Getting Started with ASP.NET MVC 5

11/23/2017 • 1 min to read • Edit Online

Introduction to ASP.NET MVC 5

An updated version of this tutorial is available here using the latest version of Visual Studio. The new tutorial uses ASP.NET Core MVC, which provides **many** improvements over this tutorial.

This tutorial teaches ASP.NET Core MVC with controllers and views. Razor Pages is a new alternative in ASP.NET Core 2.0, a page-based programming model that makes building web UI easier and more productive. We recommend you try the Razor Pages tutorial before the MVC version. The Razor Pages tutorial:

- Is easier to follow.
- Covers more features.
- Is the preferred approach for new application development.

This following tutorial series covers ASP.NET MVC: Source located on GitHub

- Getting Started
- Adding a Controller
- Adding a View
- Adding a Model
- Creating a Connection String and Working with SQL Server LocalDB
- Accessing Your Model's Data from a Controller
- Examining the Edit Methods and Edit View
- Adding Search
- Adding a New Field
- Adding Validation
- Examining the Details and Delete Methods

Getting Started with ASP.NET MVC 5

1/24/2018 • 3 min to read • Edit Online

by Rick Anderson

An updated version of this tutorial is available here using the latest version of Visual Studio. The new tutorial uses ASP.NET Core MVC, which provides **many** improvements over this tutorial.

This tutorial teaches ASP.NET Core MVC with controllers and views. Razor Pages is a new alternative in ASP.NET Core 2.0, a page-based programming model that makes building web UI easier and more productive. We recommend you try the Razor Pages tutorial before the MVC version. The Razor Pages tutorial:

- Is easier to follow.
- Covers more features.
- Is the preferred approach for new application development.

This tutorial will teach you the basics of building an ASP.NET MVC 5 web app using Visual Studio 2017. Final Source for tutorial located on GitHub

This tutorial was written by Scott Guthrie (twitter@scottgu), Scott Hanselman (twitter: @shanselman), and Rick Anderson (@RickAndMSFT)

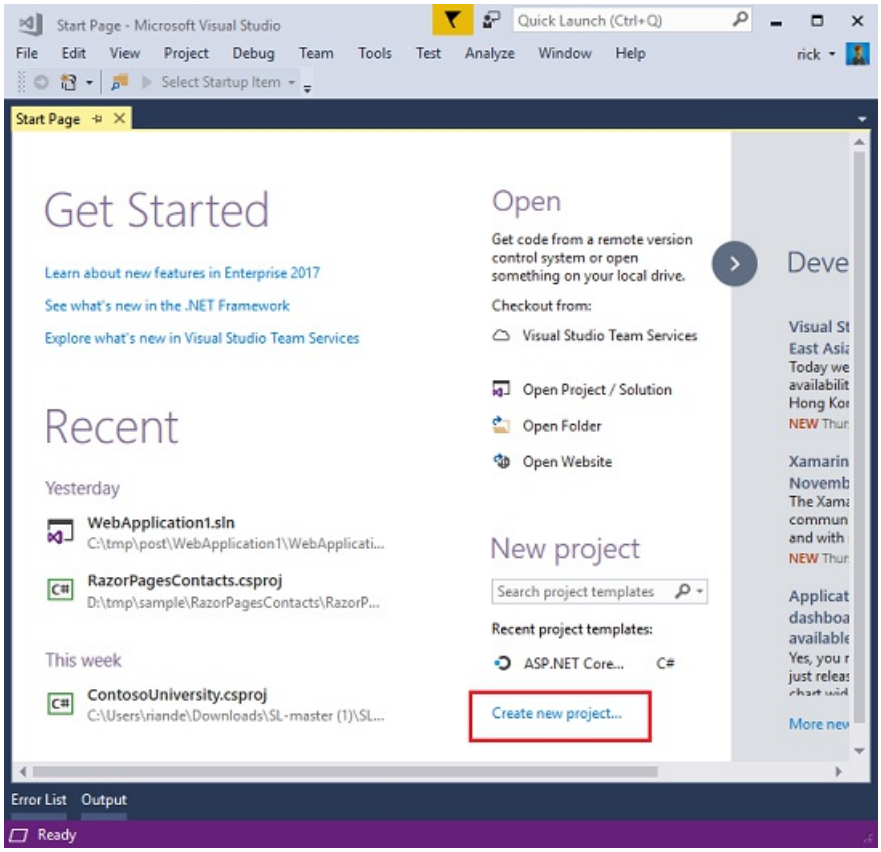
You need an Azure account to deploy this app to Azure:

- You can open an Azure account for free - You get credits you can use to try out paid Azure services, and even after they're used up you can keep the account and use free Azure services.
- You can activate MSDN subscriber benefits - Your MSDN subscription gives you credits every month that you can use for paid Azure services.

Getting Started

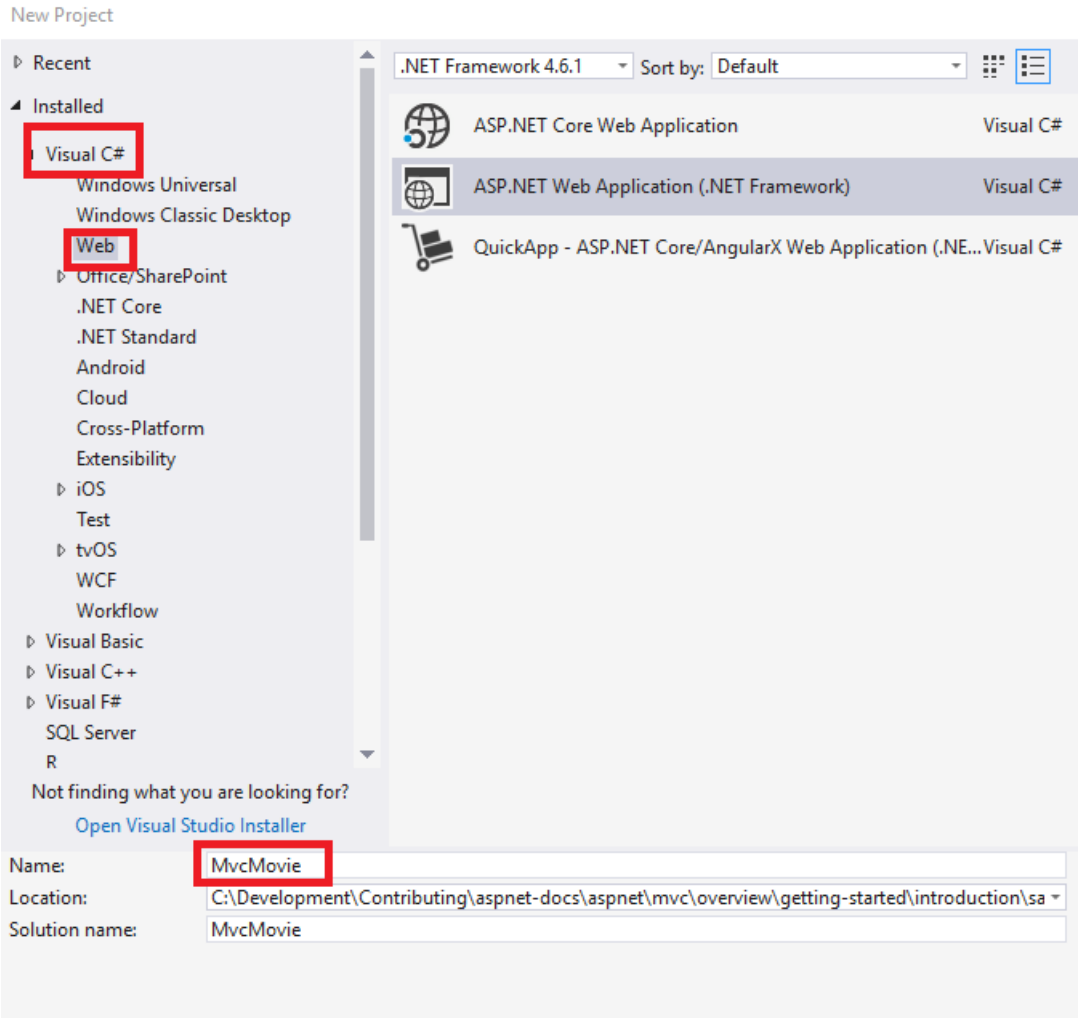
Start by installing and running Visual Studio 2017.

Visual Studio is an IDE, or integrated development environment. Just like you use Microsoft Word to write documents, you'll use an IDE to create applications. In Visual Studio there's a list along the bottom showing various options available to you. There's also a menu that provides another way to perform tasks in the IDE. (For example, instead of selecting **New Project** from the **Start** page, you can use the menu and select **File** > **New Project**.)

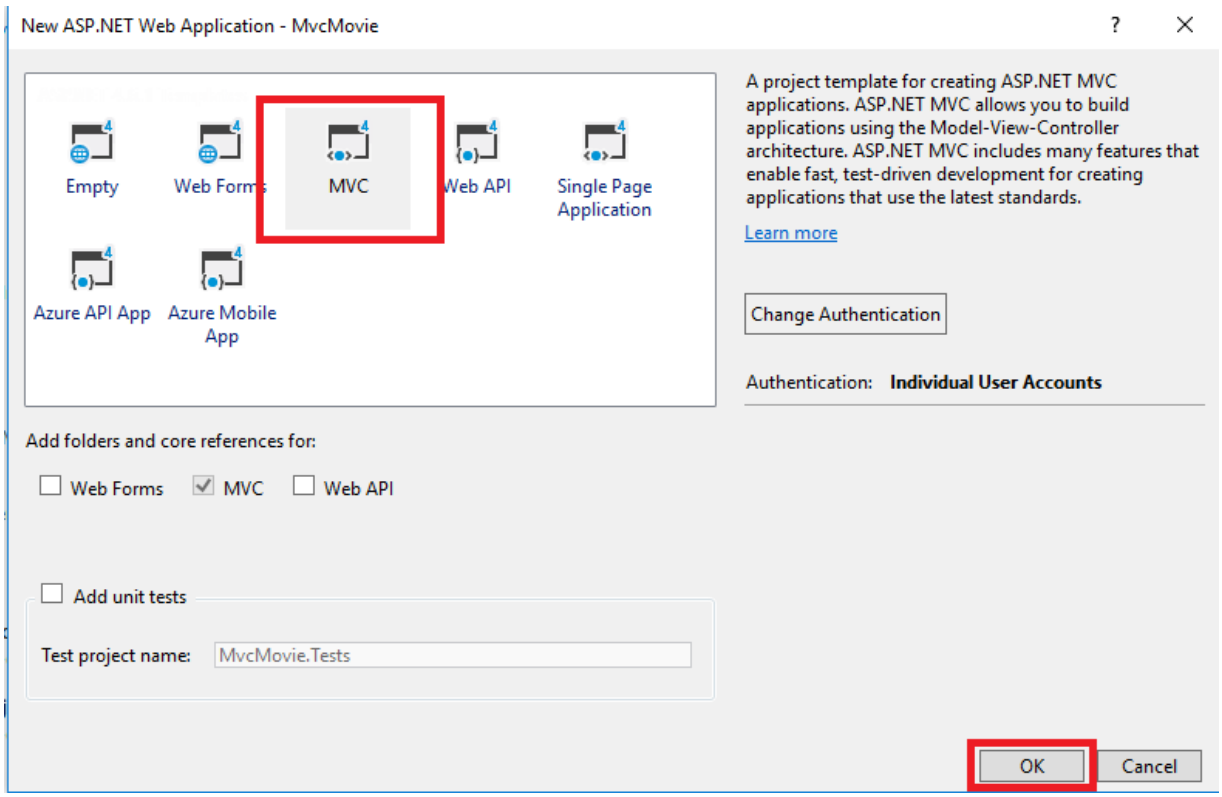


Creating Your First Application

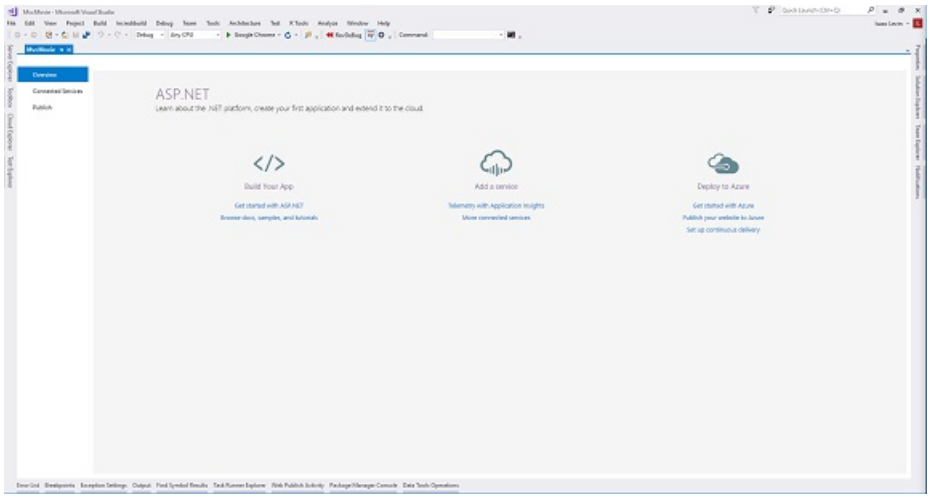
Click **New Project**, then select Visual C# on the left, then **Web** and then select **ASP.NET Web Application (.NET Framework)**. Name your project "MvcMovie" and then click **OK**.



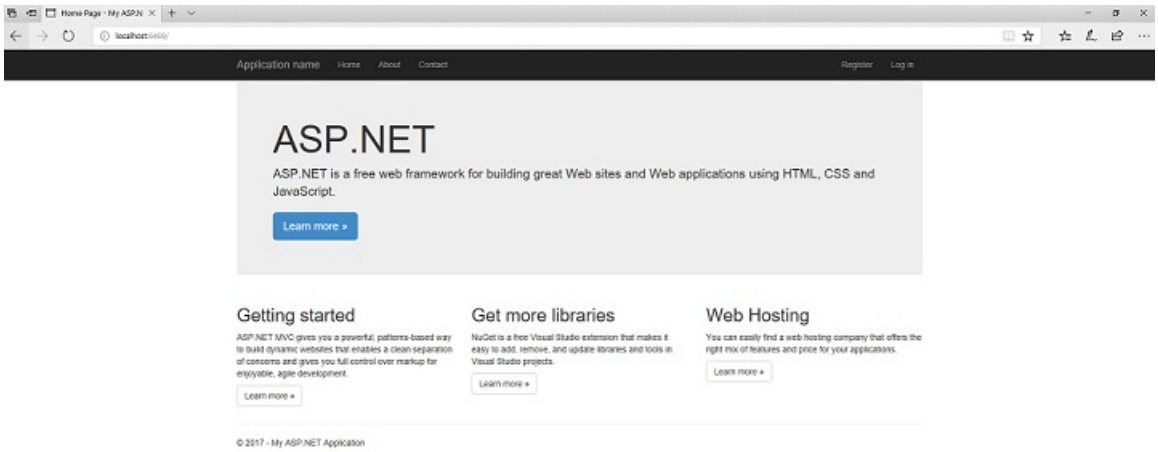
In the **New ASP.NET Project** dialog, click **MVC** and then click **OK**.



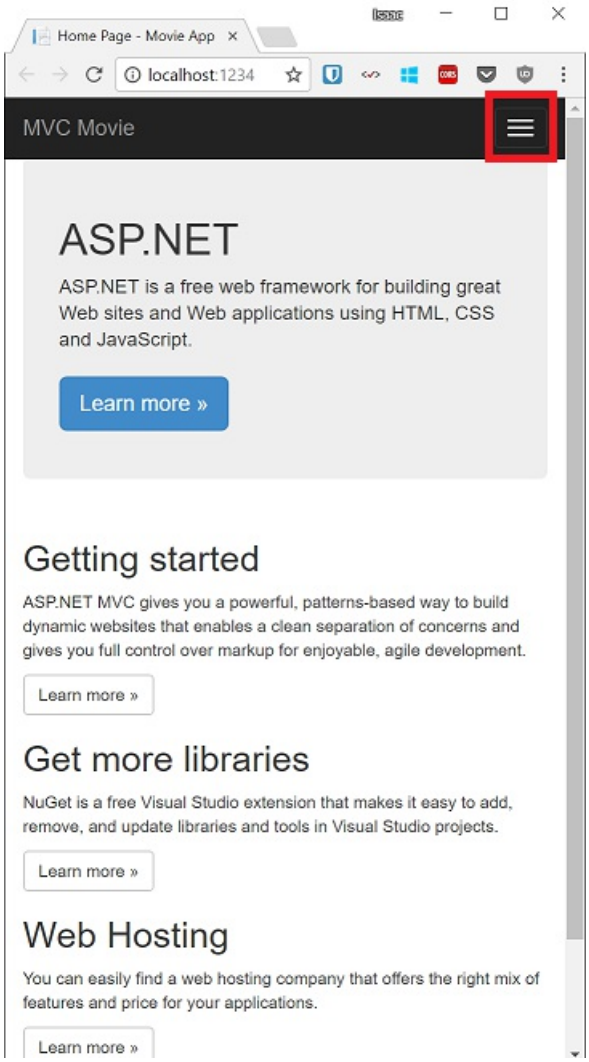
Visual Studio used a default template for the ASP.NET MVC project you just created, so you have a working application right now without doing anything! This is a simple "Hello World!" project, and it's a good place to start your application.



Click F5 to start debugging. F5 causes Visual Studio to start IIS Express and run your web app. Visual Studio then launches a browser and opens the application's home page. Notice that the address bar of the browser says `localhost: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 application you just built. When Visual Studio runs a web project, a random port is used for the web server. In the image below, the port number is 1234. When you run the application, you'll see a different port number.



Right out of the box this default template gives you Home, Contact and About pages. The image above doesn't show the **Home**, **About** and **Contact** links. Depending on the size of your browser window, you might need to click the navigation icon to see these links.



The application also provides support to register and log in. The next step is to change how this application works and learn a little bit about ASP.NET MVC. Close the ASP.NET MVC application and let's change some code.

For a list of current tutorials, see MVC recommended articles.

See this App Running on Azure

Would you like to see the finished site running as a live web app? You can deploy a complete version of the app to your Azure account by simply clicking the following button.

Deploy to Azure

You need an Azure account to deploy this solution to Azure. If you do not already have an account, you have the following options:

- Open an Azure account for free - You get credits you can use to try out paid Azure services, and even after they're used up you can keep the account and use free Azure services.
- Activate MSDN subscriber benefits - Your MSDN subscription gives you credits every month that you can use for paid Azure services.

NEXT

Adding a Controller

1/24/2018 • 6 min to read • Edit Online

by Rick Anderson

> **NOTE**
>
> This document is part of the Getting Started with ASP.NET MVC 5 tutorial. Final Source for tutorial located on GitHub

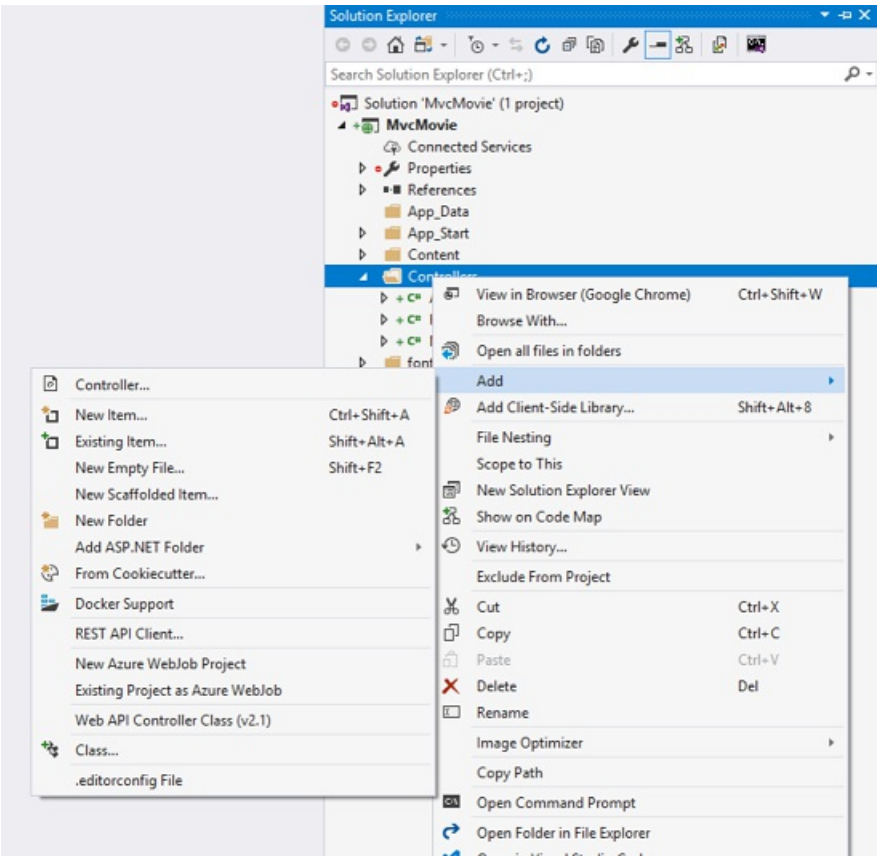
MVC stands for *model-view-controller*. MVC is a pattern for developing applications that are well architected, testable and easy to maintain. MVC-based applications contain:

- **M** odels: Classes that represent the data of the application and that use validation logic to enforce business rules for that data.
- **V** iews: Template files that your application uses to dynamically generate HTML responses.
- **C** ontrollers: Classes that handle incoming browser requests, retrieve model data, and then specify view templates that return a response to the browser.

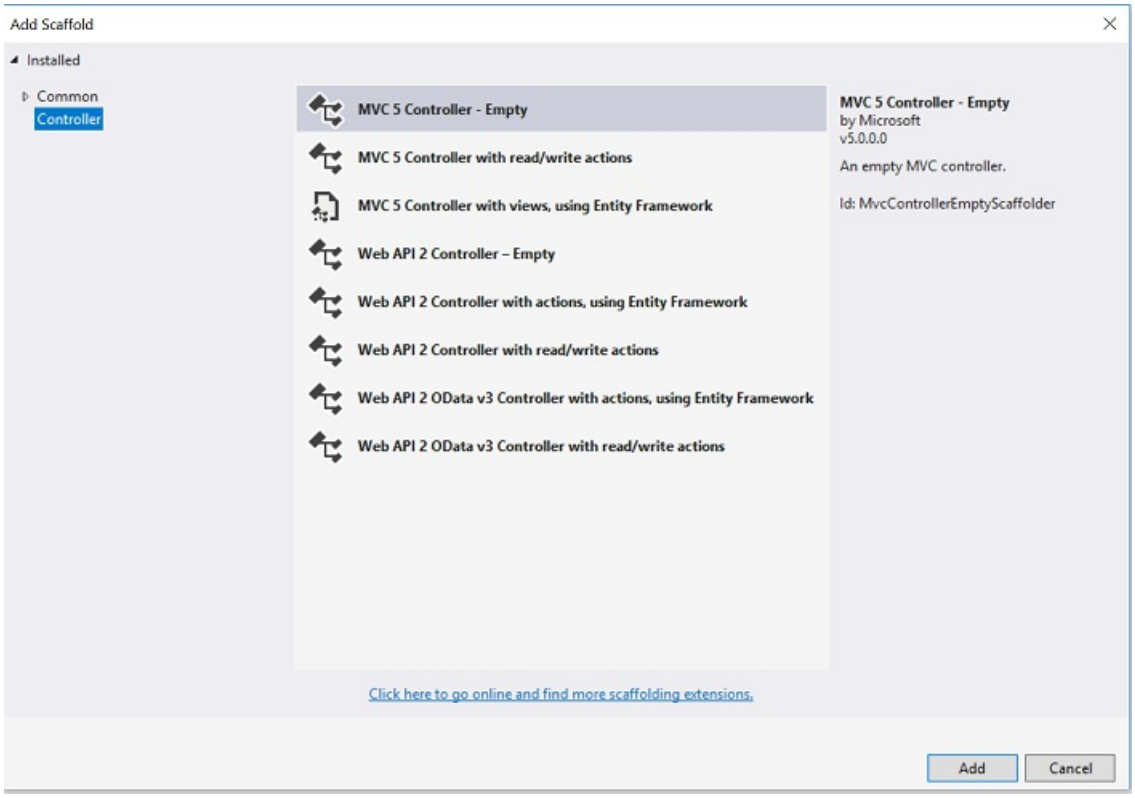
We'll be covering all these concepts in this tutorial series and show you how to use them to build an application.

> **NOTE**
>
> In the previous step the Default MVC template was selected. This creates Home, Account and Manage Controllers by default.

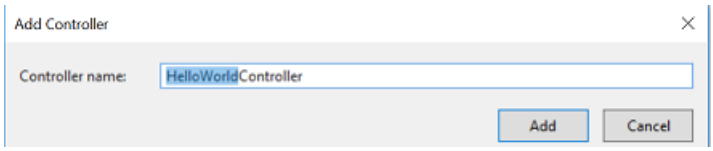
Let's begin by creating a controller class. In **Solution Explorer**, right-click the *Controllers* folder and then click **Add**, then **Controller**.



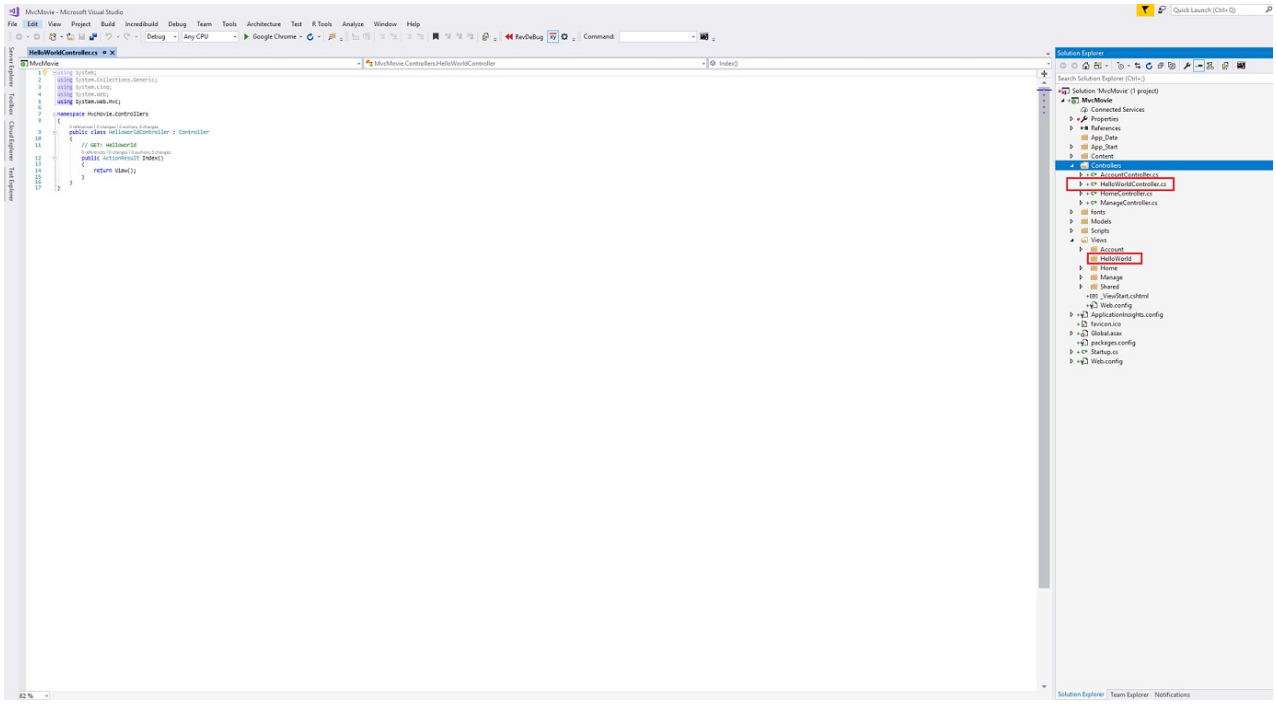
In the **Add Scaffold** dialog box, click **MVC 5 Controller - Empty**, and then click **Add**.



Name your new controller "HelloWorldController" and click **Add**.



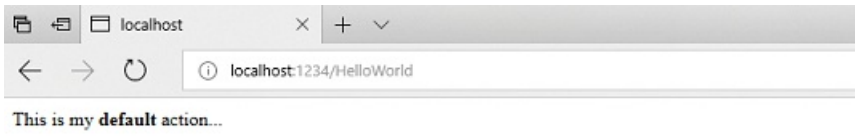
Notice in **Solution Explorer** that a new file has been created named *HelloWorldController.cs* and a new folder *Views\HelloWorld*. The controller is open in the IDE.



Replace the contents of the file with the following code.

```
using System.Web;
using System.Web.Mvc;

namespace MvcMovie.Controllers
{
    public class HelloWorldController : Controller
    {
        //
        // GET: /HelloWorld/

        public string Index()
        {
            return "This is my <b>default</b> action...";
        }

        //
        // GET: /HelloWorld/Welcome/

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

The controller methods will return a string of HTML as an example. The controller is named `HelloWorldController` and the first method is named `Index`. Let's invoke it from a browser. Run the application (press F5 or Ctrl+F5). In the browser, append "HelloWorld" to the path in the address bar. (For example, in the illustration below, it's `http://localhost:1234/HelloWorld.`) The page in the browser will look like the following screenshot. In the method above, the code returned a string directly. You told the system to just return some HTML, and it did!



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

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

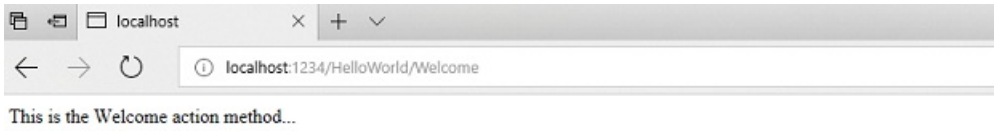
You set the format for routing in the *App_Start/RouteConfig.cs* file.

```
public static void RegisterRoutes(RouteCollection routes)
{
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

    routes.MapRoute(
        name: "Default",
        url: "{controller}/{action}/{id}",
        defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }
    );
}
```

When you run the application and don't supply any URL segments, it defaults to the "Home" controller and the "Index" action method specified in the defaults section of the code above.

The first part of the URL determines the controller class to execute. So */HelloWorld* maps to the `HelloWorldController` class. The second part of the URL determines the action method on the class to execute. So */HelloWorld/Index* would cause the `Index` method of the `HelloWorldController` class to execute. Notice that we only had to browse to */HelloWorld* and the `Index` method was used by default. This is because a method named `Index` is the default method that will be called on a controller if one is not explicitly specified. The third part of the URL segment (`Parameters`) 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...". The default MVC mapping is `/[Controller]/[ActionName]/[Parameters]`. For this URL, the controller is `HelloWorld` and `Welcome` is the action method. You 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 your `Welcome` method to include two parameters as shown below. Note that the code uses the C# optional-parameter feature to indicate that the `numTimes` parameter should default to 1 if no value is passed for that parameter.

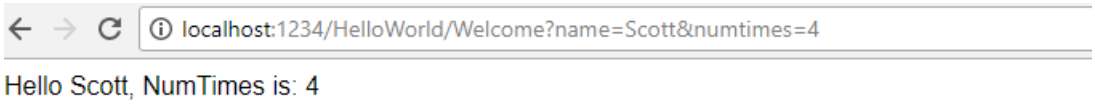
```
public string Welcome(string name, int numTimes = 1) {
     return HttpUtility.HtmlEncode("Hello " + name + ", NumTimes is: " + numTimes);
}
```

NOTE

Security Note: The code above uses HttpUtility.HtmlEncode to protect the application from malicious input (namely JavaScript). For more information see How to: Protect Against Script Exploits in a Web Application by Applying HTML Encoding to Strings.

Run your application and browse to the example URL (`http://localhost:xxxx/HelloWorld/Welcome?name=Scott&numtimes=4`). You can try different values for `name` and `numtimes` in the URL. The ASP.NET MVC model binding system automatically maps the named parameters from

the query string in the address bar to parameters in your method.



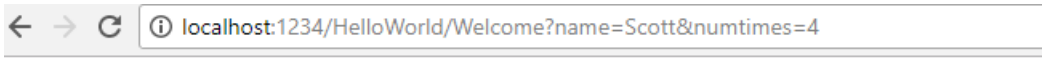
Hello Scott, NumTimes is: 4

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

Replace the Welcome method with the following code:

```
public string Welcome(string name, int ID = 1)
{
    return HttpUtility.HtmlEncode("Hello " + name + ", ID: " + ID);
}
```

Run the application and enter the following URL: `http://localhost:xxx/HelloWorld/Welcome/1?name=Scott`



Hello Scott, ID: 1

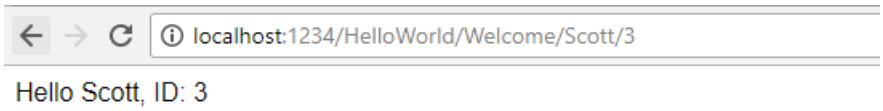
This time the third URL segment matched the route parameter `ID.` The `Welcome` action method contains a parameter (`ID`) that matched the URL specification in the `RegisterRoutes` method.

```
public static void RegisterRoutes(RouteCollection routes)
{
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

    routes.MapRoute(
        name: "Default",
        url: "{controller}/{action}/{id}",
        defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }
    );
}
```

In ASP.NET MVC applications, it's more typical to pass in parameters as route data (like we did with ID above) than passing them as query strings. You could also add a route to pass both the `name` and `numtimes` in parameters as route data in the URL. In the *App_Start\RouteConfig.cs* file, add the "Hello" route:

```
public class RouteConfig
{
   public static void RegisterRoutes(RouteCollection routes)
   {
      routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

      routes.MapRoute(
          name: "Default",
          url: "{controller}/{action}/{id}",
          defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }
      );

      routes.MapRoute(
           name: "Hello",
           url: "{controller}/{action}/{name}/{id}"
       );
   }
}
```

Run the application and browse to `/localhost:XXX/HelloWorld/Welcome/Scott/3`.



For many MVC applications, the default route works fine. You'll learn later in this tutorial to pass data using the model binder, and you won't have to modify the default route for that.

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. Ordinarily you don't want controllers returning HTML directly, since that becomes very cumbersome to code. Instead we'll typically use a separate view template file to help generate the HTML response. Let's look next at how we can do this.

PREVIOUS NEXT

Adding a View

1/24/2018 • 9 min to read • Edit Online

by Rick Anderson

> **NOTE**
>
> This document is part of the Getting Started with ASP.NET MVC 5 tutorial. Final Source for tutorial located on GitHub

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

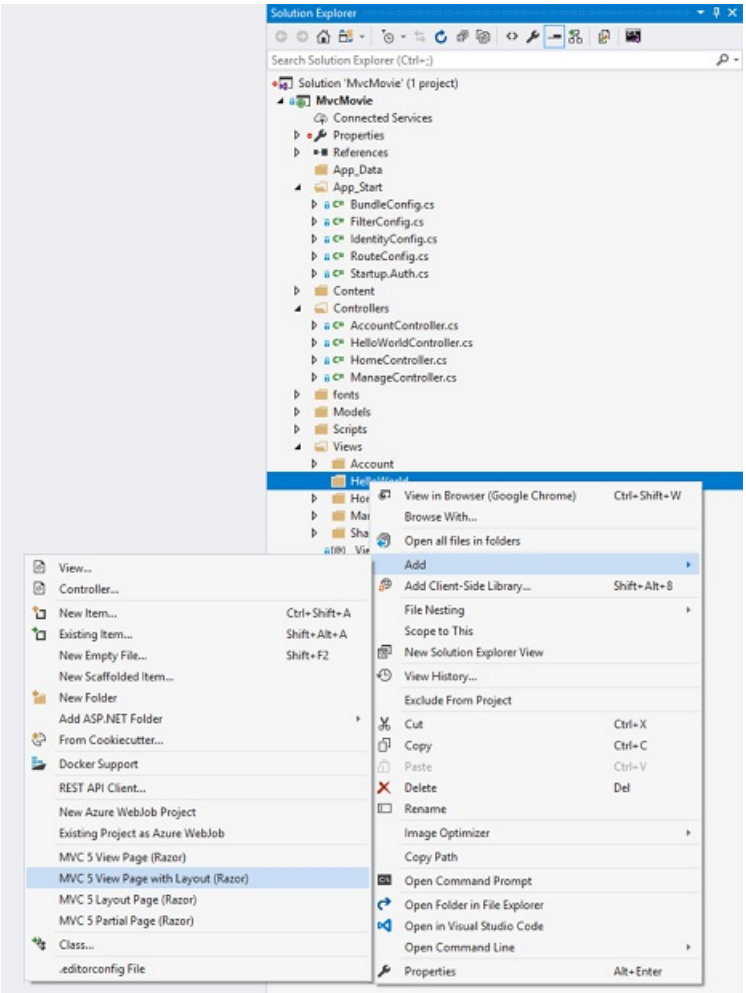
You'll create a view template file using the Razor view engine. Razor-based view templates have a *.cshtml* file extension, and provide an elegant way to create HTML output using C#. Razor minimizes the number of characters and keystrokes required when writing a view template, and enables a fast, fluid 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:

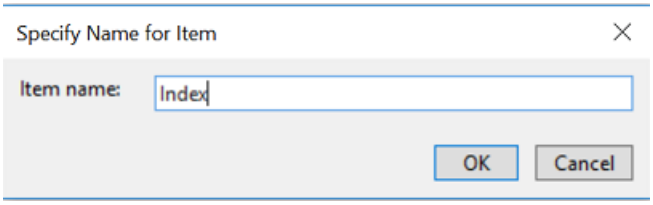
```
public ActionResult Index()
{
    return View();
}
```

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 ActionResult (or a class derived from ActionResult), not primitive types like string.

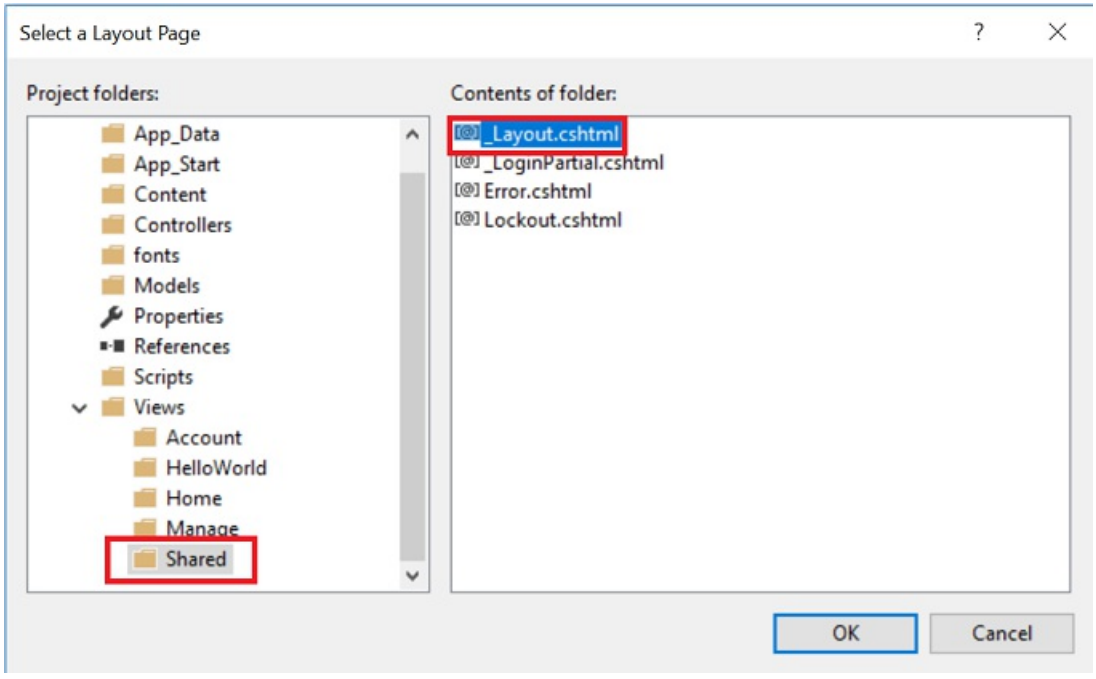
Right click the *Views\HelloWorld* folder and click **Add**, then click **MVC 5 View Page with Layout (Razor)**.



In the **Specify Name for Item** dialog box, enter *Index*, and then click **OK**.

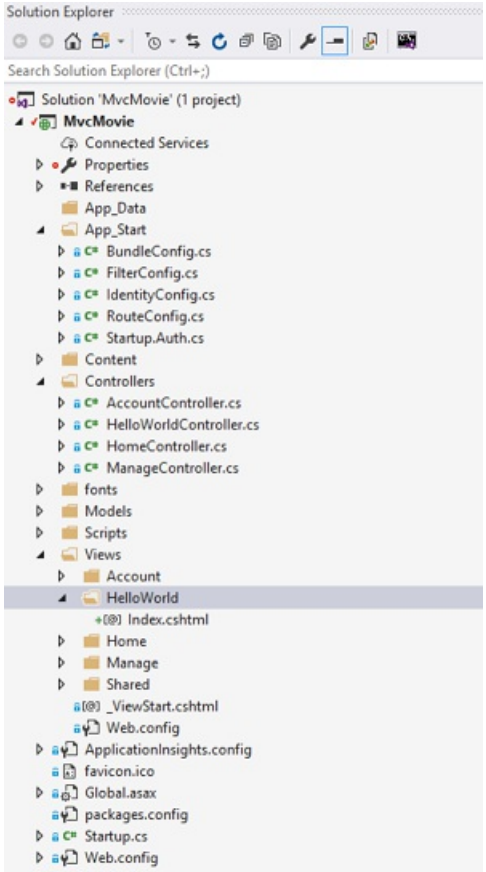


In the **Select a Layout Page** dialog, accept the default **_Layout.cshtml** and click **OK**.



In the dialog above, the *Views\Shared* folder is selected in the left pane. If you had a custom layout file in another folder, you could select it. We'll talk about the layout file later in the tutorial

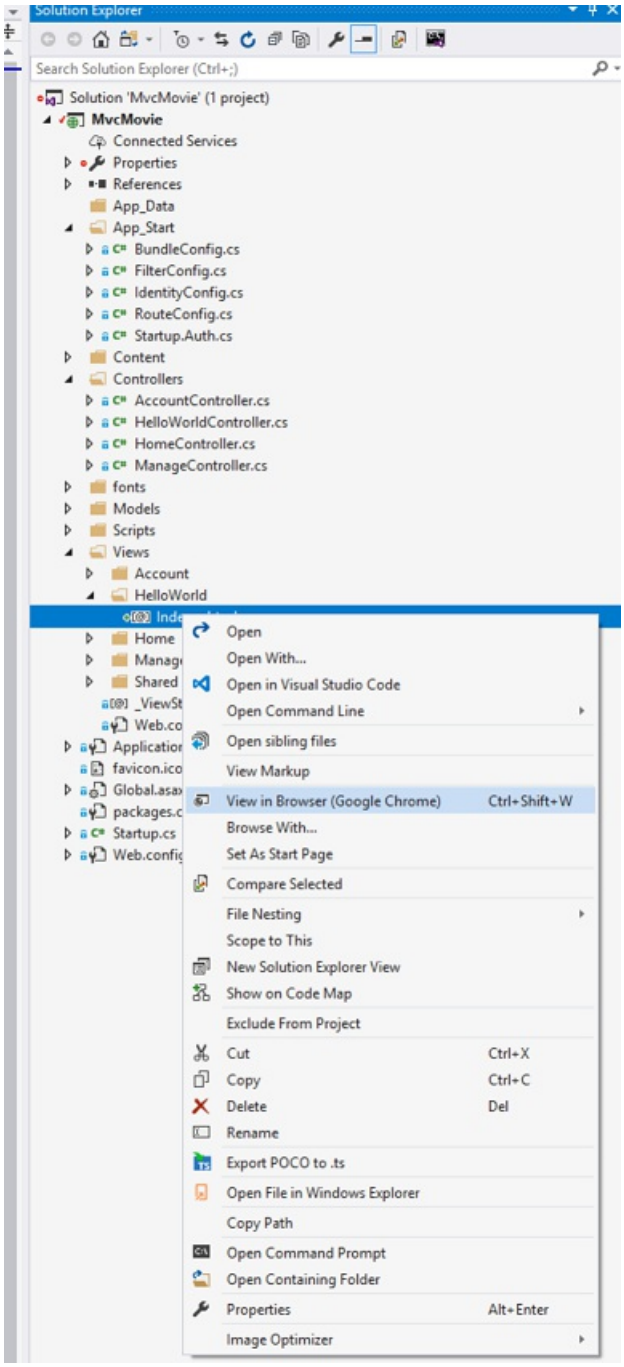
The *MvcMovie\Views\HelloWorld\Index.cshtml* file is created.



Add the following highlighted markup.

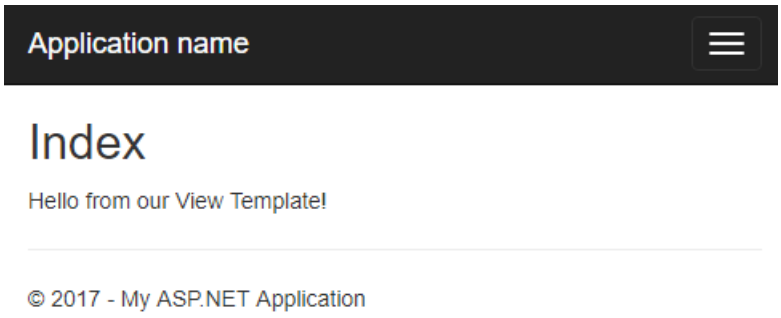
```
@{
    Layout = "~/Views/Shared/_Layout.cshtml";
}

@{
    ViewBag.Title = "Index";
}

<h2>Index</h2>

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

Right click the *Index.cshtml* file and select **View in Browser**.



You can also right click the *Index.cshtml* file and select **View in Page Inspector.** See the Page Inspector tutorial for more information.

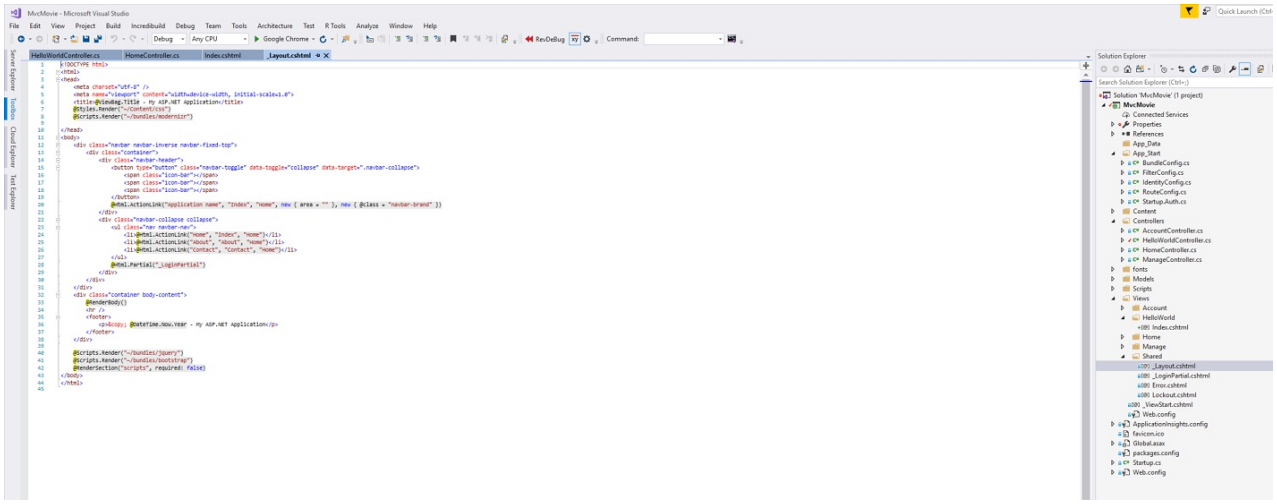
Alternatively, run the application and browse to the `HelloWorld` controller (`http://localhost:xxxx/HelloWorld`). The `Index` method in your controller didn't do much work; it simply ran the statement `return View()`, which specified that the method should use a view template file to render a response to the browser. Because you didn't explicitly specify the name of the view template file to use, ASP.NET 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.



Looks pretty good. However, notice that the browser's title bar shows "Index - My ASP.NET Appli" and the big link on the top of the page says "Application name." Depending on how small you make your browser window, you might need to click the three bars in the upper right to see the to the **Home**, **About**, **Contact**, **Register** and **Log in** links.

Changing Views and Layout Pages

First, you want to change the "Application name" link at the top of the page. That text is common to every page. It's actually implemented in only one place in the project, even though it appears on every page in the application. Go to the */Views/Shared* folder in **Solution Explorer** and open the *_Layout.cshtml* file. This file is called a *layout page* and it's in the shared folder that all other pages use.

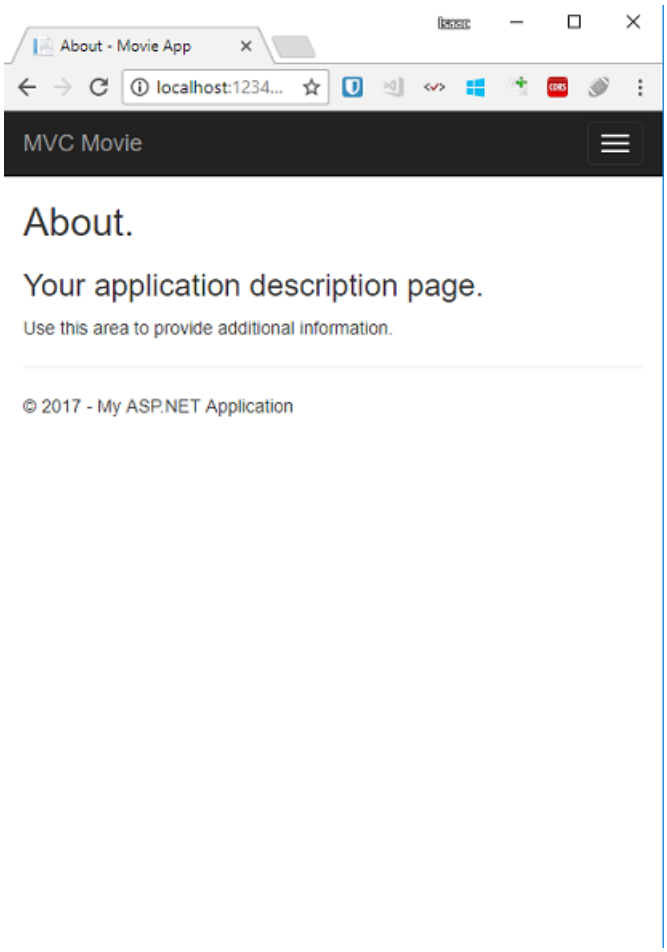


Layout templates allow you to specify the HTML container layout of your site in one place and then apply it across multiple pages in your site. Find the `@RenderBody()` line. `RenderBody` is a placeholder where all the view-specific pages you create show up, "wrapped" in the layout page. For example, if you select the **About** link, the *Views\Home\About.cshtml* view is rendered inside the `RenderBody` method.

Change the contents of the title element. Change the ActionLink in the layout template from "Application name" to "MVC Movie" and the controller from `Home` to `Movies`. The complete layout file is shown below:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>@ViewBag.Title - Movie App</title>
    @Styles.Render("~/Content/css")
    @Scripts.Render("~/bundles/modernizr")

</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="icon-bar"></span>
                    <span class="icon-bar"></span>
                    <span class="icon-bar"></span>
                </button>
                @Html.ActionLink("MVC Movie", "Index", "Movies", null, new { @class = "navbar-brand" })
            </div>
            <div class="navbar-collapse collapse">
                <ul class="nav navbar-nav">
                    <li>@Html.ActionLink("Home", "Index", "Home")</li>
                    <li>@Html.ActionLink("About", "About", "Home")</li>
                    <li>@Html.ActionLink("Contact", "Contact", "Home")</li>
                </ul>
            </div>
        </div>
    </div>
    <div class="container body-content">
        @RenderBody()
        <hr />
        <footer>
            <p>&copy; @DateTime.Now.Year - My ASP.NET Application</p>
        </footer>
    </div>

    @Scripts.Render("~/bundles/jquery")
    @Scripts.Render("~/bundles/bootstrap")
    @RenderSection("scripts", required: false)
</body>
</html>
```

Run the application and notice that it now says "MVC Movie ". Click the **About** link, and you see how that page shows "MVC Movie", too. We were able to make the change once in the layout template and have all pages on the site reflect the new title.



When we first created the *Views\HelloWorld\Index.cshtml* file, it contained the following code:

```
@{
    Layout = "~/Views/Shared/_Layout.cshtml";
}
```

The Razor code above is explicitly setting the layout page. Examine the *Views_ViewStart.cshtml* file, it contains the exact same Razor markup. The *Views_ViewStart.cshtml* file defines the common layout that all views will use, therefore you can comment out or remove that code from the *Views\HelloWorld\Index.cshtml* file.

```
@*@{
    Layout = "~/Views/Shared/_Layout.cshtml";
}*@

@{
    ViewBag.Title = "Index";
}

<h2>Index</h2>

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

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 *MvcMovie\Views\HelloWorld\Index.cshtml*. There are two places to make a change: first, the text that appears in the title of the browser, and then in the secondary header (the `<h2>` element). You'll make them slightly different so you can see which bit of code changes which part of the app.

```
@{
    ViewBag.Title = "Movie List";
}

<h2>My Movie List</h2>

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

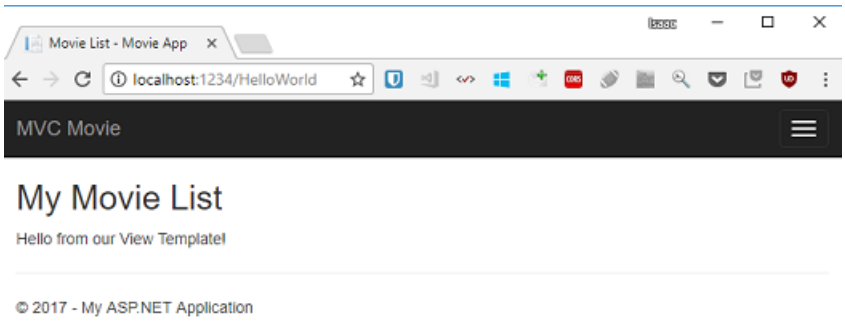
To indicate the HTML title to display, the code above sets a `Title` property of the `ViewBag` object (which is in the *Index.cshtml* view template). Notice that the layout template (*Views\Shared_Layout.cshtml*) uses this value in the `<title>` element as part of the `<head>` section of the HTML that we modified previously.

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>@ViewBag.Title - Movie App</title>
    @Styles.Render("~/Content/css")
    @Scripts.Render("~/bundles/modernizr")
</head>
```

Using this `ViewBag` approach, you can easily pass other parameters between your view template and your layout file.

Run the application. 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 the `ViewBag.Title` we set in the *Index.cshtml* view template and the additional "- Movie App" added in the layout file.

Also notice how the content in the *Index.cshtml* view template was merged with the *_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.



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 classes 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.

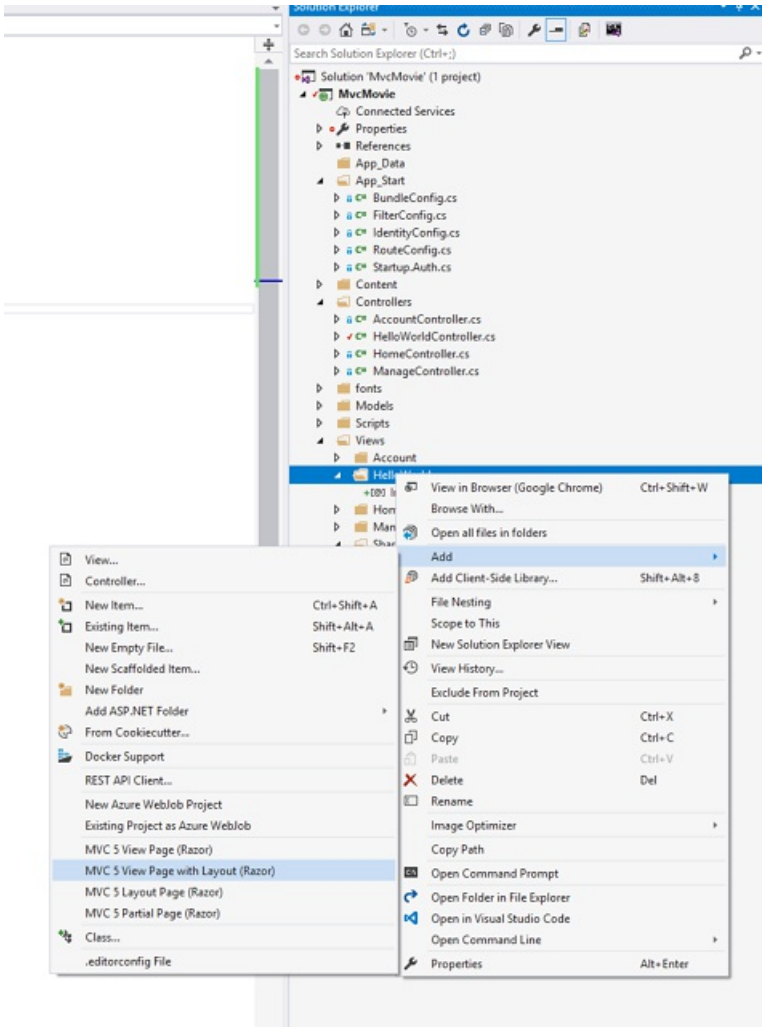
Controllers are responsible for providing whatever data or objects are required in order for a view template to render a response to the browser. A best practice: **A view template should never perform business logic or interact with a database directly**. Instead, a view template should work only with 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` action method in the `HelloWorldController` class takes a `name` and a `numTimes` 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 `ViewBag` object that the view template can then access.

Return to the *HelloWorldController.cs* file and change the `Welcome` method to add a `Message` and `NumTimes` value to the `ViewBag` object. `ViewBag` is a dynamic object, which means you can put whatever you want in to it; the `ViewBag` object has no defined properties until you put something inside it. The ASP.NET 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:

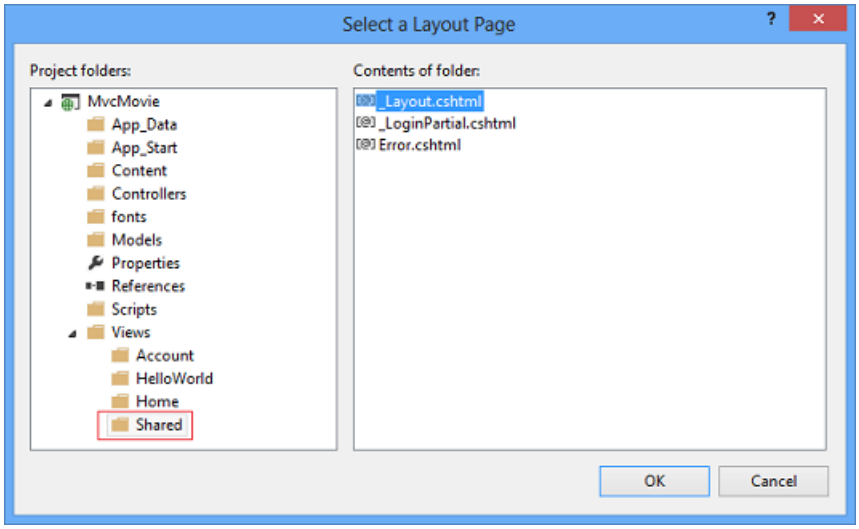
```
using System.Web;
using System.Web.Mvc;

namespace MvcMovie.Controllers
{
    public class HelloWorldController : Controller
    {
        public ActionResult Index()
        {
            return View();
        }

        public ActionResult Welcome(string name, int numTimes = 1)
        {
            ViewBag.Message = "Hello " + name;
            ViewBag.NumTimes = numTimes;

            return View();
        }
    }
}
```

Now the `ViewBag` object contains data that will be passed to the view automatically. Next, you need a Welcome view template! In the **Build** menu, select **Build Solution** (or Ctrl+Shift+B) to make sure the project is compiled. Right click the *Views\HelloWorld* folder and click **Add**, then click **MVC 5 View Page with Layout (Razor)**.



In the **Specify Name for Item** dialog box, enter *Welcome*, and then click **OK**.

In the **Select a Layout Page** dialog, accept the default **_Layout.cshtml** and click **OK**.



The *MvcMovie\Views\HelloWorld\Welcome.cshtml* file is created.

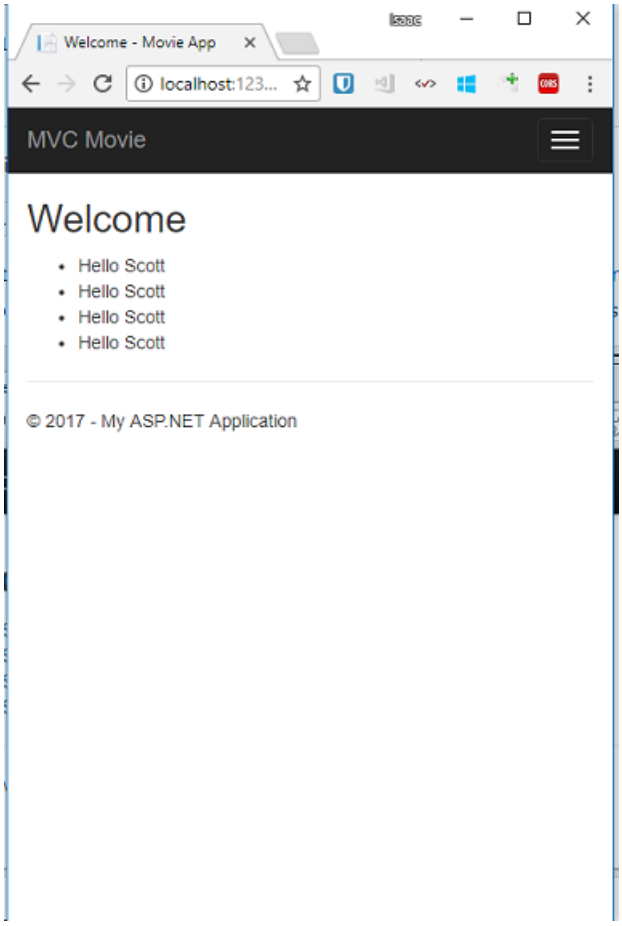
Replace the markup in the *Welcome.cshtml* file. You'll create a loop that says "Hello" as many times as the user says it should. The complete *Welcome.cshtml* file is shown below.

```
@{
    ViewBag.Title = "Welcome";
}

<h2>Welcome</h2>

<ul>
    @for (int i = 0; i < ViewBag.NumTimes; i++)
    {
        <li>@ViewBag.Message</li>
    }
</ul>
```

Run the application and browse to the following URL:

`http://localhost:xx/HelloWorld/Welcome?name=Scott&numtimes=4`

Now data is taken from the URL and passed to the controller using the model binder. The controller packages the data into a `ViewBag` object and passes that object to the view. The view then displays the data as HTML to the user.



In the sample above, we used a `ViewBag` object 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 view bag approach. See the blog entry Dynamic V Strongly Typed Views for more information.

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

PREVIOUS NEXT

Adding a Model

2/4/2018 • 2 min to read • Edit Online

by Rick Anderson

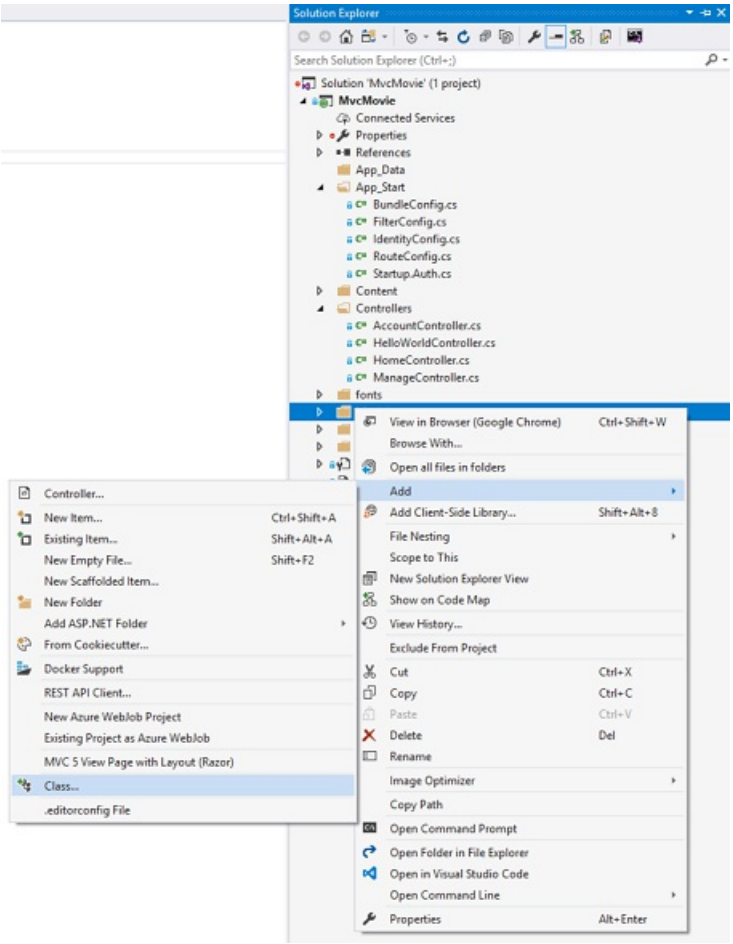
> **NOTE**
>
> This document is part of the Getting Started with ASP.NET MVC 5 tutorial. Final Source for tutorial located on GitHub

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

You'll use a .NET Framework data-access technology known as the Entity Framework to define and work with these model classes. The Entity Framework (often referred to as EF) supports a development paradigm called *Code First*. Code First allows you to create model objects by writing simple classes. (These are also known as POCO classes, from "plain-old CLR objects.") You can then have the database created on the fly from your classes, which enables a very clean and rapid development workflow. If you are required to create the database first, you can still follow this tutorial to learn about MVC and EF app development. You can then follow Tom Fizmakens ASP.NET Scaffolding tutorial, which covers the database first approach.

Adding Model Classes

In **Solution Explorer**, right click the *Models* folder, select **Add**, and then select **Class**.



Enter the *class* name "Movie".

Add the following five properties to the `Movie` class:

```
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; }
    }
}
```

We'll use the `Movie` class to represent movies in a database. Each instance of a `Movie` object will correspond to a row within a database table, and each property of the `Movie` class will map to a column in the table.

Note: In order to use System.Data.Entity, and the related class, you need to install the Entity Framework NuGet Package. Follow the link for further instructions.

In the same file, add the following `MovieDBContext` class:

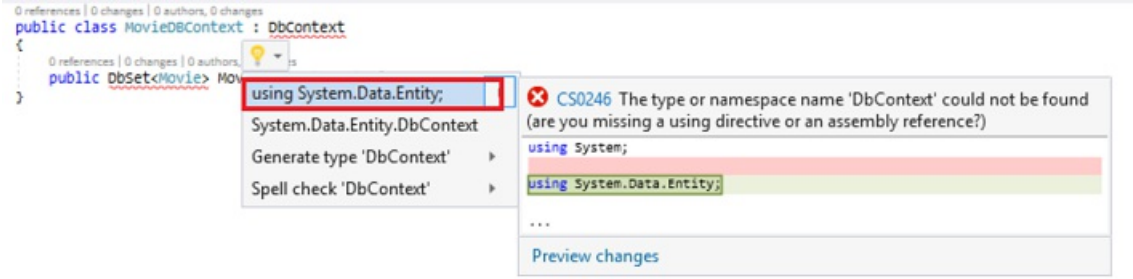
```
using System;
using System.Data.Entity;

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; }
    }

    public class MovieDBContext : DbContext
    {
        public DbSet<Movie> Movies { get; set; }
    }
}
```

The `MovieDBContext` class represents the Entity Framework movie database context, which handles fetching, storing, and updating `Movie` class instances in a database. The `MovieDBContext` derives from the `DbContext` base class provided by the Entity Framework.

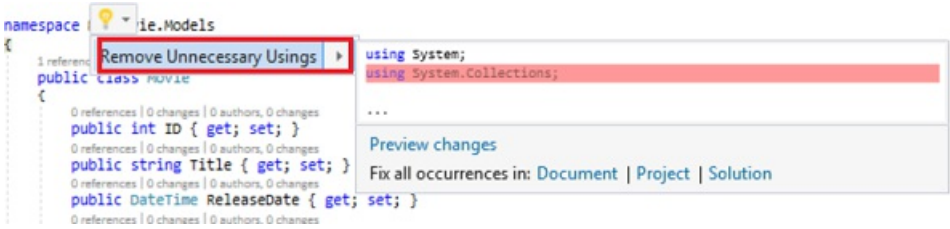
In order to be able to reference `DbContext` and `DbSet`, you need to add the following `using` statement at the top of the file:

```
using System.Data.Entity;
```

You can do this by manually adding the using statement, or you can hover over the red squiggly lines, click `Show potential fixes` and click `using System.Data.Entity;`



Note: Several unused `using` statements have been removed. Visual Studio will show unused dependencies as gray. You can remove unused dependencies by hovering over the gray dependencies, click `Show potential fixes` and click **Remove Unused Usings.**



We've finally added a model (the M in MVC). In the next section you'll work with the database connection string.

PREVIOUS NEXT

Creating a Connection String and Working with SQL Server LocalDB

1/24/2018 • 3 min to read • Edit Online

by Rick Anderson

> **NOTE**
>
> This document is part of the Getting Started with ASP.NET MVC 5 tutorial. Final Source for tutorial located on GitHub

Creating a Connection String and Working with SQL Server LocalDB

The `MovieDBContext` class you created handles the task of connecting to the database and mapping `Movie` objects to database records. One question you might ask, though, is how to specify which database it will connect to. You don't actually have to specify which database to use, Entity Framework will default to using LocalDB. In this section we'll explicitly add a connection string in the *Web.config* file of the application.

SQL Server Express LocalDB

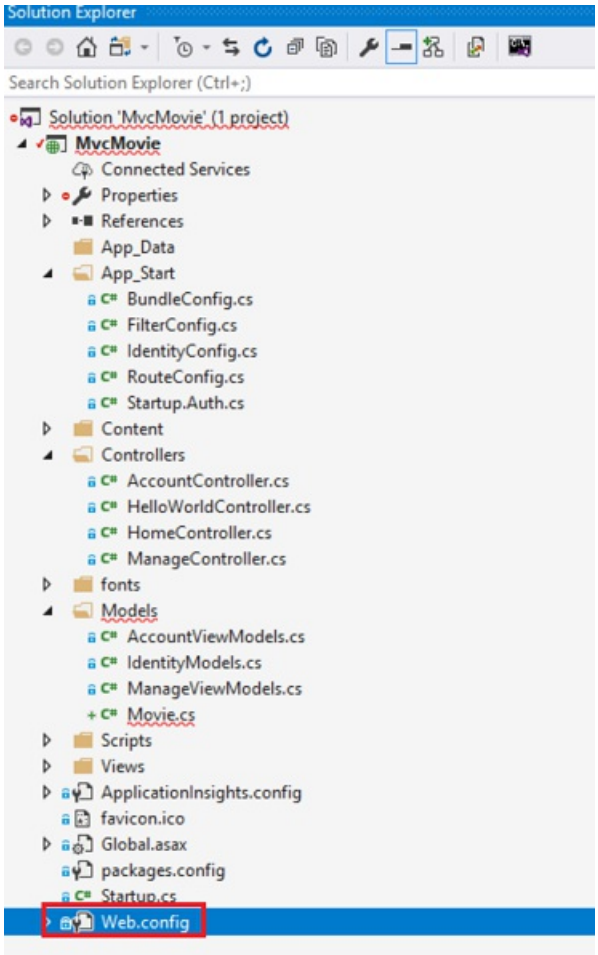
LocalDB is a lightweight version of the SQL Server Express Database Engine that starts on demand and runs in user mode. LocalDB runs in a special execution mode of SQL Server Express that enables you to work with databases as *.mdf* files. Typically, LocalDB database files are kept in the *App_Data* folder of a web project.

SQL Server Express is not recommended for use in production web applications. LocalDB in particular should not be used for production with a web application because it is not designed to work with IIS. However, a LocalDB database can be easily migrated to SQL Server or SQL Azure.

In Visual Studio 2017, LocalDB is installed by default with Visual Studio.

By default, the Entity Framework looks for a connection string named the same as the object context class (`MovieDBContext` for this project). For more information see SQL Server Connection Strings for ASP.NET Web Applications.

Open the application root *Web.config* file shown below. (Not the *Web.config* file in the *Views* folder.)



Find the `<connectionStrings>` element:

```
<?xml version="1.0" encoding="utf-8"?>
<!--
  For more information on how to configure your ASP.NET application, please visit
  https://go.microsoft.com/fwlink/?LinkId=301880
  -->
<configuration>
  <configSections>
    <!-- For more information on Entity Framework configuration, visit http://go.microsoft.com/fwlink/?LinkID=237468 --
    <section name="entityFramework" type="System.Data.Entity.Internal.ConfigFile.EntityFrameworkSection, EntityFramewor
  </configSections>
  <connectionStrings>
    <add name="DefaultConnection" connectionString="Data Source=(LocalDb)\MSSQLLocalDB;Initial Catalog=aspnet-MvcMovie-
  </connectionStrings>
  <appSettings>
    <add key="webpages:Version" value="3.0.0.0" />
    <add key="webpages:Enabled" value="false" />
    <add key="ClientValidationEnabled" value="true" />
    <add key="UnobtrusiveJavaScriptEnabled" value="true" />
  </appSettings>
  <system.web>
    <authentication mode="None" />
    <compilation debug="true" targetFramework="4.6.1" />
    <httpRuntime targetFramework="4.6.1" />
  </system.web>
```

Add the following connection string to the `<connectionStrings>` element in the *Web.config* file.

```
<add name="MovieDBContext"
   connectionString="Data Source=(LocalDb)\MSSQLLocalDB;Initial Catalog=aspnet-MvcMovie;Integrated
Security=SSPI;AttachDBFilename=|DataDirectory|\Movies.mdf"
   providerName="System.Data.SqlClient"
/>
```

The following example shows a portion of the *Web.config* file with the new connection string added:

```
<connectionStrings>
  <add name="DefaultConnection" connectionString="Data Source=(LocalDb)\MSSQLLocalDB;Initial Catalog=aspnet-
MvcMovie-fefdc1f0-bd81-4ce9-b712-93a062e01031;Integrated
Security=SSPI;AttachDBFilename=|DataDirectory|\aspnet-MvcMovie-fefdc1f0-bd81-4ce9-b712-93a062e01031.mdf"
providerName="System.Data.SqlClient" />
  <add name="MovieDBContext" connectionString="Data Source=(LocalDb)\MSSQLLocalDB;Initial Catalog=aspnet-
MvcMovie;Integrated Security=SSPI;AttachDBFilename=|DataDirectory|\Movies.mdf"
providerName="System.Data.SqlClient" />
</connectionStrings>
```

The two connection strings are very similar. The first connection string is named `DefaultConnection` and is used for the membership database to control who can access the application. The connection string you've added specifies a LocalDB database named *Movie.mdf* located in the *App_Data* folder. We won't use the membership database in this tutorial, for more information on membership, authentication and security, see my tutorial Create an ASP.NET MVC app with auth and SQL DB and deploy to Azure App Service.

The name of the connection string must match the name of the DbContext class.

```
using System;
using System.Data.Entity;

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; }
    }

    public class MovieDBContext : DbContext
    {
        public DbSet<Movie> Movies { get; set; }
    }
}
```

You don't actually need to add the `MovieDBContext` connection string. If you don't specify a connection string, Entity Framework will create a LocalDB database in the users directory with the fully qualified name of the DbContext class (in this case `MvcMovie.Models.MovieDBContext`). You can name the database anything you like, as long as it has the *.MDF* suffix. For example, we could name the database *MyFilms.mdf*.

Next, you'll build a new `MoviesController` class that you can use to display the movie data and allow users to create new movie listings.

PREVIOUS NEXT

Accessing Your Model's Data from a Controller

1/24/2018 • 7 min to read • Edit Online

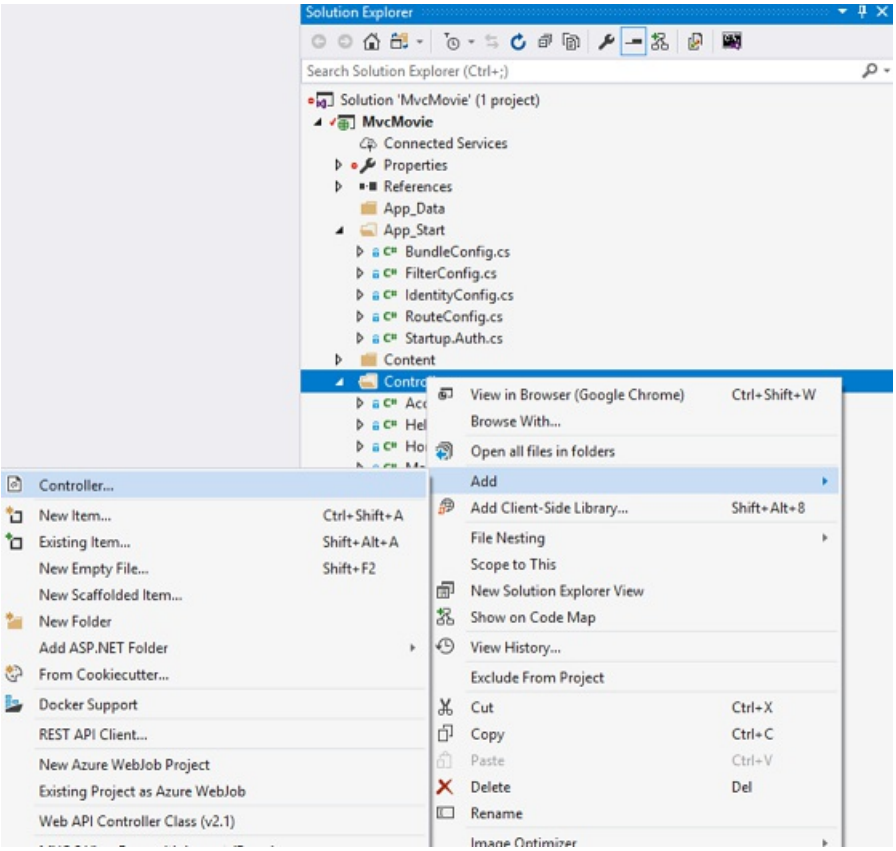
by Rick Anderson

> **NOTE**
>
> This document is part of the Getting Started with ASP.NET MVC 5 tutorial. Final Source for tutorial located on GitHub

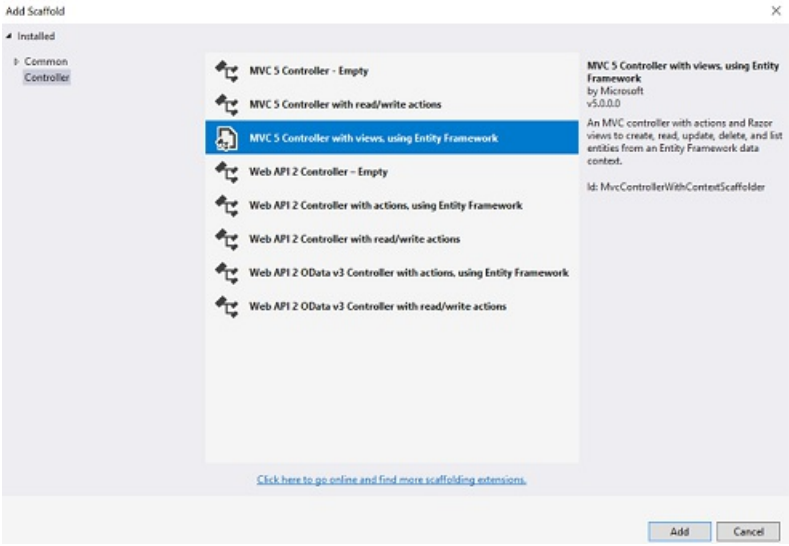
In this section, you'll create a new `MoviesController` class and write code that retrieves the movie data and displays it in the browser using a view template.

Build the application before going on to the next step. If you don't build the application, you'll get an error adding a controller.

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

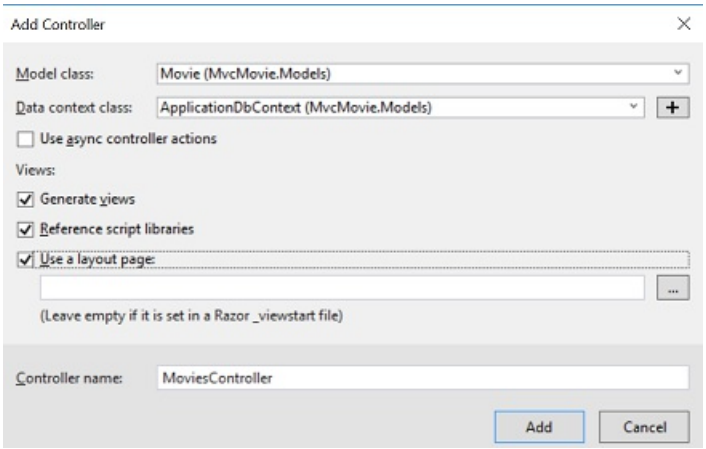


In the **Add Scaffold** dialog box, click **MVC 5 Controller with views, using Entity Framework**, and then click **Add**.



- Select **Movie (MvcMovie.Models)** for the Model class.
- Select **MovieDBContext (MvcMovie.Models)** for the Data context class.
- For the Controller name enter **MoviesController**.

  The image below shows the completed dialog.

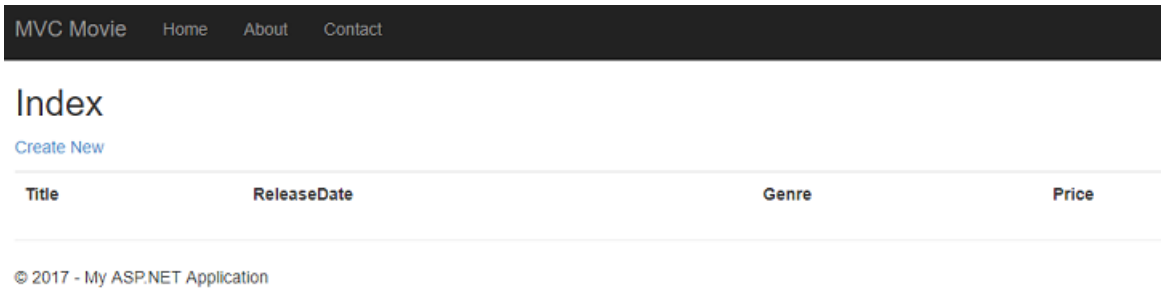


Click **Add**. (If you get an error, you probably didn't build the application before starting adding the controller.) Visual Studio creates the following files and folders:

- *A MoviesController.cs* file in the *Controllers* folder.
- A *Views\Movies* folder.
- *Create.cshtml, Delete.cshtml, Details.cshtml, Edit.cshtml*, and *Index.cshtml* in the new *Views\Movies* folder.

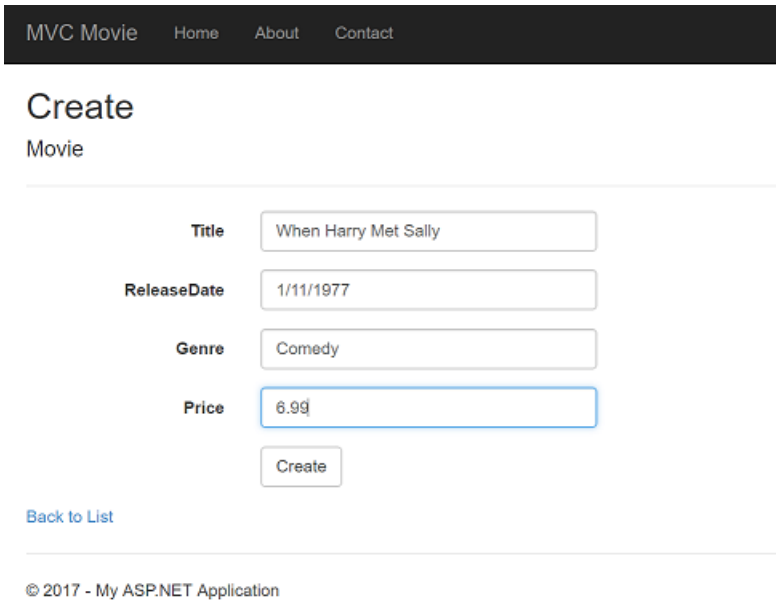
Visual Studio automatically created the CRUD (create, read, update, and delete) action methods and views for you (the automatic creation of CRUD action methods and views is known as scaffolding). You now have a fully functional web application that lets you create, list, edit, and delete movie entries.

Run the application and click on the **MVC Movie** link (or browse to the `Movies` controller by appending */Movies* to the URL in the address bar of your browser). Because the application is relying on the default routing (defined in the *App_Start\RouteConfig.cs* file), the browser request `http://localhost:xxxxx/Movies` is routed to the default `Index` action method of the `Movies` controller. In other words, the browser request `http://localhost:xxxxx/Movies` is effectively the same as the browser request `http://localhost:xxxxx/Movies/Index`. The result is an empty list of movies, because you haven't added any yet.



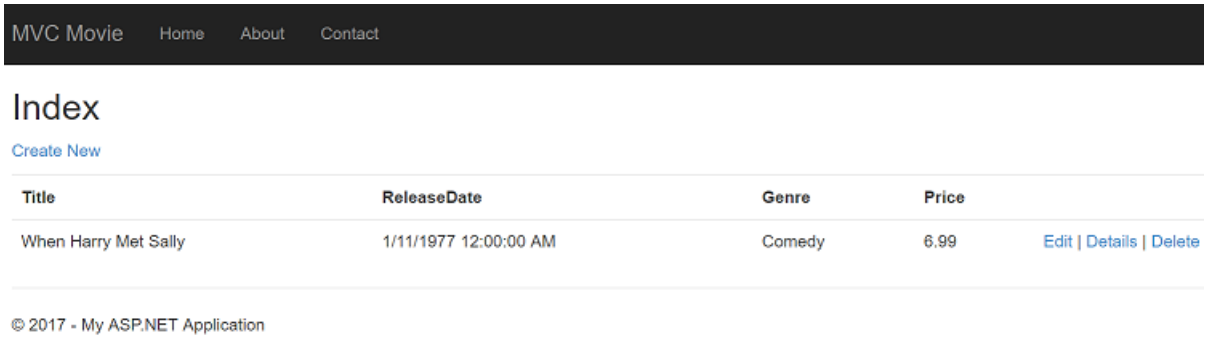
Creating a Movie

Select the **Create New** link. Enter some details about a movie and then click the **Create** button.



> **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 include *globalize.js* and your specific *cultures/globalize.cultures.js* file(from https://github.com/jquery/globalize) and JavaScript to use `Globalize.parseFloat`. I'll show how to do this in the next tutorial. For now, just enter whole numbers like 10.

Clicking the **Create** button causes the form to be posted to the server, where the movie information is saved in the database. You're 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 MovieDBContext db = new MovieDBContext();

    // GET: /Movies/
    public ActionResult Index()
    {
        return View(db.Movies.ToList());
    }
```

A request to the `Movies` controller returns all the entries in the `Movies` table and then passes the results to the `Index` view. The following line from the `MoviesController` class instantiates a movie database context, as described previously. You can use the movie database context to query, edit, and delete movies.

```
private MovieDBContext db = new MovieDBContext();
```

Strongly Typed Models and the @model Keyword

Earlier in this tutorial, you saw how a controller can pass data or objects to a view template using the `ViewBag` object. The `ViewBag` 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 template. This strongly typed approach enables better compile-time checking of your code and richer IntelliSense in the Visual Studio editor. The scaffolding mechanism in Visual Studio used this approach (that is, passing a *strongly* typed model) with the `MoviesController` class and view templates when it created the methods and views.

In the *Controllers\MoviesController.cs* file examine the generated `Details` method. The `Details` method is shown below.

```
public ActionResult Details(int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    Movie movie = db.Movies.Find(id);
    if (movie == null)
    {
        return HttpNotFound();
    }
    return View(movie);
}
```

The `id` parameter is generally passed as route data, for example `http://localhost:1234/movies/details/1` will set the controller to the movie controller, the action to `details` and the `id` to 1. 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

@{
    ViewBag.Title = "Details";
}

<h2>Details</h2>

<div>
    <h4>Movie</h4>
<hr />
    <dl class="dl-horizontal">
        <dt>
            @Html.DisplayNameFor(model => model.Title)
        </dt>
         @*Markup omitted for clarity.*@
    </dl>
</div>
<p>
    @Html.ActionLink("Edit", "Edit", new { id = Model.ID }) |
    @Html.ActionLink("Back to List", "Index")
</p>
```

By including a `@model` statement at the top of the view template 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* template, 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 view templates also pass a movie model object.

Examine the *Index.cshtml* view template and the `Index` method in the *MoviesController.cs* file. Notice how the code creates a `List` object when it calls the `View` helper method in the `Index` action method. The code then passes this `Movies` list from the `Index` action method to the view:

```
public ActionResult Index()
{
    return View(db.Movies.ToList());
}
```

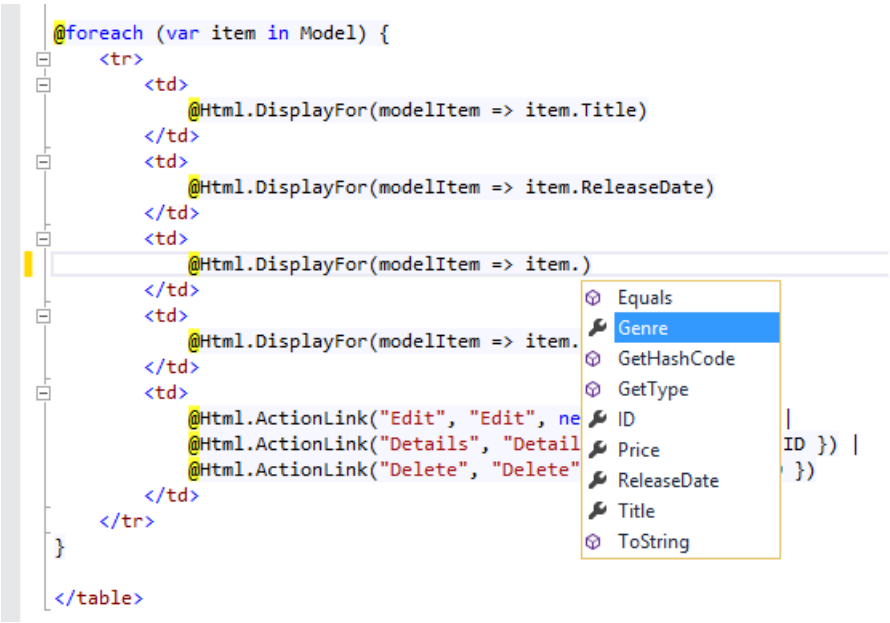
When you created the movie controller, Visual Studio automatically included the following `@model` statement at the top of the *Index.cshtml* file:

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

This `@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* template, the code loops through the movies by doing a `foreach` statement over the strongly typed `Model` object:

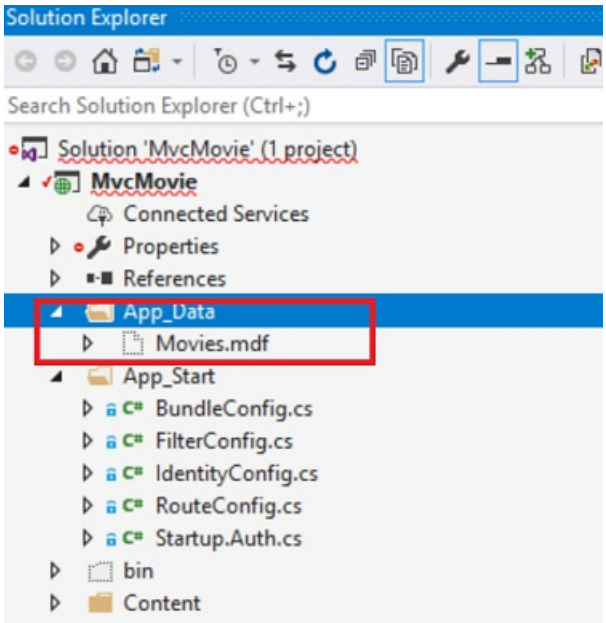
```
@foreach (var item in Model) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.Title)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.ReleaseDate)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Genre)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Price)
        </td>
         <th>
            @Html.DisplayFor(modelItem => item.Rating)
        </th>
        <td>
            @Html.ActionLink("Edit", "Edit", new { id=item.ID }) |
            @Html.ActionLink("Details", "Details", new { id=item.ID })  |
            @Html.ActionLink("Delete", "Delete", new { id=item.ID })
        </td>
    </tr>
}
```

Because the `Model` object is strongly typed (as an `IEnumerable<Movie>` object), each `item` object 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:

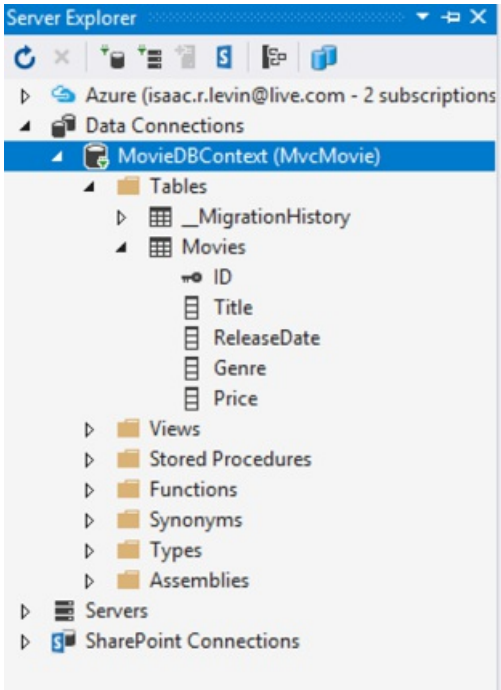


Working with SQL Server LocalDB

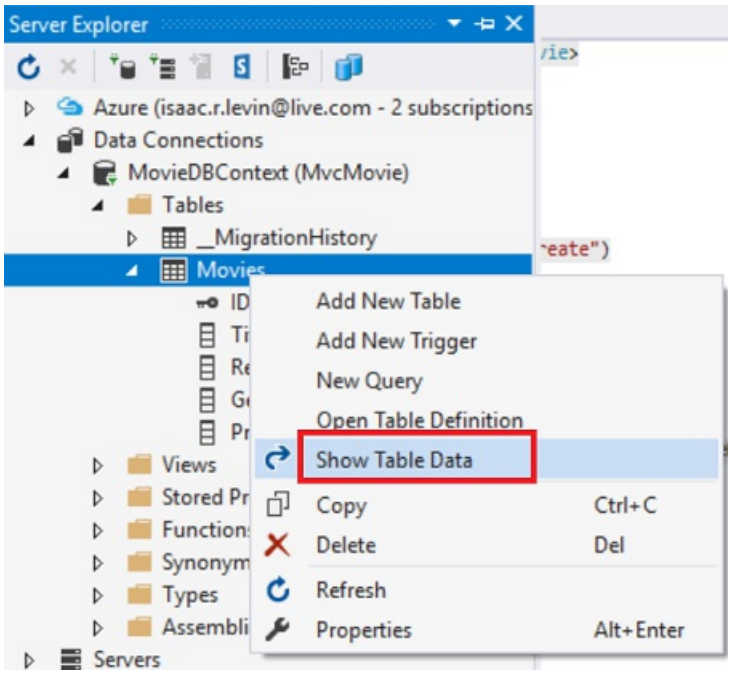
Entity Framework Code First detected that the database connection string that was provided pointed to a `Movies` database that didn't exist yet, so Code First created the database automatically. You can verify that it's been created by looking in the *App_Data* folder. If you don't see the *Movies.mdf* file, click the **Show All Files** button in the **Solution Explorer** toolbar, click the **Refresh** button, and then expand the *App_Data* folder.

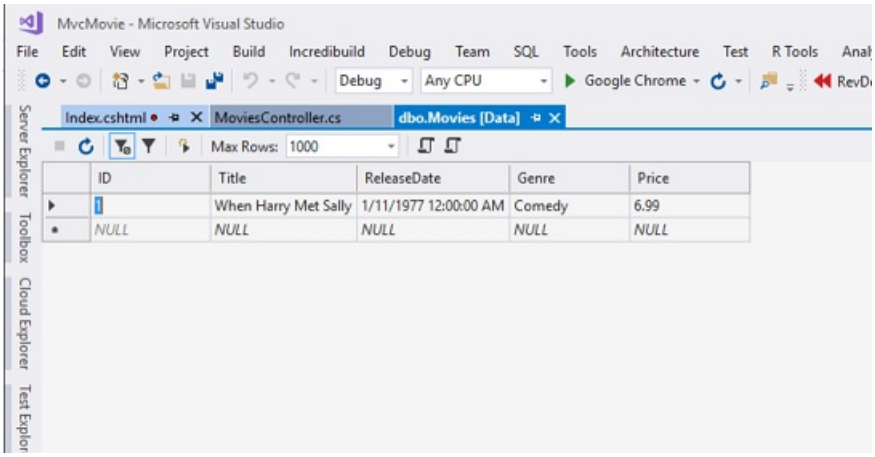


Double-click *Movies.mdf* to open **SERVER EXPLORER**, then expand the **Tables** folder to see the Movies table. Note the key icon next to ID. By default, EF will make a property named ID the primary key. For more information on EF and MVC, see Tom Dykstra's excellent tutorial on MVC and EF.

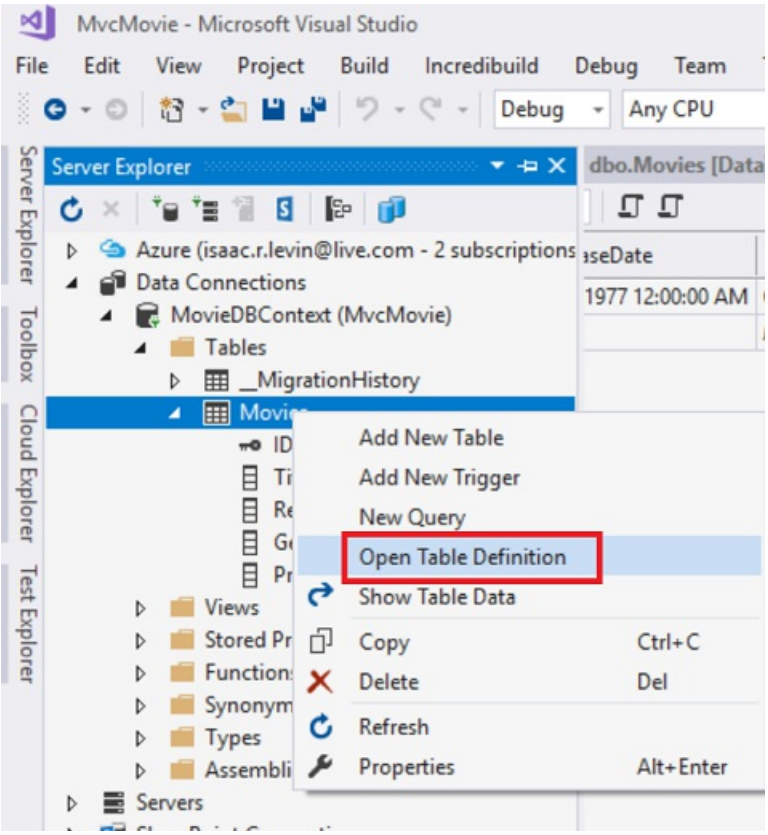


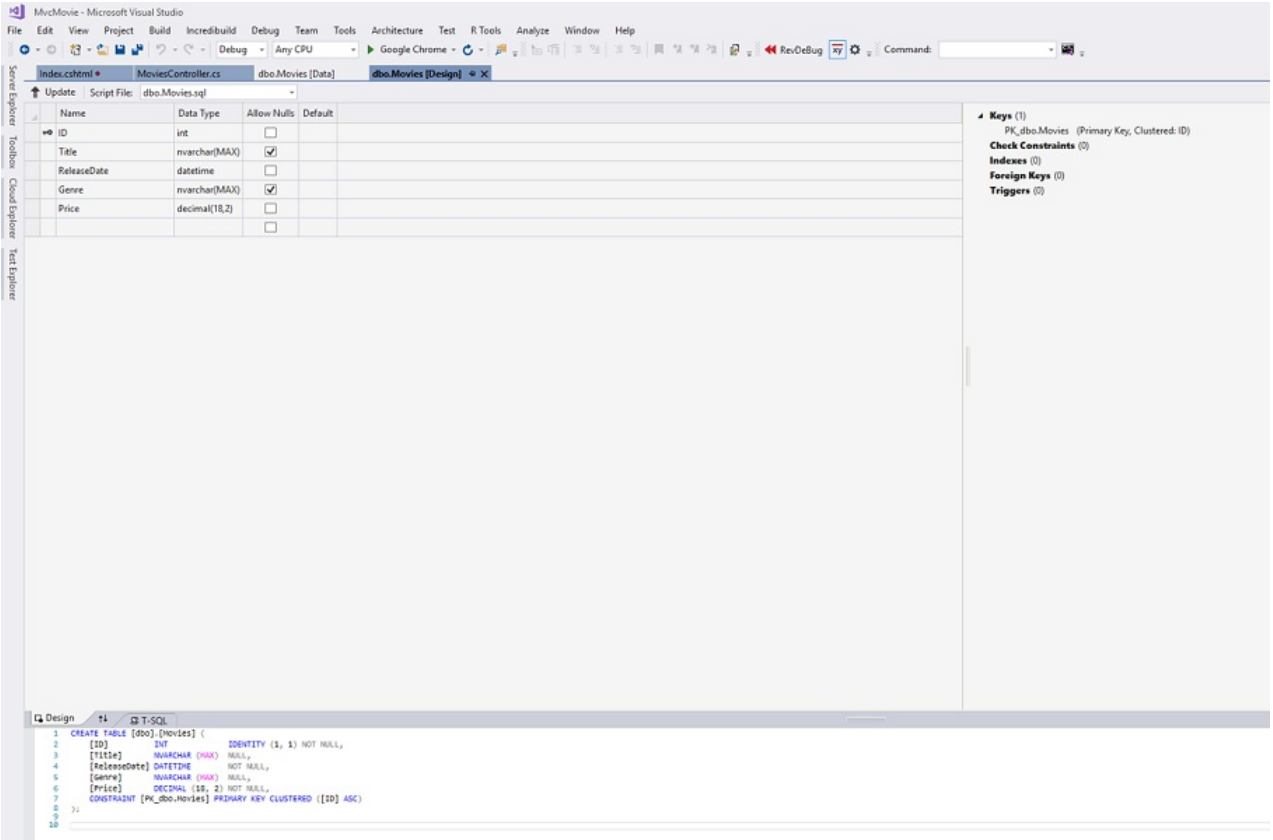
Right-click the `Movies` table and select **Show Table Data** to see the data you created.





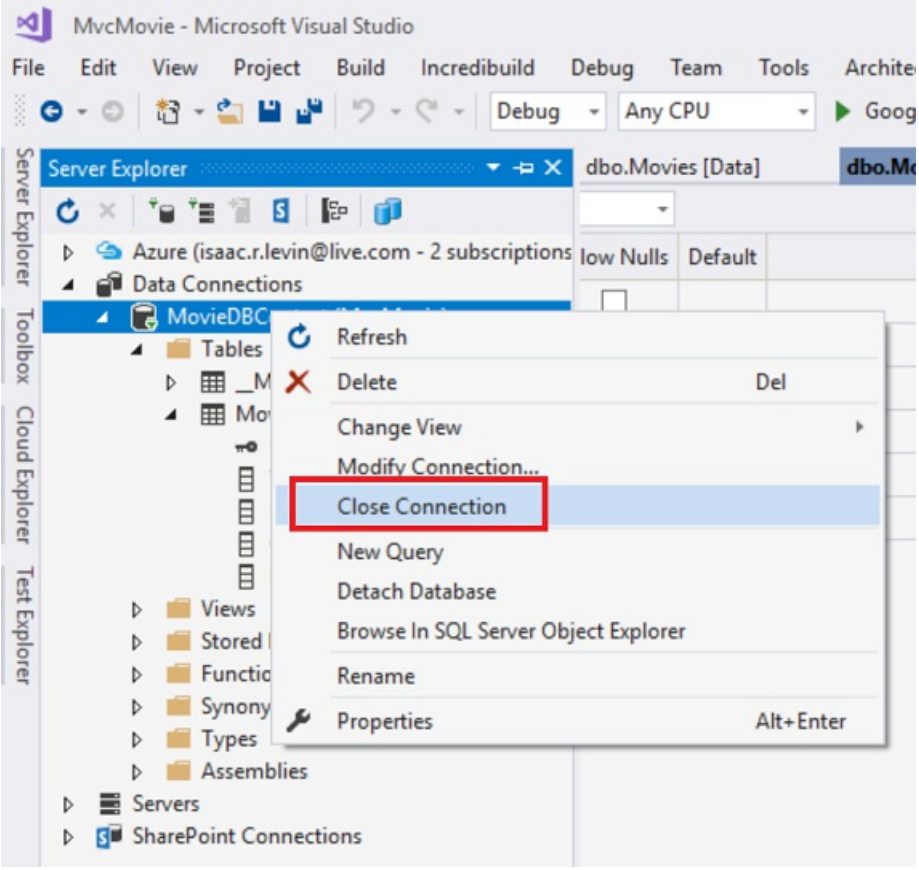
Right-click the `Movies` table and select **Open Table Definition** to see the table structure that Entity Framework Code First created for you.





Notice how the schema of the `Movies` table maps to the `Movie` class you created earlier. Entity Framework Code First automatically created this schema for you based on your `Movie` class.

When you're finished, close the connection by right clicking *MovieDBContext* and selecting **Close Connection**. (If you don't close the connection, you might get an error the next time you run the project).



You now have a database and pages to display, edit, update and delete data. In the next tutorial, we'll examine the rest of the scaffolded code and add a `SearchIndex` method and a `SearchIndex` view that lets you search for movies in this database. For more information on using Entity Framework with MVC, see Creating an Entity Framework Data Model for an ASP.NET MVC Application.

PREVIOUS NEXT

Examining the Edit Methods and Edit View

1/24/2018 • 10 min to read • Edit Online

by Rick Anderson

> **NOTE**
>
> This document is part of the Getting Started with ASP.NET MVC 5 tutorial. Final Source for tutorial located on GitHub

In this section, you'll examine the generated `Edit` action methods and views for the movie controller. But first will take a short diversion to make the release date look better. Open the *Models\Movie.cs* file and add the highlighted lines shown below:

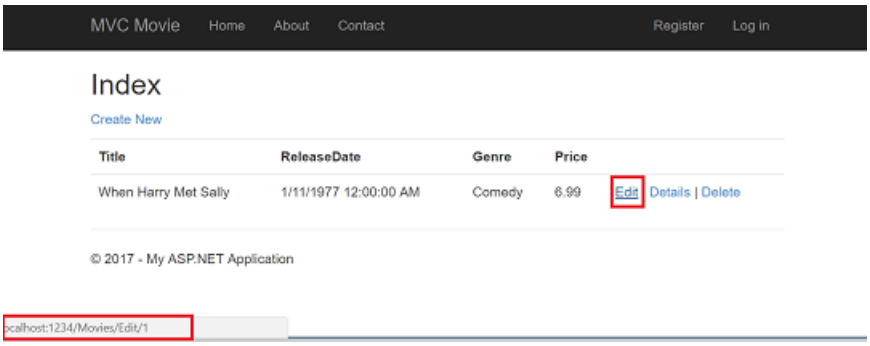
```
using System;
using System.ComponentModel.DataAnnotations;
using System.Data.Entity;

namespace MvcMovie.Models
{
    public class Movie
    {
        public int ID { get; set; }
        public string Title { get; set; }

        [Display(Name = "Release Date")]
        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
        public DateTime ReleaseDate { get; set; }
        public string Genre { get; set; }
        public decimal Price { get; set; }
    }

    public class MovieDBContext : DbContext
    {
        public DbSet<Movie> Movies { get; set; }
    }
}
```

You can also make the date culture specific like this:

```
[Display(Name = "Release Date")]
[DataType(DataType.Date)]
[DisplayFormat(DataFormatString = "{0:d}", ApplyFormatInEditMode = true)]
public DateTime ReleaseDate { get; set; }
```

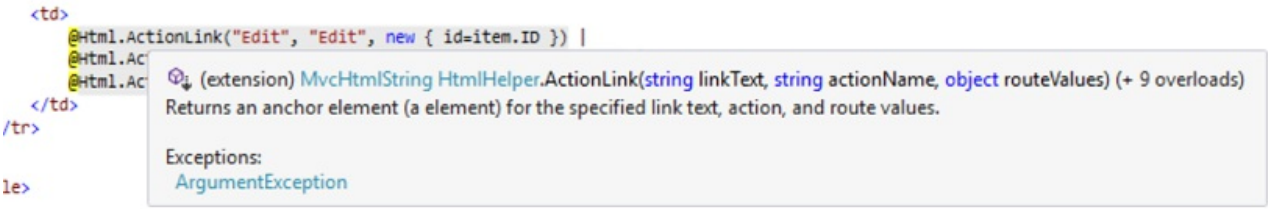
We'll cover DataAnnotations in the next tutorial. The Display attribute specifies what to display for the name of a field (in this case "Release Date" instead of "ReleaseDate"). The DataType attribute specifies the type of the data, in this case it's a date, so the time information stored in the field is not displayed. The DisplayFormat attribute is needed for a bug in the Chrome browser that renders date formats incorrectly.

Run the application and browse to the `Movies` controller. Hold the mouse pointer over an **Edit** link to see the URL that it links to.



The **Edit** link was generated by the `Html.ActionLink` method in the *Views\Movies\Index.cshtml* view:

```
@Html.ActionLink("Edit", "Edit", new { id=item.ID })
```

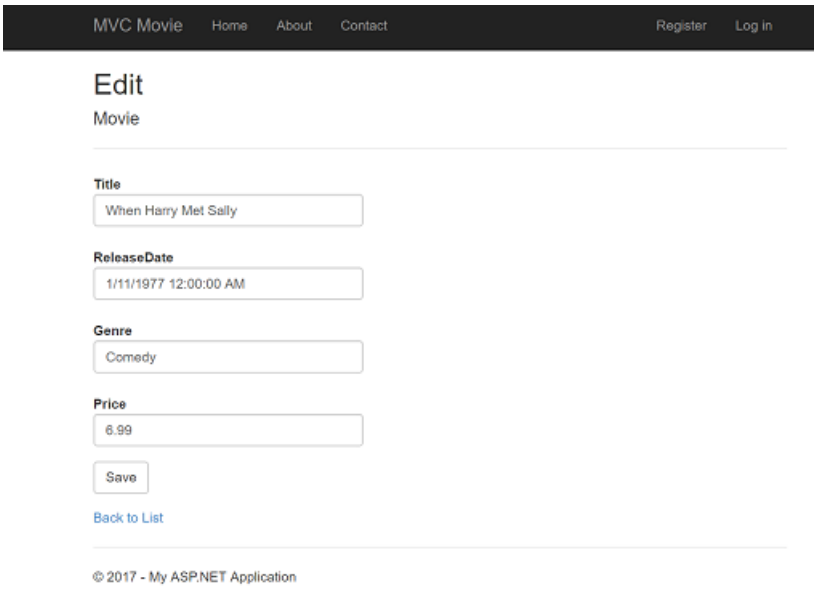


The `Html` object is a helper that's exposed using a property on the System.Web.Mvc.WebViewPage base class. The `ActionLink` method of the helper makes it easy to dynamically generate HTML hyperlinks that link to action methods on controllers. The first argument to the `ActionLink` method is the link text to render (for example, `<a>Edit Me</a>`). The second argument is the name of the action method to invoke (In this case, the `Edit` action). The final argument is an anonymous object that generates the route data (in this case, the ID of 4).

The generated link shown in the previous image is `http://localhost:1234/Movies/Edit/4`. The default route (established in *App_Start\RouteConfig.cs*) takes the URL pattern `{controller}/{action}/{id}`. Therefore, ASP.NET translates `http://localhost:1234/Movies/Edit/4` into a request to the `Edit` action method of the `Movies` controller with the parameter `ID` equal to 4. Examine the following code from the *App_Start\RouteConfig.cs* file. The MapRoute method is used to route HTTP requests to the correct controller and action method and supply the optional ID parameter. The MapRoute method is also used by the HtmlHelpers such as `ActionLink` to generate URLs given the controller, action method and any route data.

```
public static void RegisterRoutes(RouteCollection routes)
{
    routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

    routes.MapRoute(
        name: "Default",
        url: "{controller}/{action}/{id}",
        defaults: new { controller = "Home", action = "Index",
            id = UrlParameter.Optional }
    );
}
```

You can also pass action method parameters using a query string. For example, the URL `http://localhost:1234/Movies/Edit?ID=3` also passes the parameter `ID` of 3 to the `Edit` action method of the `Movies` controller.



Open the `Movies` controller. The two `Edit` action methods are shown below.

```
// GET: /Movies/Edit/5
public ActionResult Edit(int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    Movie movie = db.Movies.Find(id);
    if (movie == null)
    {
        return HttpNotFound();
    }
    return View(movie);
}

// POST: /Movies/Edit/5
// To protect from overposting attacks, please enable the specific properties you want to bind to, for
// more details see http://go.microsoft.com/fwlink/?LinkId=317598.
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Edit([Bind(Include="ID,Title,ReleaseDate,Genre,Price")] Movie movie)
{
    if (ModelState.IsValid)
    {
        db.Entry(movie).State = EntityState.Modified;
        db.SaveChanges();
        return RedirectToAction("Index");
    }
    return View(movie);
}
```

Notice the second `Edit` action method is preceded by the `HttpPost` attribute. This attribute specifies that the overload of the `Edit` method can be invoked only for POST requests. You could apply the `HttpGet` attribute to the first edit method, but that's not necessary because it's the default. (We'll refer to action methods that are implicitly assigned the `HttpGet` attribute as `HttpGet` methods.) The Bind attribute is another important security mechanism that keeps hackers from over-posting data to your model. You should only include properties in the bind attribute that you want to change. You can read about overposting and the bind attribute in my overposting security note. In the simple model used in this tutorial, we will be binding all the data in the model. The ValidateAntiForgeryToken attribute is used to prevent forgery of a request and is paired up with `@Html.AntiForgeryToken()` in the edit view file (*Views\Movies\Edit.cshtml*), a portion is shown below:

```
@model MvcMovie.Models.Movie

@{
    ViewBag.Title = "Edit";
}
<h2>Edit</h2>
@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()
    <div class="form-horizontal">
        <h4>Movie</h4>
        <hr />
        @Html.ValidationSummary(true)
        @Html.HiddenFor(model => model.ID)

        <div class="form-group">
            @Html.LabelFor(model => model.Title, new { @class = "control-label col-md-2" })
            <div class="col-md-10">
                @Html.EditorFor(model => model.Title)
                @Html.ValidationMessageFor(model => model.Title)
            </div>
        </div>
```

`@Html.AntiForgeryToken()` generates a hidden form anti-forgery token that must match in the `Edit` method of the `Movies` controller. You can read more about Cross-site request forgery (also known as XSRF or CSRF) in my tutorial XSRF/CSRF Prevention in MVC.

The `HttpGet` `Edit` method takes the movie ID parameter, looks up the movie using the Entity Framework `Find` method, and returns the selected movie to the Edit view. If a movie cannot be found, HttpNotFound is returned. 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

@{
    ViewBag.Title = "Edit";
}
<h2>Edit</h2>
@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()
    <div class="form-horizontal">
        <h4>Movie</h4>
        <hr />
        @Html.ValidationSummary(true)
        @Html.HiddenFor(model => model.ID)

        <div class="form-group">
            @Html.LabelFor(model => model.Title, new { @class = "control-label col-md-2" })
            <div class="col-md-10">
                @Html.EditorFor(model => model.Title)
                @Html.ValidationMessageFor(model => model.Title)
            </div>
        </div>
        <div class="form-group">
            @Html.LabelFor(model => model.ReleaseDate, new { @class = "control-label col-md-2" })
            <div class="col-md-10">
                @Html.EditorFor(model => model.ReleaseDate)
                @Html.ValidationMessageFor(model => model.ReleaseDate)
            </div>
        </div>
        @*Genre and Price 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>
}
<div>
    @Html.ActionLink("Back to List", "Index")
</div>
@section Scripts {
    @Scripts.Render("~/bundles/jqueryval")
}
```

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 *helper methods* to streamline the HTML markup. The `Html.LabelFor` helper displays the name of the field ("Title", "ReleaseDate", "Genre", or "Price"). The `Html.EditorFor` helper renders an HTML `<input>` element. The `Html.ValidationMessageFor` 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 HTML for the form element is shown below.

```
<form action="/movies/Edit/4" method="post">
   <input name="__RequestVerificationToken" type="hidden" value="UxY6bkQyJCXO3Kn5AXg-
6TXxOj6yVBi9tghHaQ5Lq_qwKvcojNXEEfcbn-FGh_0vuw4tS_BRk7QQQHlJp8AP4_X4orVNoQnp2cd8kXhykS01" />  <fieldset
class="form-horizontal">
      <legend>Movie</legend>

      <input data-val="true" data-val-number="The field ID must be a number." data-val-required="The ID field
is required." id="ID" name="ID" type="hidden" value="4" />

      <div class="control-group">
         <label class="control-label" for="Title">Title</label>
         <div class="controls">
            <input class="text-box single-line" id="Title" name="Title" type="text" value="GhostBusters" />
            <span class="field-validation-valid help-inline" data-valmsg-for="Title" data-valmsg-
replace="true"></span>
         </div>
      </div>

      <div class="control-group">
         <label class="control-label" for="ReleaseDate">Release Date</label>
         <div class="controls">
            <input class="text-box single-line" data-val="true" data-val-date="The field Release Date must be
a date." data-val-required="The Release Date field is required." id="ReleaseDate" name="ReleaseDate"
type="date" value="1/1/1984" />
            <span class="field-validation-valid help-inline" data-valmsg-for="ReleaseDate" data-valmsg-
replace="true"></span>
         </div>
      </div>

      <div class="control-group">
         <label class="control-label" for="Genre">Genre</label>
         <div class="controls">
            <input class="text-box single-line" id="Genre" name="Genre" type="text" value="Comedy" />
            <span class="field-validation-valid help-inline" data-valmsg-for="Genre" data-valmsg-
replace="true"></span>
         </div>
      </div>

      <div class="control-group">
         <label class="control-label" for="Price">Price</label>
         <div class="controls">
            <input class="text-box single-line" 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" type="text" value="7.99" />
            <span class="field-validation-valid help-inline" data-valmsg-for="Price" data-valmsg-
replace="true"></span>
         </div>
      </div>

      <div class="form-actions no-color">
         <input type="submit" value="Save" class="btn" />
      </div>
   </fieldset>
</form>
```

The `<input>` elements are in an HTML `<form>` element whose `action` attribute is set to post to the */Movies/Edit* URL. The form data will be posted to the server when the **Save** button is clicked. The second line shows the hidden XSRF token generated by the `@Html.AntiForgeryToken()` call.

Processing the POST Request

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

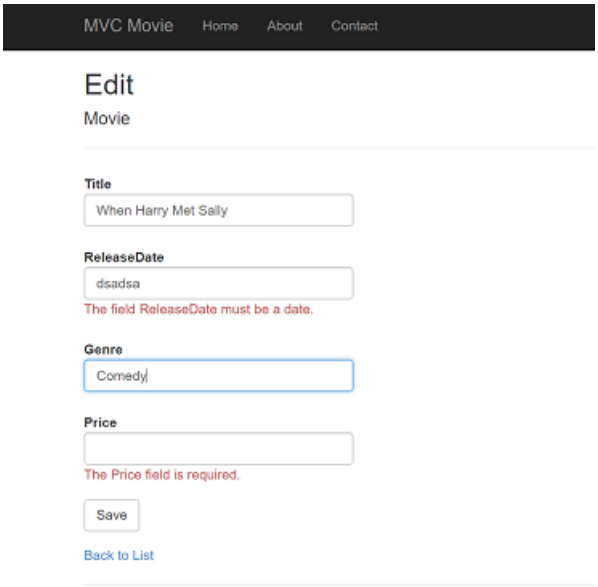
```
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Edit([Bind(Include="ID,Title,ReleaseDate,Genre,Price")] Movie movie)
{
    if (ModelState.IsValid)
    {
        db.Entry(movie).State = EntityState.Modified;
        db.SaveChanges();
        return RedirectToAction("Index");
    }
    return View(movie);
}
```

The ValidateAntiForgeryToken attribute validates the XSRF token generated by the `@Html.AntiForgeryToken()` call in the view.

The ASP.NET MVC model binder 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, the movie data is saved to the `Movies` collection of the `db(MovieDBContext` instance). The new movie data is saved to the database by calling the `SaveChanges` method of `MovieDBContext`. 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.

As soon as the client side validation determines the values of a field are not valid, an error message is displayed. If you disable JavaScript, you won't have client side validation but the server will detect the posted values are not valid, and the form values will be redisplayed with error messages. Later in the tutorial we examine validation in more detail.

The `Html.ValidationMessageFor` helpers in the *Edit.cshtml* view template take care of displaying appropriate error messages.

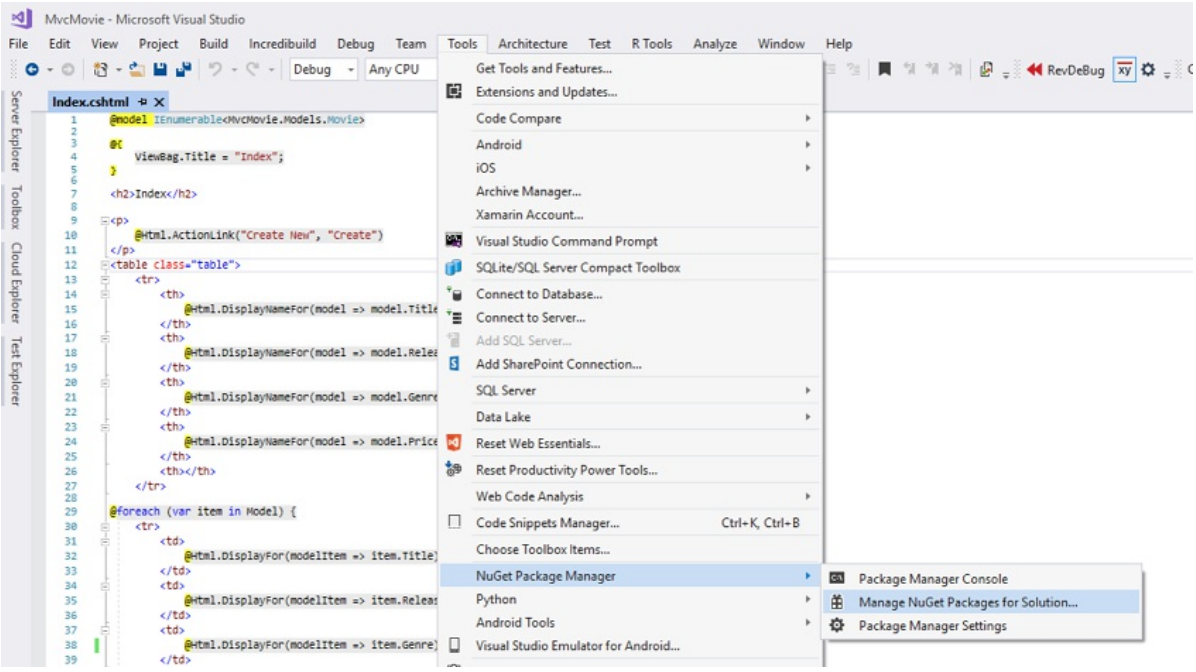


All the `HttpGet` methods follow a similar pattern. They get a movie object (or list of objects, in the case of `Index`), and pass the model to the view. The `Create` method passes an empty movie object to the Create view. All the methods that create, edit, delete, or otherwise modify data do so in the `HttpPost` overload of the method. Modifying data in an HTTP GET method is a security risk, as described in the blog post entry ASP.NET MVC Tip #46 – Don't use Delete Links because they create Security Holes. Modifying data in a 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.

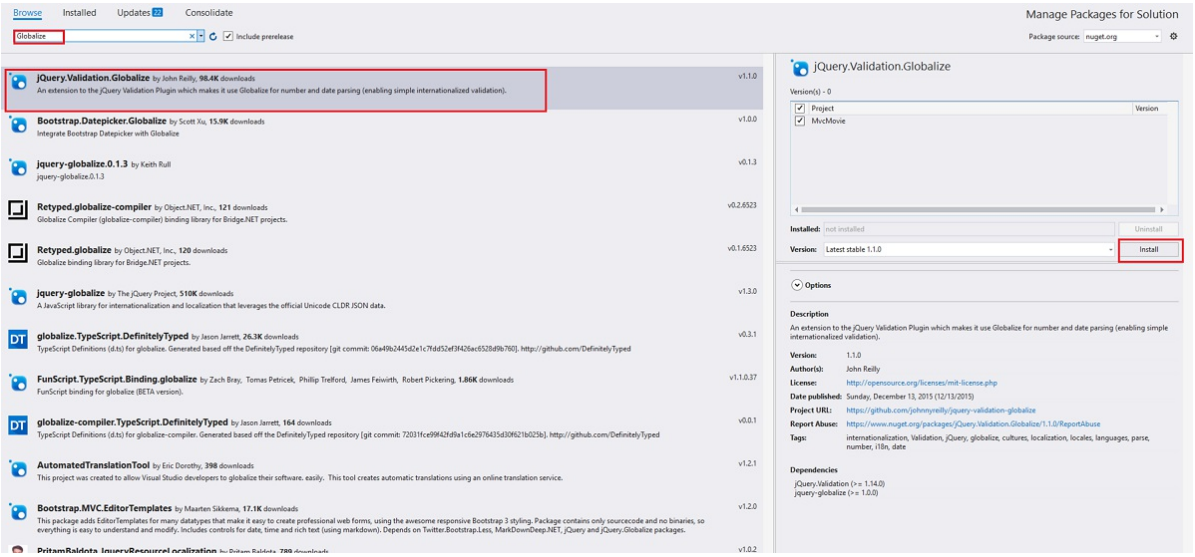
If you are using a US-English computer, you can skip this section and go to the next tutorial. You can download the Globalize version of this tutorial here. For an excellent two part tutorial on Internationalization, see Nadeem's ASP.NET MVC 5 Internationalization.

> **NOTE**
>
> to support jQuery validation for non-English locales that use a comma (",") for a decimal point, and non US-English date formats, you must include *globalize.js* and your specific *cultures/globalize.cultures.js* file(from https://github.com/jquery/globalize) and JavaScript to use `Globalize.parseFloat`. You can get the jQuery non-English validation from NuGet. (Don't install Globalize if you are using a English locale.)

1. From the **Tools** menu click **NuGetLibrary Package Manager**, and then click **Manage NuGet Packages for Solution**.



2. On the left pane, select **Browse*.***(See the image below.)
3. In the input box, enter *Globalize**.



Choose `jQuery.Validation.Globalize`, choose `MvcMovie` and click **Install**. The *Scripts\jquery.globalize\globalize.js* file will be added to your project. The *Scripts\jquery.globalize\cultures* folder will contain many culture JavaScript files. Note, it may take five minutes to install this package.

The following code shows the modifications to the Views\Movies\Edit.cshtml file:

```
@section Scripts {
    @Scripts.Render("~/bundles/jqueryval")

<script src="~/Scripts/globalize/globalize.js"></script>
<script
src="~/Scripts/globalize/cultures/globalize.culture.@(System.Threading.Thread.CurrentThread.CurrentCulture.Nam
e).js"></script>
<script>
    $.validator.methods.number = function (value, element) {
        return this.optional(element) ||
            !isNaN(Globalize.parseFloat(value));
    }
    $(document).ready(function () {
        Globalize.culture('@(System.Threading.Thread.CurrentThread.CurrentCulture.Name)');
    });
</script>
<script>
    jQuery.extend(jQuery.validator.methods, {
        range: function (value, element, param) {
            //Use the Globalization plugin to parse the value
            var val = Globalize.parseFloat(value);
            return this.optional(element) || (
                val >= param[0] && val <= param[1]);
        }
    });
    $.validator.methods.date = function (value, element) {
        return this.optional(element) ||
            Globalize.parseDate(value) ||
            Globalize.parseDate(value, "yyyy-MM-dd");
    }
</script>
}
```

To avoid repeating this code in every Edit view, you can move it to the layout file. To optimize the script download, see my tutorial Bundling and Minification.

For more information see ASP.NET MVC 3 Internationalization and ASP.NET MVC 3 Internationalization - Part 2 (NerdDinner).

As a temporary fix, if you can't get validation working in your locale, you can force your computer to use US English or you can disable JavaScript in your browser. To force your computer to use US English, you can add the globalization element to the projects root *web.config* file. The following code shows the globalization element with the culture set to United States English.

```
<system.web>
    <globalization culture ="en-US" />
    <!--elements removed for clarity-->
  </system.web>
```

In the next tutorial, we'll implement search functionality.

PREVIOUS NEXT

Search

1/24/2018 • 8 min to read • Edit Online

by Rick Anderson

> **NOTE**
> This document is part of the Getting Started with ASP.NET MVC 5 tutorial. Final Source for tutorial located on GitHub

Adding a Search Method and Search View

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

Updating the Index Form

Start by updating the `Index` action method to the existing `MoviesController` class. Here's the code:

```
public ActionResult Index(string searchString)
{
    var movies = from m in db.Movies
                 select m;

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

    return View(movies);
}
```

The first line of the `Index` method creates the following LINQ query to select the movies:

```
var movies = from m in db.Movies
                 select m;
```

The query is defined at this point, but hasn't yet 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));
}
```

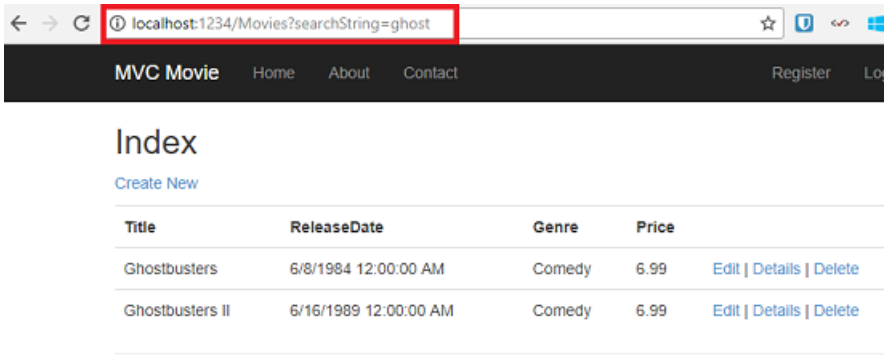
The `s => s.Title` code above is a Lambda Expression. Lambdas are used in method-based LINQ queries as arguments to standard query operator methods such as the Where method used in the above code. LINQ queries are not executed when they are defined or when they are modified by calling a method such as `Where` 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 `ToList` method is called. In the `Search` sample, the query is executed in the

Index.cshtml view. 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.

Now you can update the `Index` view that will display the form to the user.

Run the application and navigate to */Movies/Index*. Append a query string such as `?searchString=ghost` to the URL. The filtered movies are displayed.



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

```
{controller}/{action}/{id}
```

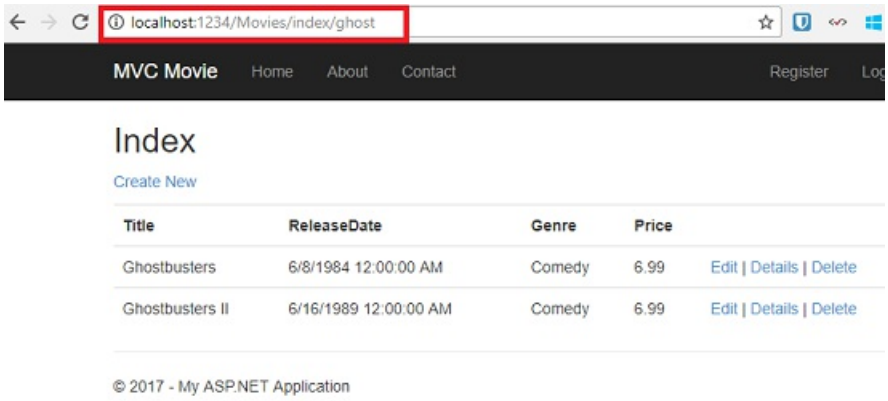
The original `Index` method looks like this::

```
public ActionResult Index(string searchString)
{
    var movies = from m in db.Movies
                 select m;

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

    return View(movies);
}
```

The modified `Index` method would look as follows:

```
public ActionResult Index(string id)
{
    string searchString = id;
    var movies = from m in db.Movies
                 select m;

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

    return View(movies);
}
```

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 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 your `Index` method takes a string parameter named `searchString`:

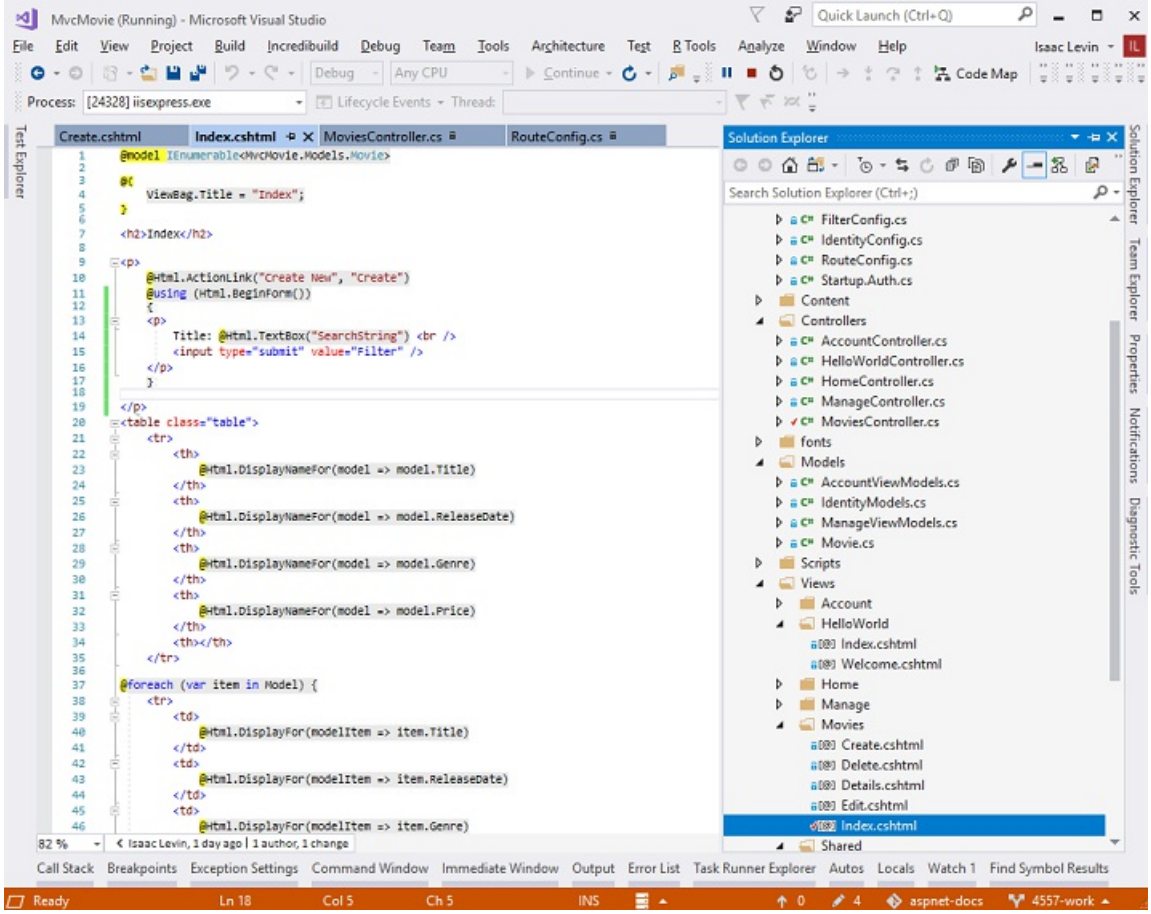
```
public ActionResult Index(string searchString)
{
 var movies = from m in db.Movies
                 select m;

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

    return View(movies);
}
```

Open the *Views\Movies\Index.cshtml* file, and just after `@Html.ActionLink("Create New", "Create")`, add the form markup highlighted below:

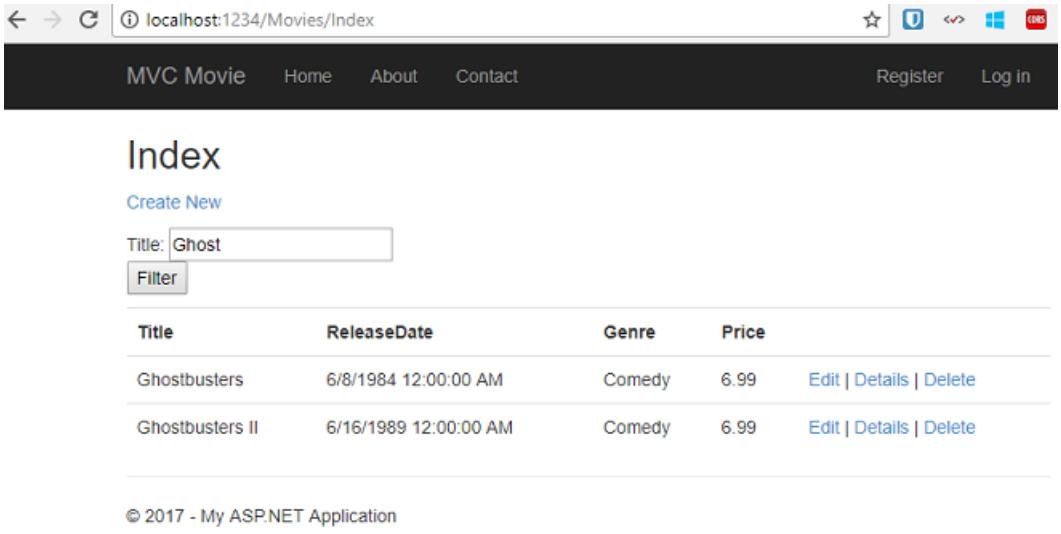
```
@model IEnumerable<MvcMovie.Models.Movie>

@{
    ViewBag.Title = "Index";
}

<h2>Index</h2>

<p>
    @Html.ActionLink("Create New", "Create")

     @using (Html.BeginForm()){
         <p> Title: @Html.TextBox("SearchString") <br />
         <input type="submit" value="Filter" /></p>
        }
</p>
```

The `Html.BeginForm` helper creates an opening `<form>` tag. The `Html.BeginForm` helper causes the form to post to itself when the user submits the form by clicking the **Filter** button.

Visual Studio 2013 has a nice improvement when displaying and editing View files. When you run the application with a view file open, Visual Studio 2013 invokes the correct controller action method to display the view.



With the Index view open in Visual Studio (as shown in the image above), tap Ctr F5 or F5 to run the application and then try searching for a movie.



There's no `HttpPost` overload of the `Index` method. You don't need it, because the method isn't changing the state of the application, just filtering data.

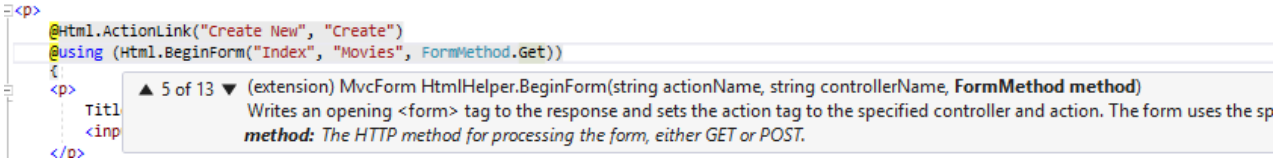
You could add the following `HttpPost Index` method. In that case, the action invoker would match the `HttpPost Index` method, and the `HttpPost Index` method would run as shown in the image below.

```
[HttpPost]
public string Index(FormCollection fc, string searchString)
{
    return "<h3> From [HttpPost]Index: " + searchString + "</h3>";
}
```

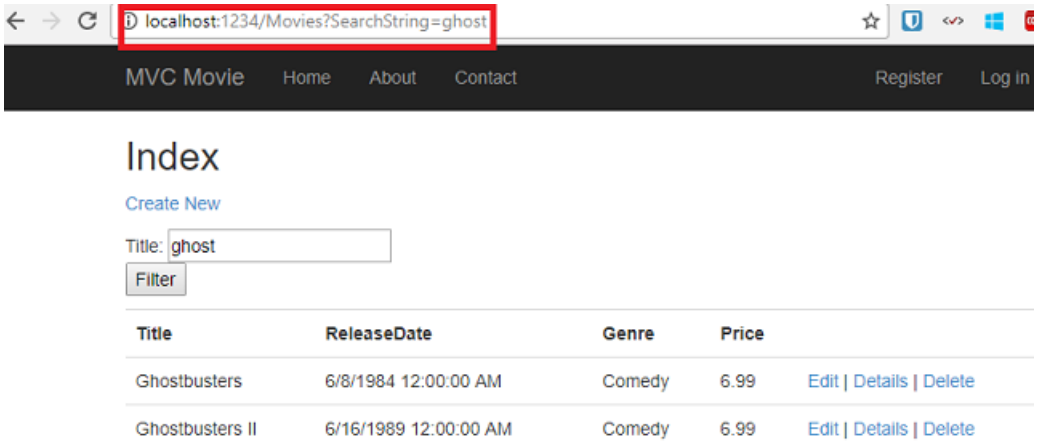
However, even if you add this `HttpPost` version of the `Index` method, there's a limitation in how this has all been implemented. Imagine that you want to bookmark a particular search or you want to send a link to friends that they can click in order to see the same filtered list of movies. Notice that 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 itself. Right now, the search string information is sent to the server as a form field value. This means you can't capture that search information to bookmark or send to friends in a URL.

The solution is to use an overload of `BeginForm` that specifies that the POST request should add the search information to the URL and that it should be routed to the `HttpGet` version of the `Index` method. Replace the existing parameterless `BeginForm` method with the following markup:

```
@using (Html.BeginForm("Index","Movies",FormMethod.Get))
```



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



Adding Search by Genre

If you added the `HttpPost` version of the `Index` method, delete it now.

Next, you'll add a feature to let users search for movies by genre. Replace the `Index` method with the following code:

```
public ActionResult Index(string movieGenre, string searchString)
{
    var GenreLst = new List<string>();

    var GenreQry = from d in db.Movies
                   orderby d.Genre
                   select d.Genre;

    GenreLst.AddRange(GenreQry.Distinct());
    ViewBag.movieGenre = new SelectList(GenreLst);

    var movies = from m in db.Movies
                 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);
    }

    return View(movies);
}
```

This version of the `Index` method takes an additional parameter, namely `movieGenre`. The first few lines of code create a `List` object to hold movie genres from the database.

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

```
var GenreQry = from d in db.Movies
                   orderby d.Genre
                   select d.Genre;
```

The code uses the `AddRange` method of the generic `List` collection to add all the distinct genres to the list. (Without the `Distinct` modifier, duplicate genres would be added — for example, comedy would be added twice in our sample). The code then stores the list of genres in the `ViewBag.MovieGenre` object. Storing category data (such a movie genre's) as a SelectList object in a `ViewBag`, then accessing the category data in a dropdown list box is a typical approach for MVC applications.

The following code shows how to check the `movieGenre` parameter. If it's not empty, the code further constrains the movies query to limit the selected movies to the specified genre.

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

As stated previously, the query is not run on the data base until the movie list is iterated over (which happens in the View, after the `Index` action method returns).

Adding Markup to the Index View to Support Search by Genre

Add an `Html.DropDownList` helper to the *Views\Movies\Index.cshtml* file, just before the `TextBox` helper. The completed markup is shown below:

```
@model IEnumerable<MvcMovie.Models.Movie>
@{
    ViewBag.Title = "Index";
}
<h2>Index</h2>
<p>
    @Html.ActionLink("Create New", "Create")
    @using (Html.BeginForm("Index", "Movies", FormMethod.Get))
    {
    <p>
        Genre: @Html.DropDownList("movieGenre", "All")
        Title: @Html.TextBox("SearchString")
        <input type="submit" value="Filter" />
    </p>
    }
</p>
<table class="table">
```

In the following code:

```
@Html.DropDownList("movieGenre", "All")
```

The parameter "MovieGenre" provides the key for the `DropDownList` helper to find a `IEnumerable<SelectListItem>` in the `ViewBag`. The `ViewBag` was populated in the action method:

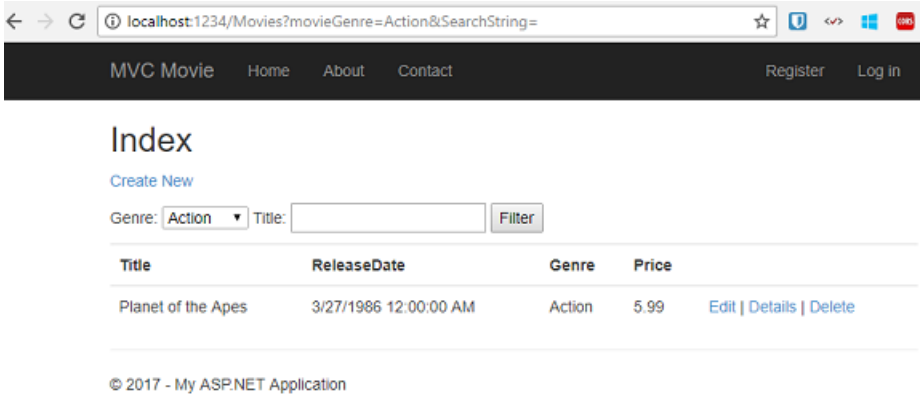
```
public ActionResult Index(string movieGenre, string searchString)
{
    var GenreLst = new List<string>();

    var GenreQry = from d in db.Movies
                   orderby d.Genre
                   select d.Genre;

    GenreLst.AddRange(GenreQry.Distinct());
    ViewBag.movieGenre = new SelectList(GenreLst);

    var movies = from m in db.Movies
                 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);
    }

    return View(movies);
}
```

The parameter "All" provides an option label. If you inspect that choice in your browser, you'll see that its "value" attribute is empty. Since our controller only filters `if` the string is not `null` or empty, submitting an empty value for `movieGenre` shows all genres.

You can also set an option to be selected by default. If you wanted "Comedy" as your default option, you would change the code in the Controller like so:

```
ViewBag.movieGenre = new SelectList(GenreLst, "Comedy");
```

Run the application and browse to */Movies/Index*. Try a search by genre, by movie name, and by both criteria.



In this section you created a search action method and view that let users search by movie title and genre. In the next section, you'll look at how to add a property to the `Movie` model and how to add an initializer that will automatically create a test database.

PREVIOUS NEXT

Adding a New Field

1/24/2018 • 10 min to read • Edit Online

by Rick Anderson

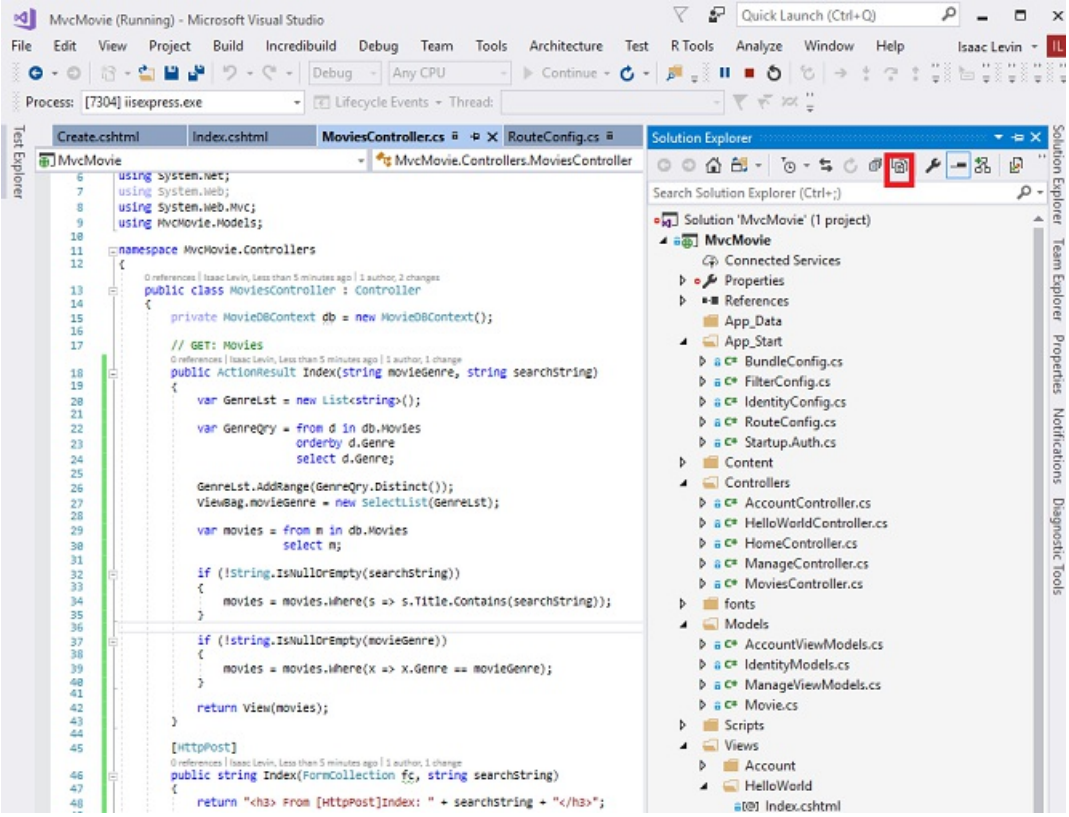
> **NOTE**
>
> This document is part of the Getting Started with ASP.NET MVC 5 tutorial. Final Source for tutorial located on GitHub

In this section you'll use Entity Framework Code First Migrations to migrate some changes to the model classes so the change is applied to the database.

By default, when you use Entity Framework Code First to automatically create a database, as you did earlier in this tutorial, 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, the Entity Framework throws an error. This makes it easier to track down issues at development time that you might otherwise only find (by obscure errors) at run time.

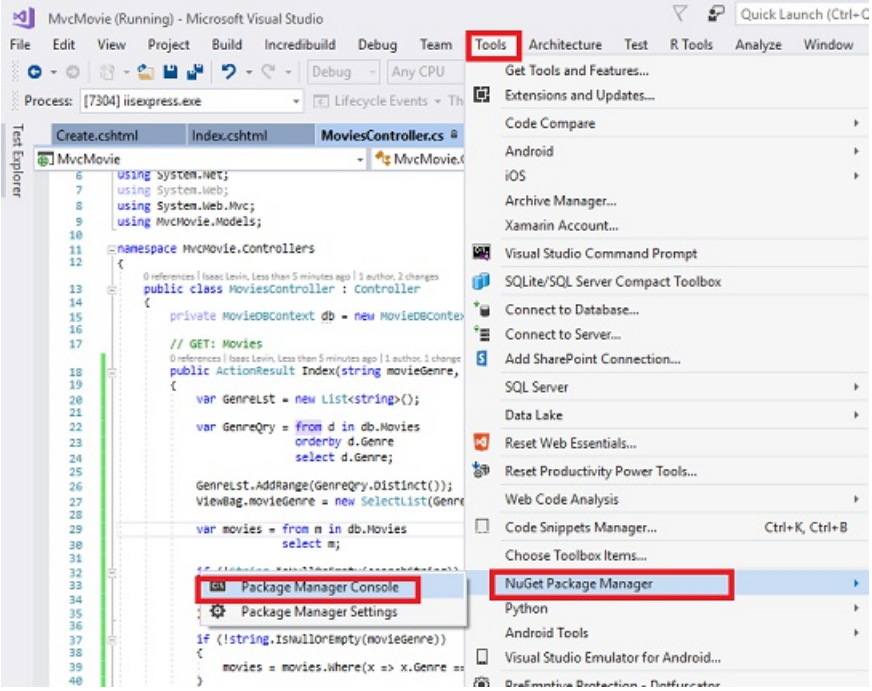
Setting up Code First Migrations for Model Changes

Navigate to Solution Explorer. Right click on the *Movies.mdf* file and select **Delete** to remove the movies database. If you don't see the *Movies.mdf* file, click on the **Show All Files** icon shown below in the red outline.



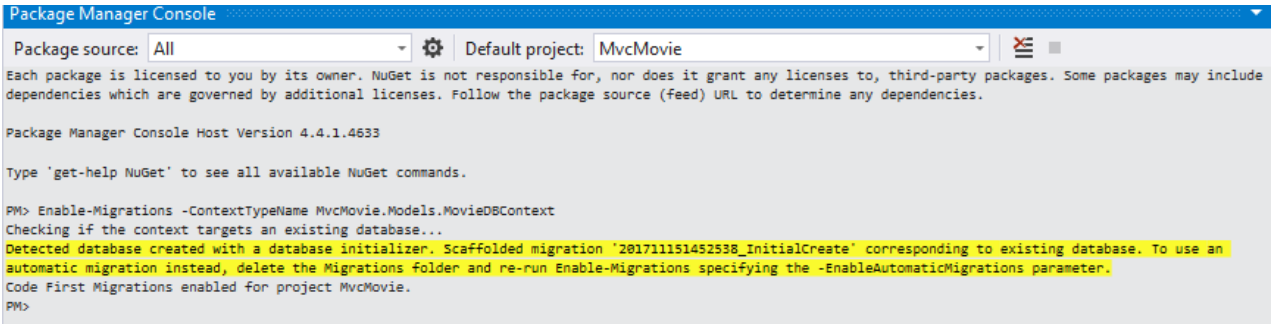
Build the application to make sure there are no errors.

From the **Tools** menu, click **NuGet Package Manager** and then **Package Manager Console**.

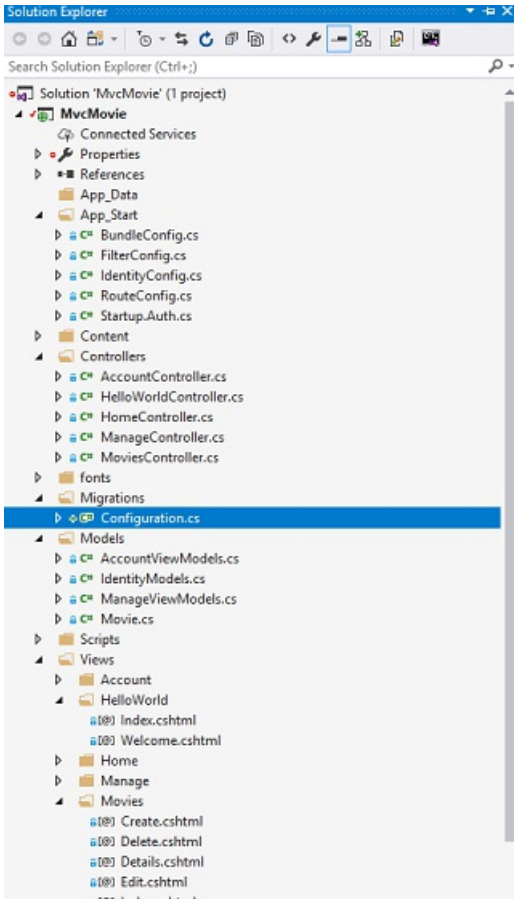


In the **Package Manager Console** window at the `PM>` prompt enter

Enable-Migrations -ContextTypeName MvcMovie.Models.MovieDBContext



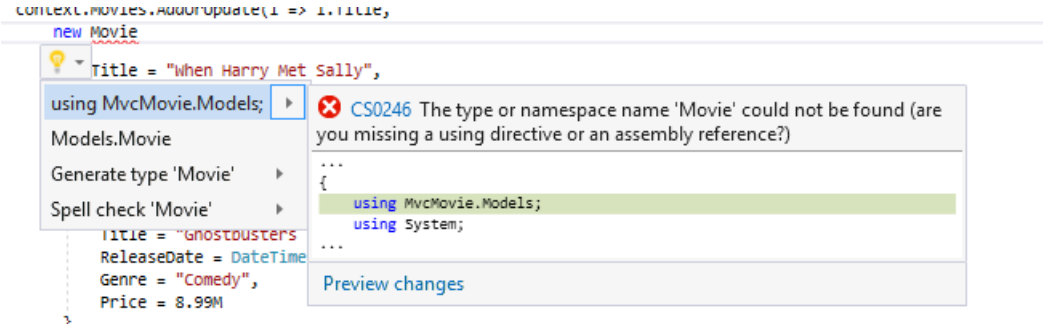
The **Enable-Migrations** command (shown above) creates a *Configuration.cs* file in a new *Migrations* folder.



Visual Studio opens the *Configuration.cs* file. Replace the `Seed` method in the *Configuration.cs* file with the following code:

```
protected override void Seed(MvcMovie.Models.MovieDBContext context)
{
    context.Movies.AddOrUpdate( i => i.Title,
        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
        }
    );

}
```

Hover over the red squiggly line under `Movie` and click `Show Potential Fixes` and then click **using MvcMovie.Models;**



Doing so adds the following using statement:

```
using MvcMovie.Models;
```

NOTE

Code First Migrations calls the `Seed` method after every migration (that is, calling **update-database** in the Package Manager Console), and this method updates rows that have already been inserted, or inserts them if they don't exist yet.

The AddOrUpdate method in the following code performs an "upsert" operation:

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

Because the Seed method runs with every migration, you can't just insert data, because the rows you are trying to add will already be there after the first migration that creates the database. The "upsert" operation prevents errors that would happen if you try to insert a row that already exists, but it overrides any changes to data that you may have made while testing the application. With test data in some tables you might not want that to happen: in some cases when you change data while testing you want your changes to remain after database updates. In that case you want to do a conditional insert operation: insert a row only if it doesn't already exist.

The first parameter passed to the AddOrUpdate method specifies the property to use to check if a row already exists. For the test movie data that you are providing, the `Title` property can be used for this purpose since each title in the list is unique:

```
context.Movies.AddOrUpdate(i => i.Title,
```

This code assumes that titiles are unique. If you manually add a duplicate title, you'll get the following exception the next time you perform a migration.

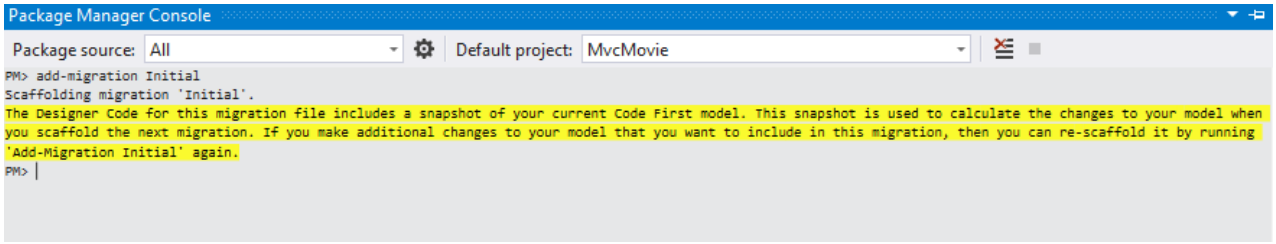
Sequence contains more than one element

For more information about the AddOrUpdate method, see Take care with EF 4.3 AddOrUpdate Method..

Press CTRL-SHIFT-B to build the project.(The following steps will fail if you don't build at this point.)

The next step is to create a `DbMigration` class for the initial migration. This migration creates a new database, that's why you deleted the *movie.mdf* file in a previous step.

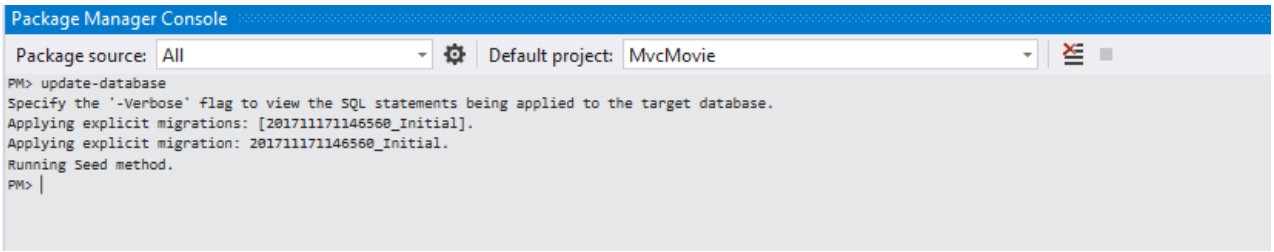
In the **Package Manager Console** window, enter the command `add-migration Initial` to create the initial migration. The name "Initial" is arbitrary and is used to name the migration file created.



Code First Migrations creates another class file in the *Migrations* folder (with the name *{DateStamp}_Initial.cs*), and this class contains code that creates the database schema. The migration filename is pre-fixed with a timestamp to help with ordering. Examine the *{DateStamp}_Initial.cs* file, it contains the instructions to create the `Movies` table for the Movie DB. When you update the database in the instructions below, this *{DateStamp}_Initial.cs* file will run and create the DB schema. Then the **Seed** method will run to populate the DB with test data.

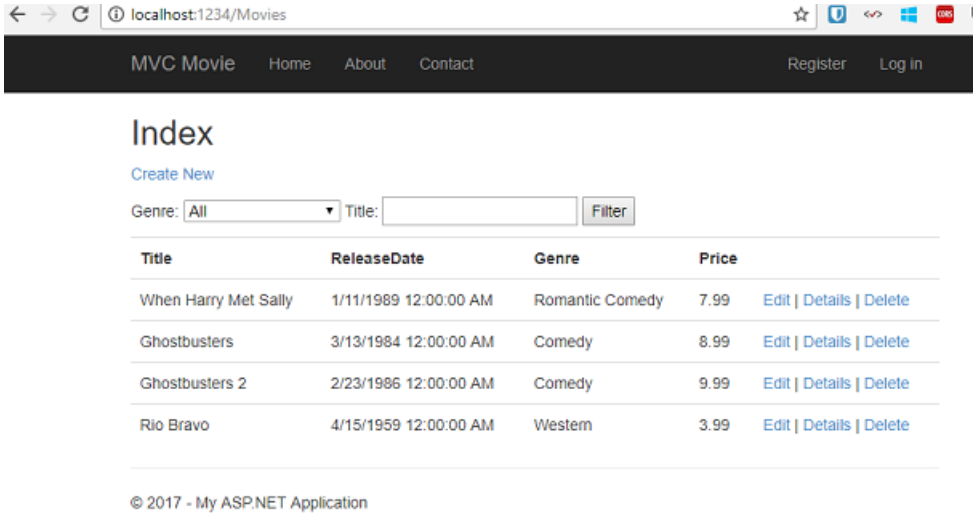
In the **Package Manager Console**, enter the command `update-database` to create the database and run the `Seed`

method.



If you get an error that indicates a table already exists and can't be created, it is probably because you ran the application after you deleted the database and before you executed `update-database`. In that case, delete the *Movies.mdf* file again and retry the `update-database` command. If you still get an error, delete the migrations folder and contents then start with the instructions at the top of this page (that is delete the *Movies.mdf* file then proceed to Enable-Migrations). If you still get an eror, open SQL Server Object Explorer and remove the database from the list.

Run the application and navigate to the */Movies* URL. The seed data is displayed.



Adding a Rating Property to the Movie Model

Start by adding a new `Rating` property to the existing `Movie` class. Open the *Models\Movie.cs* file and add the `Rating` property like this one:

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

The complete `Movie` class now looks like the following code:

```
public class Movie
{
    public int ID { get; set; }
    public string Title { get; set; }

    [Display(Name = "Release Date")]
    [DataType(DataType.Date)]
    [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
    public DateTime ReleaseDate { get; set; }
    public string Genre { get; set; }
    public decimal Price { get; set; }
    public string Rating { get; set; }
}
```

Build the application (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(Include = "ID,Title,ReleaseDate,Genre,Price,Rating")]
```

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

Open the *\Views\Movies\Index.cshtml* file and add a `<th>Rating</th>` column heading just after the **Price** column. Then add a `<td>` column near the end of the template to render the `@item.Rating` value. Below is what the updated *Index.cshtml* view template looks like:

```
@model IEnumerable<MvcMovie.Models.Movie>
@{
    ViewBag.Title = "Index";
}
<h2>Index</h2>
<p>
    @Html.ActionLink("Create New", "Create")
    @using (Html.BeginForm("Index", "Movies", FormMethod.Get))
    {
    <p>
        Genre: @Html.DropDownList("movieGenre", "All")
        Title: @Html.TextBox("SearchString")
        <input type="submit" value="Filter" />
    </p>
    }
</p>
<table class="table">
    <tr>
        <th>
            @Html.DisplayNameFor(model => model.Title)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.ReleaseDate)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Genre)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Price)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Rating)
        </th>

        <th></th>
    </tr>

@foreach (var item in Model) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.Title)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.ReleaseDate)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Genre)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Price)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Rating)
        </td>

        <td>
            @Html.ActionLink("Edit", "Edit", new { id=item.ID }) |
            @Html.ActionLink("Details", "Details", new { id=item.ID }) |
            @Html.ActionLink("Delete", "Delete", new { id=item.ID })
        </td>
    </tr>
}

</table>
```

Next, open the *\Views\Movies\Create.cshtml* file and add the `Rating` field with the following highlighed markup.

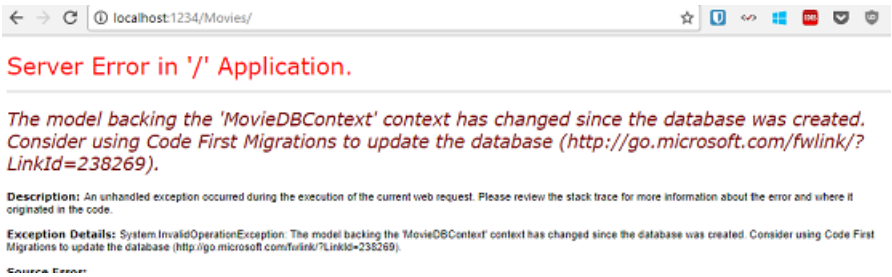
This renders a text box so that you can specify a rating when a new movie is created.

```
<div class="form-group">
            @Html.LabelFor(model => model.Price, new { @class = "control-label col-md-2" })
            <div class="col-md-10">
                @Html.EditorFor(model => model.Price, new { htmlAttributes = new { @class = "form-control" }
})
                @Html.ValidationMessageFor(model => model.Price, "", new { @class = "text-danger" })
            </div>
        </div>

        <div class="form-group">
            @Html.LabelFor(model => model.Rating, new { @class = "control-label col-md-2" })
            <div class="col-md-10">
                @Html.EditorFor(model => model.Rating, new { htmlAttributes = new { @class = "form-control" }
})
                @Html.ValidationMessageFor(model => model.Rating, "", new { @class = "text-danger" })
            </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>
}

<div>
    @Html.ActionLink("Back to List", "Index")
</div>

@section Scripts {
    @Scripts.Render("~/bundles/jqueryval")
}
```

You've now updated the application code to support the new `Rating` property.

Run the application and navigate to the */Movies* URL. When you do this, though, you'll see one of the following errors:

The model backing the 'MovieDBContext' context has changed since the database was created. Consider using Code First Migrations to update the database (https://go.microsoft.com/fwlink/?LinkId=238269).



You're seeing this error because the updated `Movie` model class in the application is now 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 develope an application. For more information on Entity Framework database initializers, see ASP.NET MVC/Entity Framework tutorial.
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 Seed method so that it provides a value for the new column. Open Migrations\Configuration.cs file and add a Rating field to each Movie object.

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

Build the solution, and then open the **Package Manager Console** window and enter the following command:

```
add-migration Rating
```

The `add-migration` command tells the migration framework to examine the current movie model with the current movie DB schema and create the necessary code to migrate the DB to the new model. The name *Rating* is arbitrary and is used to name the migration file. It's helpful to use a meaningful name for the migration step.

When this command finishes, Visual Studio opens the class file that defines the new `DbMigration` derived class, and in the `Up` method you can see the code that creates the new column.

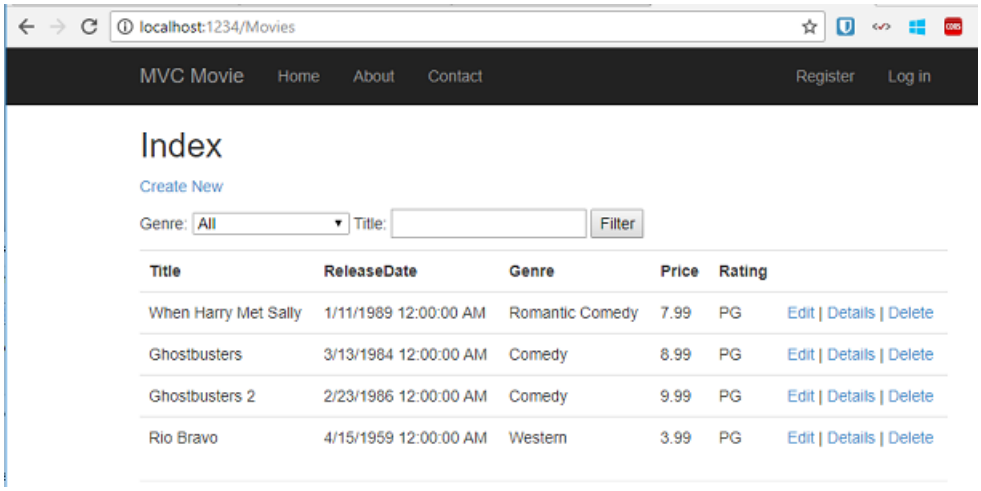
```
public partial class AddRatingMig : DbMigration
{
    public override void Up()
    {
        AddColumn("dbo.Movies", "Rating", c => c.String());
    }

    public override void Down()
    {
        DropColumn("dbo.Movies", "Rating");
    }
}
```

Build the solution, and then enter the `update-database` command in the **Package Manager Console** window.

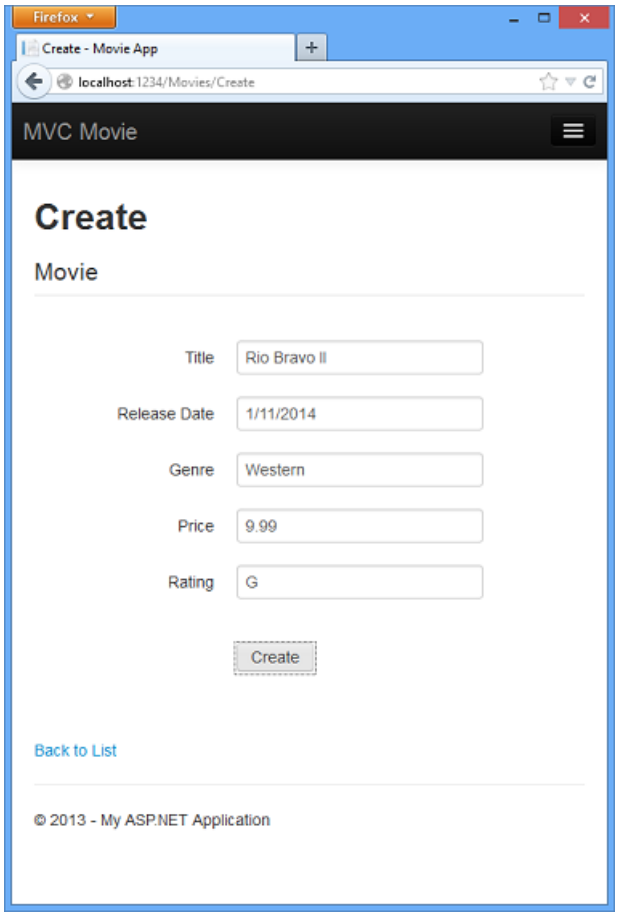
The following image shows the output in the **Package Manager Console** window (The date stamp prepending *Rating* will be different.)

```
PM> update-database
Specify the '-Verbose' flag to view the SQL statements being applied to the target database.
Applying explicit migrations: [201711171332278_Rating].
Applying explicit migration: 201711171332278_Rating.
Running Seed method.
PM>
```

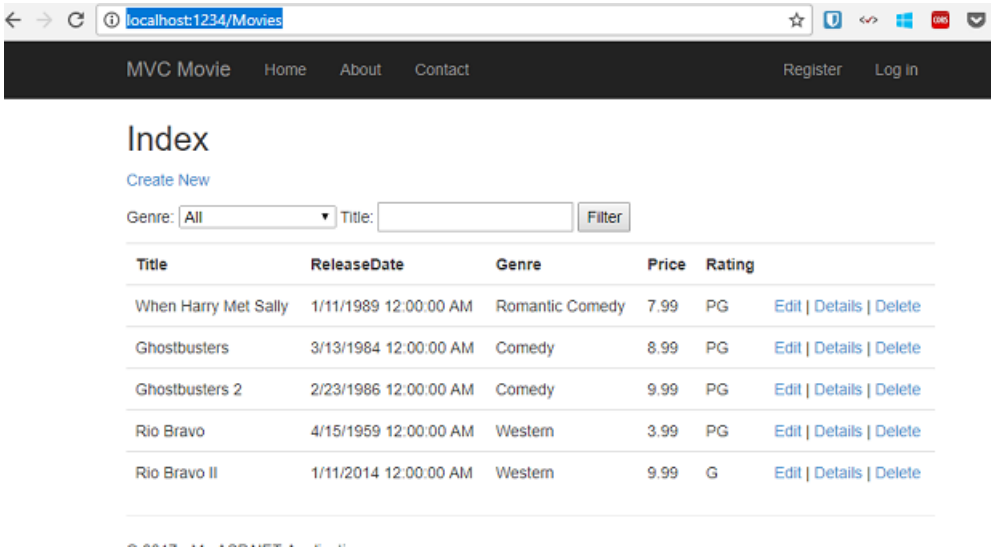
Re-run the application and navigate to the /Movies URL. You can see the new Rating field.



Click the **Create New** link to add a new movie. Note that you can add a rating.



Click **Create**. The new movie, including the rating, now shows up in the movies listing:



Now that the project is using migrations, you won't need to drop the database when you add a new field or otherwise update the schema. In the next section, we'll make more schema changes and use migrations to update the database.

You should also add the `Rating` field to the Edit, Details, and Delete view templates.

You could enter the "update-database" command in the **Package Manager Console** window again and no migration code would run, because the schema matches the model. However, running "update-database" will run the `Seed` method again, and if you changed any of the Seed data, the changes will be lost because the `Seed` method upserts data. You can read more about the `Seed` method in Tom Dykstra's popular ASP.NET MVC/Entity Framework tutorial.

In this section you saw how you can modify model objects and keep the database in sync with the changes. You also learned a way to populate a newly created database with sample data so you can try out scenarios. This was

just a quick introduction to Code First, see Creating an Entity Framework Data Model for an ASP.NET MVC Application for a more complete tutorial on the subject. Next, let's look at how you can add richer validation logic to the model classes and enable some business rules to be enforced.

PREVIOUS NEXT

Adding Validation

2/12/2018 • 12 min to read • Edit Online

by Rick Anderson

> **NOTE**
> This document is part of the Getting Started with ASP.NET MVC 5 tutorial. Final Source for tutorial located on GitHub

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 using the application.

Keeping Things DRY

One of the core design tenets of ASP.NET 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 application. This reduces the amount of code you need to write and makes the code you do write less error prone and easier to maintain.

The validation support provided by ASP.NET MVC and Entity Framework 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 application.

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

Adding Validation Rules to the Movie Model

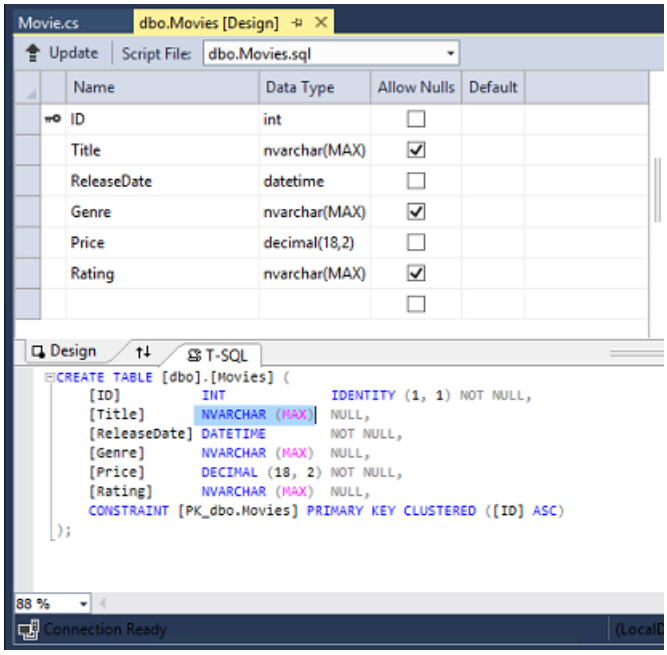
You'll begin by adding some validation logic to the `Movie` class.

Open the *Movie.cs* file. Notice the `System.ComponentModel.DataAnnotations` namespace does not contain `System.Web`. DataAnnotations provides a built-in set of validation attributes that you can apply declaratively to any class or property. (It also contains formatting attributes like DataType that help with formatting and don't provide any validation.)

Now update the `Movie` class to take advantage of the built-in `Required`, `StringLength`, RegularExpression, and `Range` validation attributes. Replace the `Movie` class with the following:

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

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

    [Display(Name = "Release Date")]
    [DataType(DataType.Date)]
    [DisplayFormat(DataFormatString = "{0:yyyy-MM-dd}", ApplyFormatInEditMode = true)]
    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 `StringLength` attribute sets the maximum length of the string, and it sets this limitation on the database, therefore the database schema will change. Right click on the **Movies** table in **Server explorer** and click **Open Table Definition**:



In the image above, you can see all the string fields are set to NVARCHAR (MAX). We will use migrations to update the schema. Build the solution, and then open the **Package Manager Console** window and enter the following commands:

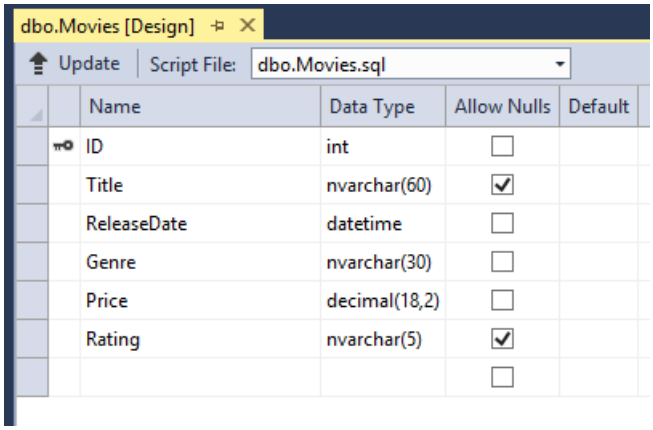
```
add-migration DataAnnotations
update-database
```

When this command finishes, Visual Studio opens the class file that defines the new `DbMIgration` derived class with the name specified (`DataAnnotations`), and in the `Up` method you can see the code that updates the schema constraints:

```
public override void Up()
{
    AlterColumn("dbo.Movies", "Title", c => c.String(maxLength: 60));
    AlterColumn("dbo.Movies", "Genre", c => c.String(nullable: false, maxLength: 30));
    AlterColumn("dbo.Movies", "Rating", c => c.String(maxLength: 5));
}
```

The `Genre` field is are no longer nullable (that is, you must enter a value). The `Rating` field has a maximum length of 5 and `Title` has a maximum length of 60. The minimum length of 3 on `Title` and the range on `Price` did not create schema changes.

Examine the Movie schema:



The string fields show the new length limits and `Genre` is no longer checked as nullable.

The validation attributes specify behavior that you want to enforce on the model properties they are applied to. The `Required` and `MinimumLength` attributes indicates that a property must have a value; but nothing prevents a user from entering white space to satisfy this validation. The RegularExpression attribute is used to limit what characters can be input. In the code above, `Genre` and `Rating` must use only letters (white space, numbers and special characters are not allowed). The `Range` attribute constrains a value to within a specified range. The `StringLength` attribute lets you set the maximum length of a string property, and optionally its minimum length. Value types (such as `decimal, int, float, DateTime`) are inherently required and don't need the `Required` attribute.

Code First ensures that the validation rules you specify on a model class are enforced before the application saves changes in the database. For example, the code below will throw a DbEntityValidationException exception when the `SaveChanges` method is called, because several required `Movie` property values are missing:

```
MovieDBContext db = new MovieDBContext();
Movie movie = new Movie();
movie.Title = "Gone with the Wind";
db.Movies.Add(movie);
db.SaveChanges();        // <= Will throw server side validation exception
```

The code above throws the following exception:

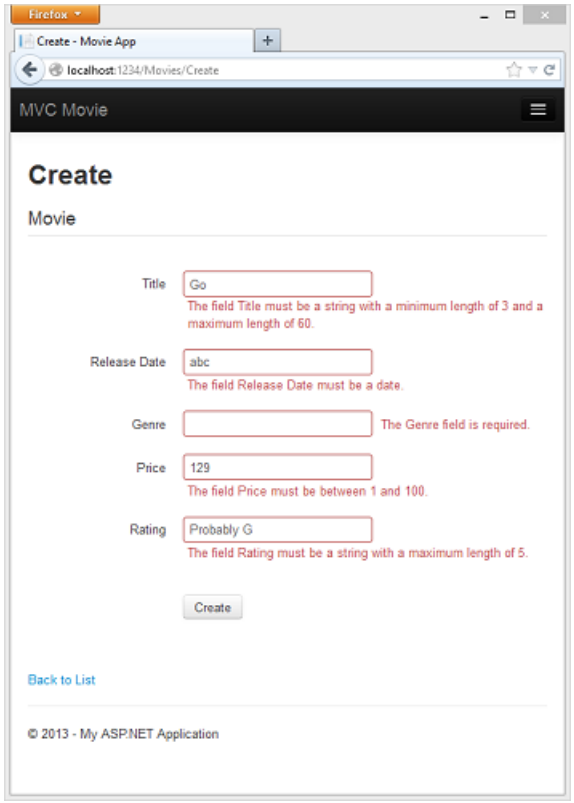
Validation failed for one or more entities. See 'EntityValidationErrors' property for more details.

Having validation rules automatically enforced by the .NET Framework helps make your application 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 ASP.NET MVC

Run the application and navigate to the */Movies* URL.

Click 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**
> to support jQuery validation for non-English locales that use a comma (",") for a decimal point, you must include the NuGet globalize as described previously in this tutorial.

Notice how the form has automatically used a red border color to highlight the text boxes that contain invalid data and has emitted an appropriate validation error message next to each one. The errors are enforced both client-side (using JavaScript and jQuery) and server-side (in case a user has JavaScript disabled).

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

The form data is not sent to the server until there are no client side validation errors. You can verify this by putting a break point in the HTTP Post method, by using the fiddler tool, or the IE F12 developer tools.

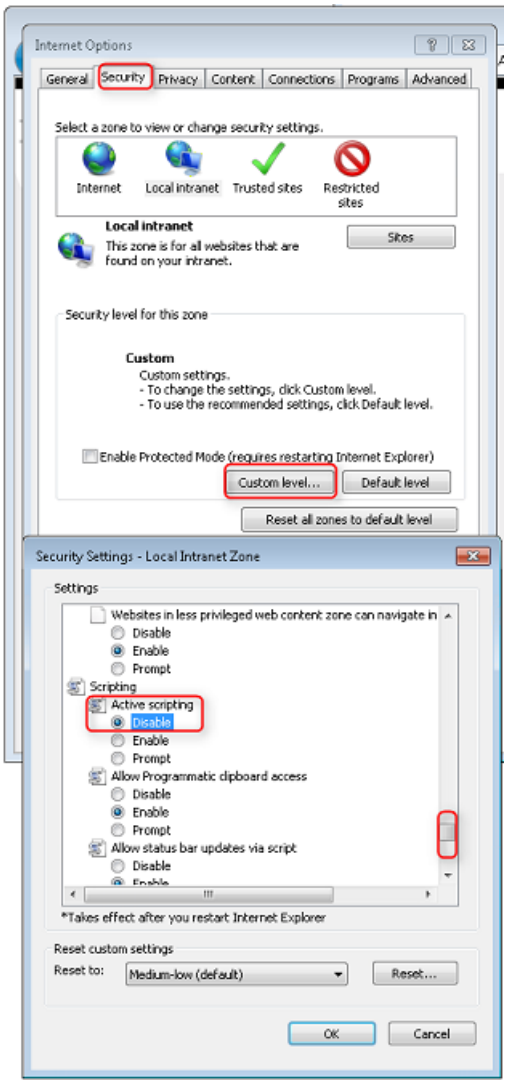
How Validation Occurs in the Create View and Create Action Method

You might wonder how the validation UI was generated without any updates to the code in the controller or views. The next listing shows what the `Create` methods in the `MovieController` class look like. They're unchanged from how you created them earlier in this tutorial.

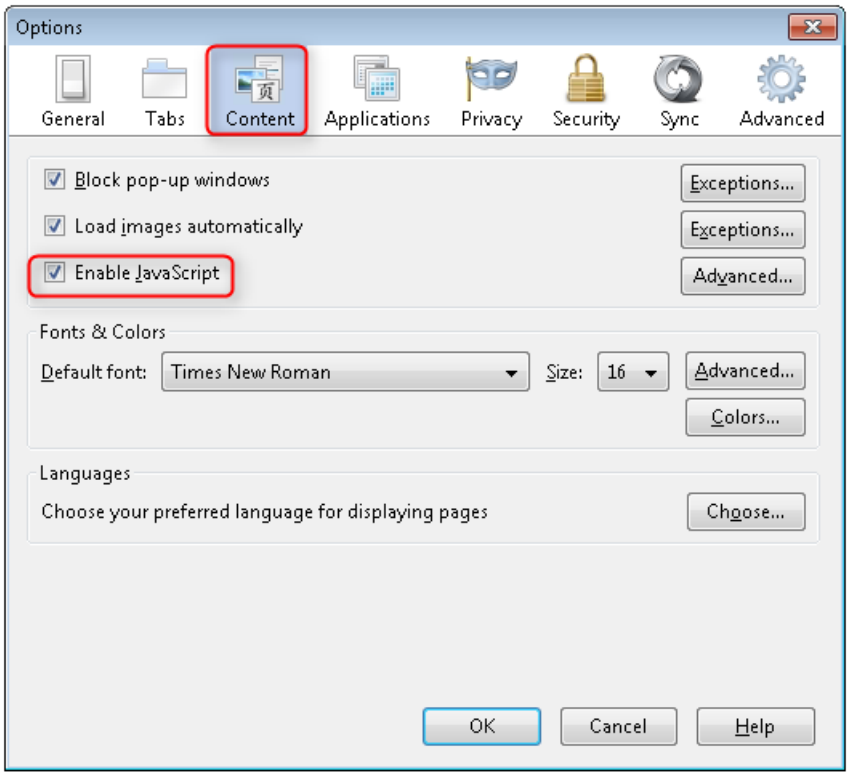
```
public ActionResult Create()
{
    return View();
}
// POST: /Movies/Create
// To protect from overposting attacks, please enable the specific properties you want to bind to, for
// more details see http://go.microsoft.com/fwlink/?LinkId=317598.
[HttpPost]
[ValidateAntiForgeryToken]
public ActionResult Create([Bind(Include = "ID,Title,ReleaseDate,Genre,Price,Rating")] Movie movie)
{
    if (ModelState.IsValid)
    {
        db.Movies.Add(movie);
        db.SaveChanges();
        return RedirectToAction("Index");
    }
    return View(movie);
}
```

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

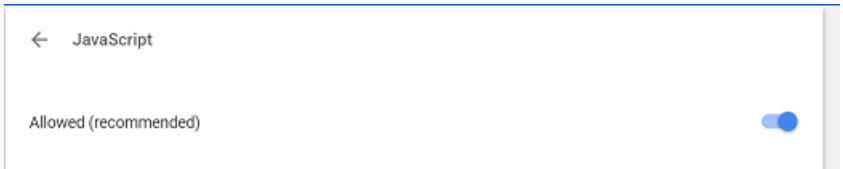
You can set a break point in the `HttpPost Create` method and verify the method is never called, client side validation will not submit the form data when validation errors are detected. If you disable JavaScript in your browser, then submit the form with errors, the break point will be hit. You still 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.



Below is 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.

```
@model MvcMovie.Models.Movie
@{
    ViewBag.Title = "Create";
}
<h2>Create</h2>
@using (Html.BeginForm())
{
    @Html.AntiForgeryToken()
    <div class="form-horizontal">
        <h4>Movie</h4>
        <hr />
        @Html.ValidationSummary(true)
        <div class="form-group">
            @Html.LabelFor(model => model.Title, new { @class = "control-label col-md-2" })
            <div class="col-md-10">
                @Html.EditorFor(model => model.Title)
                @Html.ValidationMessageFor(model => model.Title)
            </div>
        </div>
        @*Fields removed for brevity.*@




        <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>
}
<div>
    @Html.ActionLink("Back to List", "Index")
</div>
@section Scripts {
    @Scripts.Render("~/bundles/jqueryval")
}
```

Notice how the code uses an `Html.EditorFor` helper to output the `<input>` element for each `Movie` property. Next to this helper is a call to the `Html.ValidationMessageFor` helper method. These two helper methods work with the model object that's passed by the controller to the view (in this case, a `Movie` object). They automatically look for validation attributes specified on the model and display error messages as appropriate.

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.

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

Using DataType Attributes

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

```
[DataType(DataType.Date)]
public DateTime ReleaseDate { get; set; }

[DataType(DataType.Currency)]
public decimal Price { get; set; }
```

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

`DataType.Date` does not specify the format of the date that is displayed. By default, the data field is displayed according to the default formats based on the server'sCultureInfo.

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

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

The `ApplyFormatInEditMode` setting specifies that the specified 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 might not want the currency symbol in the text box for editing.)

You can use the DisplayFormat attribute by itself, but it's generally a good idea to use the DataType attribute also. 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 yourlocale.
- TheDataType attribute can enable MVC to choose the right field template to render the data (the DisplayFormat if used by itself uses the string template). For more information, see Brad Wilson'sASP.NET MVC 2 Templates. (Though written for MVC 2, this article still applies to the current version of ASP.NET MVC.)

If you use the `DataType` attribute with a date field, you have to specify the `DisplayFormat` attribute also in order to ensure that the field renders correctly in Chrome browsers. For more information, see this StackOverflow thread.

NOTE

jQuery validation does not work with theRange attribute andDateTime. 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 withDateTime. It's generally not a good practice to compile hard dates in your models, so using theRange attribute andDateTime is discouraged.

The following code shows combining attributes on one line:

```
public class Movie
{
   public int ID { get; set; }
   [Required,StringLength(60, MinimumLength = 3)]
   public string Title { get; set; }
   [Display(Name = "Release Date"),DataType(DataType.Date)]
   public DateTime ReleaseDate { get; set; }
   [Required]
   public string Genre { get; set; }
   [Range(1, 100),DataType(DataType.Currency)]
   public decimal Price { get; set; }
   [Required,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.

PREVIOUS NEXT

Examining the Details and Delete Methods

1/24/2018 • 4 min to read • Edit Online

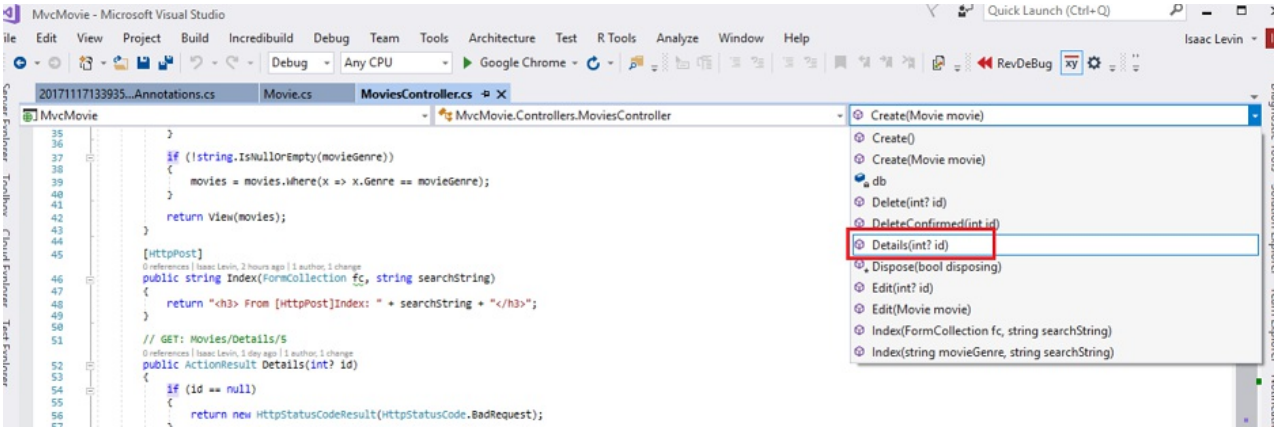
by Rick Anderson

> **NOTE**
>
> This document is part of the Getting Started with ASP.NET MVC 5 tutorial. Final Source for tutorial located on GitHub

In this part of the tutorial, you'll examine the automatically generated `Details` and `Delete` methods.

Examining the Details and Delete Methods

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




```
public ActionResult Details(int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    Movie movie = db.Movies.Find(id);
    if (movie == null)
    {
        return HttpNotFound();
    }
    return View(movie);
}
```

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, the `Movies` controller, the `Details` method and a `ID` value.

Code First makes it easy to search for data using the `Find` method. An important security feature built into the method is that the code verifies that the `Find` 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, a null movie would result in a database error.

Examine the `Delete` and `DeleteConfirmed` methods.

```
// GET: /Movies/Delete/5
public ActionResult Delete(int? id)
{
    if (id == null)
    {
        return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
    }
    Movie movie = db.Movies.Find(id);
    if (movie == null)
    {
        return HttpNotFound();
    }
    return View(movie);
}

// POST: /Movies/Delete/5
[HttpPost, ActionName("Delete")]
[ValidateAntiForgeryToken]
public ActionResult DeleteConfirmed(int id)
{
    Movie movie = db.Movies.Find(id);
    db.Movies.Remove(movie);
    db.SaveChanges();
    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. For more information about this, see Stephen Walther's blog entry ASP.NET MVC Tip #46 — Don't use Delete Links because they create Security Holes.

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 ActionResult Delete(int? id)

//
// POST: /Movies/Delete/5
[HttpPost, ActionName("Delete")]
public ActionResult DeleteConfirmed(int id)
```

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

To sort this out, you can do a couple of things. 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. This effectively performs mapping for the routing system so that a URL that includes */Delete/* for a POST request will find the `DeleteConfirmed` method.

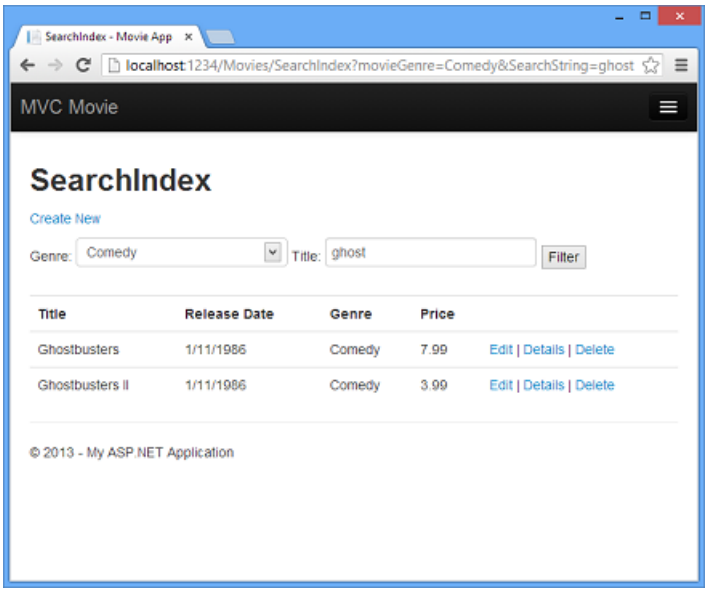
Another common way to avoid a problem with methods that have identical names and signatures is to artificially change the signature of the POST method to include an unused parameter. For example, some developers add a

parameter type `FormCollection` that is passed to the POST method, and then simply don't use the parameter:

```
public ActionResult Delete(FormCollection fcNotUsed, int id = 0)
{
    Movie movie = db.Movies.Find(id);
    if (movie == null)
    {
        return HttpNotFound();
    }
    db.Movies.Remove(movie);
    db.SaveChanges();
    return RedirectToAction("Index");
}
```

Summary

You now have a complete ASP.NET MVC application that stores data in a local DB database. You can create, read, update, delete, and search for movies.



Next Steps

After you have built and tested a web application, the next step is to make it available to other people to use over the Internet. To do that, you have to deploy it to a web hosting provider. Microsoft offers free web hosting for up to 10 web sites in a free Azure trial account. I suggest you next follow my tutorial Deploy a Secure ASP.NET MVC app with Membership, OAuth, and SQL Database to Azure. An excellent tutorial is Tom Dykstra's intermediate-level Creating an Entity Framework Data Model for an ASP.NET MVC Application. Stackoverflow and the ASP.NET MVC forums are a great places to ask questions. Follow me on twitter so you can get updates on my latest tutorials.

Feedback is welcome.

— Rick Anderson twitter: @RickAndMSFT
— Scott Hanselman twitter: @shanselman

PREVIOUS