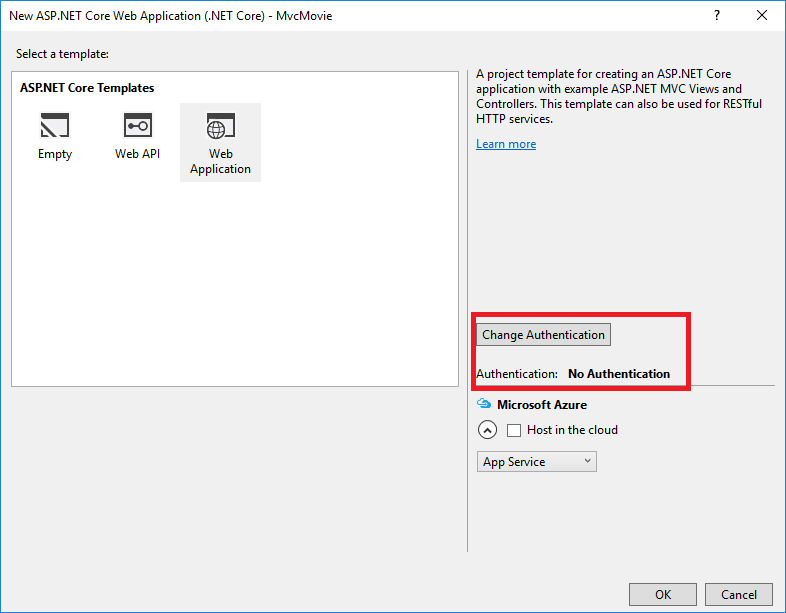


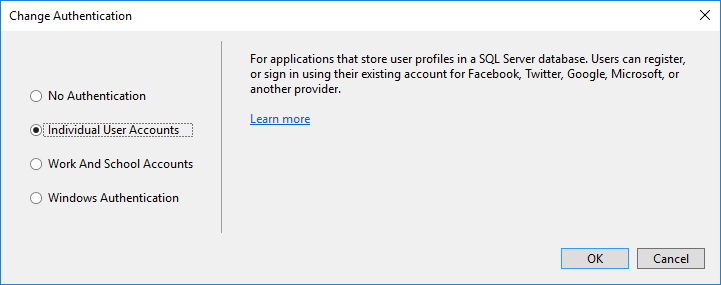
**Warning**

You must have the **Authentication** set to **Individual User Accounts** in this release for the scaffolding engine to work.

In the **New ASP.NET Core Web Application - MvcMovie** dialog:

* tap **Web Application**
* tap the **Change Authentication** button and change the authentication to **Individual User Accounts** and tap **OK**





Follow the instructions in Change the title and menu link in the layout file so you can tap the**MvcMovie** link to invoke the Movie controller. We’ll scaffold the movies controller in this tutorial.

### Adding data model classes

In Solution Explorer, right click the Models folder > **Add** > **Class**. Name the class **Movie** and add the following properties:

**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**; }

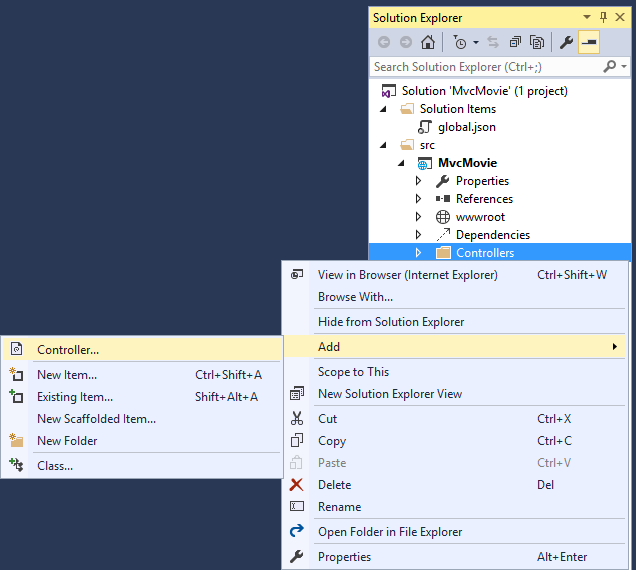
}

}

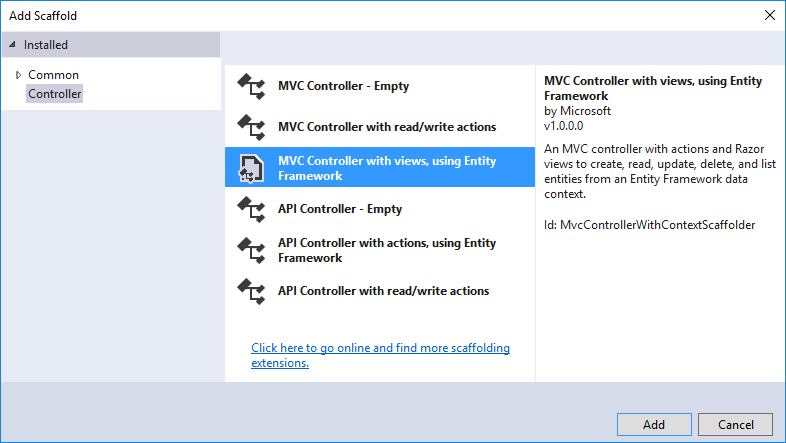
In addition to the properties you’d expect to model a movie, the ID field is required by the DB for the primary key. Build the project. If you don’t build the app, you’ll get an error in the next section. We’ve finally added a **M**odel to our **M**VC app.

### Scaffolding a controller

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

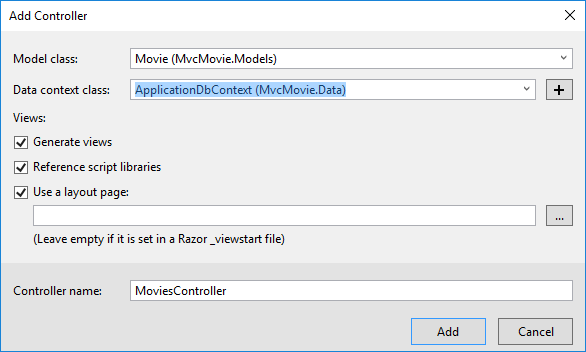


In the **Add Scaffold** dialog, tap **MVC Controller with views, using Entity Framework > Add**.



Complete the **Add Controller** dialog

* **Model class:** Movie(MvcMovie.Models)
* **Data context class:** ApplicationDbContext(MvcMovie.Models)
* **Views:**: Keep the default of each option checked
* **Controller name:** Keep the default MoviesController
* Tap **Add**

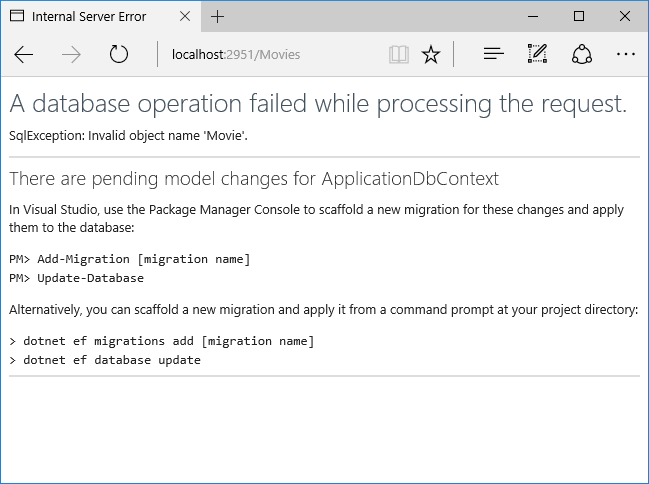
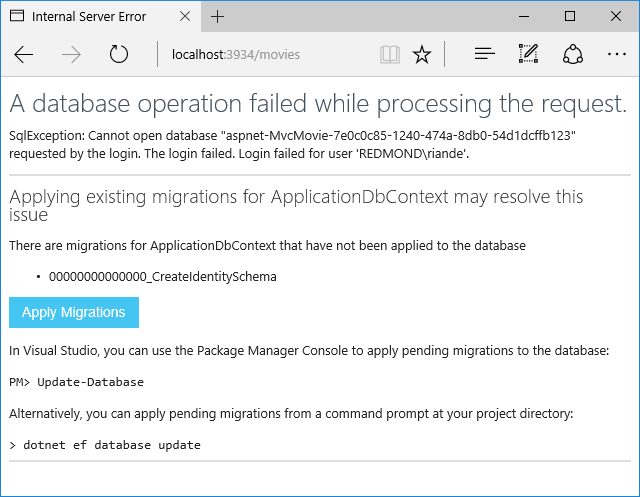


The Visual Studio scaffolding engine creates the following:

* A movies controller (Controllers/MoviesController.cs)
* Create, Delete, Details, Edit and Index Razor view files (Views/Movies)

Visual Studio automatically created the CRUD (create, read, update, and delete) action methods and views for you (the automatic creation of CRUD action methods and views is known asscaffolding). You’ll soon have a fully functional web application that lets you create, list, edit, and delete movie entries.

If you run the app and click on the **Mvc Movie** link, you’ll get the following errors:



We’ll follow those instructions to get the database ready for our Movie app.

### Update the database

**Warning**

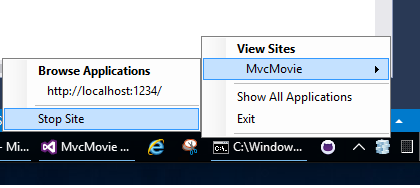
You must stop IIS Express before you update the database.

## To Stop IIS Express:

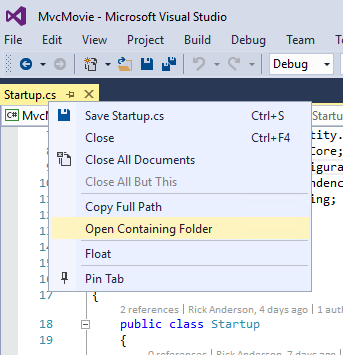
* Right click the IIS Express system tray icon in the notification area

[](https://docs.asp.net/en/latest/_images/iisExIcon.png)

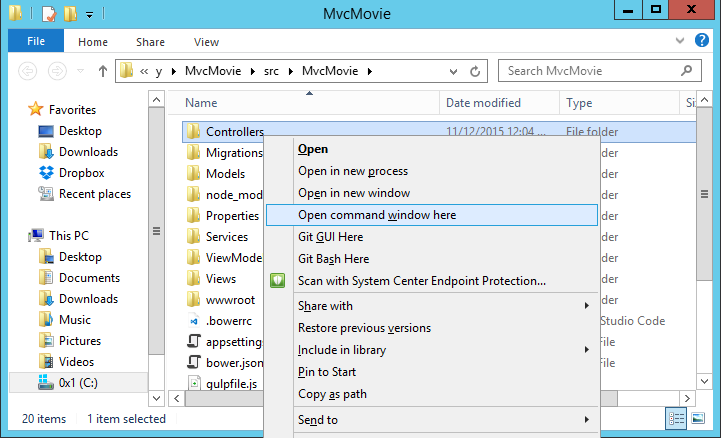
* Tap **Exit** or **Stop Site**



* Alternatively, you can exit and restart Visual Studio
* Open a command prompt in the project directory (MvcMovie/src/MvcMovie). Follow these instructions for a quick way to open a folder in the project directory.
  + Open a file in the root of the project (for this example, use Startup.cs.)
  + Right click on Startup.cs **> Open Containing Folder**.



* Shift + right click a folder > **Open command window here**



* Run cd .. to move back up to the project directory
* Run the following commands in the command prompt:

dotnet ef migrations add Initial

dotnet ef database update

**Note**

If IIS-Express is running, you’ll get the error CS2012: Cannot open ‘MvcMovie/bin/Debug/netcoreapp1.0/MvcMovie.dll’ for writing – ‘The process cannot access the file ‘MvcMovie/bin/Debug/netcoreapp1.0/MvcMovie.dll’ because it is being used by another process.’

## dotnet ef commands

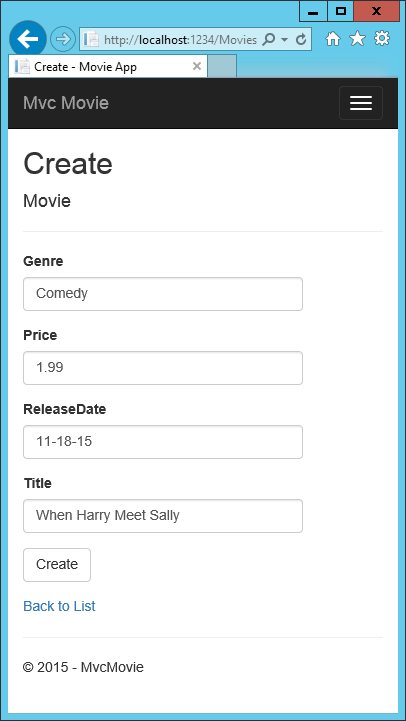
* dotnet (.NET Core) is a cross-platform implementation of .NET. You can read about it here
* dotnet ef migrations add Initial Runs the Entity Framework .NET Core CLI migrations command and creates the initial migration. The parameter “Initial” is arbitrary, but customary for the first (initial) database migration. This operation creates the Data/Migrations/<date-time>\_Initial.csfile containing the migration commands to add (or drop) the Movie table to the database
* dotnet ef database update Updates the database with the migration we just created

### Test the app

**Note**

If your browser is unable to connect to the movie app you might need to wait for IIS Express to load the app. It can sometimes take up to 30 seconds to build the app and have it ready to respond to requests.

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



**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.

**Note**

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

**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**; }

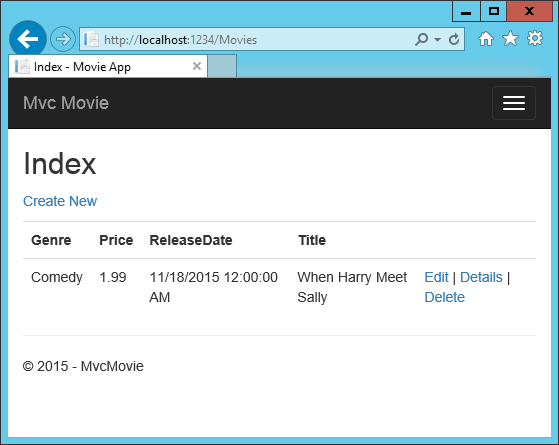
**public** **string** Genre { **get**; **set**; }

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

}

}

Tapping **Create** causes the form to be posted to the server, where the movie information is saved in a database. You are then redirected to the /Movies URL, where you can see the newly created movie in the listing.



Create a couple more movie entries. Try the **Edit**, **Details**, and **Delete** links, which are all functional.

### Examining the Generated Code

Open the Controllers/MoviesController.cs file and examine the generated Index method. A portion of the movie controller with the Index method is shown below:

**public** **class** **MoviesController** : Controller

{

**private** **readonly** ApplicationDbContext \_context;

**public** **MoviesController**(ApplicationDbContext context)

{

\_context = context;

}

*// GET: Movies*

**public** **async** Task<IActionResult> Index()

{

**return** **View**(**await** \_context.Movie.ToListAsync());

}

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

A request to the Movies controller returns all the entries in the Movies table and then passes the data to the Index view.

## 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 theViewData 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 objects to a view. This strongly typed approach enables better compile-time checking of your code and richer IntelliSense in Visual Studio (VS). The scaffolding mechanism in VS used this approach (that is, passing a strongly typed model) with the MoviesController class and views when it created the methods and views.

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

*// 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 examplehttp://localhost:1234/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 could also pass in the id with a query string as follows:

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

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

**return** **View**(movie);

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

@model MvcMovie.Models.Movie

@{

ViewData["Title"] = "Details";

}

<h2>Details</h2>

<div>

<h4>Movie</h4>

<hr />

<dl class**=**"dl-horizontal">

<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>

<dt>

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

</dt>

<dd>

@Html.DisplayFor(model => model.ReleaseDate)

</dd>

<dt>

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

</dt>

<dd>

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

</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:

@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:

*// GET: Movies*

**public** **async** Task<IActionResult> Index()

{

**return** **View**(**await** \_context.Movie.ToListAsync());

}

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

@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:

@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.Genre)

</th>

<th>

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

</th>

<th>

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

</th>

<th>

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

</th>

<th></th>

</tr>

</thead>

<tbody>

@foreach (var item in Model) {

<tr>

<td>

@Html.DisplayFor(modelItem => item.Genre)

</td>

<td>

@Html.DisplayFor(modelItem => item.Price)

</td>

<td>

@Html.DisplayFor(modelItem => item.ReleaseDate)

</td>

<td>

@Html.DisplayFor(modelItem => item.Title)

</td>

@\*<snippet\_1>\*@

<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>

@\*</snippet\_1>\*@

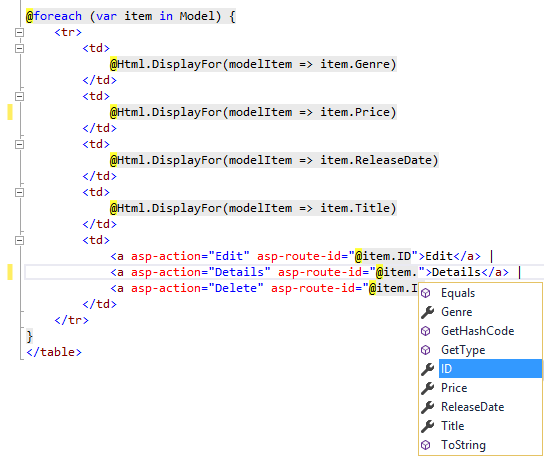
</tr>

}

</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 and full IntelliSense support in the code editor:



You now have a database and pages to display, edit, update and delete data. In the next tutorial, we’ll work with the database.

# Working with SQL Server LocalDB

The ApplicationDbContext class handles the task of connecting to the database and mapping Movieobjects to database records. The database context is registered with the Dependency Injectioncontainer in the ConfigureServices method in the Startup.cs file:

**public** **void** **ConfigureServices**(IServiceCollection services)

{

*// Add framework services.*

services.AddDbContext<ApplicationDbContext>(options =>

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

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

{

"ConnectionStrings"**:** {

"DefaultConnection"**:** "Server=(localdb)\\mssqllocaldb;Database=aspnet-MvcMovie-4ae3798a;Trusted\_Connection=True;MultipleActiveResultSets=true"

},

"Logging"**:** {

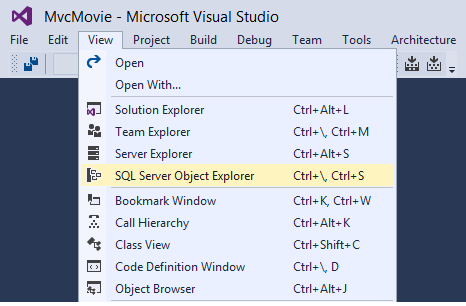
"IncludeScopes"**:** **false**,

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 .

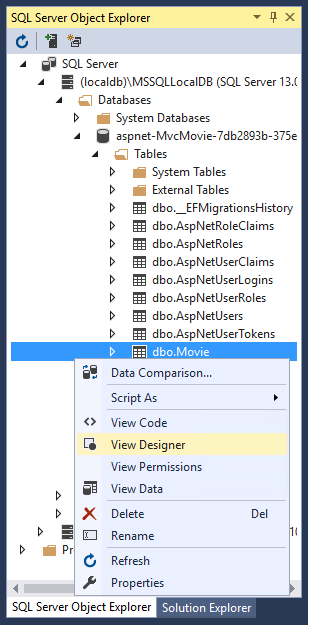
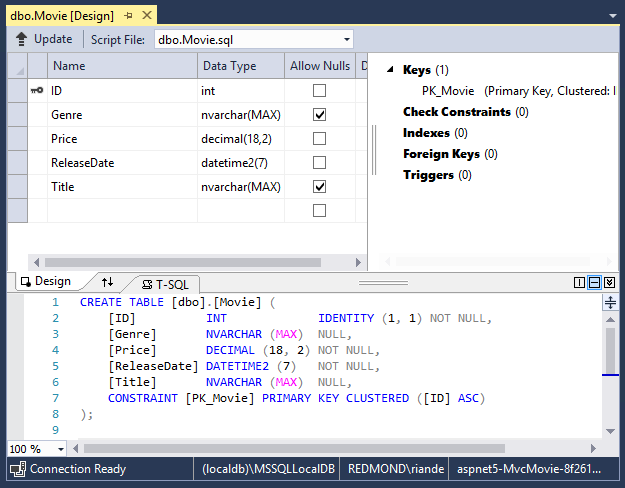
## SQL Server Express LocalDB

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).

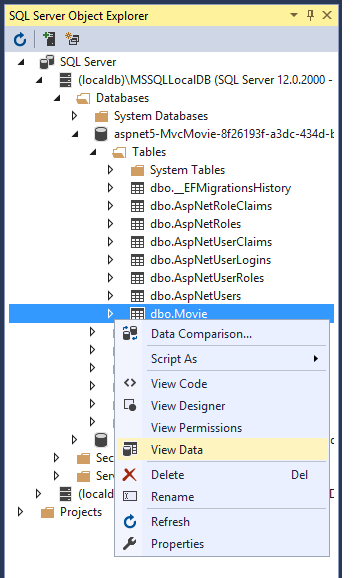
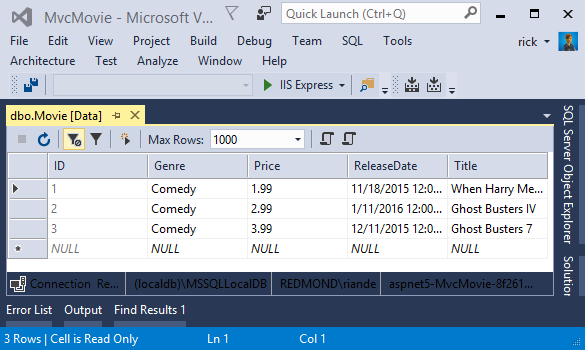


* Right click on the Movie table **> View Designer**

[](https://docs.asp.net/en/latest/_images/design.png) 

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

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

## Seed the database

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

**using** Microsoft.EntityFrameworkCore;

**using** Microsoft.Extensions.DependencyInjection;

**using** MvcMovie.Data;

**using** System;

**using** System.Linq;

**namespace** MvcMovie.Models

{

**public** **static** **class** **SeedData**

{

**public** **static** **void** **Initialize**(IServiceProvider serviceProvider)

{

**using** (**var** context = **new** ApplicationDbContext(

serviceProvider.GetRequiredService<DbContextOptions<ApplicationDbContext>>()))

{

*// 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();

}

}

}

}

Notice if there are any movies in the DB, the seed initializer returns.

**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:

app**.**UseMvc(routes **=>**

{

routes**.**MapRoute(

name: "default",

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

});

*#endregion*

SeedData**.**Initialize(app**.**ApplicationServices);

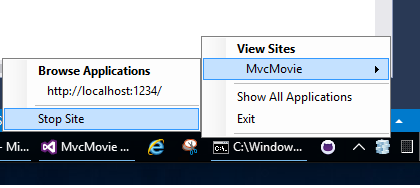
}

**//** End of Configure**.**

## 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**

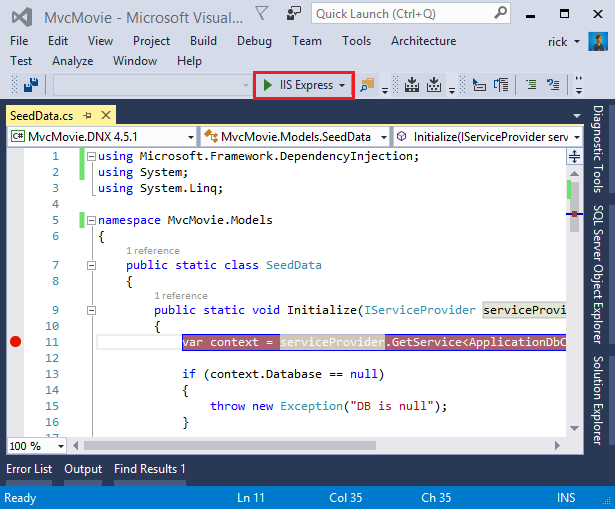
[](https://docs.asp.net/en/latest/_images/iisExIcon.png)



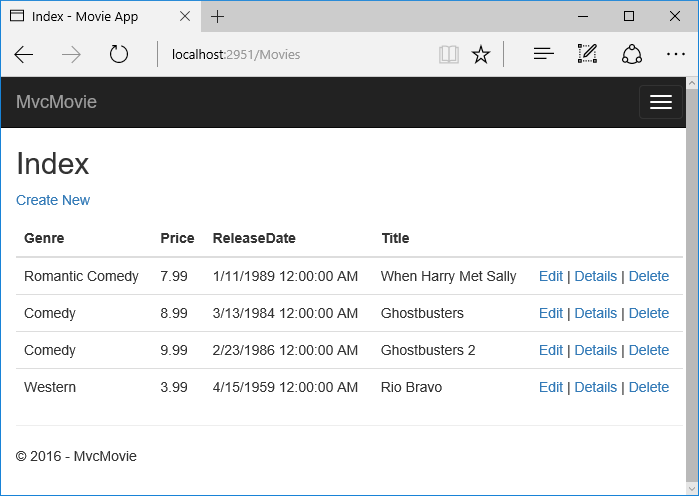
* If you were running VS in non-debug mode, press F5 to run in debug mode
* If you were running VS in debug mode, stop the debugger and press ^F5

**Note**

If the database doesn’t initialize, put a break point on the line if (context.Movie.Any()) and start debugging.

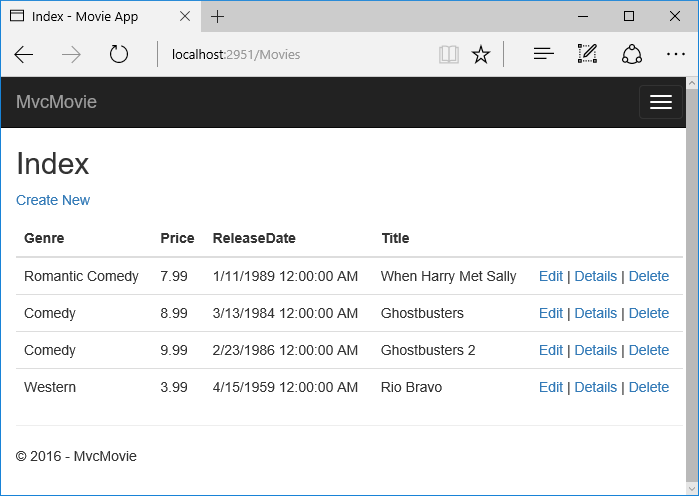


The app shows the seeded data.



# Controller methods and views

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:

**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**; }

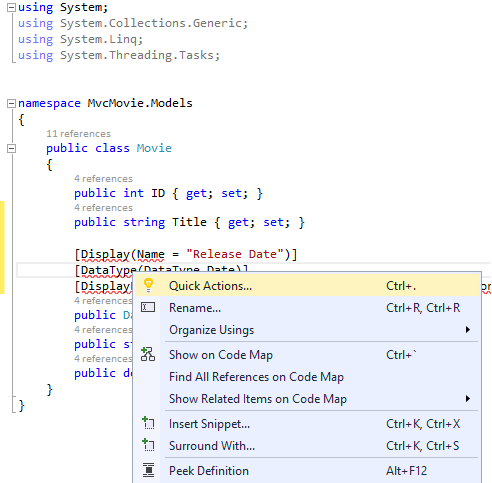
**public** **string** Genre { **get**; **set**; }

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

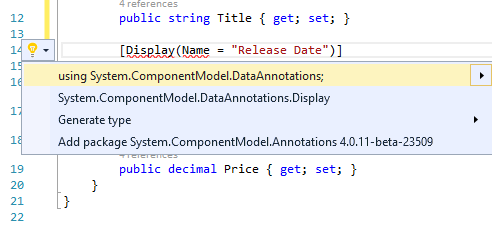
}

}

* Right click on a red squiggly line **> Quick Actions**.

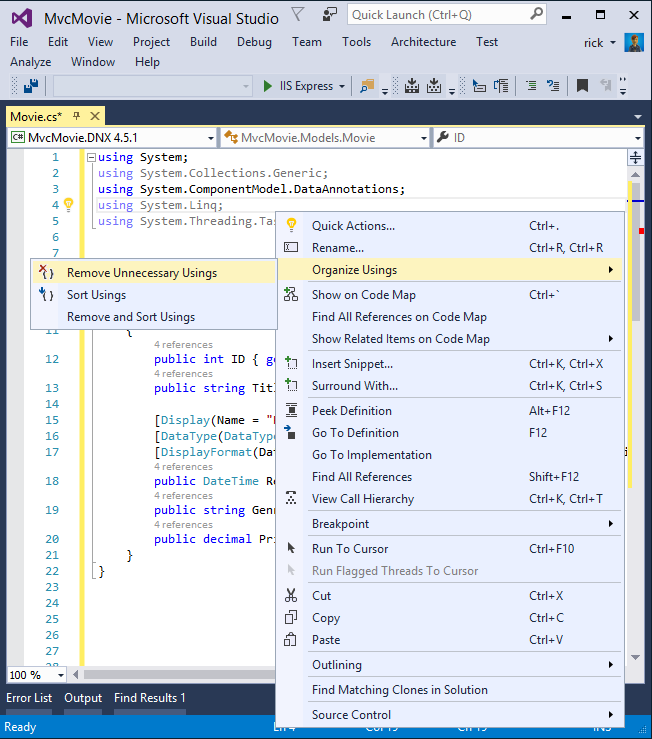


* Tap using System.ComponentModel.DataAnnotations;



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 **> Organize Usings > Remove Unnecessary Usings**.



The updated code:

**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**; }

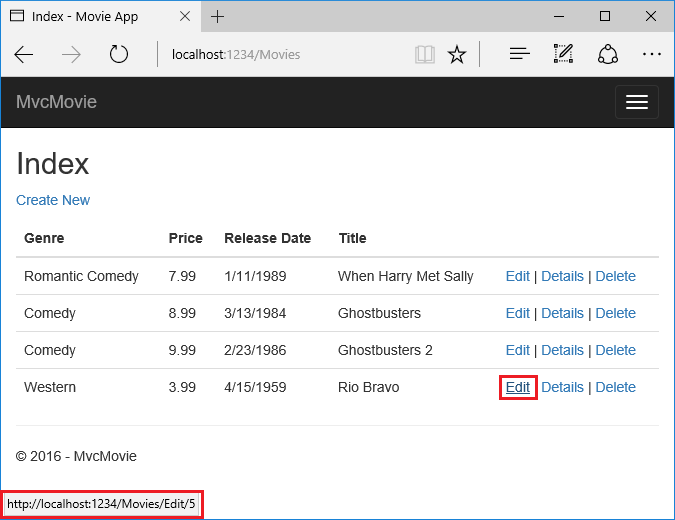
**public** **decimal** Price { **get**; **set**; }

}

}

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, in this case it’s a 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.



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

<td>

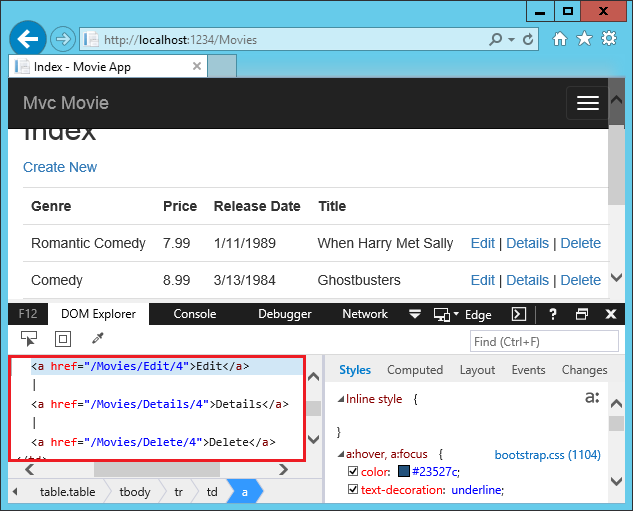
<a asp-action**=**"Edit" asp-route-id**=**"@item.ID">Edit</a> |

<a asp-action**=**"Details" asp-route-id**=**"@item.ID">Details</a> |

<a asp-action**=**"Delete" asp-route-id**=**"@item.ID">Delete</a>

</td>

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 **F12** tools to examine the generated markup. The **F12** tools are shown below.



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

app.UseMvc(routes =>

{

routes.MapRoute(

name: "default",

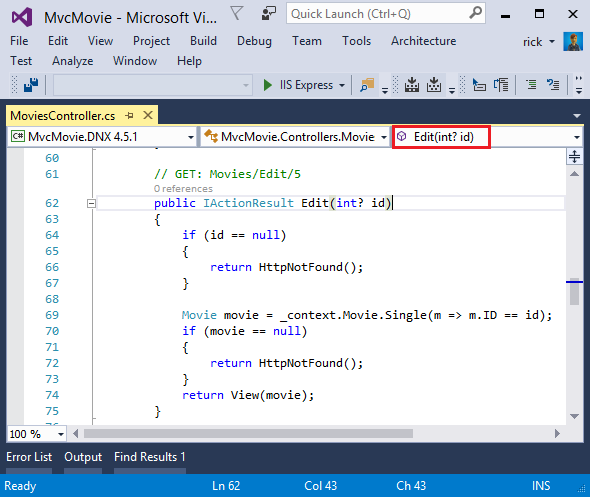
template: "{controller=Home}/{action=Index}/{id?}");

});

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 resourcesfor more information.

Open the Movies controller and examine the two Edit action methods:



*// 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);

}

[HttpPost]

[ValidateAntiForgeryToken]

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

[HttpPost]

[ValidateAntiForgeryToken]

**public** **async** Task<IActionResult> Edit(**int** id, [Bind("ID,Genre,Price,ReleaseDate,Title")] 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 **HttpPostAttribute** attribute specifies that this Edit method can be invoked *only* for POSTrequests. You could apply the [HttpGet] attribute to the first edit method, but that’s not necessary because [HttpGet] is the default.

The **ValidateAntiForgeryTokenAttribute** 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.

<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.

*// 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:

@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**=**"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">

<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**=**"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">

<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 — this 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.

<form action**=**"/Movies/Edit/7" method**=**"post">

<div class**=**"form-horizontal">

<h4>Movie</h4>

<hr />

<div class**=**"text-danger" />

<input type**=**"hidden" data-val**=**"true" data-val-required**=**"The ID field is required." id**=**"ID" name**=**"ID" value**=**"7" />

<div class**=**"form-group">

<label class**=**"control-label col-md-2" for**=**"Genre" />

<div class**=**"col-md-10">

<input class**=**"form-control" type**=**"text" id**=**"Genre" name**=**"Genre" value**=**"Western" />

<span class**=**"text-danger field-validation-valid" data-valmsg-for**=**"Genre" data-valmsg-replace**=**"true"></span>

</div>

</div>

<div class**=**"form-group">

<label class**=**"control-label col-md-2" for**=**"Price" />

<div class**=**"col-md-10">

<input class**=**"form-control" type**=**"text" data-val**=**"true" data-val-number**=**"The field Price must be a number." data-val-required**=**"The Price field is required." id**=**"Price" name**=**"Price" value**=**"3.99" />

<span class**=**"text-danger field-validation-valid" data-valmsg-for**=**"Price" data-valmsg-replace**=**"true"></span>

</div>

</div>

*<!-- Markup removed for brevity -->*

<div class**=**"form-group">

<div class**=**"col-md-offset-2 col-md-10">

<input type**=**"submit" value**=**"Save" class**=**"btn btn-default" />

</div>

</div>

</div>

<input name**=**"\_\_RequestVerificationToken" type**=**"hidden" value**=**"CfDJ8Inyxgp63fRFqUePGvuI5jGZsloJu1L7X9le1gy7NCIlSduCRx9jDQClrV9pOTTmqUyXnJBXhmrjcUVDJyDUMm7-MF\_9rK8aAZdRdlOri7FmKVkRe\_2v5LIHGKFcTjPrWPYnc9AdSbomkiOSaTEg7RU" />

</form>

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

## Processing the POST Request

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

[HttpPost]

[ValidateAntiForgeryToken]

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

}

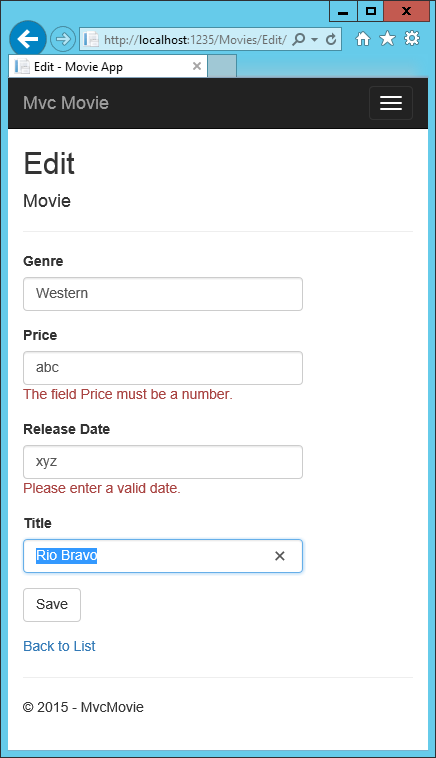
**return** **View**(movie);

}

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 validation in more detail. The Validation Tag Helper in the*Views/Book/Edit.cshtml* view template takes care of displaying appropriate error messages.



All the HttpGet methods in the movie controller follow a similar pattern. They get a movie object (or list of objects, in the case of Index), and pass the object (model) to the view. The Createmethod 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, as in ASP.NET MVC Tip #46 – Don’t use Delete Links because they create Security Holes. Modifying data in a 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.

# Adding Search

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

Update the Index action method to enable search:

**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:

**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, using the following code:

**if** (!String.IsNullOrEmpty(searchString))

{

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

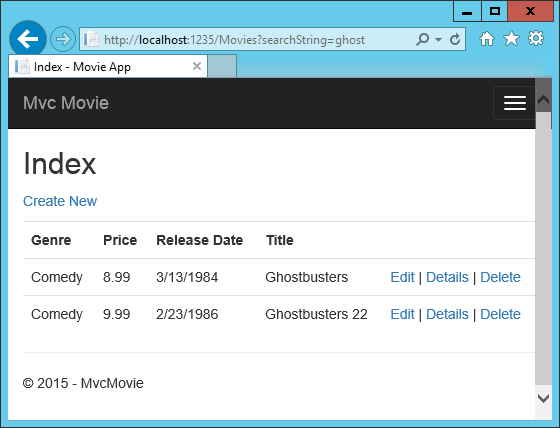
}

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. Instead, query execution is deferred, which 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 the c# code above. On the database,Contains maps to SQL LIKE, which is case insensitive.

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 idparameter will match the optional {id} placeholder for the default routes set in Startup.cs.

app.UseMvc(routes =>

{

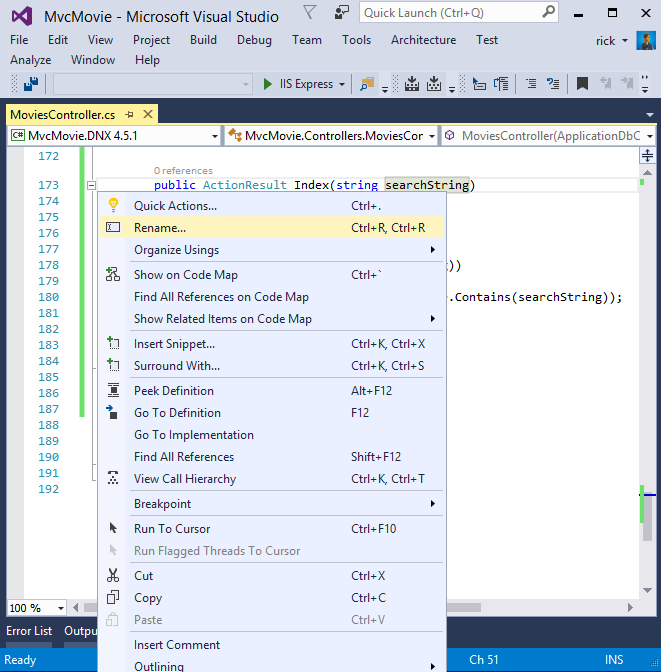
routes.MapRoute(

name: "default",

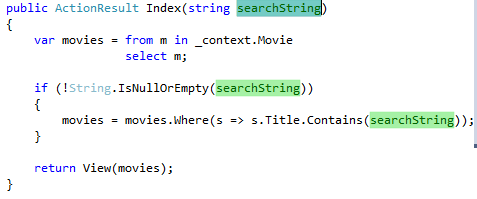
template: "{controller=Home}/{action=Index}/{id?}");

});

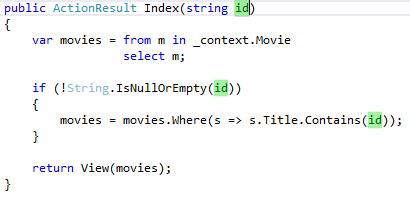
You can quickly rename the searchString parameter to id with the **rename** command. Right click on searchString **> Rename**.



The rename targets are highlighted.



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



The previous Index method:

**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:

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

{

**var** movies = **from** m **in** \_context.Movie

**select** m;

**if** (!String.IsNullOrEmpty(id))

{

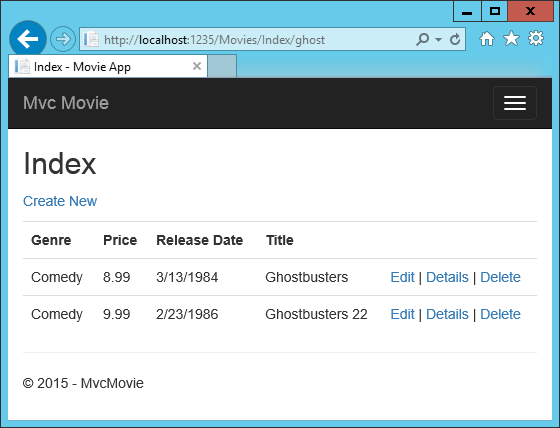
movies = movies.Where(s => s.Title.Contains(id));

}

**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:

**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:

@{

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">

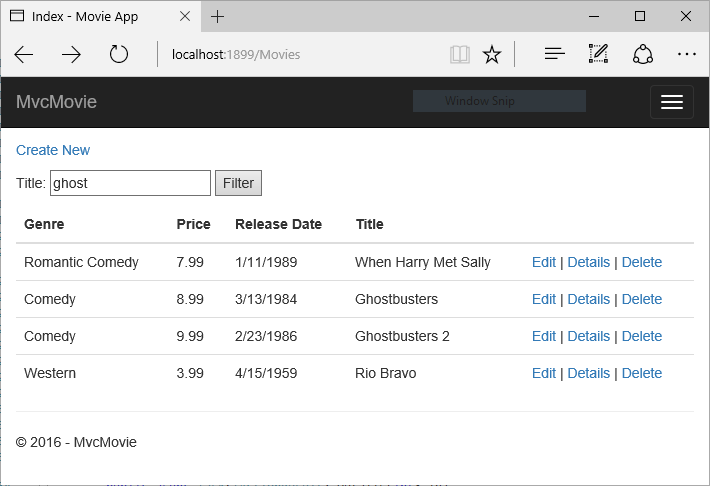
<input type**=**"submit" value**=**"Filter" />

</p>

</form>

<table class**=**"table">

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.



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.

[HttpPost]

**public** **string** **Index**(**string** searchString, **bool** notUsed)

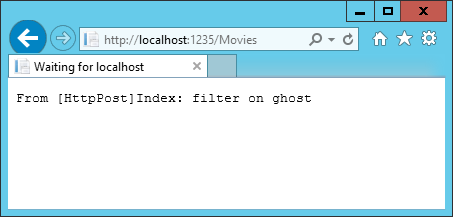
{

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

}

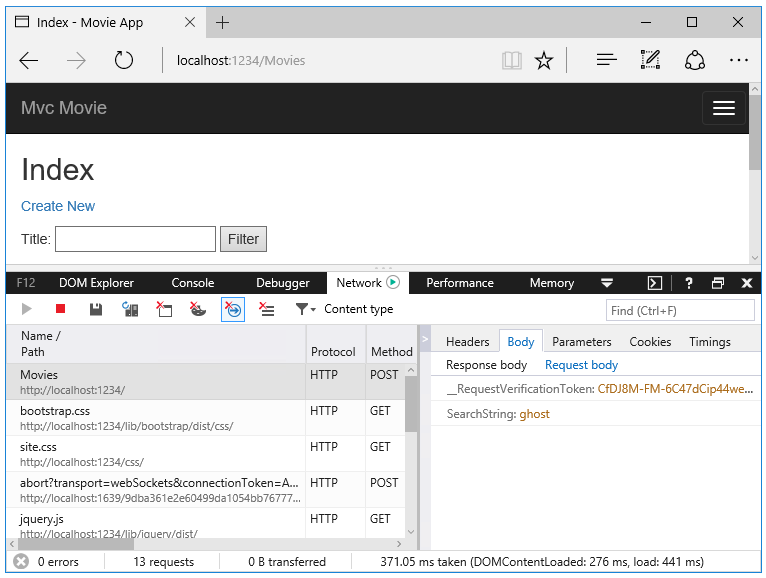
The notUsed parameter is used to create an overload for the Index method. We’ll talk about that later in the tutorial.

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.



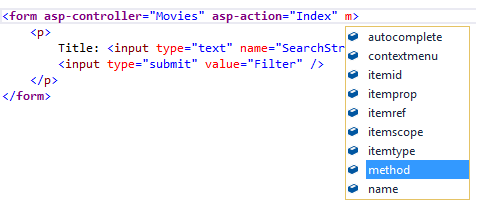
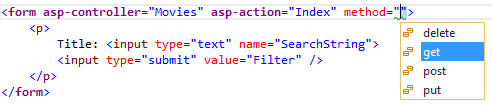
However, even if you add this [HttpPost] version of the Index method, there’s a limitation in how this has all been implemented. Imagine that you want to bookmark a particular search or you want to send a link to friends that they can click in order to see the same filtered list of movies. Notice that 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 F12 Developer tools or the excellent Fiddler tool. Start the F12 tool:

Tap the **http://localhost:xxx/Movies HTTP POST 200** line and then tap **Body > Request Body**.

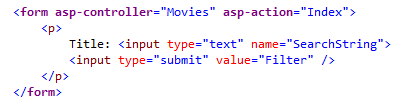


You can see the search parameter and XSRF token in the request body. Note, as mentioned in the previous tutorial, the Form Tag Helper generates an XSRF anti-forgery token. We’re not modifying data, so we don’t need to validate the token in the controller method.

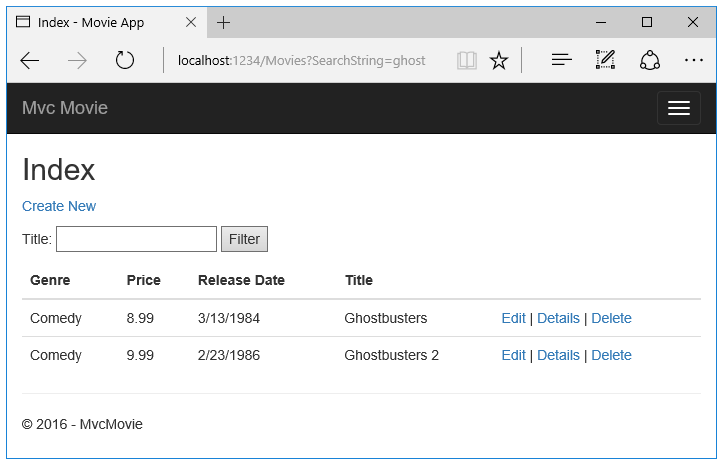
Because the search parameter is in the request body and not the URL, you can’t capture that search information to bookmark or share with others. We’ll fix this by specifying the request should be HTTP GET. Notice how intelliSense helps us update the markup.

Notice the distinctive font in the <form> tag. That distinctive font indicates the tag is supported byTag Helpers.



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:

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

## Adding Search by Genre

Add the following MovieGenreViewModel class to the Models folder:

**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 with the following code:

**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.

IQueryable<**string**> genreQuery = **from** m **in** \_context.Movie

**orderby** m.Genre

**select** m.Genre;

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

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

## Adding search by genre to the Index view

@model 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">

<tr>

<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].ReleaseDate)

</th>

<th>

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

</th>

<th></th>

</tr>

<tbody>

@foreach (var item in Model.movies)

{

<tr>

<td>

@Html.DisplayFor(modelItem => item.Genre)

</td>

<td>

@Html.DisplayFor(modelItem => item.Price)

</td>

<td>

@Html.DisplayFor(modelItem => item.ReleaseDate)

</td>

<td>

@Html.DisplayFor(modelItem => item.Title)

</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>

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

# Adding a New Field

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 track down issues at development time that you might otherwise only find (by obscure errors) at run time.

## Adding a Rating Property to the Movie Model

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

**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. Update the [Bind] attribute for Create and Edit action methods to include the Rating property:

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

You also need to update the view templates in order to display, create and edit the new Ratingproperty in the browser view.

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

<table class**=**"table">

<tr>

<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].ReleaseDate)

</th>

<th>

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

</th>

<th>

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

</th>

<th></th>

</tr>

<tbody>

@foreach (var item in Model.movies)

{

<tr>

<td>

@Html.DisplayFor(modelItem => item.Genre)

</td>

<td>

@Html.DisplayFor(modelItem => item.Price)

</td>

<td>

@Html.DisplayFor(modelItem => item.ReleaseDate)

</td>

<td>

@Html.DisplayFor(modelItem => item.Title)

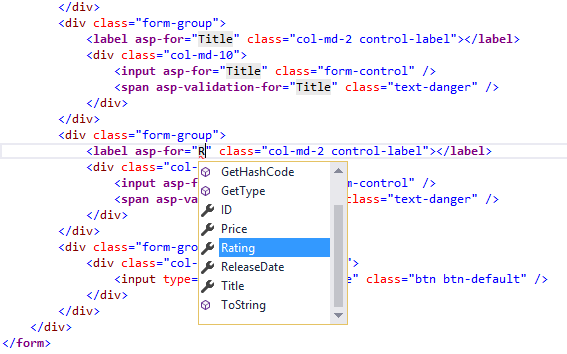
</td>

<td>

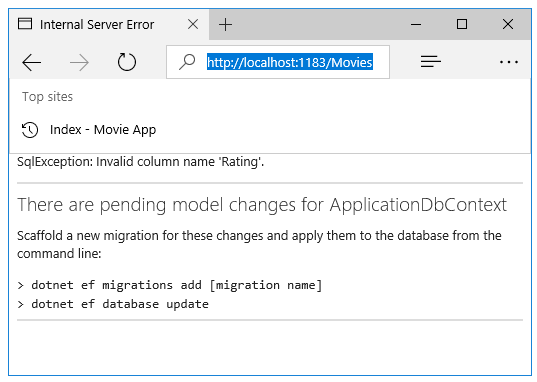
@Html.DisplayFor(modelItem => item.Rating)

</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.



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:



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.

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.

**new** Movie

{

Title = "When Harry Met Sally",

ReleaseDate = DateTime.Parse("1989-1-11"),

Genre = "Romantic Comedy",

Rating = "R",

Price = 7.99M

},

**Warning**

You must stop IIS Express before you run the dotnet ef commands. See To Stop IIS Express:

Build the solution then open a command prompt. Enter the following commands:

dotnet ef migrations add Rating

dotnet ef database update

The migrations add 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 step.

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.

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

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 attempts to create or edit 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.

The validation support provided by MVC and Entity Framework Core Code First is a great 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.

Let’s look at how you can take advantage of this validation support in the movie 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 DataTypethat 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.

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; }

}

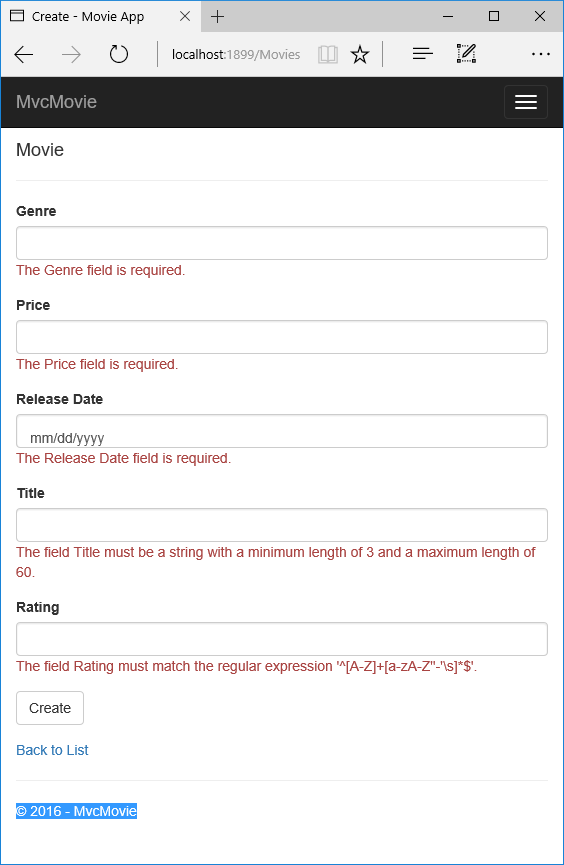
The validation attributes specify behavior that you want to enforce on the model properties they are applied to. The Required and MinimumLength attributes indicates that a property must have a value; but nothing prevents a user from entering white space to satisfy this validation. TheRegularExpression attribute is used to limit what characters can be input. In the code above, Genreand 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.

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.



**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 MoviesControllerclass 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 theEdit 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 Occurs in the Create View and Create Action Method

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

**public** IActionResult **Create**()

{

**return** **View**();

}

*// POST: Movies/Create*

*// 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> Create([Bind("ID,Genre,Price,ReleaseDate,Title,Rating")] Movie movie)

{

**if** (ModelState.IsValid)

{

\_context.Add(movie);

**await** \_context.SaveChangesAsync();

**return** **RedirectToAction**("Index");

}

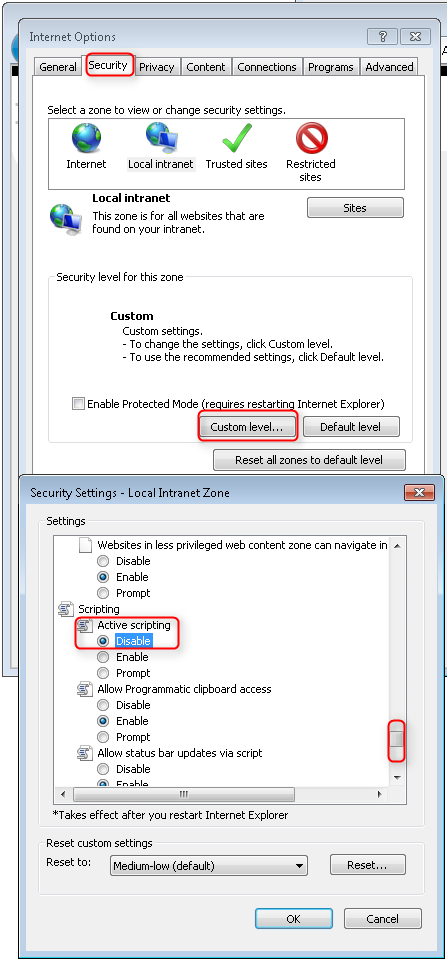
**return** **View**(movie);

}

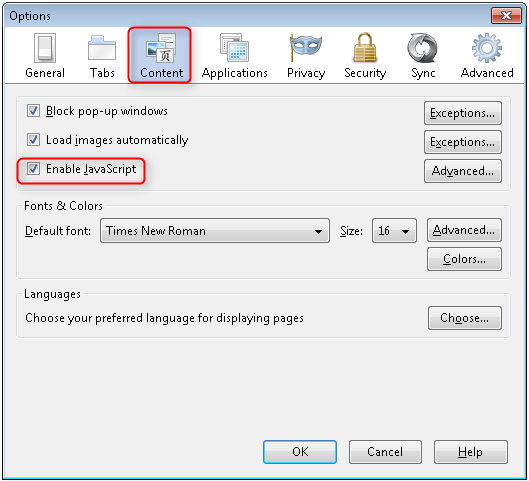
**#region snippet\_edit\_get**

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.

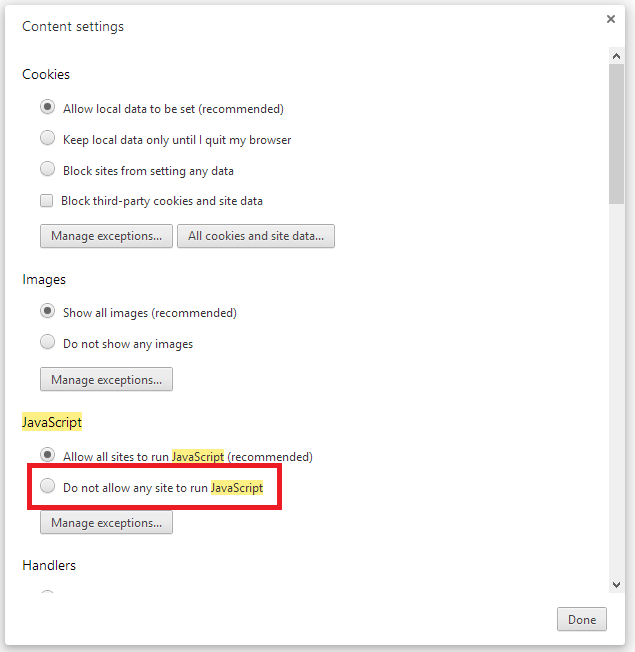
You can set a break point in the [HttpPost] Create method and verify the method is never called, client side validation will not submit the form data when validation errors are detected. If you disable JavaScript in your browser, then submit the form with errors, the break point will be hit. You still get full validation without JavaScript. The following image shows how to disable JavaScript in Internet Explorer.



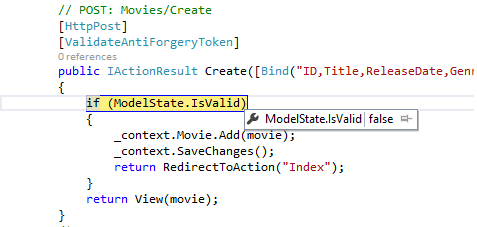
The following image shows how to disable JavaScript in the FireFox browser.



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



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



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.

<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**=**"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>

@\*Markup removed for brevity.\*@

<div class**=**"form-group">

<label asp-for**=**"Rating" class**=**"col-md-2 control-label"></label>

<div class**=**"col-md-10">

<input asp-for**=**"Rating" class**=**"form-control" />

<span asp-validation-for**=**"Rating" class**=**"text-danger"></span>

</div>

</div>

<div class**=**"form-group">

<div class**=**"col-md-offset-2 col-md-10">

<input type**=**"submit" value**=**"Create" class**=**"btn btn-default" />

</div>

</div>

</div>

</form>

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

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

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

[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 theRegularExpression 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 DataTypeEnumeration 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 forDataType.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:

[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 DataTypeattribute. 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:

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

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

The following code shows combining attributes on one line:

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.

# Examining the Details and Delete methods

Open the Movie controller and examine the Details method:

**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);

}

**#endregion**

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

*#region snippet\_1*

app**.**UseMvc(routes **=>**

{

routes**.**MapRoute(

name: "default",

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

});

*#endregion*

Code First 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 the code 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.

Examine the Delete and DeleteConfirmed methods.

**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:

*// GET: Movies/Delete/5*

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

*// POST: Movies/Delete/*

[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 twoDelete 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:

[ValidateAntiForgeryToken]

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

{

**var** movie = **await** \_context.Movie.SingleOrDefaultAsync(m => m.ID == id);

\_context.Movie.Remove(movie);

**await** \_context.SaveChangesAsync();

**return** **RedirectToAction**("Index");

}