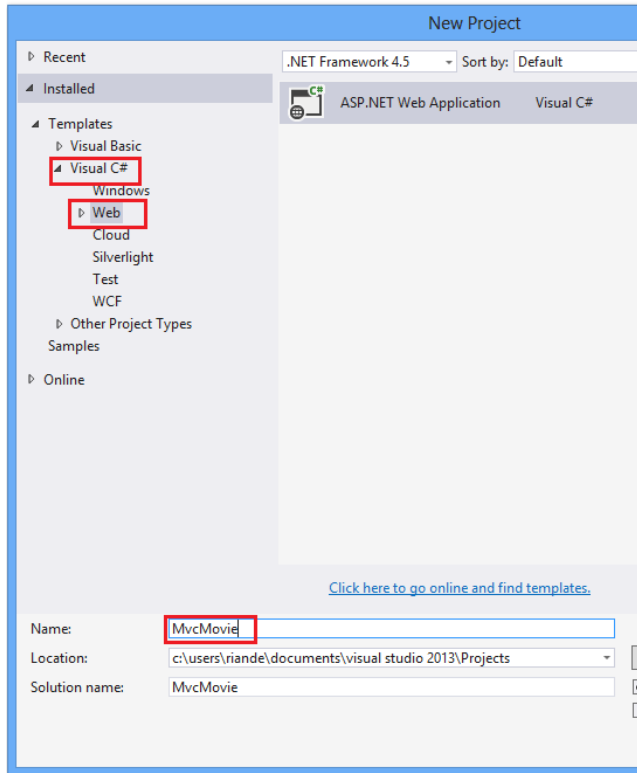


Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

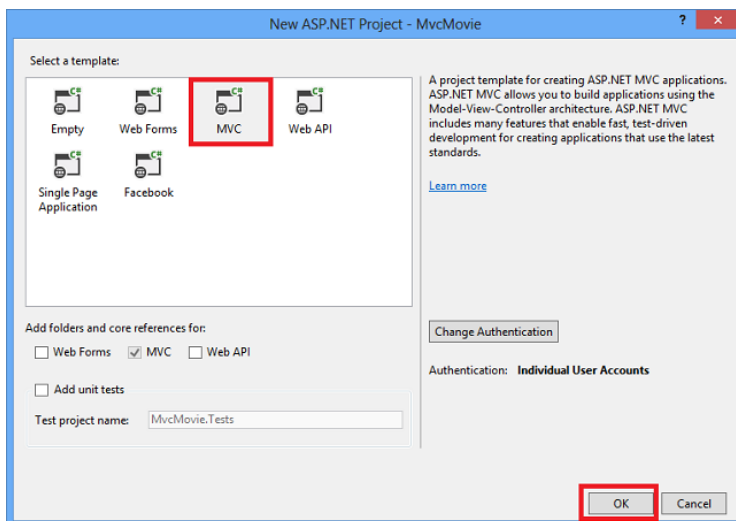
Creating Your First Application:

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

<http://www.asp.net/mvc/tutorials/mvc-5/introduction/getting-started>

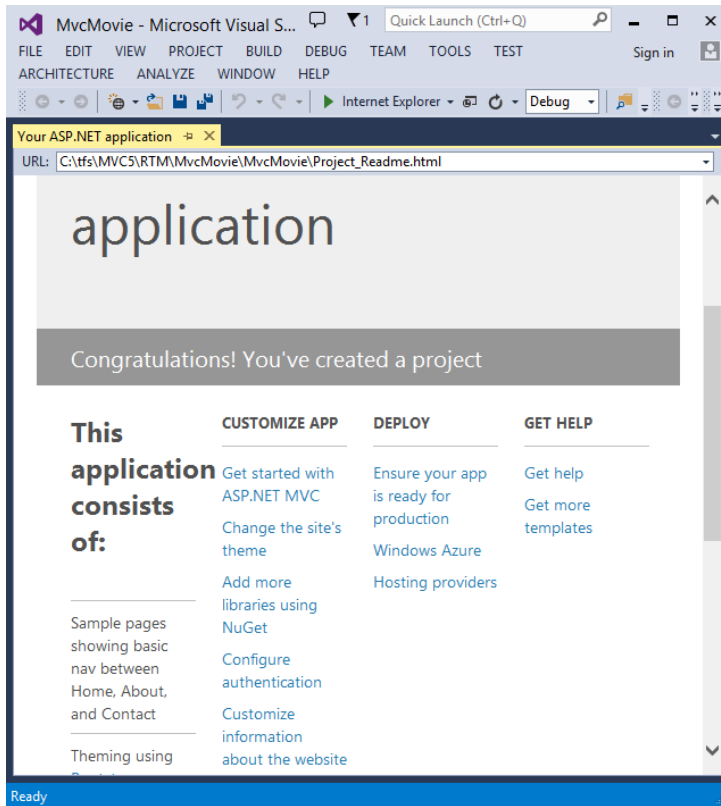


In the **New ASP.NET Project** dialog, click **MVC** and then click **OK**.

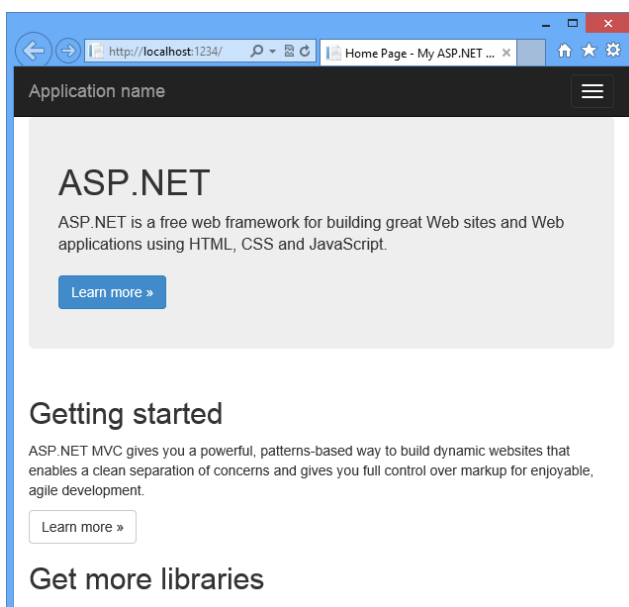


Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

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.

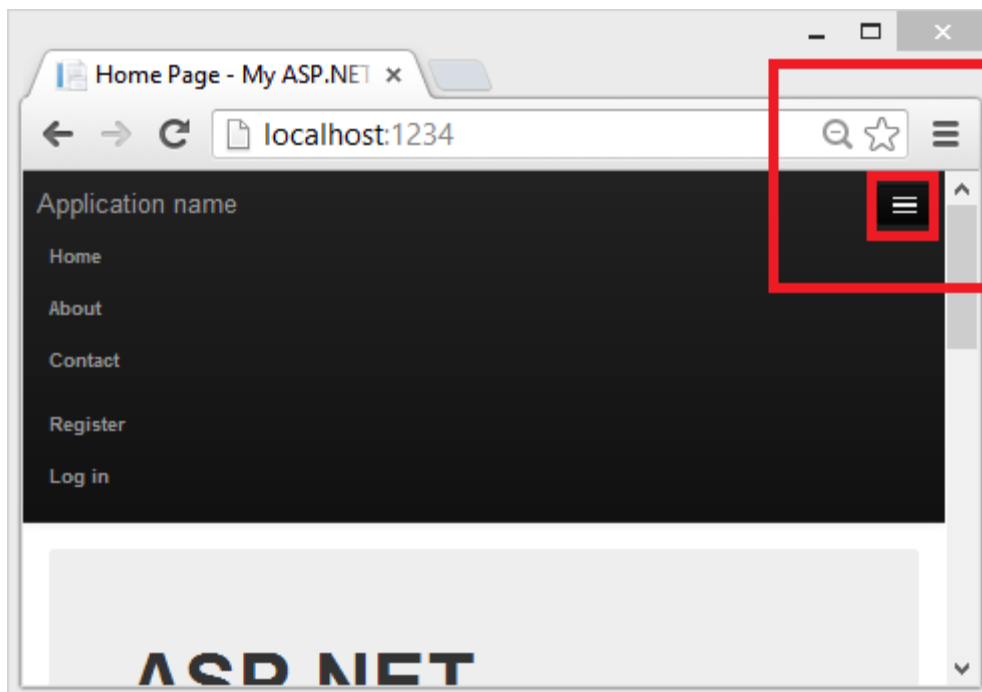
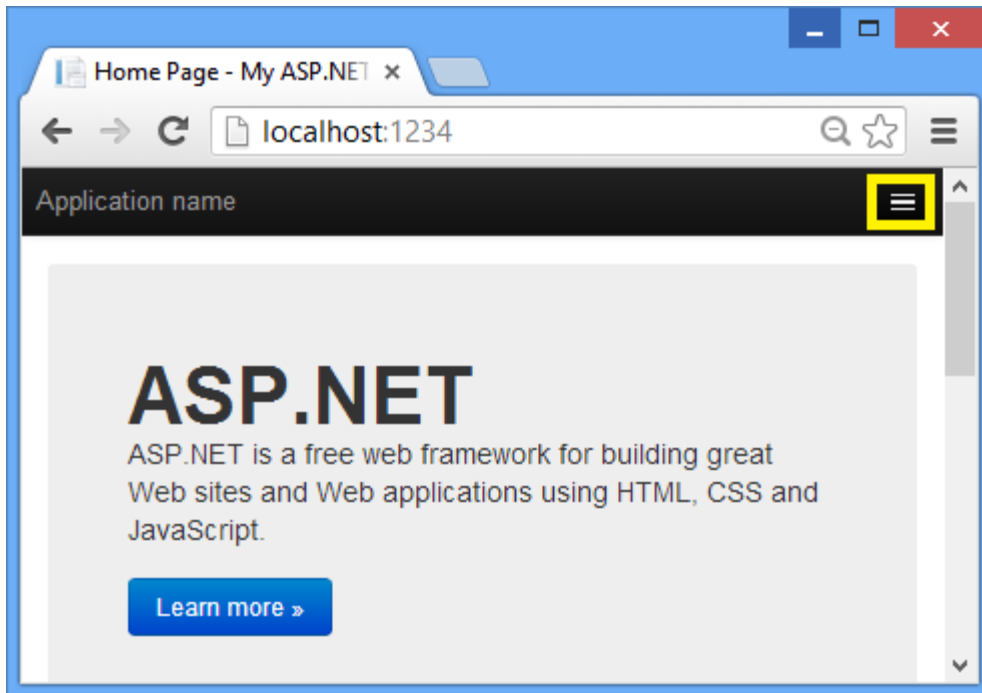


Click F5 to start debugging. F5 causes Visual Studio to start [IIS Express](#) and run your web app. Visual Studio then launches a browser and opens the application's home page. Notice that the address bar of the browser says **localhost:port#** and not something like **example.com**. That's because **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.



Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

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.

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

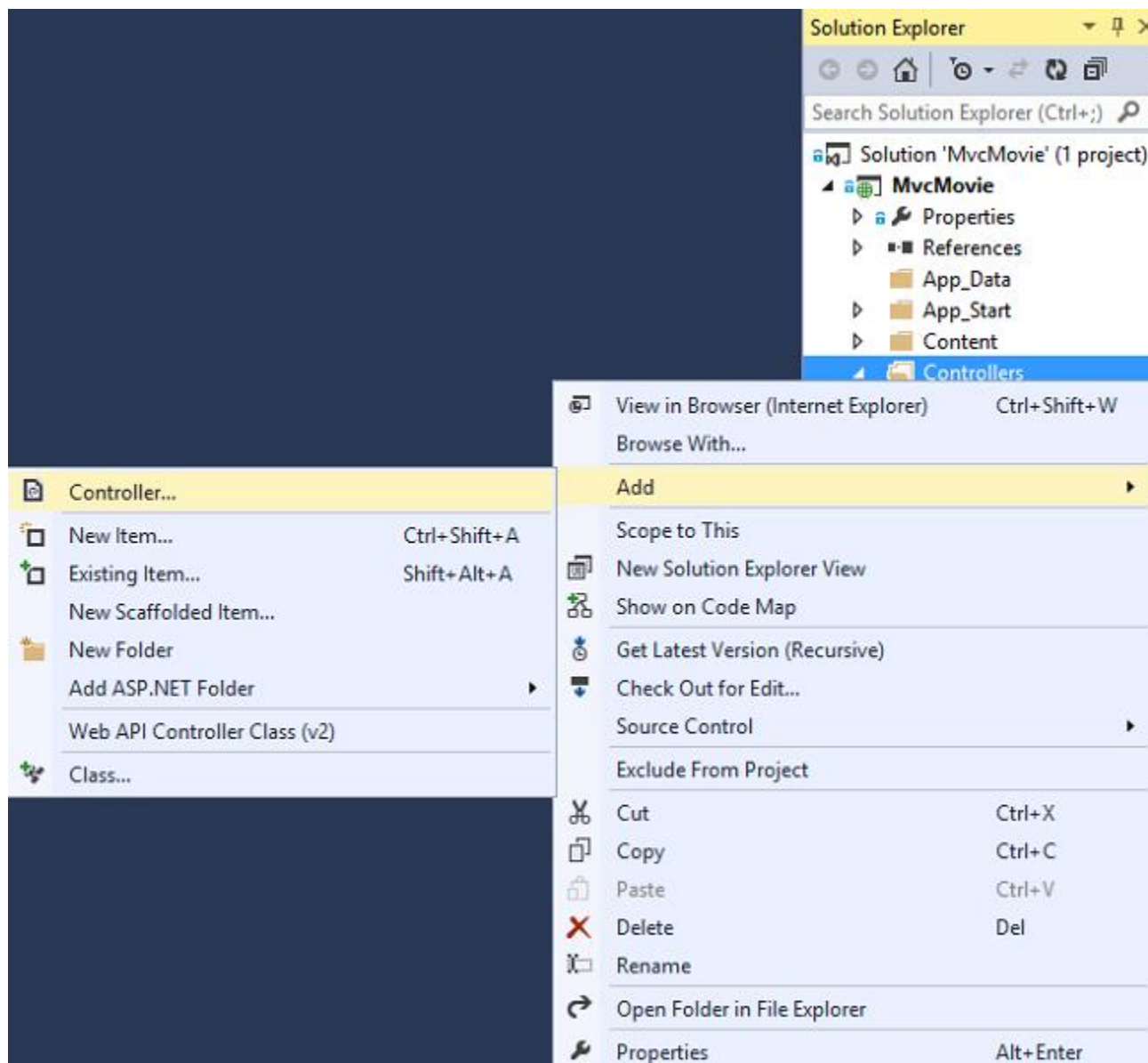
Adding a Controller:

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:

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

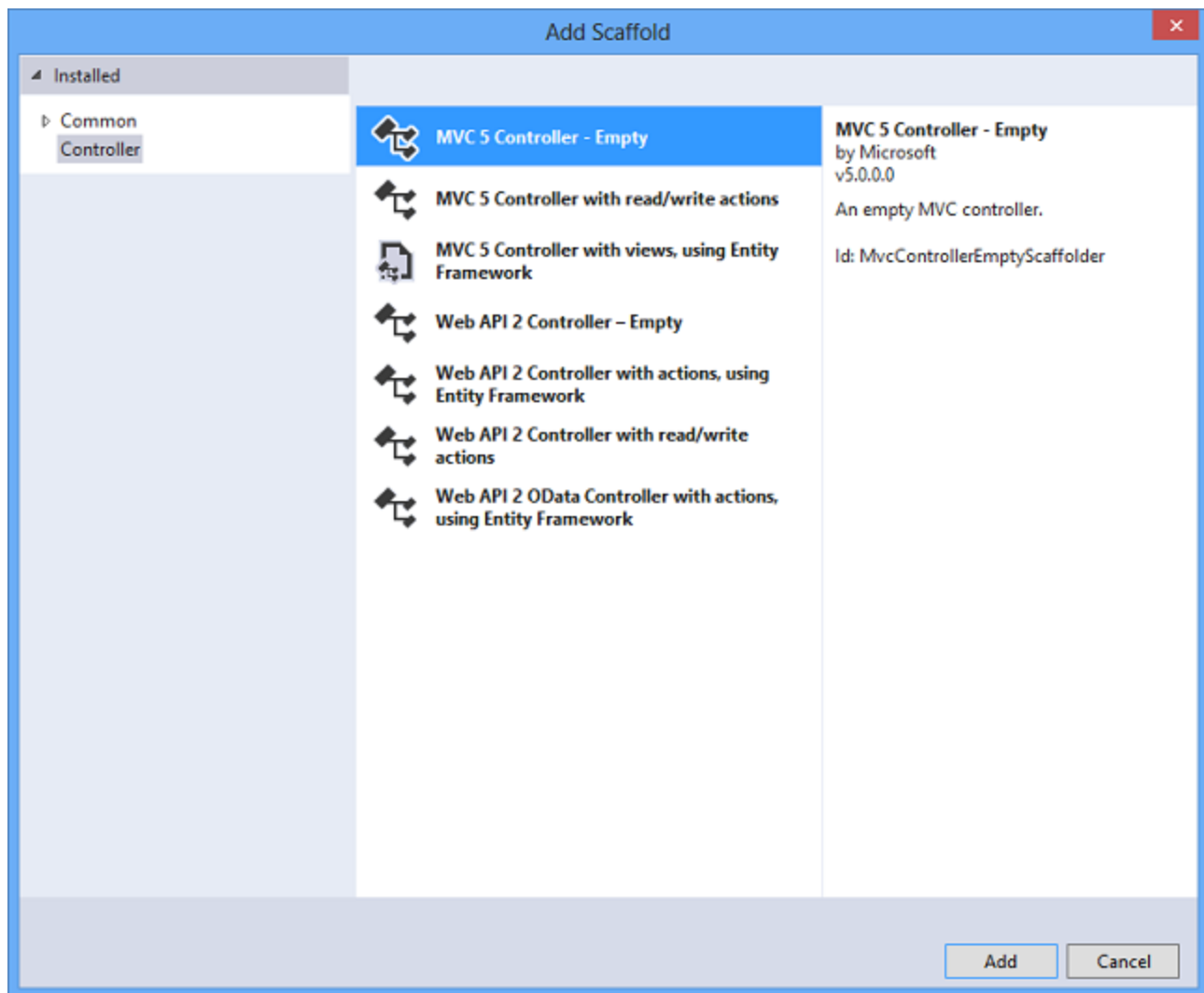
We'll be covering all these concepts in this tutorial series and show you how to use them to build an application.

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

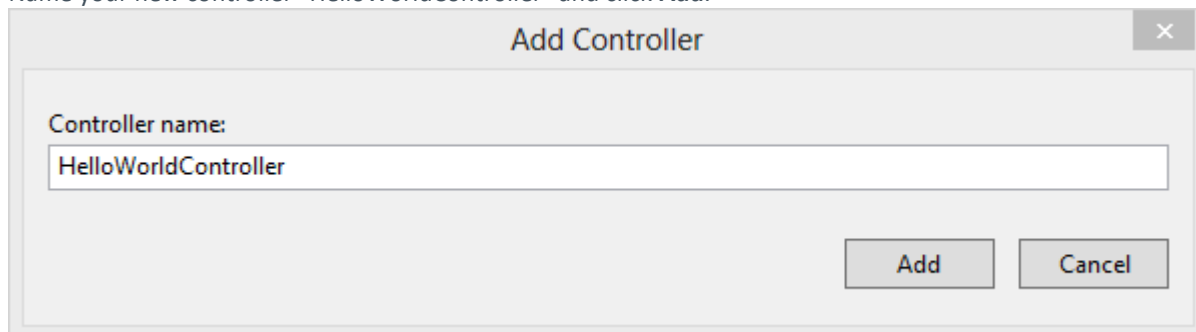


Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

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

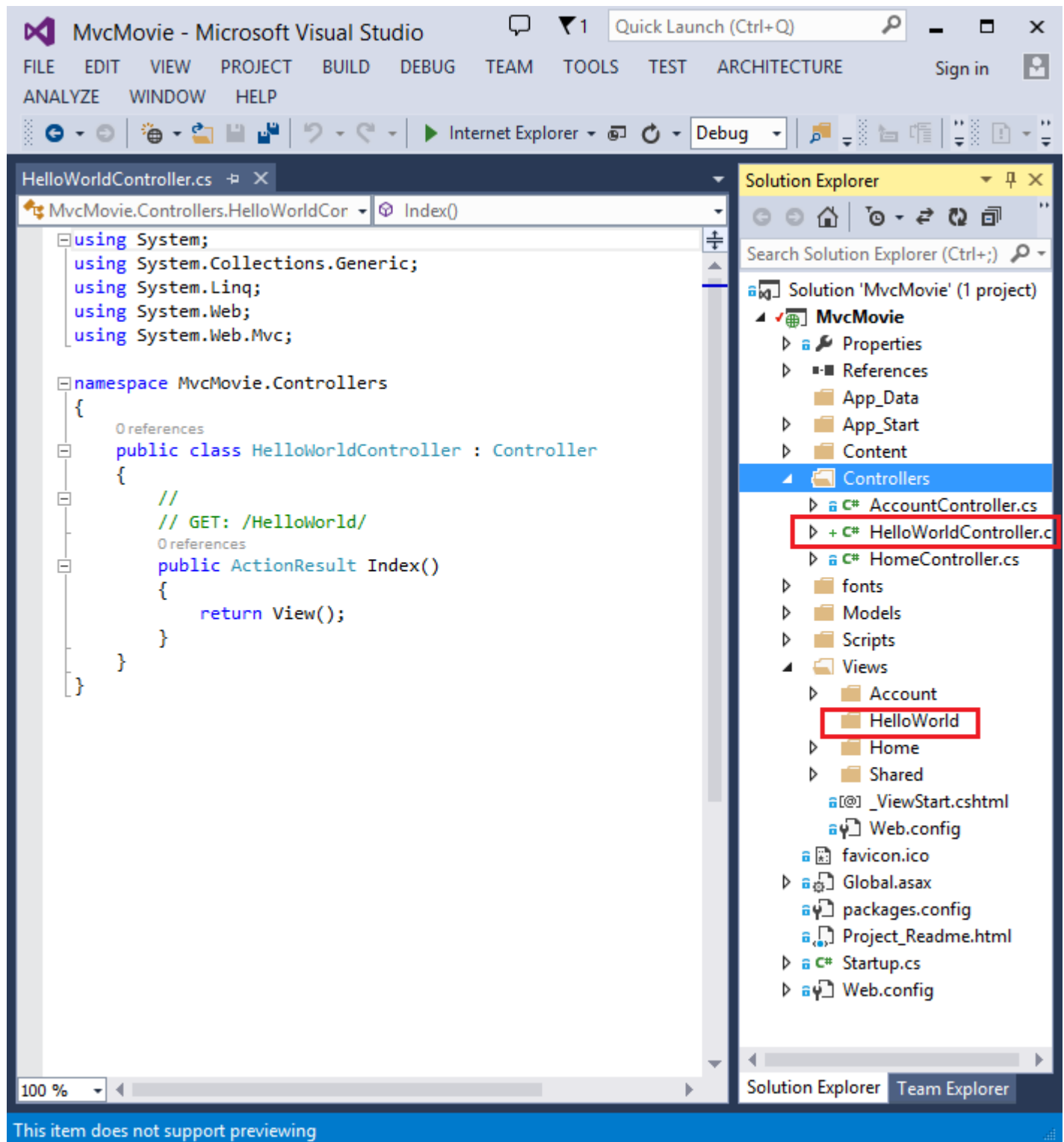


Name your new controller "HelloWorldController" and click **Add**.



Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

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.



Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

Replace the contents of the file with the following code.

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

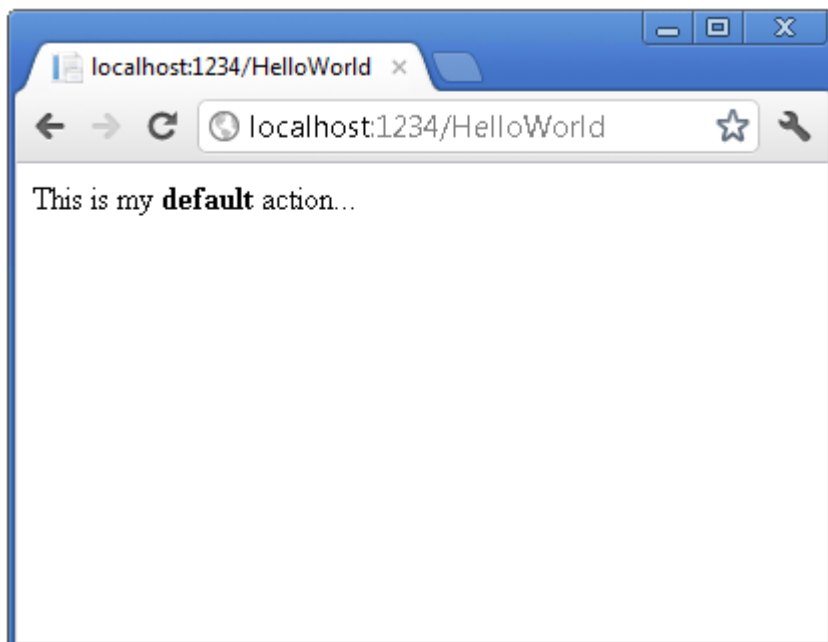
namespace MvcMovie.Controllers
{
    public class HelloWorldController : Controller
    {
        //
        // GET: /HelloWorld/

        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!



Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

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 }

    );

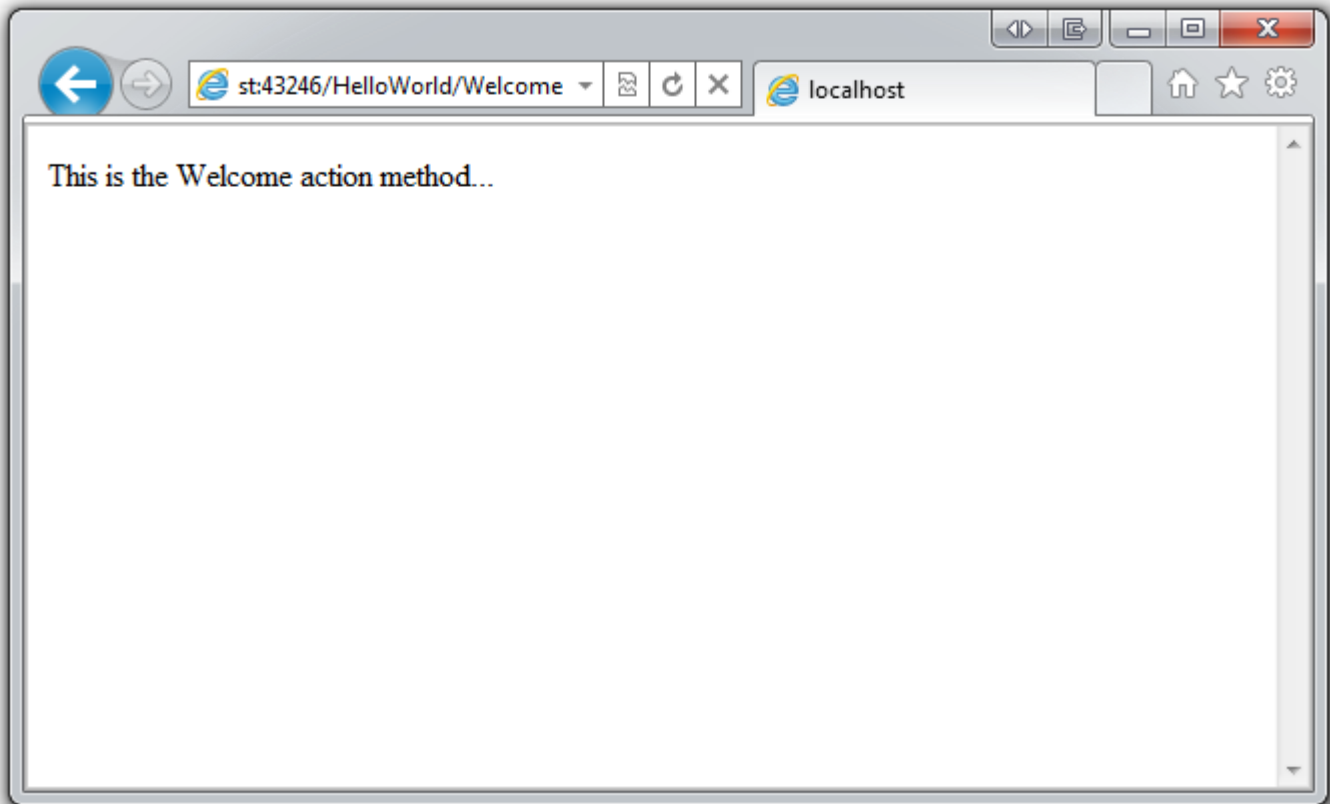
}
```

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.

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

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.



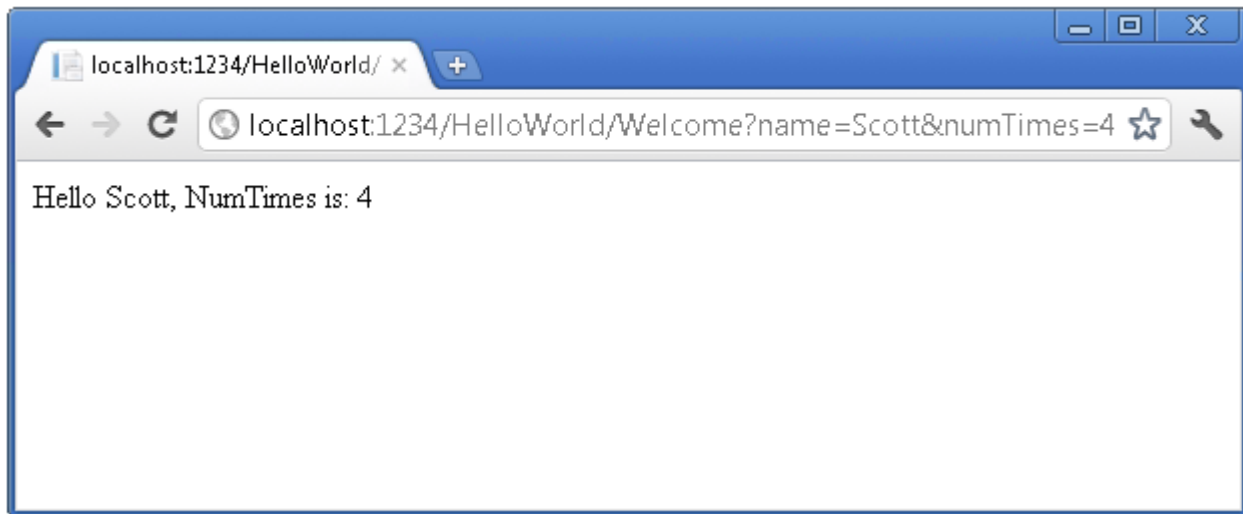
Let's modify the example slightly so that you can pass some parameter information from the URL to the controller (for example, `/HelloWorld/Welcome?name=Scott&numtimes=4`). Change your **Welcome** method to include two parameters as shown below. Note that the code uses the C# optional-parameter feature 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);  
}
```

Security Note: The code above uses `HttpServerUtility.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.

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..



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:

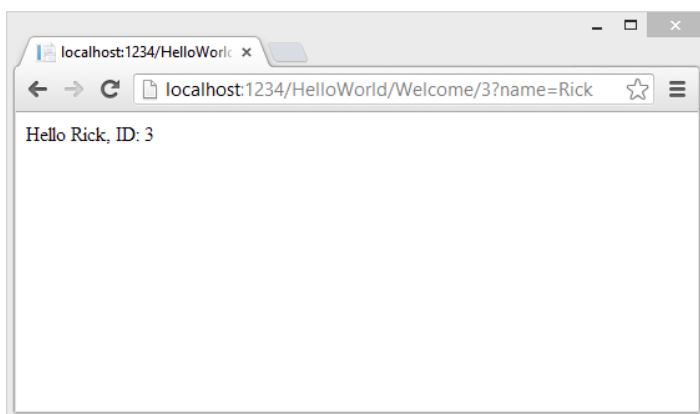
```
public string Welcome(string name, int ID = 1)

{

    return HttpUtility.HtmlEncode("Hello " + name + ", ID: " + ID);

}
```

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



Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

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}",
```

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

```
defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }

);

routes.MapRoute(

    name: "Hello",

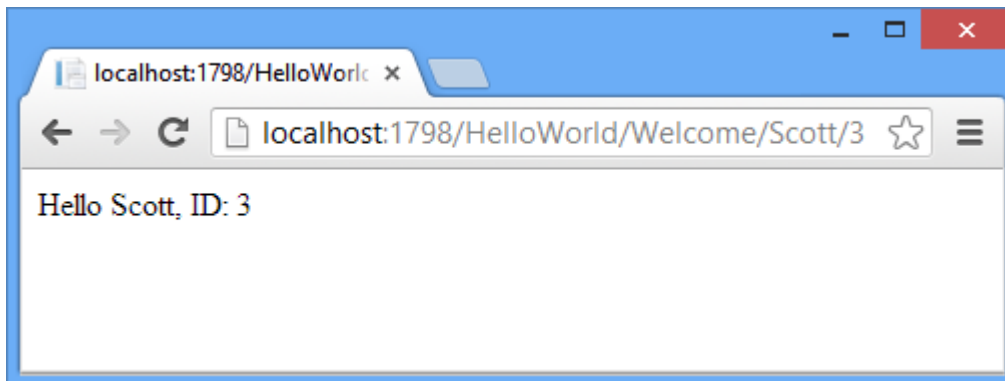
    url: "{controller}/{action}/{name}/{id}"

);

}

}
```

Run the application and browse to [/localhost:XXX/HelloWorld/Welcome/Scott/3](http://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.

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

Adding a “View”

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()

{

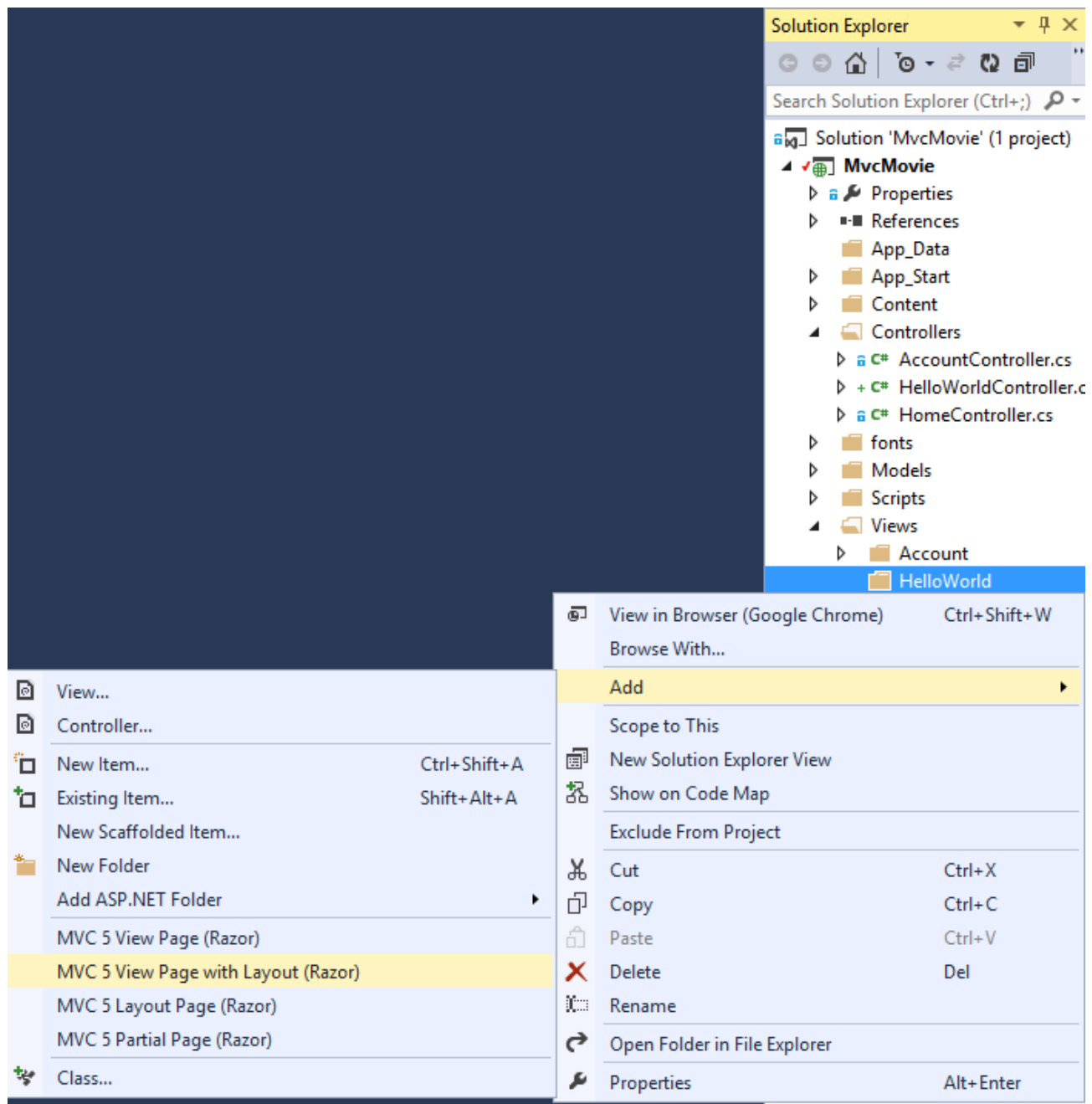
    return View();

}
```

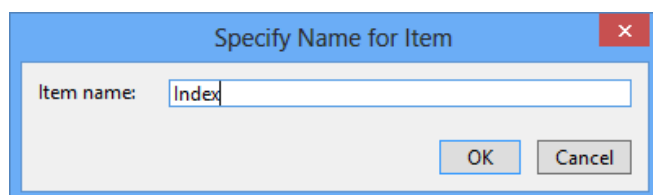
The `Index` method above uses a view template to generate an HTML response to the browser. Controller methods (also known as `action methods`), such as the `Index` method above, generally return an `ActionResult` (or a class derived from `ActionResult`), not primitive types like string.

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

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

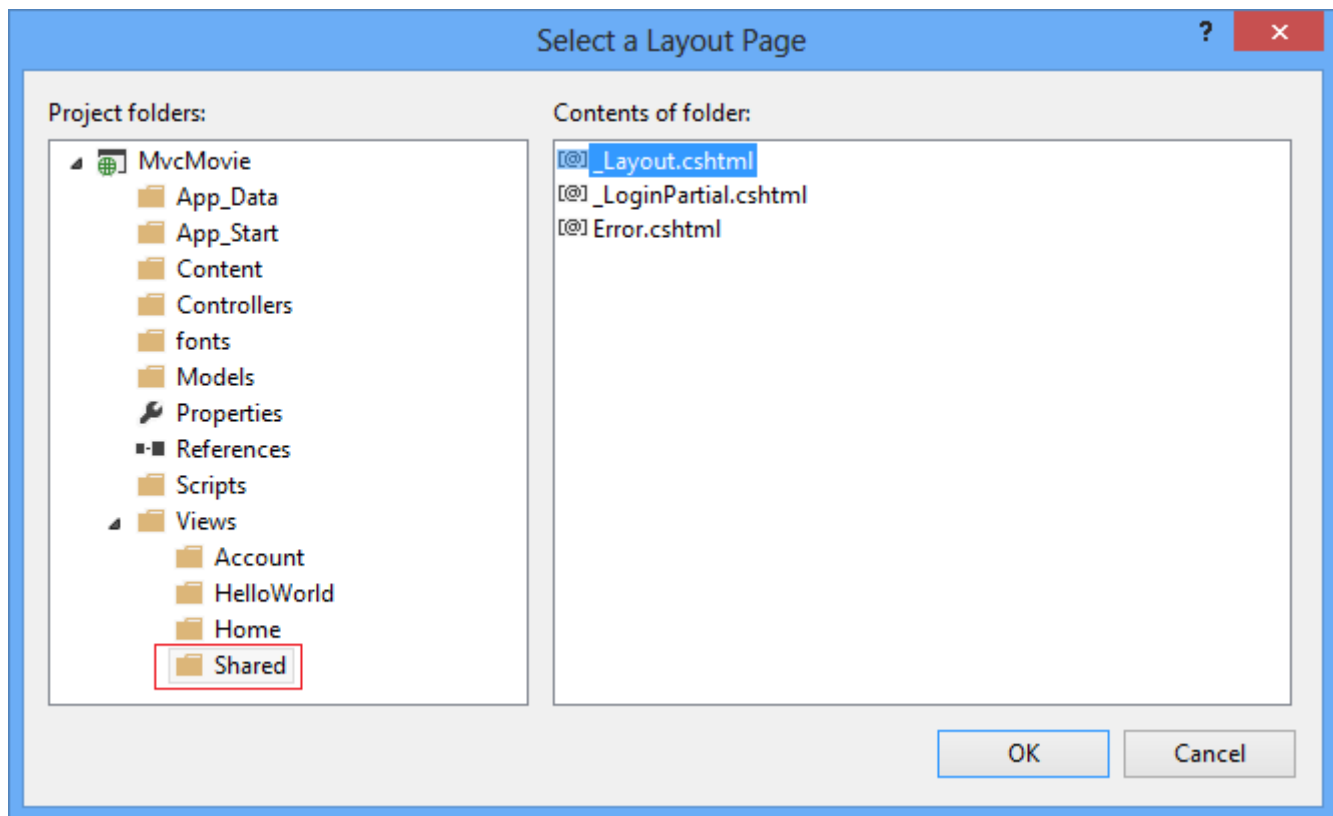


In the **Specify Name for Item** dialog box, enter *Index*, and then click **OK**.



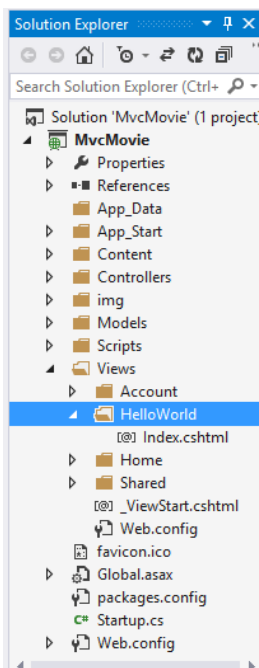
Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

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.



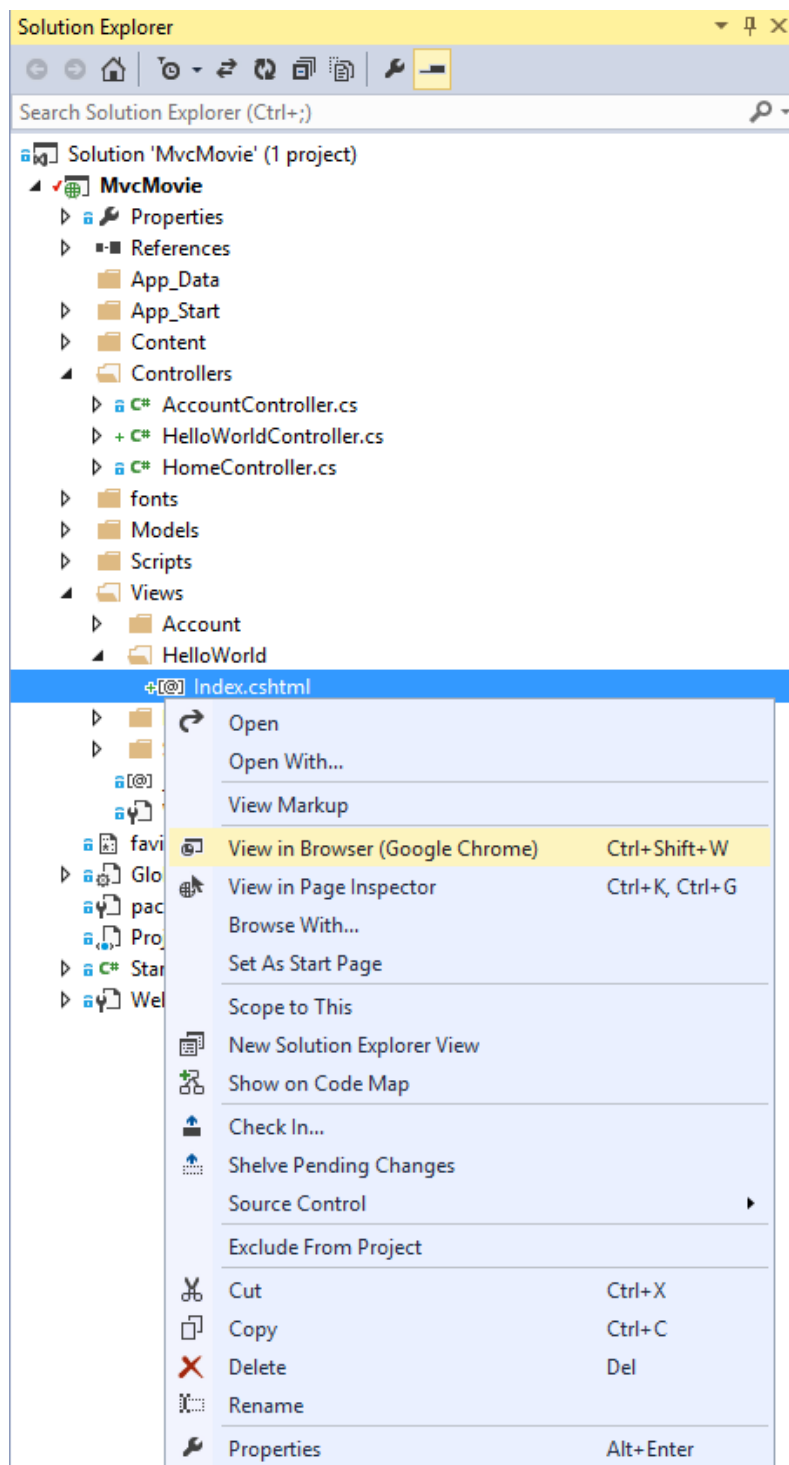
Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

Add the following highlighted markup.

```
@{  
  
    Layout = "~/Views/Shared/_Layout.cshtml";  
  
}  
  
@{  
  
    ViewBag.Title = "Index";  
  
}  
  
<h2>Index</h2>  
  
<p>Hello from our View Template!</p>
```


Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

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

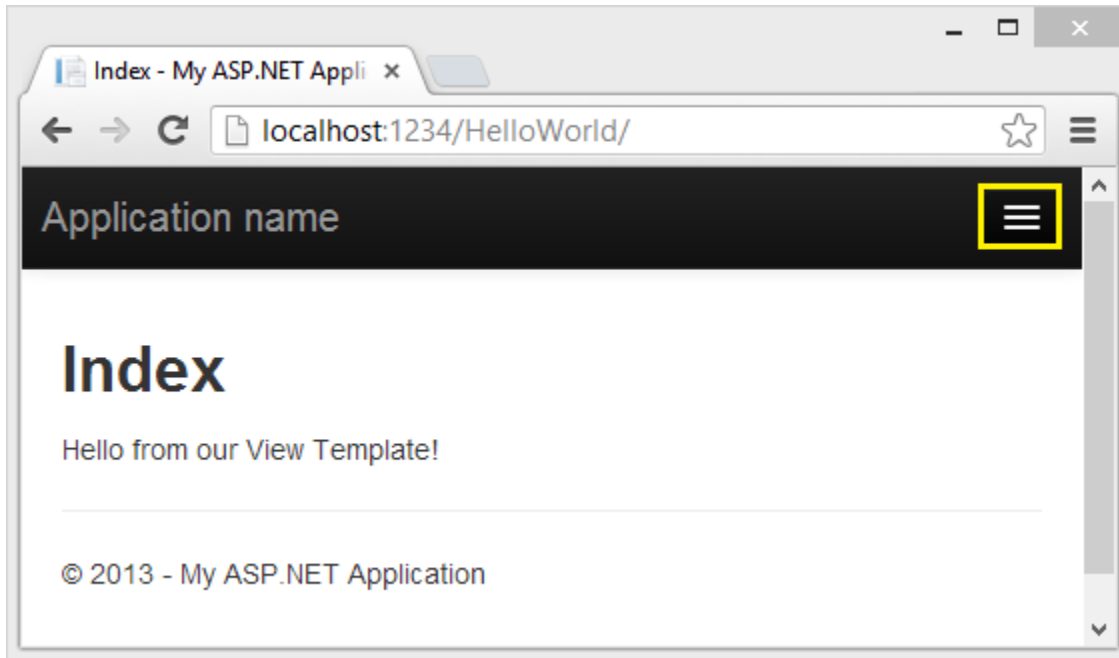


You can also right click the *Index.cshtml* file and select **View in Page Inspector**. See the [Page Inspector tutorial](#) for more information.

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.

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

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.

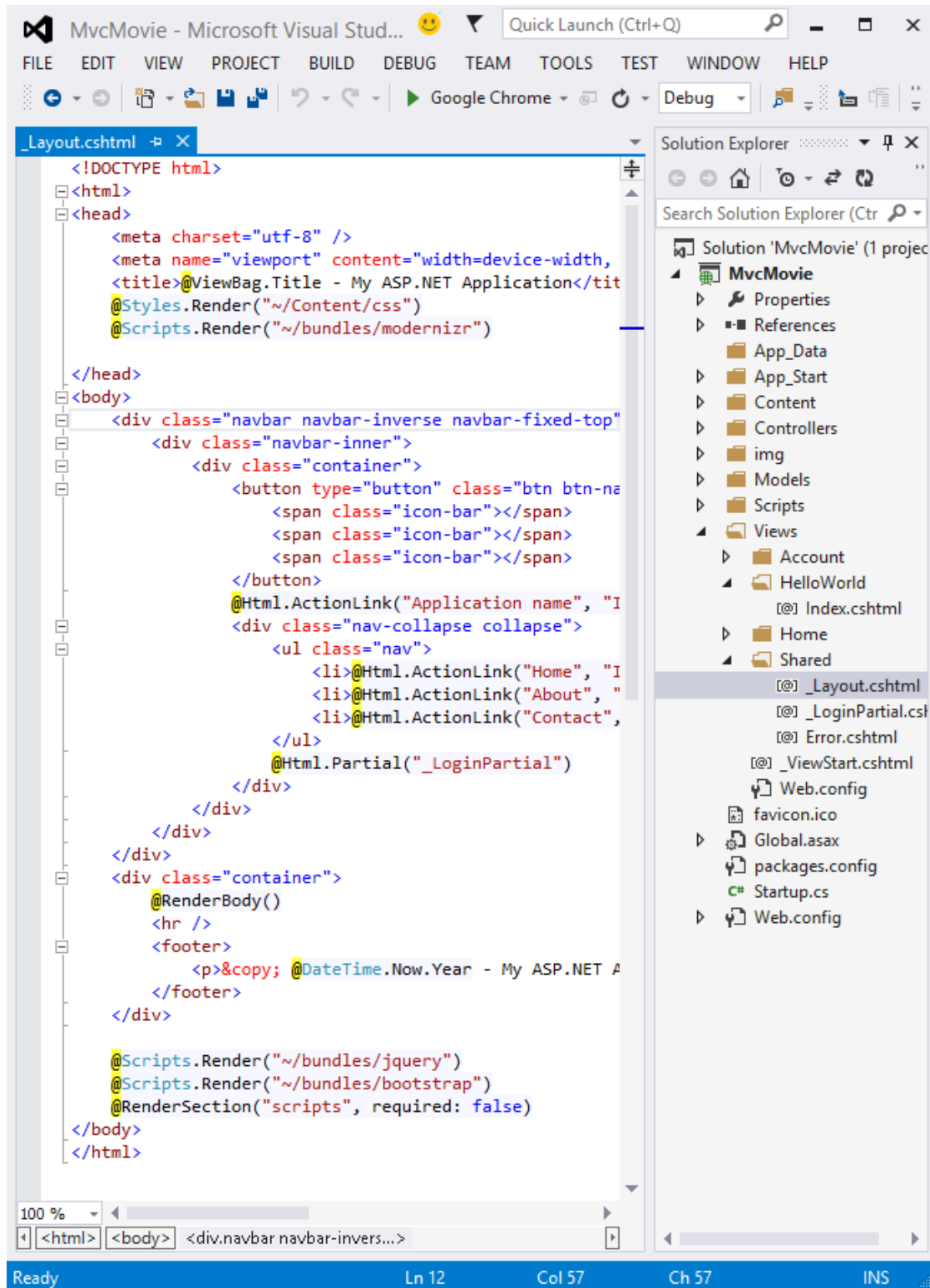


Looks pretty good. However, notice that the browser's title bar shows "Index My ASP.NET Appli" and the big link on the top of the page says "Application name." **Depending on how small you make your browser window**, you might need to click the three bars in the upper right to see the **Home, About,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

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

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.



Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

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>

            </div>

        </div>

    </div>

</body>

</html>
```

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

```
</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>

    @Html.Partial("_LoginPartial")

</div>

</div>

</div>

<div class="container body-content">

    @RenderBody()

    <hr />

    <footer>

        <p>&copy; @DateTime.Now.Year - My ASP.NET Application</p>

    </footer>

</div>
```

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

```
@Scripts.Render("~/bundles/jquery")

@Scripts.Render("~/bundles/bootstrap")

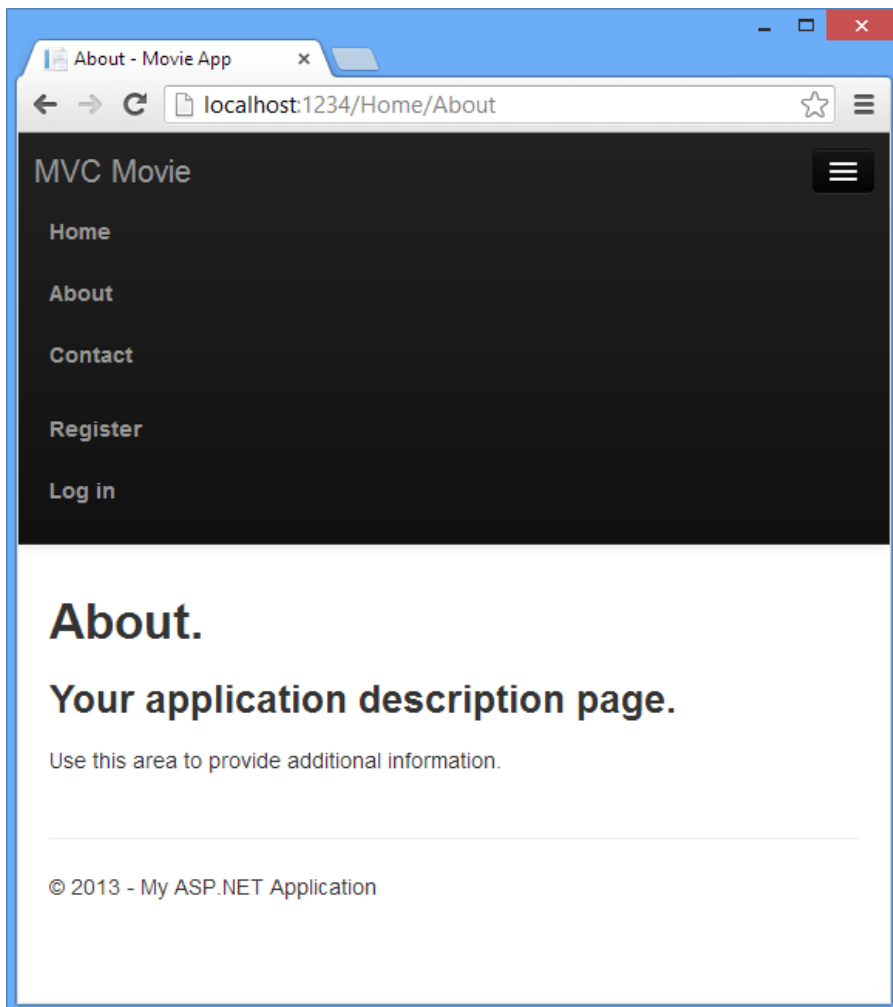
@RenderSection("scripts", required: false)

</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:

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

```
@{  
  
    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).

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..
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.

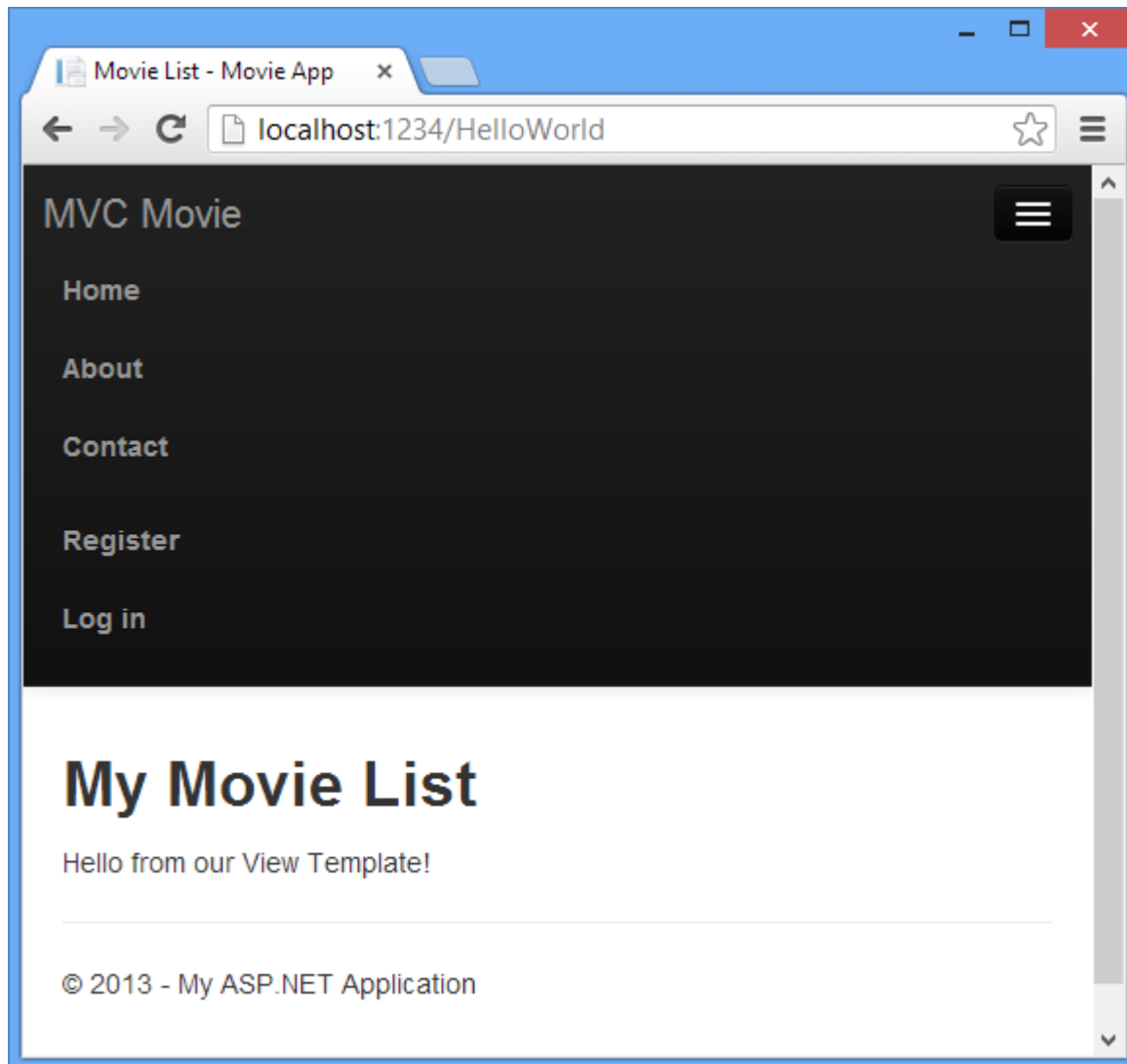
```
<!DOCTYPE html>  
  
<html>  
  
<head>  
  
    <meta charset="utf-8" />  
  
    <meta name="viewport" content="width=device-width, initial-scale=1.0">  
  
    <title>@ViewBag.Title - Movie App</title>  
  
    @Styles.Render("~/Content/css")  
  
    @Scripts.Render("~/bundles/modernizr")  
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

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

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 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.

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

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;

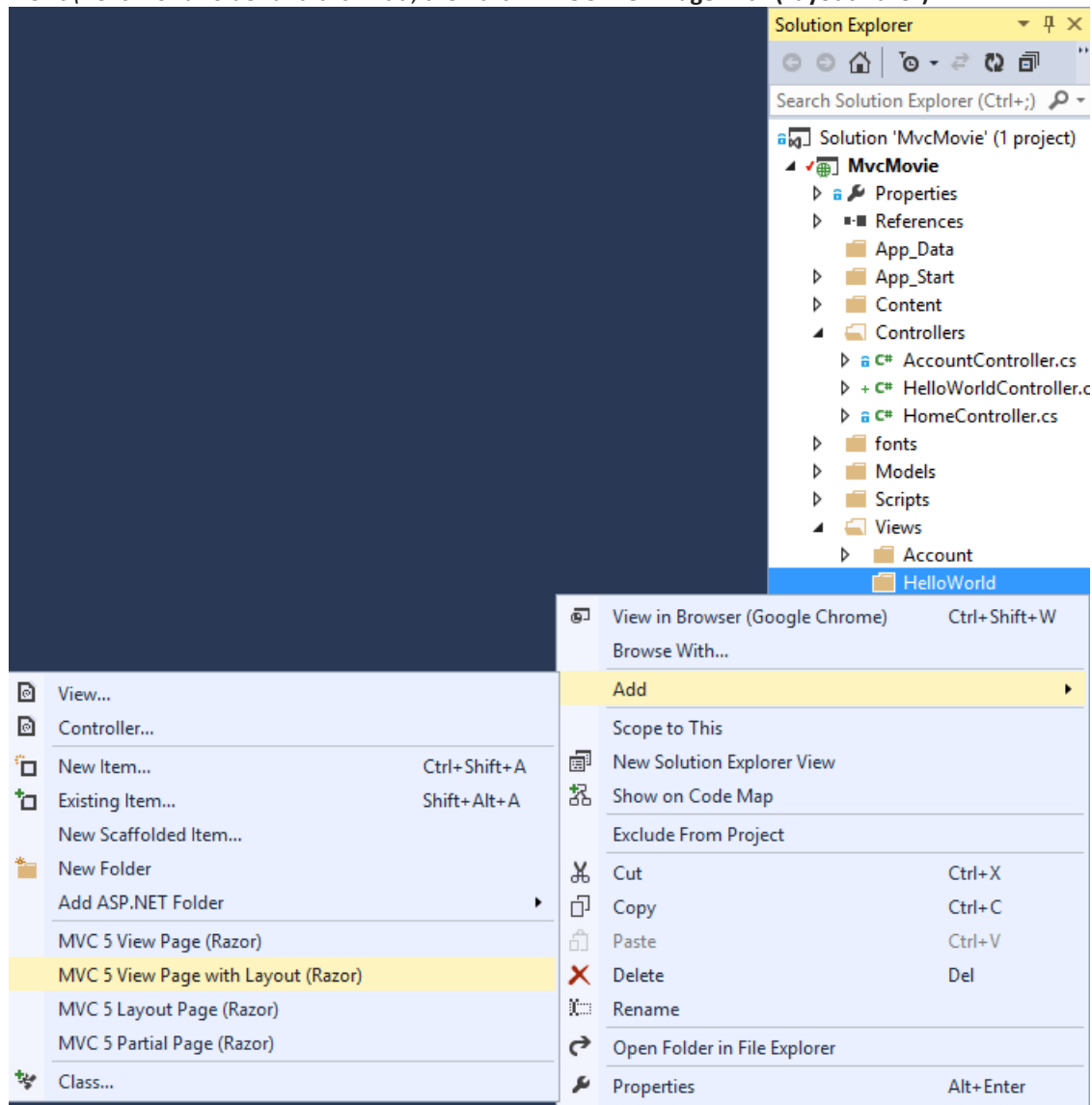
            ViewBag.NumTimes = numTimes;

            return View();
        }
    }
}
```

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

```
}  
  
}  
  
}
```

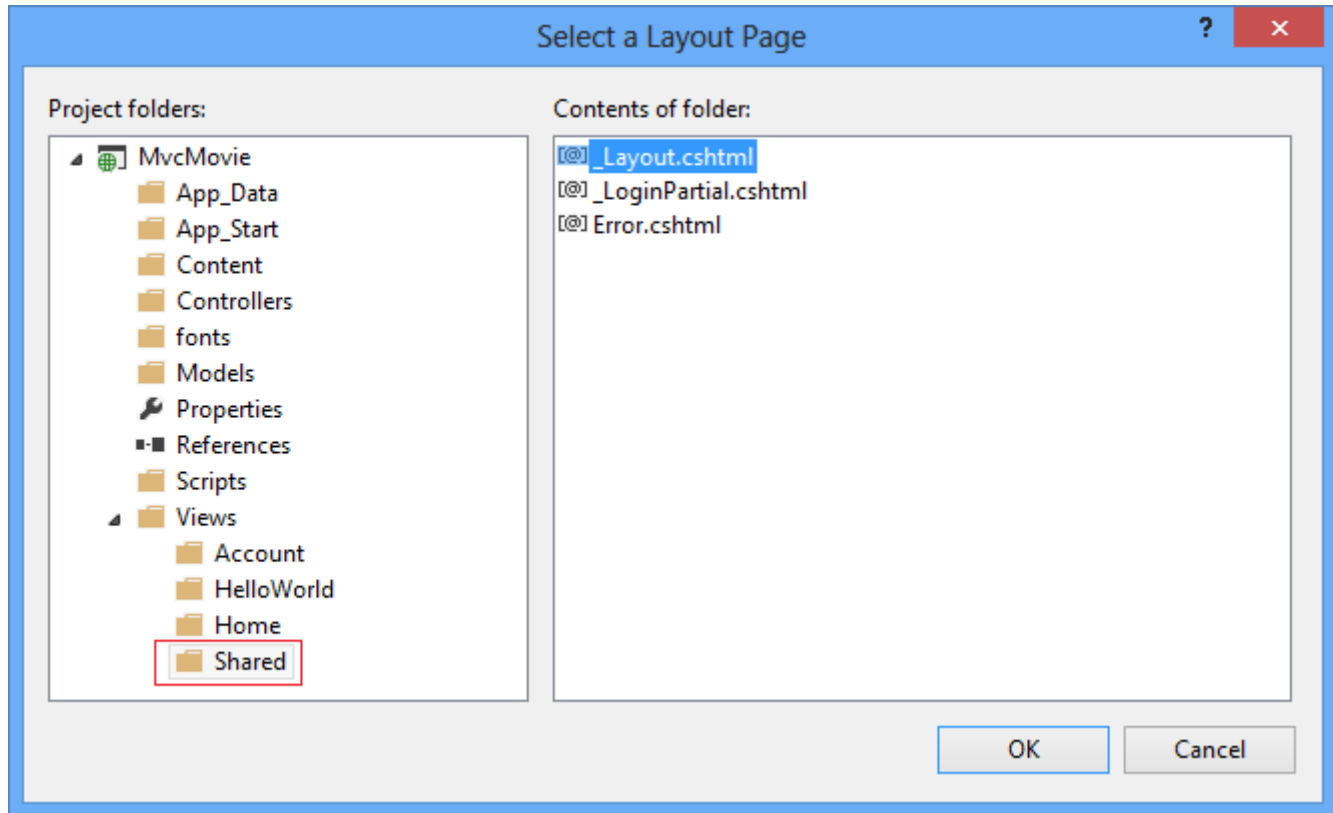
Now the **ViewBag** object contains data that will be passed to the view automatically. Next, you need a Welcome view template! In the **Build** menu, select **Build Solution** (or Ctrl+Shift+B) to make sure the project is compiled. Right click the **Views\HelloWorld** folder and click **Add**, then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



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

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

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>  
  
    }  
  
}
```

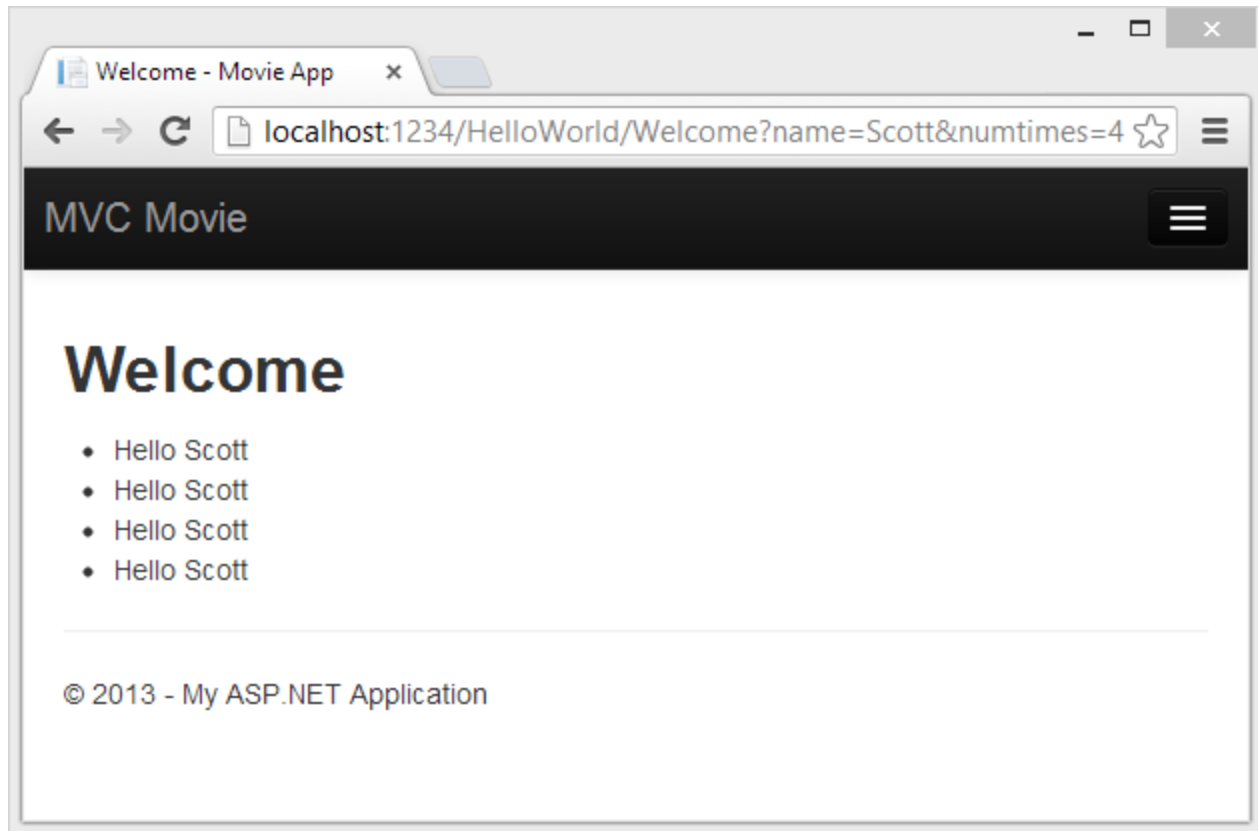
Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

```
}  
  
</ul>
```

Run the application and browse to the following URL:

[http://localhost:xx/HelloWorld/Welcome?name=Scott&numtimes=4](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. Latter 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.

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

Introduction to ASP.NET Web Programming Using the Razor Syntax (C#)

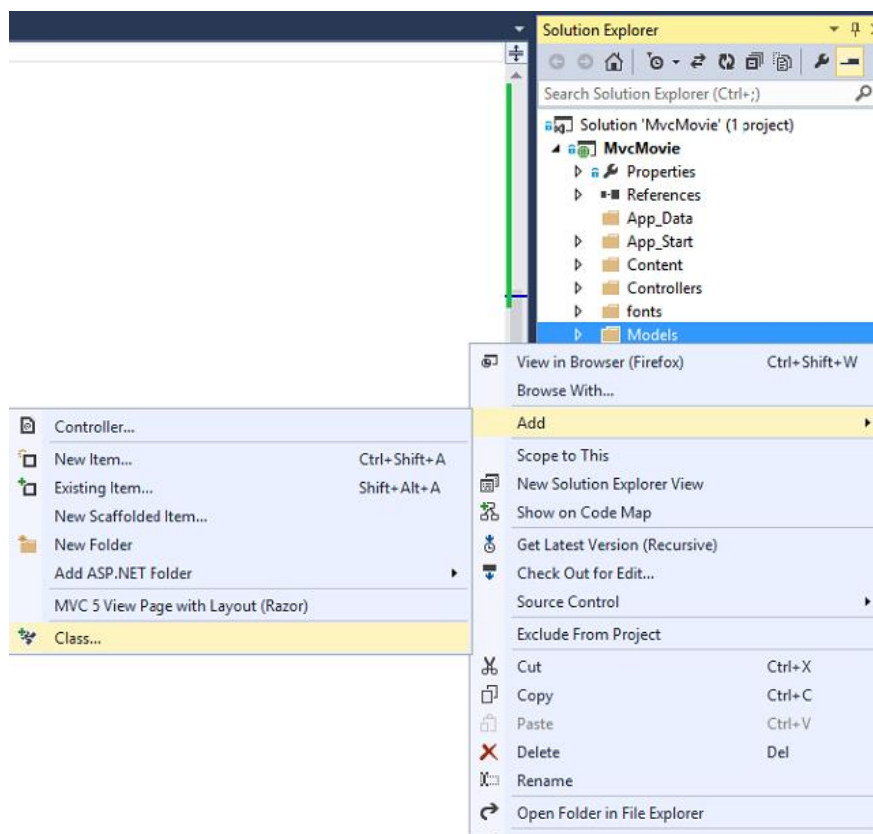
Adding a Model: [Create database with EF starts here...](#)

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".

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

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.

In the same file, add the following **MovieDbContext** class:

```
using System;

using System.Data.Entity;

namespace MvcMovie.Models

{

    public class Movie
```

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

```
{

    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; }

}

}
```

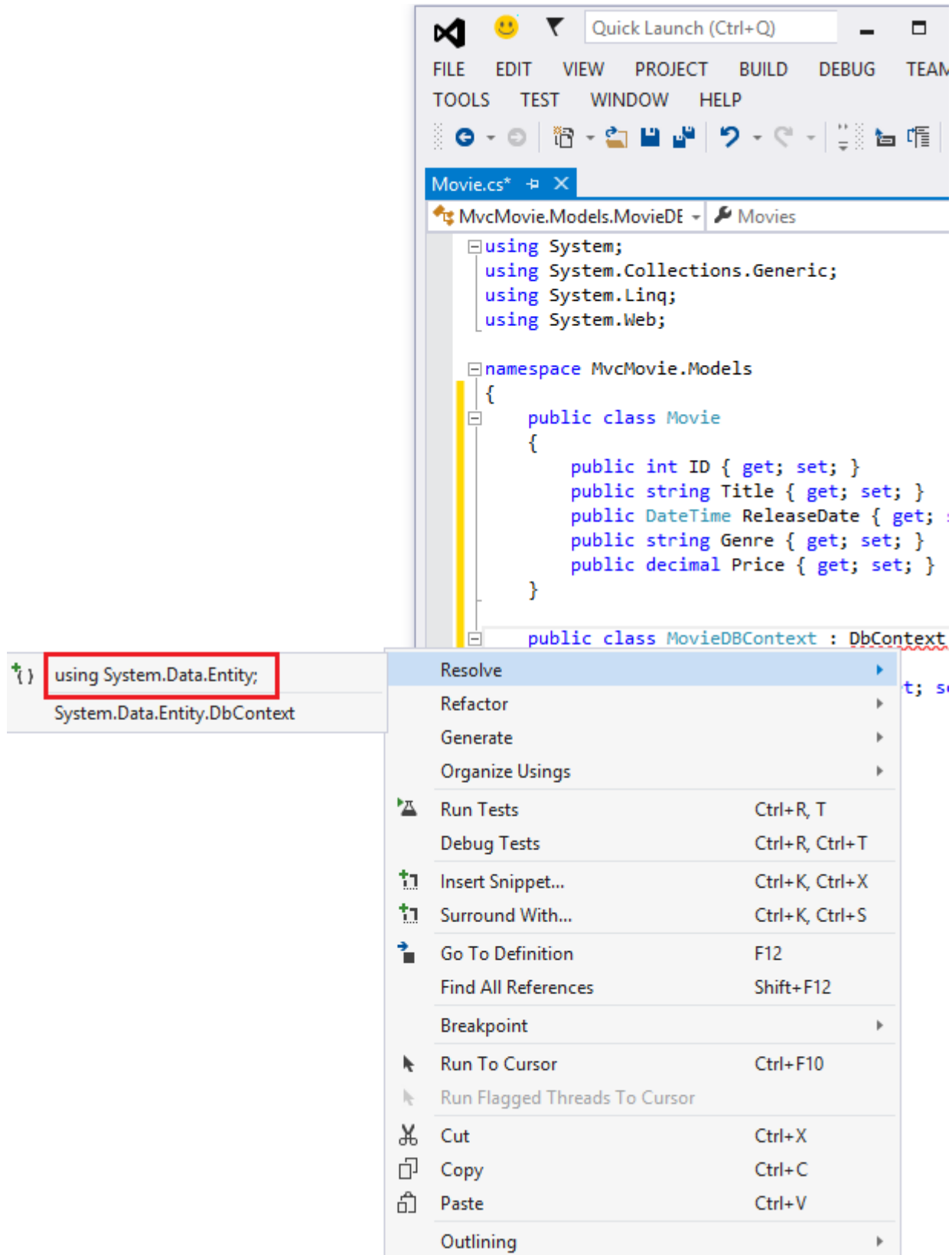
The **MovieDBContext** class represents the Entity Framework movie database context, which handles fetching, storing, and updating **Movie** class instances in a database. The **MovieDBContext** derives from the **DbContext** base class provided by the Entity Framework.

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;
```


Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

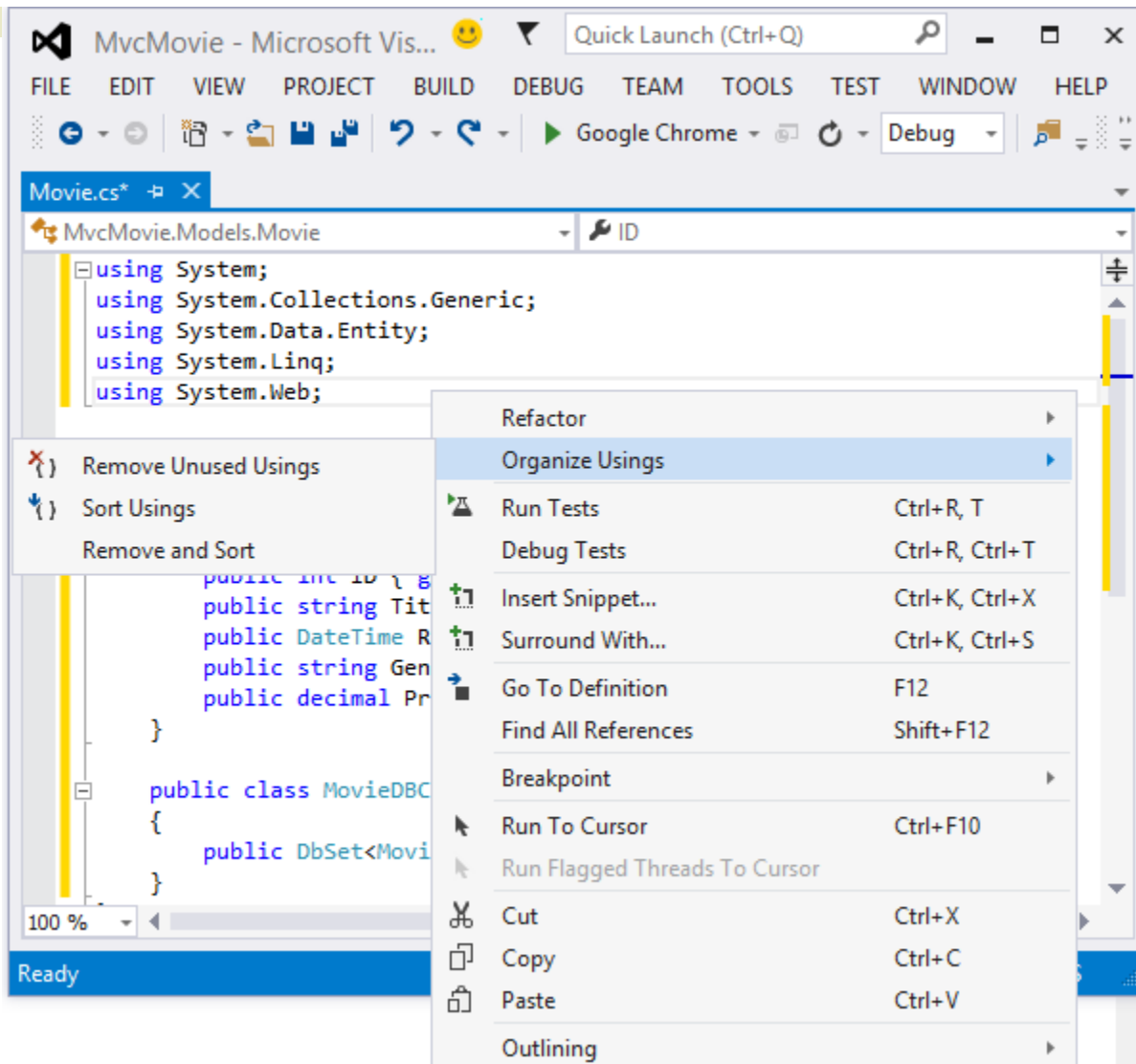
You can do this by manually adding the using statement, or you can right click on the red squiggly lines and click **Resolve**, and then click **using System.Data.Entity**.



Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

Note: Several unused `using` statements have been removed. You can do this by right clicking in the file, click **Organize**

Usings, and then 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.

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

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

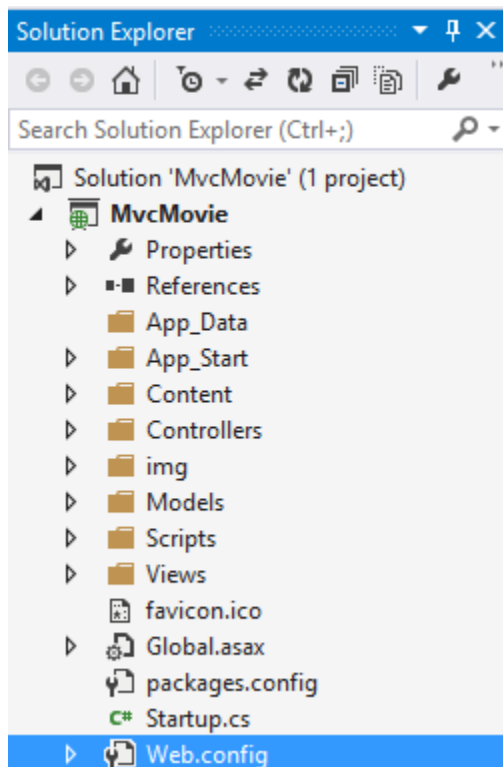
`LocalDB` is a lightweight version of the SQL Server Express Database Engine that starts on demand and runs in user mode. LocalDB runs in a special execution mode of SQL Server Express that enables you to work with databases as `.mdf` files. Typically, LocalDB database files are kept in the `App_Data` 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 2013 (and in 2012), 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.)



Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

Find the `<connectionStrings>` element:

```
<?xml version="1.0" encoding="utf-8"?>
<!--
  For more information on how to configure your ASP.NET application, please visit
  http://go.microsoft.com/fwlink/?LinkId=301880
-->
<configuration>
  <configSections>
    <!-- For more information on Entity Framework configuration, visit http://go
    <section name="entityFramework" type="System.Data.Entity.Internal.ConfigFile
  </configSections>
  <connectionStrings>
    <add name="DefaultConnection" connectionString="Data Source=(LocalDb)\v11.0;
      providerName="System.Data.SqlClient" />
  </connectionStrings>
  <appSettings>
    <add key="webpages:Version" value="3.0.0.0" />
    <add key="webpages:Enabled" value="false" />
    <add key="PreserveLoginUrl" value="true" />
    <add key="ClientValidationEnabled" value="true" />
    <add key="UnobtrusiveJavaScriptEnabled" value="true" />
  </appSettings>
  <system.web>
```

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

```
<add name="MovieDBContext"

  connectionString="Data Source=(LocalDB)\v11.0;AttachDbFilename=|DataDirectory|\Movies.mdf;Integrated
Security=True"

  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)\v11.0;AttachDbFilename=|DataDirectory|\aspnet-MvcMovie-20130603030321.mdf;Initial Catalog=aspnet-
MvcMovie-20130603030321;Integrated Security=True" providerName="System.Data.SqlClient" />

  <add name="MovieDBContext" connectionString="Data
Source=(LocalDB)\v11.0;AttachDbFilename=|DataDirectory|\Movies.mdf;Integrated Security=True"
providerName="System.Data.SqlClient" />
```

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

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 [Deploy a Secure ASP.NET MVC app with Membership, OAuth, and SQL Database to a Windows Azure Web Site](#).

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; }
    }
}
```

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

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 user's 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.

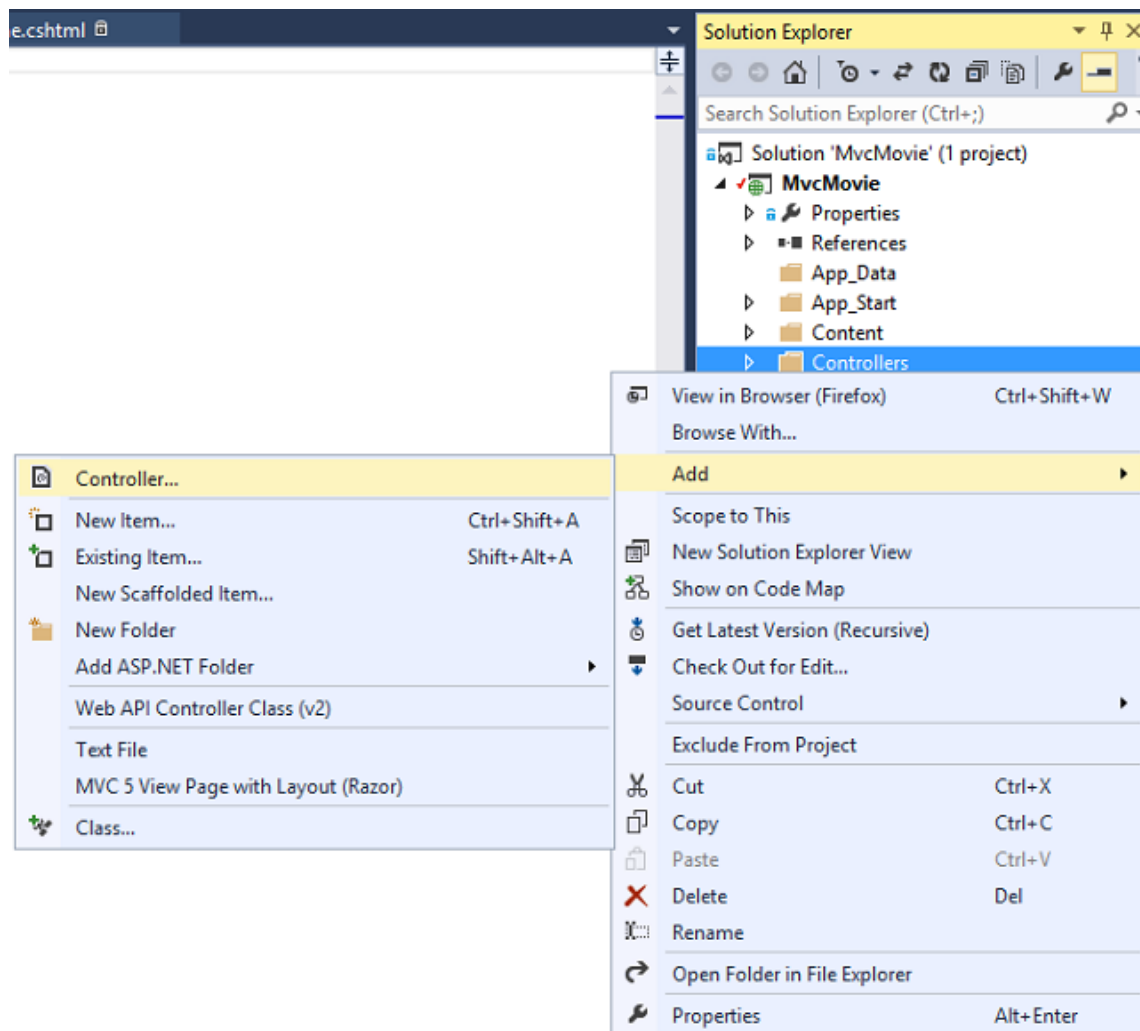
Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

Accessing Your Model's Data from a Controller

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.

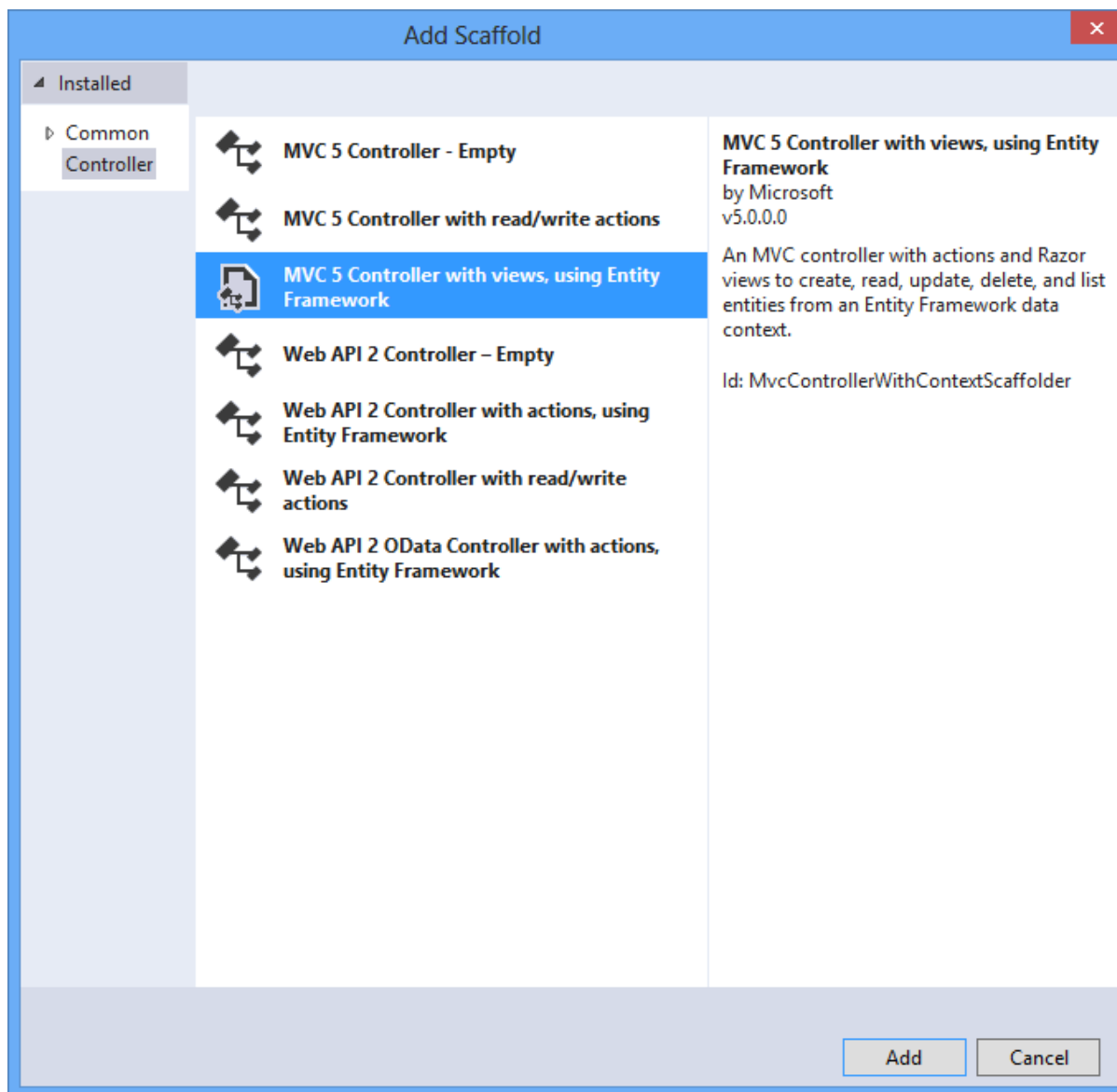
Build the application before going on to the next step. If you don't build the application, you'll get an error adding a controller.

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



Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

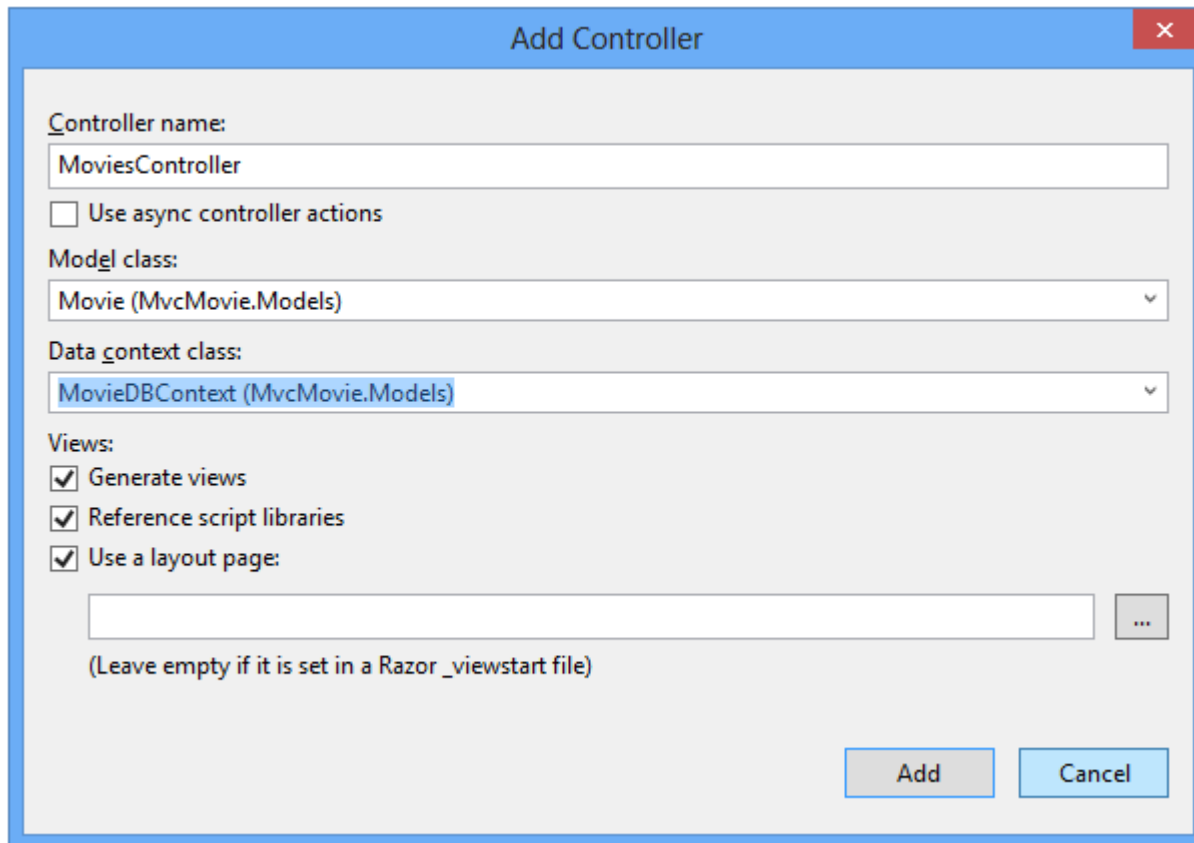
In the **Add Scaffold** dialog box, click **MVC 5 Controller with views, using Entity Framework**, and then click **Add**.



Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

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

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:

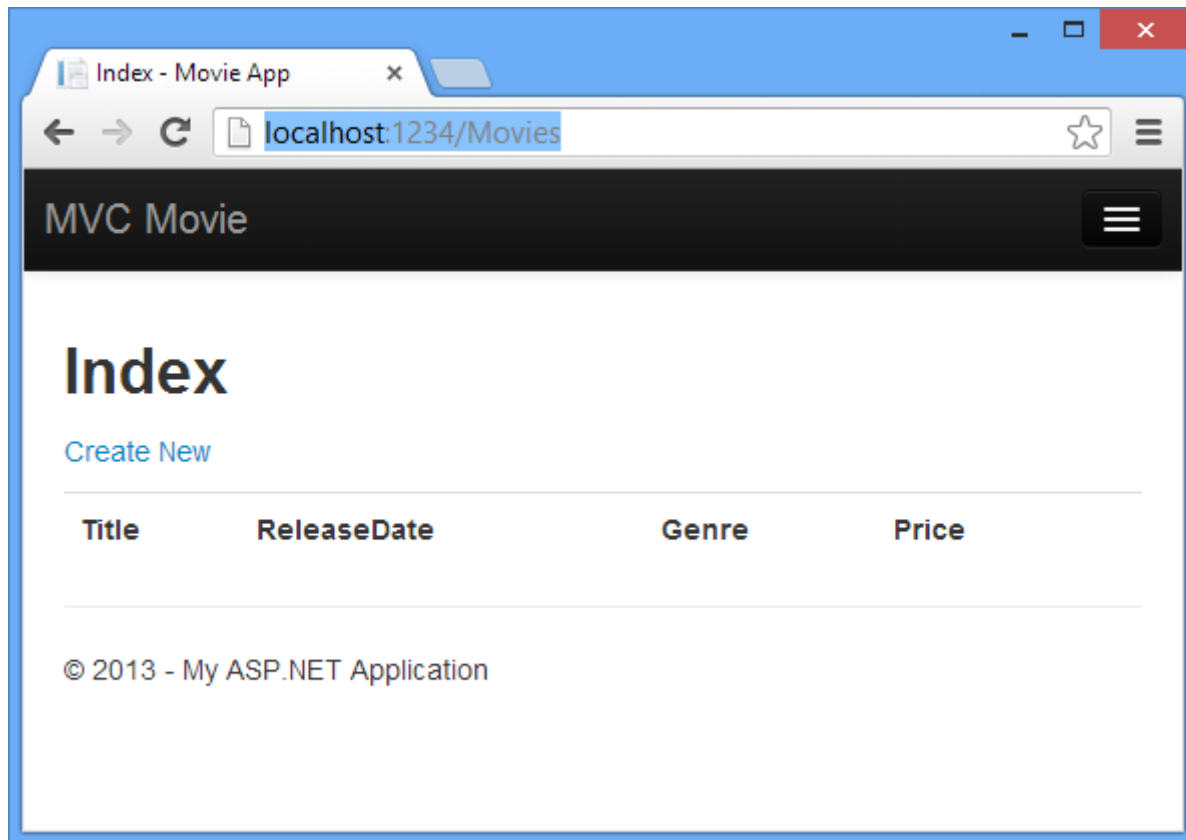
- A *MoviesController.cs* file in the *Controllers* folder.
- A *Views\Movies* folder.
- *Create.cshtml*, *Delete.cshtml*, *Details.cshtml*, *Edit.cshtml*, and *Index.cshtml* in the new *Views\Movies* folder.

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.

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

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).

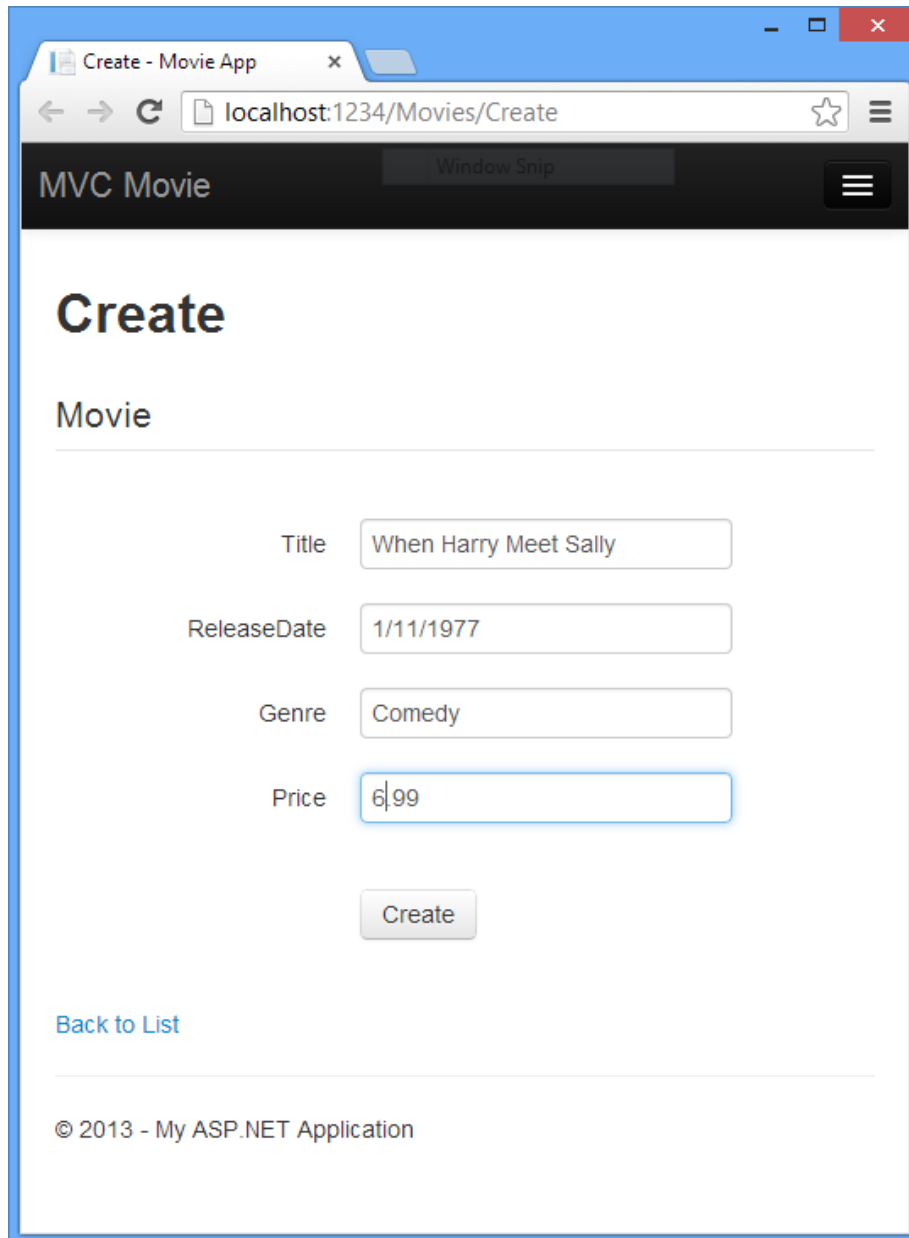
Because the application is relying on the default routing (defined in the *App_Start\RouteConfig.cs* file), the browser request *http://localhost:xxxxx/Movies* is routed to the default **Index** action method of the **Movies** controller. In other words, the browser request *http://localhost:xxxxx/Movies* 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.



Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

Creating a Movie

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

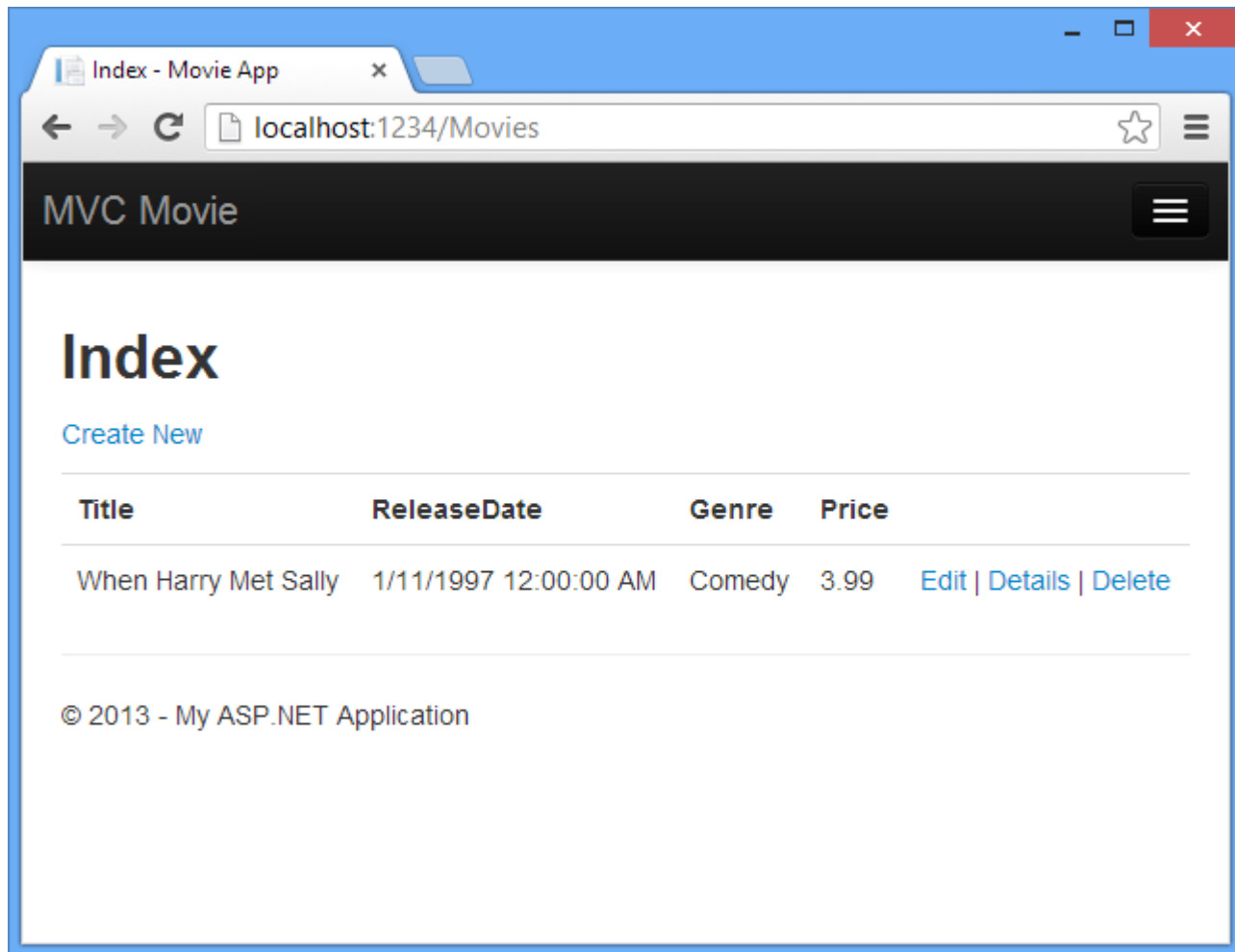


The screenshot shows a web browser window with the title 'Create - Movie App' and the address bar displaying 'localhost:1234/Movies/Create'. The page has a dark header with 'MVC Movie' and a 'Window Snip' button. The main content area is titled 'Create' and 'Movie'. It contains four input fields: 'Title' with the value 'When Harry Meet Sally', 'ReleaseDate' with '1/11/1977', 'Genre' with 'Comedy', and 'Price' with '6,99'. A 'Create' button is located below the fields. At the bottom, there is a 'Back to List' link and a copyright notice '© 2013 - My ASP.NET Application'.

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.

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

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.

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

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();
```

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

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);

}
```

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

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);
```

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

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>
```


Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

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>
```

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

```
@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", { id=item.ID }) |

    @Html.ActionLink("Delete", "Delete", { id=item.ID })

</td>

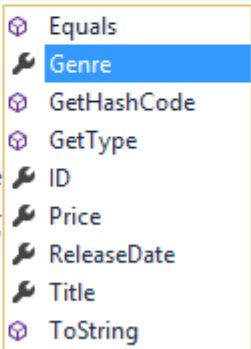
</tr>

}
```

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:

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

```
@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.)  
        </td>  
        <td>  
            @Html.DisplayFor(modelItem => item.  
        </td>  
        <td>  
            @Html.ActionLink("Edit", "Edit", ne  
            @Html.ActionLink("Details", "Detail  
            @Html.ActionLink("Delete", "Delete"  
        </td>  
    </tr>  
}  
  
</table>
```

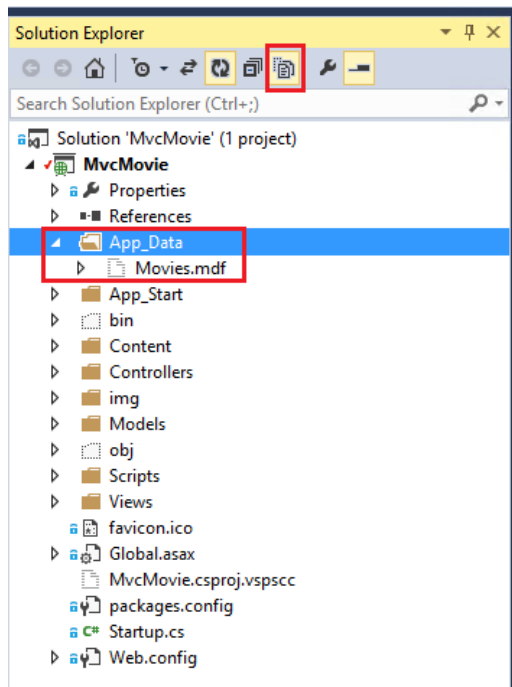


- Equals
- Genre
- GetHashCode
- GetType
- ID
- Price
- ReleaseDate
- Title
- ToString

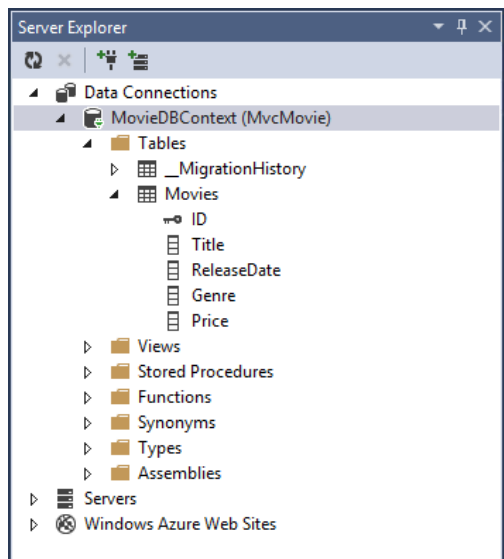
Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

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.

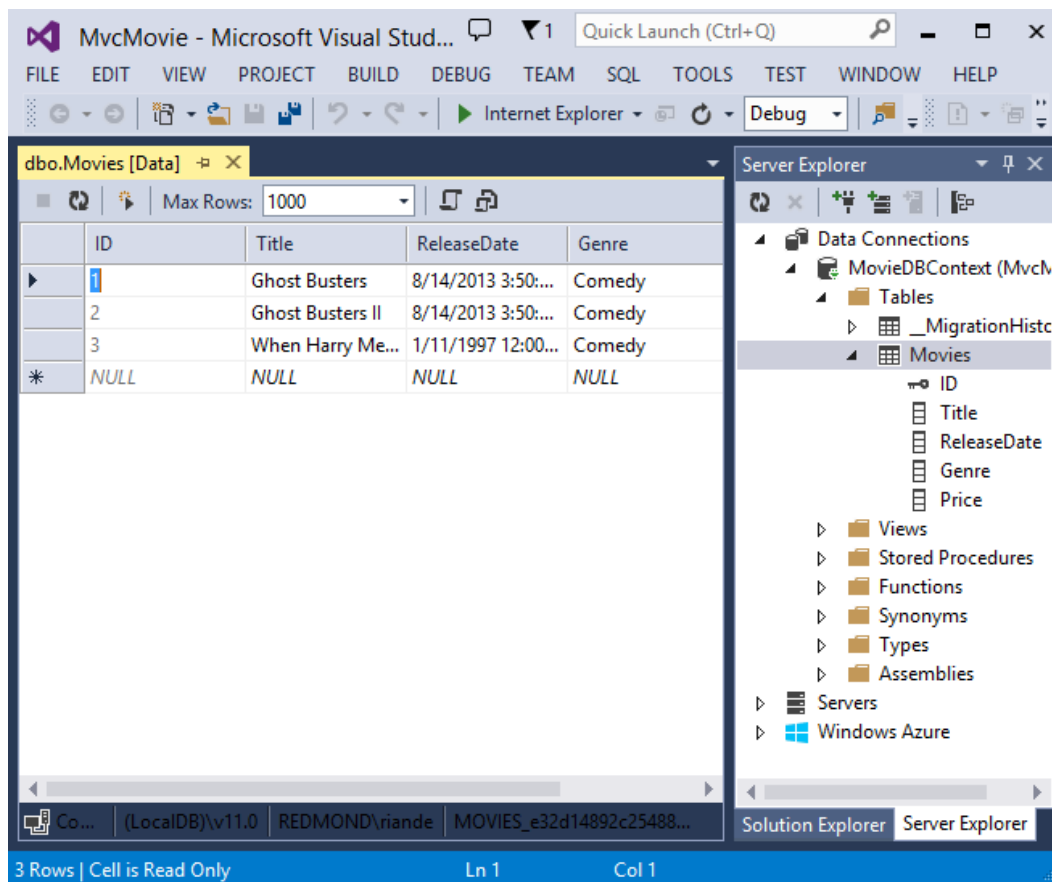
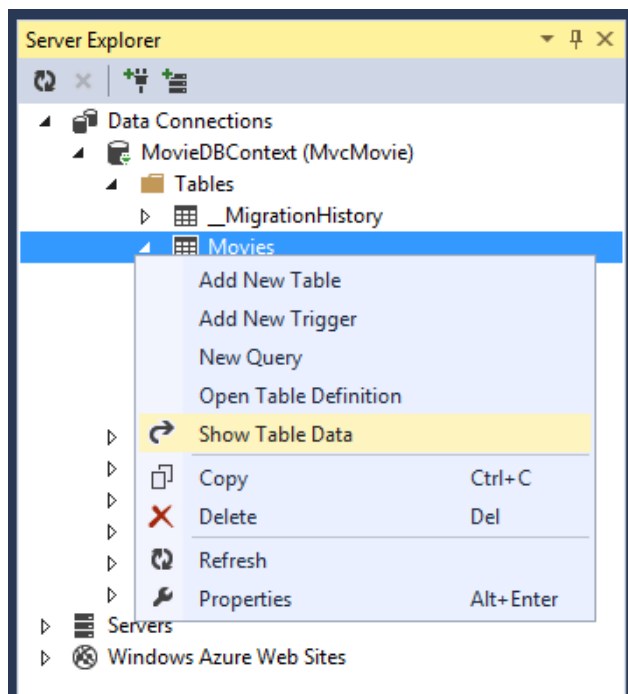


Double-click *Movies.mdf* to open **SERVER EXPLORER**, then expand the **Tables** folder to see the **Movies** table. Note the key icon next to ID. By default, EF will make a property named ID the primary key. For more information on EF and MVC, see Tom Dykstra's excellent tutorial on [MVC and EF](#).



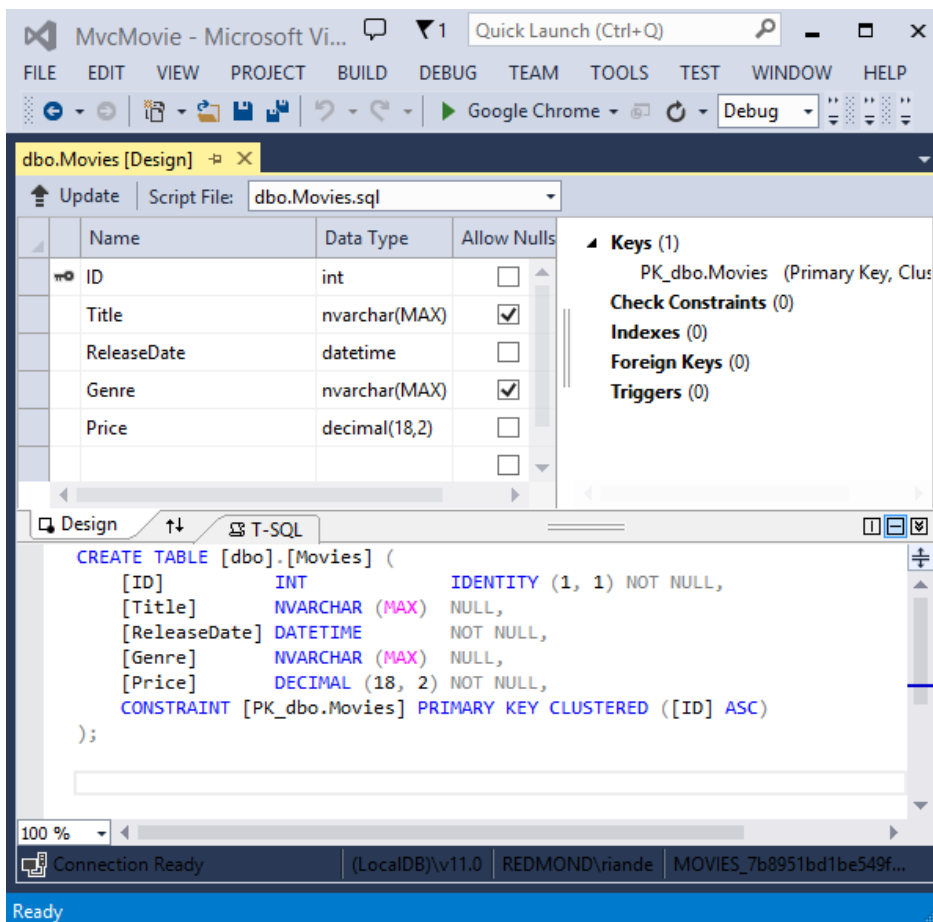
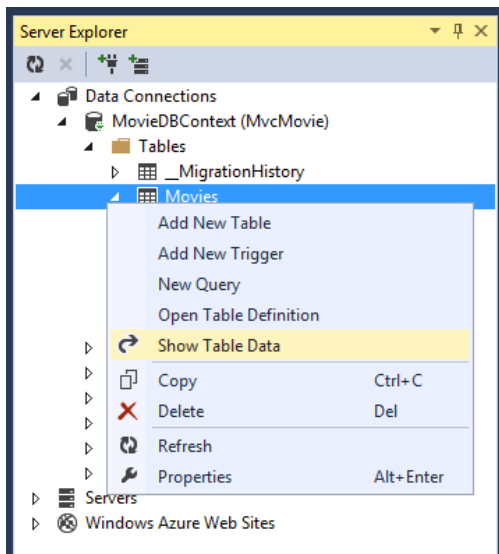
Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

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



Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

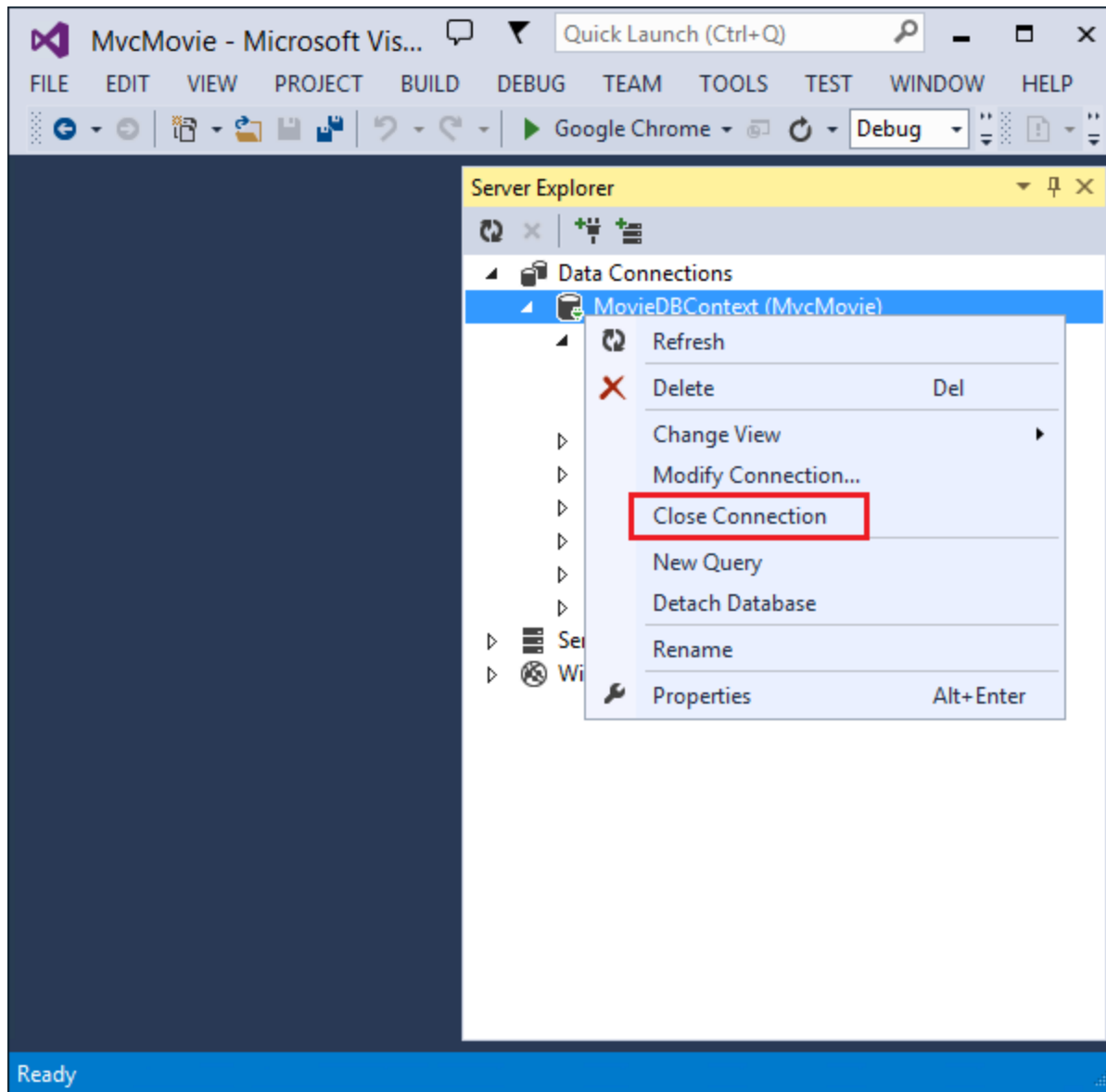
Right-click the **Movies** table and select **Open Table Definition** to see the table structure that Entity Framework Code First created for you.



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.

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

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](#).

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

Examining the Edit Methods and Edit View:

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; }
    }

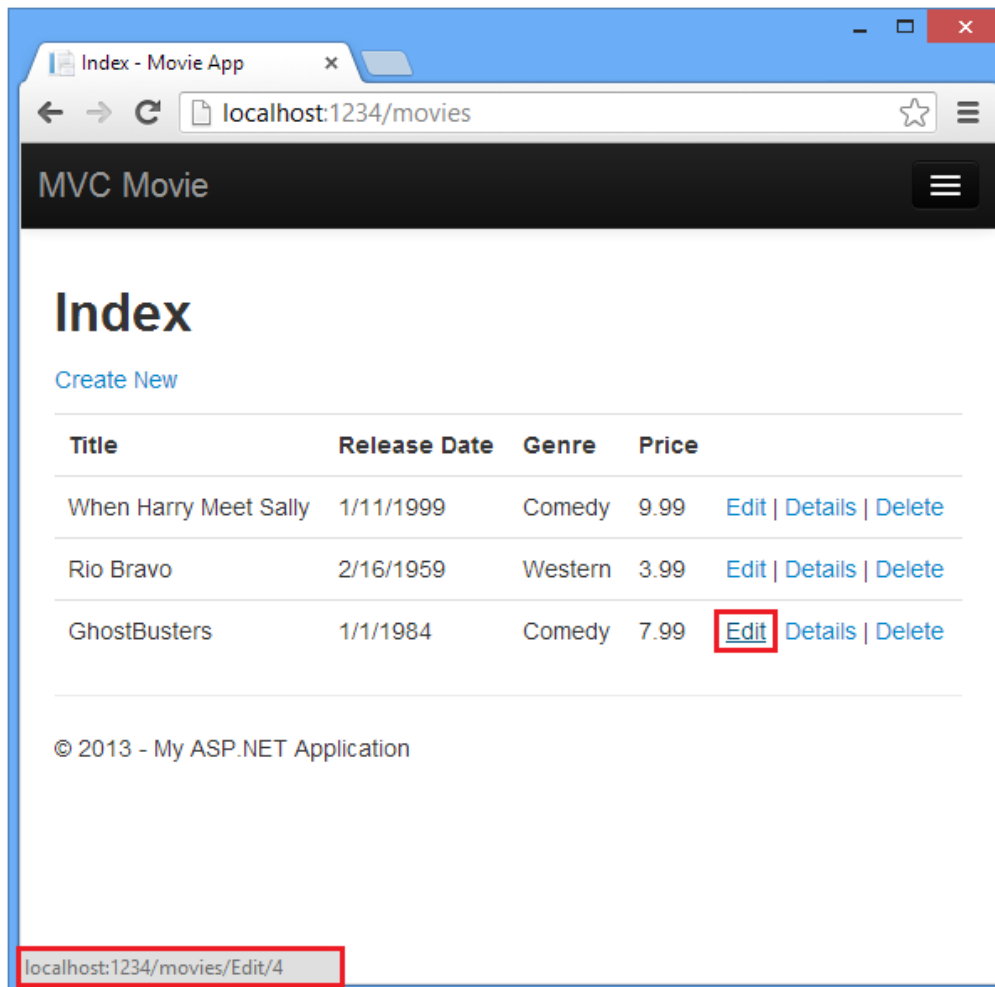
    public class MovieDbContext : DbContext
```


Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

```
{  
  
    public DbSet<Movie> Movies { 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:

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

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

```
<td>
@Html.ActionLink("Edit Me", "Edit", new { id=item.ID }) |
@Html. (extension) MvcHtmlString HtmlHelper.ActionLink(string linkText, string actionName, object routeValues)
@Html. Returns an anchor element (a element) that contains the virtual path of the specified action.

Exceptions:
    System.ArgumentException
```

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`). 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 }
    );
}
```

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

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.

Firefox

Edit - Movie App

localhost:1234/movies/Edit?id=3

MVC Movie

Edit

Movie

Title

Release Date

Genre

Price

[Back to List](#)

© 2013 - My ASP.NET Application

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

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]
```

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

```
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 that 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>
```

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

```
@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>

}
```

`@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.

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

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>
}
```

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

```
</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 {
```


Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

```
@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_qwKvcojNXEEfcfn-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>
```

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

```
<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>
```

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

```
<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);

}
```

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

```
}
```

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.

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

The [Html.ValidationMessageFor](#) helpers in the *Edit.cshtml* view template take care of displaying appropriate error messages.

Firefox

Edit - Movie App

localhost:1234/movies/Edit?id=3

MVC Movie

Edit

Movie

Title

Release Date
The field Release Date must be a date.

Genre

Price
The field Price must be a number.

[Back to List](#)

© 2013 - My ASP.NET Application

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.

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

Search:

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.

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

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:

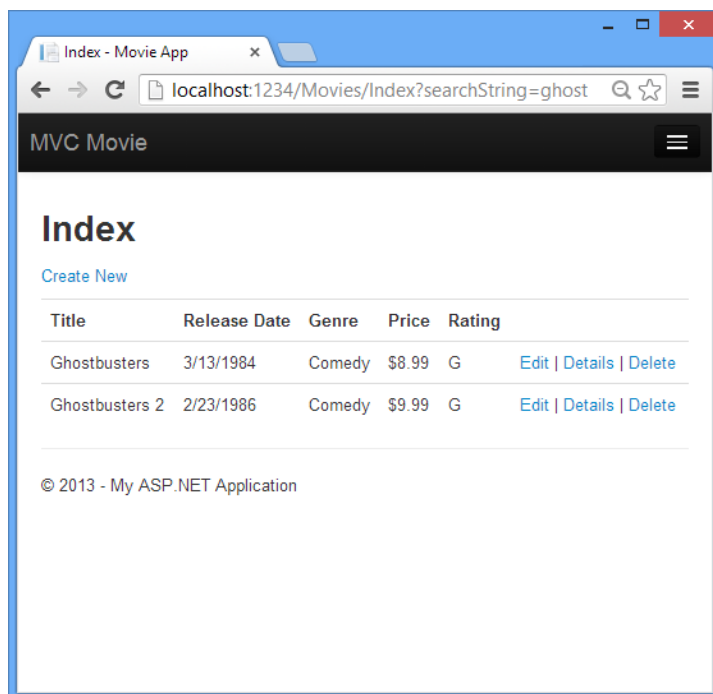
```
if (!String.IsNullOrEmpty(searchString))
{
    movies = movies.Where(s => s.Title.Contains(searchString));
}
```

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.



Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

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

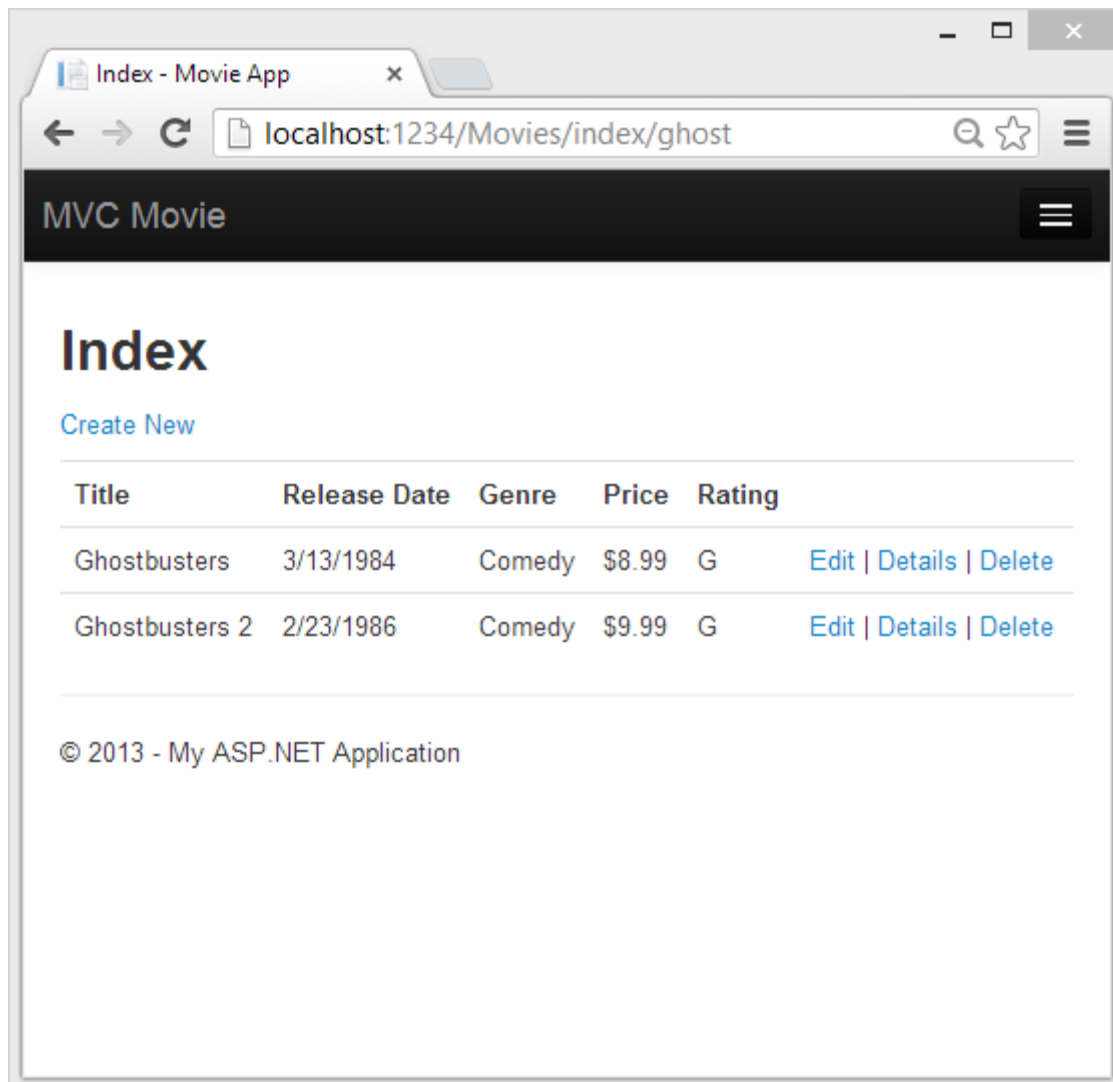
        select m;
```


Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

```
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**.



Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

However, you can't expect users to modify the URL every time they want to search for a movie. So now you'll add UI to help them filter movies. If you changed the signature of the `Index` method to test how to pass the route-bound ID parameter, change it back so that 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);

}
```

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

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>

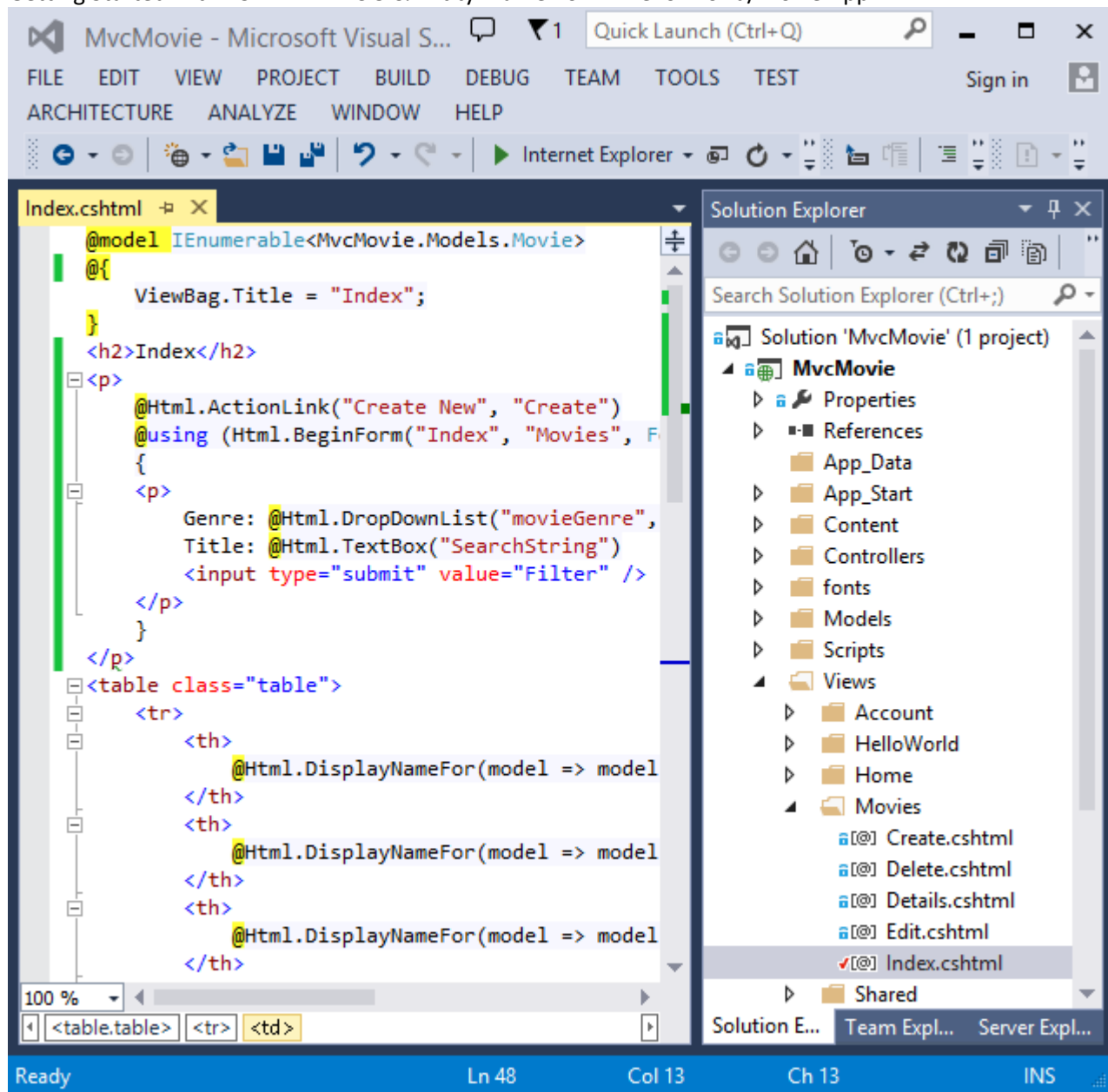
    }

</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.

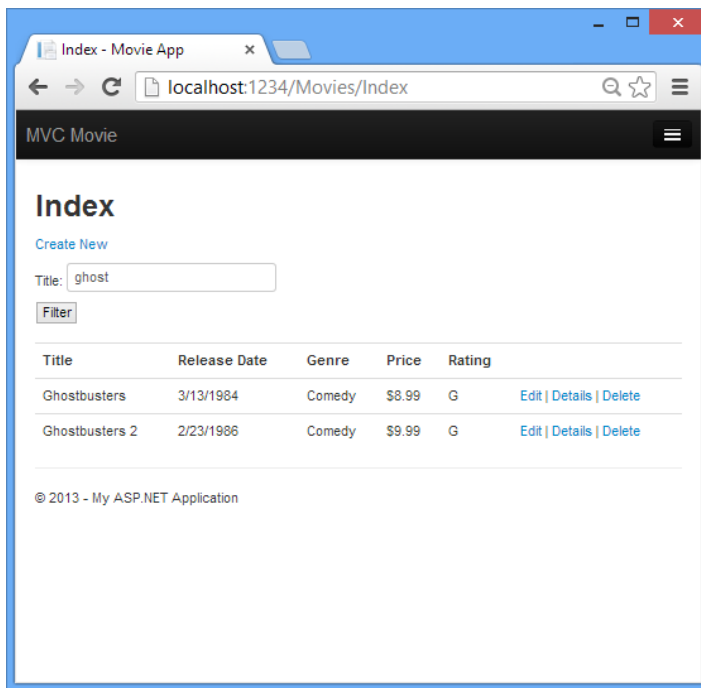
Visual Studio 2013 has a nice improvement when displaying and editing View files. When you run the application with a view file open, Visual Studio 2013 invokes the correct controller action method to display the view.

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..



With the Index view open in Visual Studio (as shown in the image above), tap Ctr F5 or F5 to run the application...

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..
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.

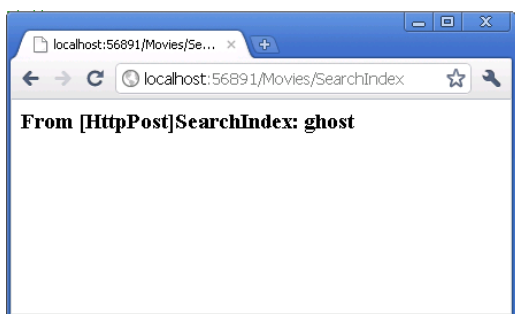
[HttpPost]

```
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.

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:

```
@Html.ActionLink("Create New", "Create")
using (Html.BeginForm("SearchIndex", "Movies", FormMethod.Get))
{
    <p>
    <input type="text" value="Search" />
    <input type="submit" value="Search" />
    </p>
    <div>
        @foreach (var movie in ViewBag.Movies)
        {
            <div>
                @Html.ActionLink(movie.Title, "Details", "Movies", new { id = movie.Id })
            </div>
        }
    </div>
}
</div>
```

Index - Movie App

localhost:1234/Movies?SearchString=ghost

MVC Movie

Index

[Create New](#)

Title:

Title	Release Date	Genre	Price	Rating	
Ghostbusters	3/13/1984	Comedy	\$8.99	G	Edit Details Delete
Ghostbusters 2	2/23/1986	Comedy	\$9.99	G	Edit Details Delete

© 2013 - My ASP.NET Application

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

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);

    }

}
```

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

```
}

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).

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

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">
```

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

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))

    {
```

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

```
movies = movies.Where(x => x.Genre == movieGenre);

}

return View(movies);

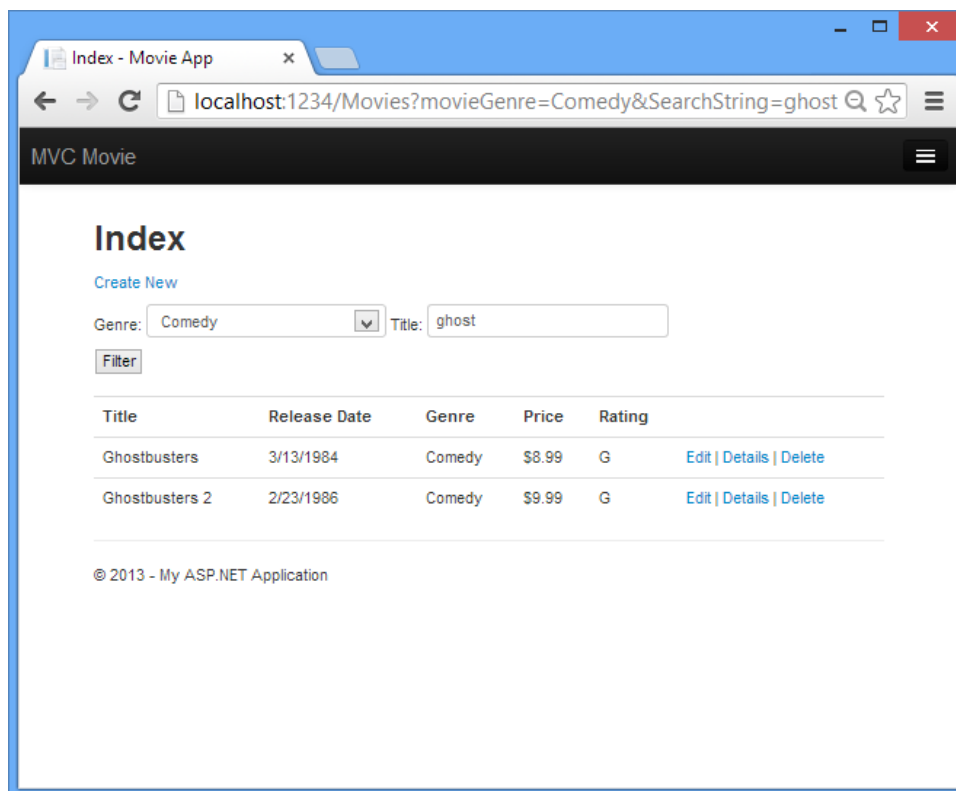
}
```

The parameter "All" provides the item in the list to be preselected. Had we used the following code:

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

And we had a movie with a "Comedy" genre in our database, "Comedy" would be preselected in the dropdown list. Because we don't have a movie genre "All", there is no "All" in the [SelectList](#), so when we post back without making a selection, the `movieGenre` query string value is empty.

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.

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

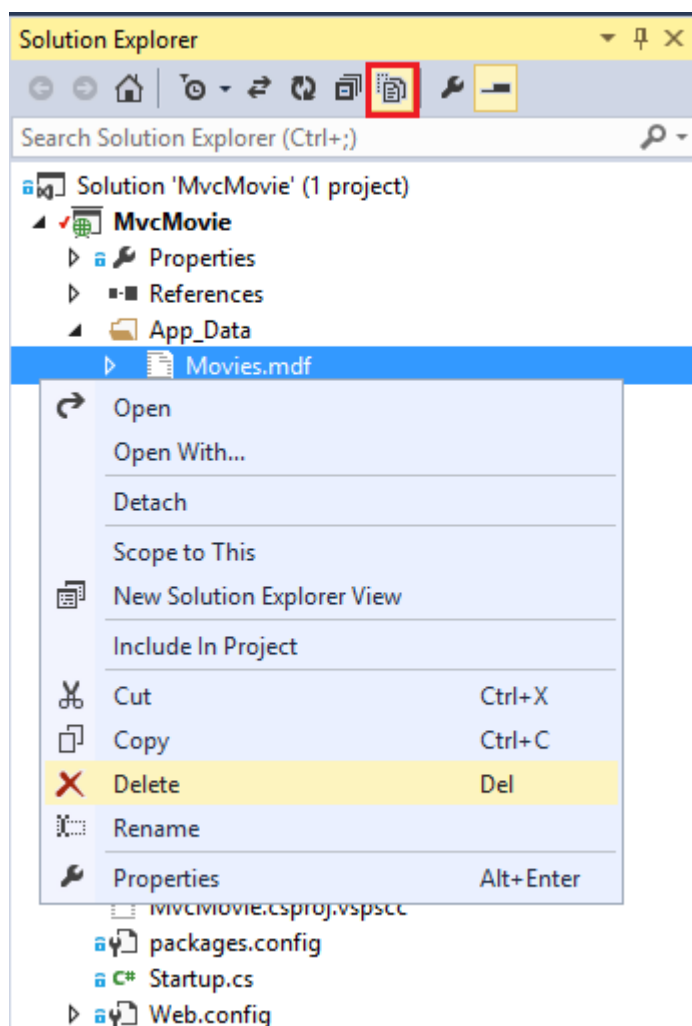
Adding a New Field:

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

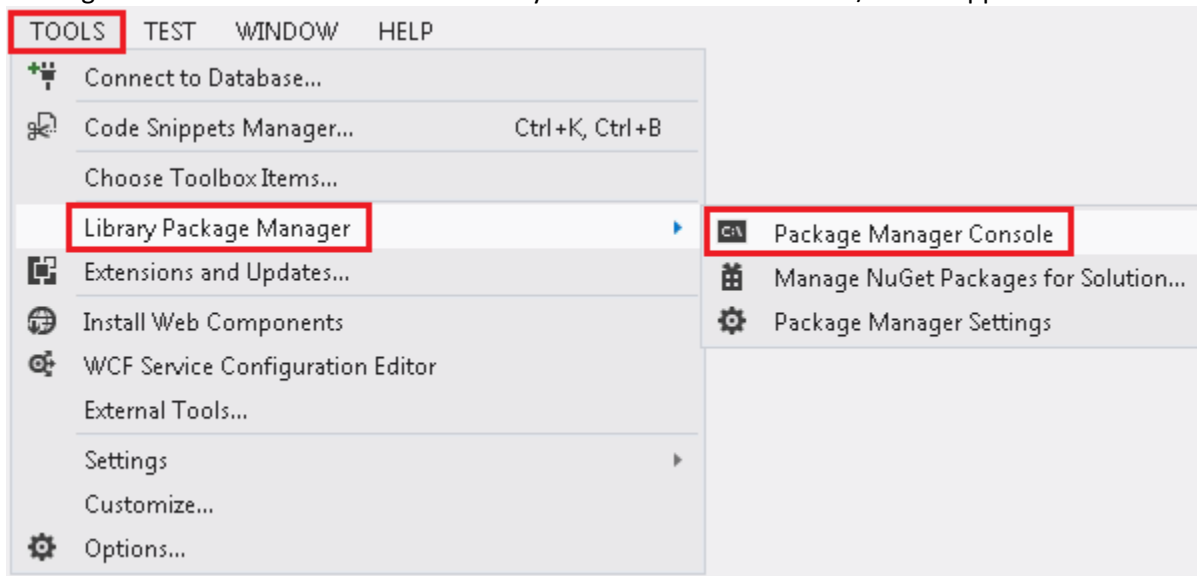
Navigate to Solution Explorer. Right click on the *Movies.mdf* file and select **Delete** to remove the movies database. If you don't see the *Movies.mdf* file, click on the **Show All Files** icon shown below in the red outline.



Build the application to make sure there are no errors.

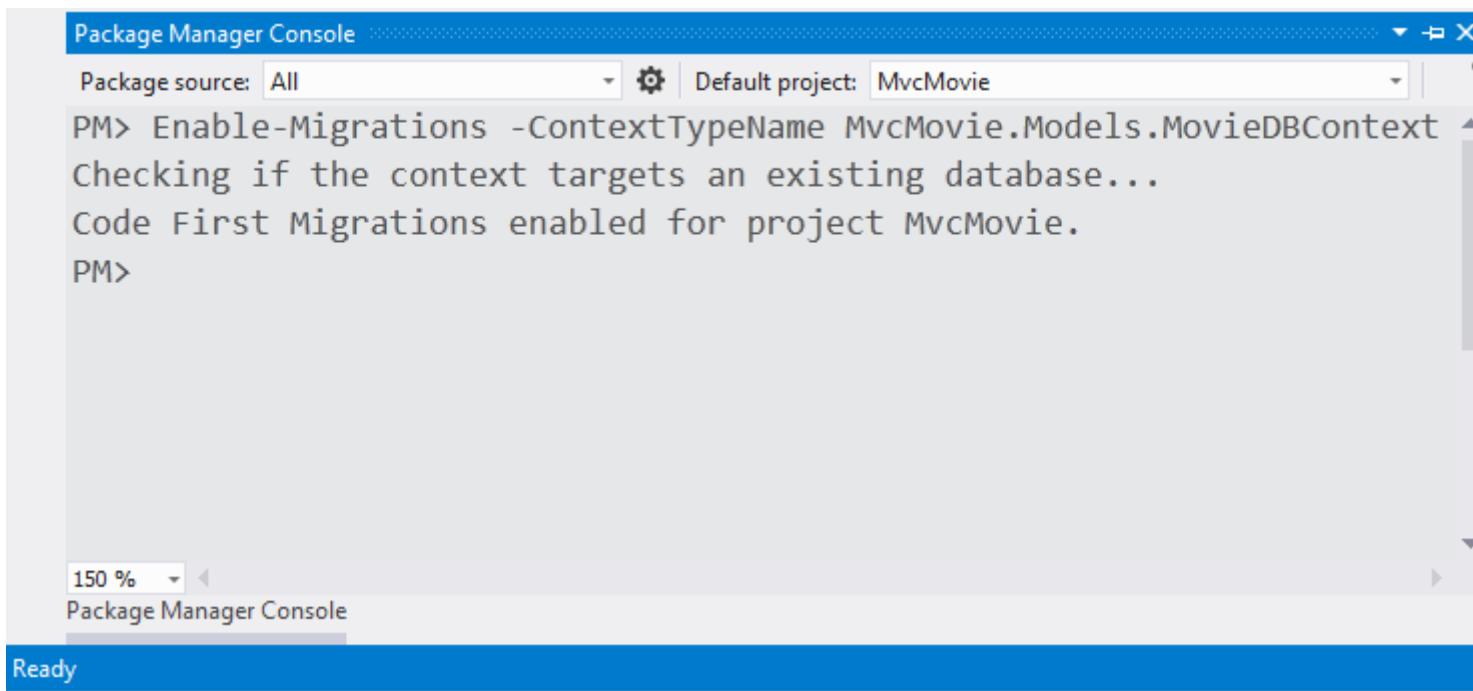
From the **Tools** menu, click **Library Package Manager** and then **Package Manager Console**.

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..



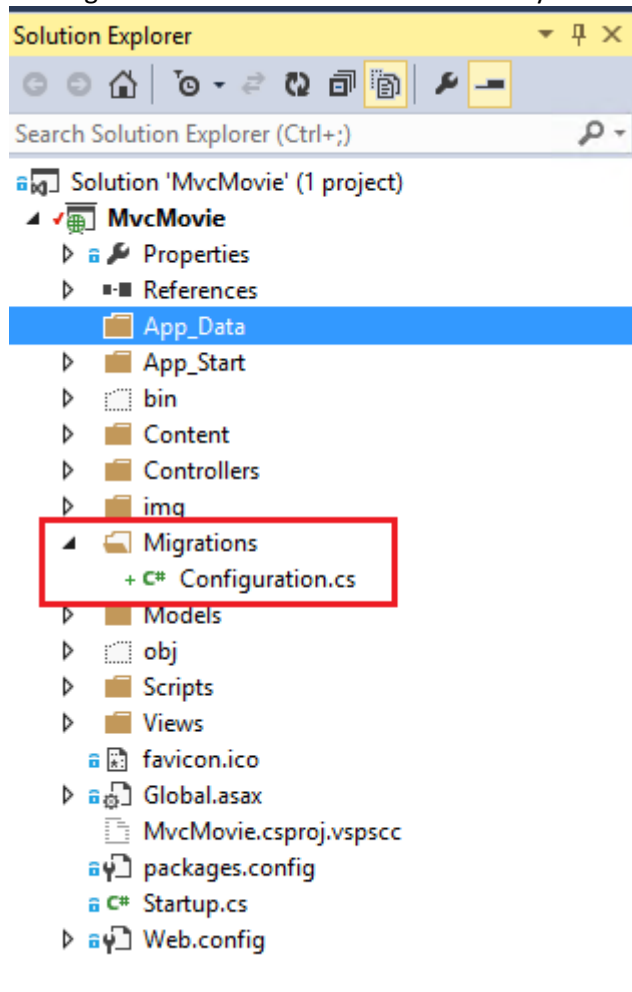
In the **Package Manager Console** window at the **PM>** prompt enter

`Enable-Migrations -ContextTypeName MvcMovie.Models.MovieDbContext`



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

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..



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",
```

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

```
        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

    {

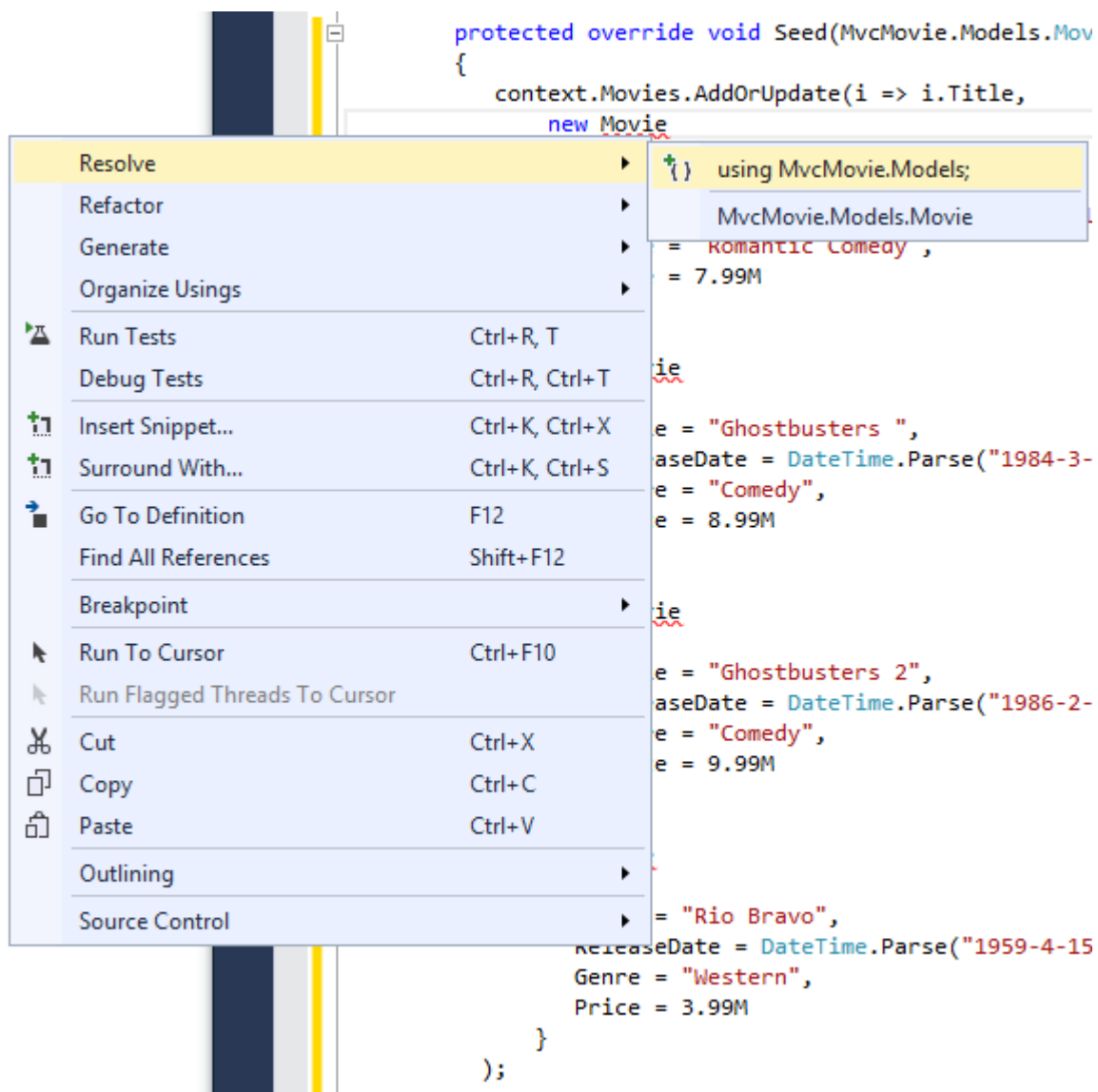
        Title = "Rio Bravo",

        ReleaseDate = DateTime.Parse("1959-4-15"),
```

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

```
Genre = "Western",  
  
Price = 3.99M  
  
}  
  
);  
  
}
```

Right click on the red squiggly line under **Movie** and select **Resolve** and then click **using MvcMovie.Models;**



Doing so adds the following using statement:

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

```
using MvcMovie.Models;
```

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 titles are unique. If you manually add a duplicate title, you'll get the following exception the next time you perform a migration.

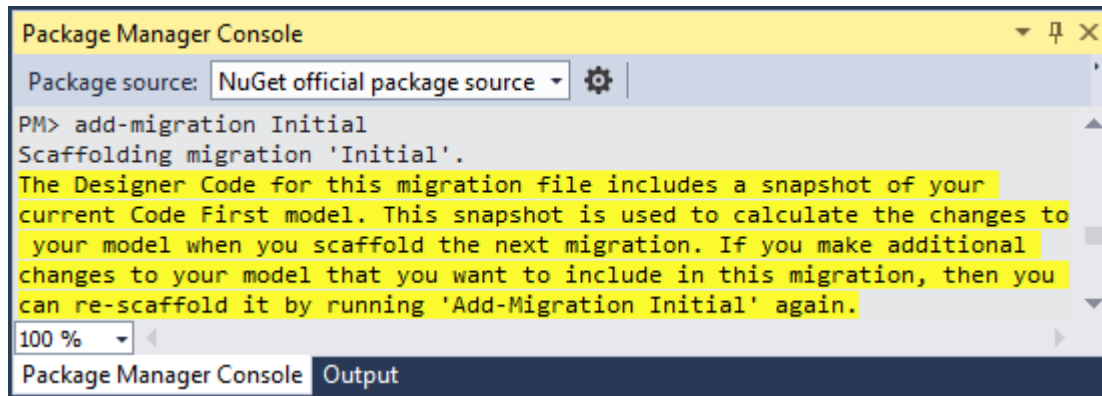
Sequence contains more than one element

For more information about the **AddOrUpdate** method, see [Take care with EF 4.3 AddOrUpdate Method..](#)

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..
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.

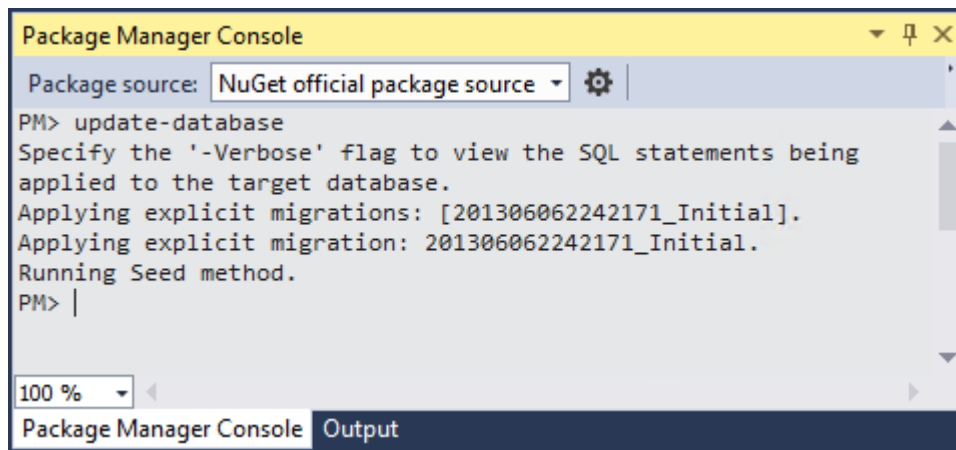
In the **Package Manager Console** window, enter the command **add-migration Initial** to create the initial migration. The name "Initial" is arbitrary and is used to name the migration file created.



```
Package Manager Console
Package source: NuGet official package source
PM> add-migration Initial
Scaffolding migration 'Initial'.
The Designer Code for this migration file includes a snapshot of your
current Code First model. This snapshot is used to calculate the changes to
your model when you scaffold the next migration. If you make additional
changes to your model that you want to include in this migration, then you
can re-scaffold it by running 'Add-Migration Initial' again.
100 %
Package Manager Console Output
```

Code First Migrations creates another class file in the *Migrations* folder (with the name *{TimeStamp}_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 *{TimeStamp}_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 *{TimeStamp}_Initial.cs* file will run and create the 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.



```
Package Manager Console
Package source: NuGet official package source
PM> update-database
Specify the '-Verbose' flag to view the SQL statements being
applied to the target database.
Applying explicit migrations: [201306062242171_Initial].
Applying explicit migration: 201306062242171_Initial.
Running Seed method.
PM> |
100 %
Package Manager Console Output
```

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).

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..
Run the application and navigate to the `/Movies` URL. The seed data is displayed.

Index - Movie App x

localhost:1234/movies

MVC Movie

Index

[Create New](#)

Genre: All Title:

Filter

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

© 2013 - My ASP.NET Application

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

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")]
```

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

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">
```

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

```
<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>
```

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

```
<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>
```

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

```
</tr>

}

</table>
```

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

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

    </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)

        @Html.ValidationMessageFor(model => model.Rating)

    </div>

</div>

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


Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

```
<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 {

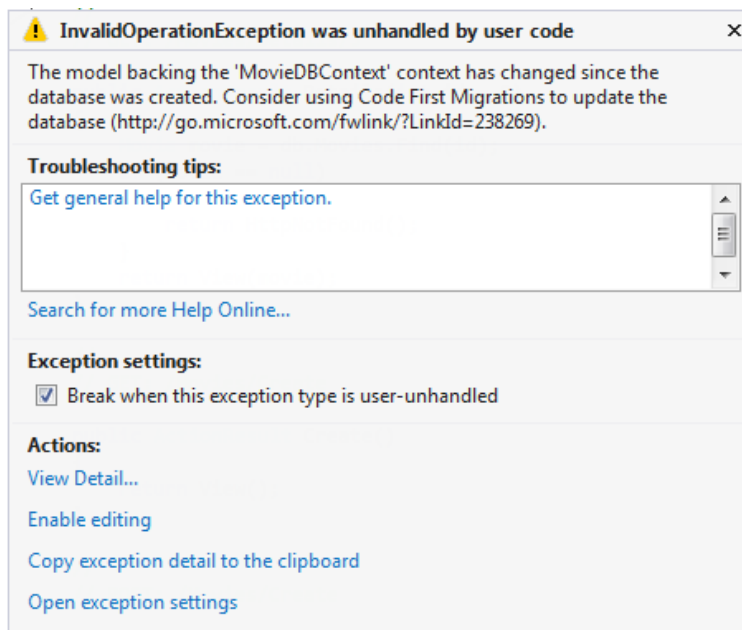
    @Scripts.Render("~/bundles/jqueryval")

}
```

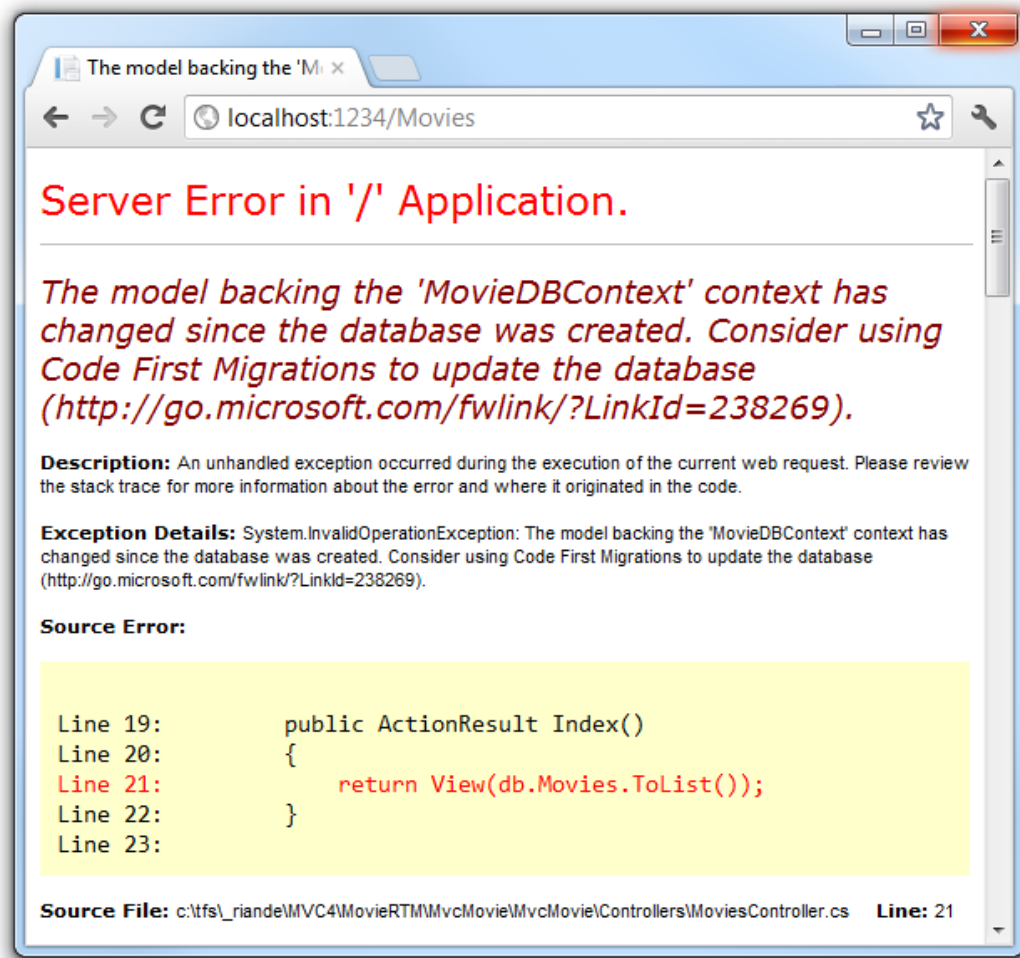
You've now updated the application code to support the new **Rating** property.

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

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 (http://go.microsoft.com/fwlink/?LinkId=238269).



Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

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 develop an application. For more information on Entity Framework database initializers, see Tom Dykstra's fantastic [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.

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

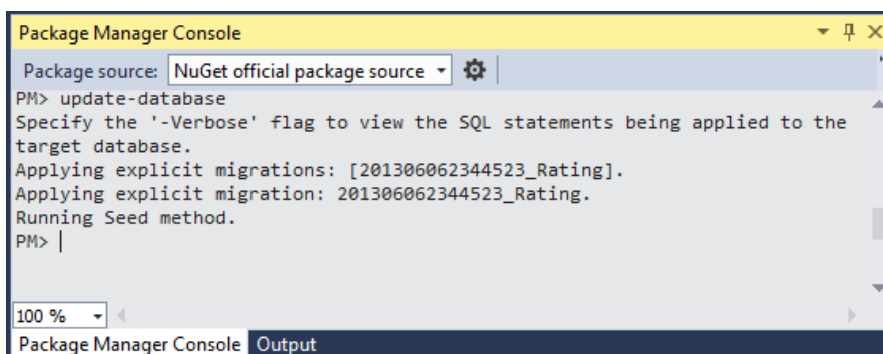
When this command finishes, Visual Studio opens the class file that defines the new **DbMigration** derived class, and in the **Up** method you can see the code that creates the new column.

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

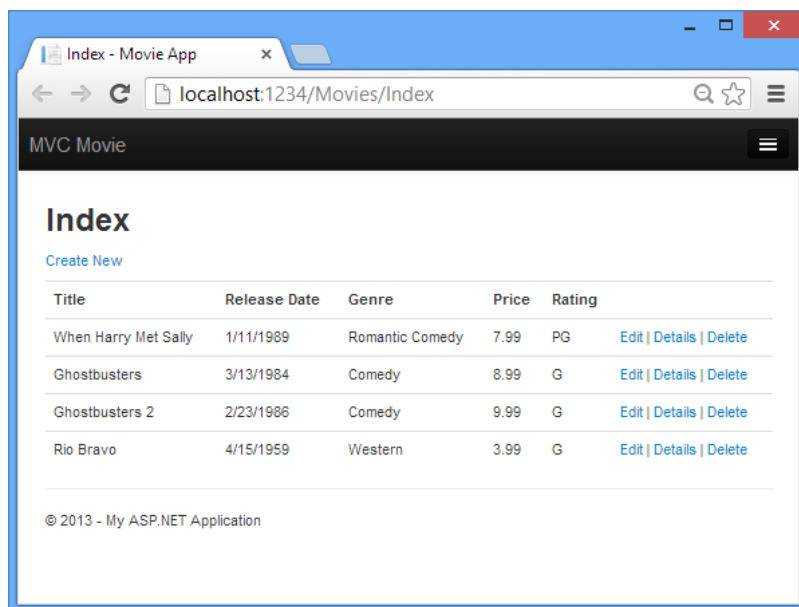
    public override void Down()
    {
        DropColumn("dbo.Movies", "Rating");
    }
}
```

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

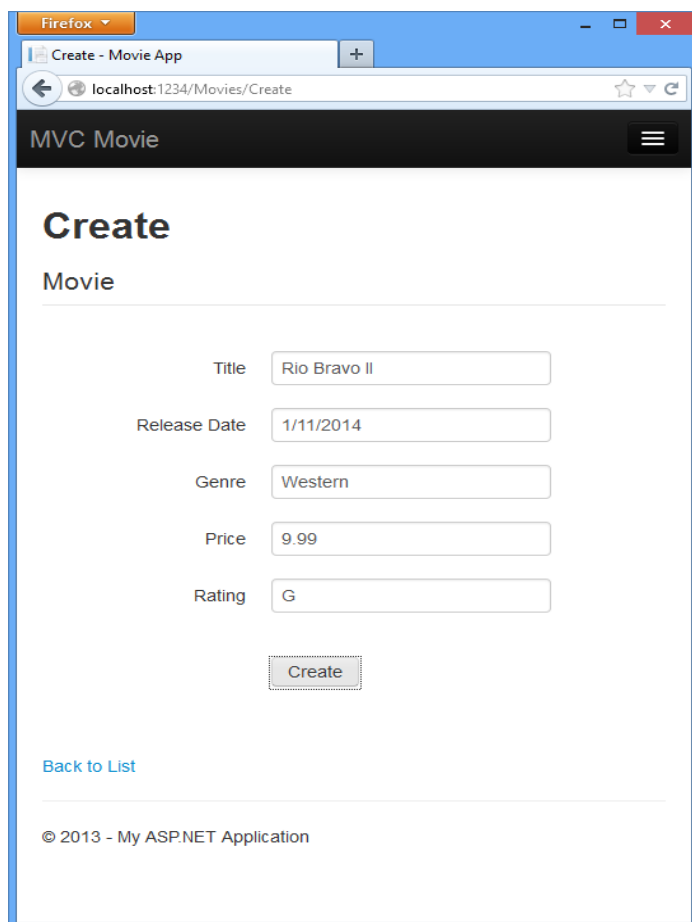
The following image shows the output in the **Package Manager Console** window (The date stamp prepending *Rating* will be different.)



Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..
Re-run the application and navigate to the /Movies URL. You can see the new Rating field.

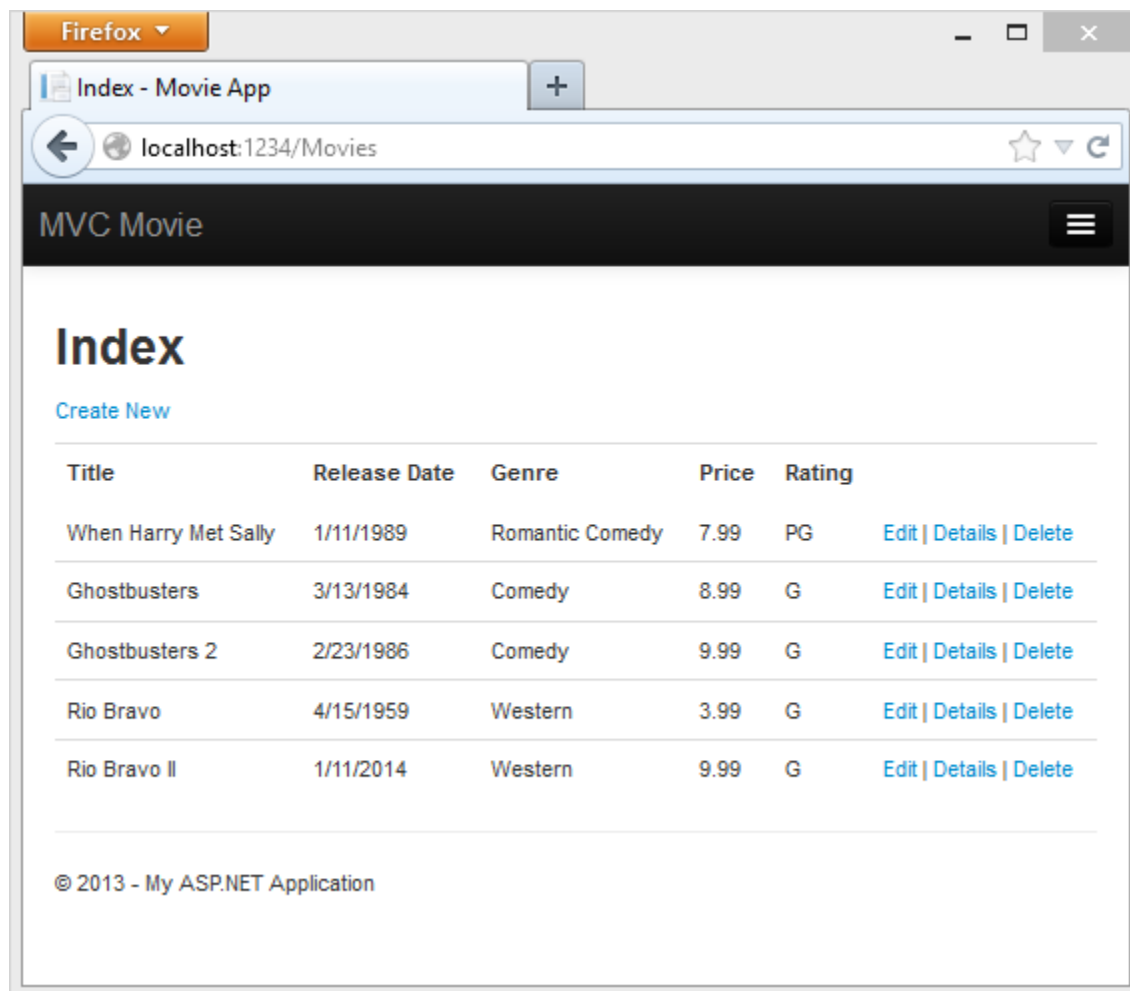


Click the **Create New** link to add a new movie. Note that you can add a rating.



Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

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 inserts 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.

Next, let's look at how you can add richer validation logic to the model classes and enable some business rules to be enforced.

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

Adding Validation:

In this 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; }
```

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

```
[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$")]

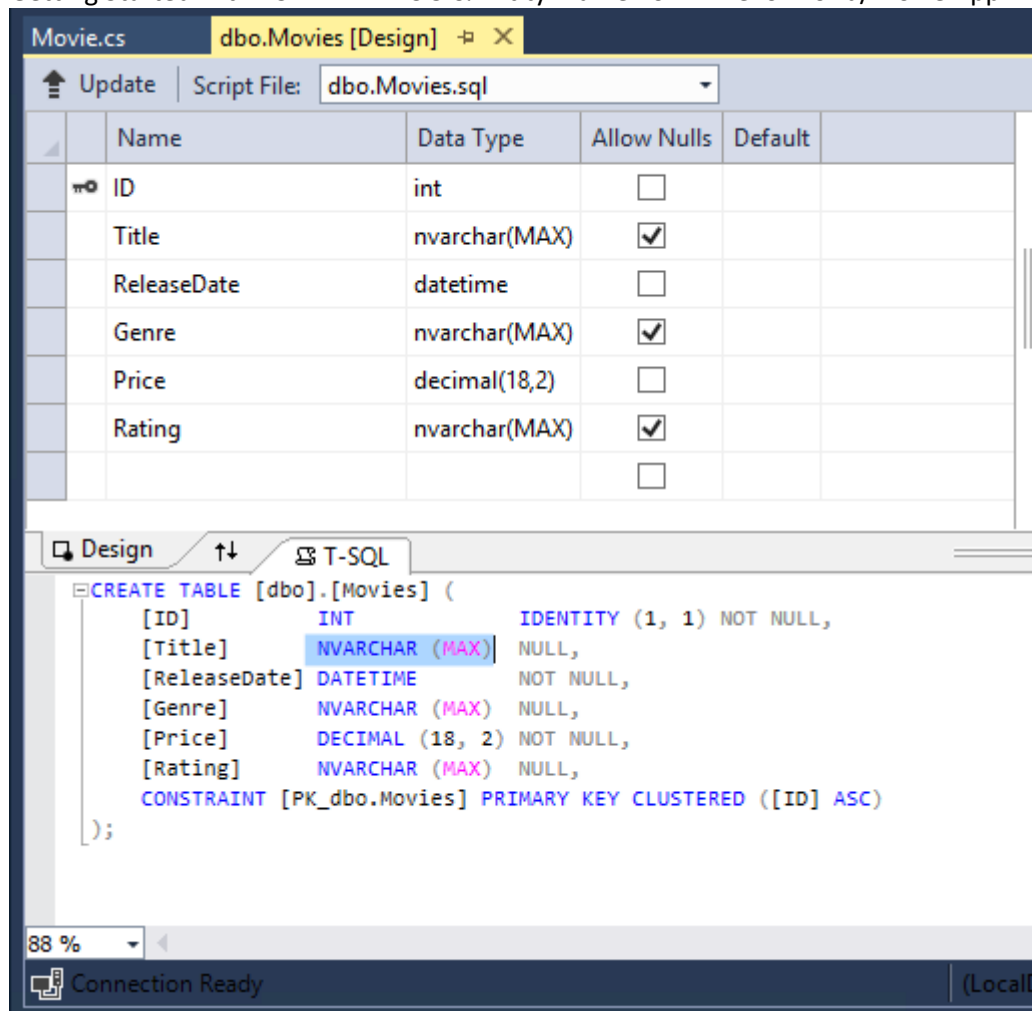
[StringLength(5)]

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**:

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..



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));  
}
```

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

```
}
```

The **Genre** field is 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:

dbo.Movies [Design] + X				
		Update Script File: <input type="text" value="dbo.Movies.sql"/>		
	Name	Data Type	Allow Nulls	Default
	ID	int	<input type="checkbox"/>	
	Title	nvarchar(60)	<input checked="" type="checkbox"/>	
	ReleaseDate	datetime	<input type="checkbox"/>	
	Genre	nvarchar(30)	<input type="checkbox"/>	
	Price	decimal(18,2)	<input type="checkbox"/>	
	Rating	nvarchar(5)	<input checked="" type="checkbox"/>	
			<input type="checkbox"/>	

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
```

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

The code above throws the following exception:

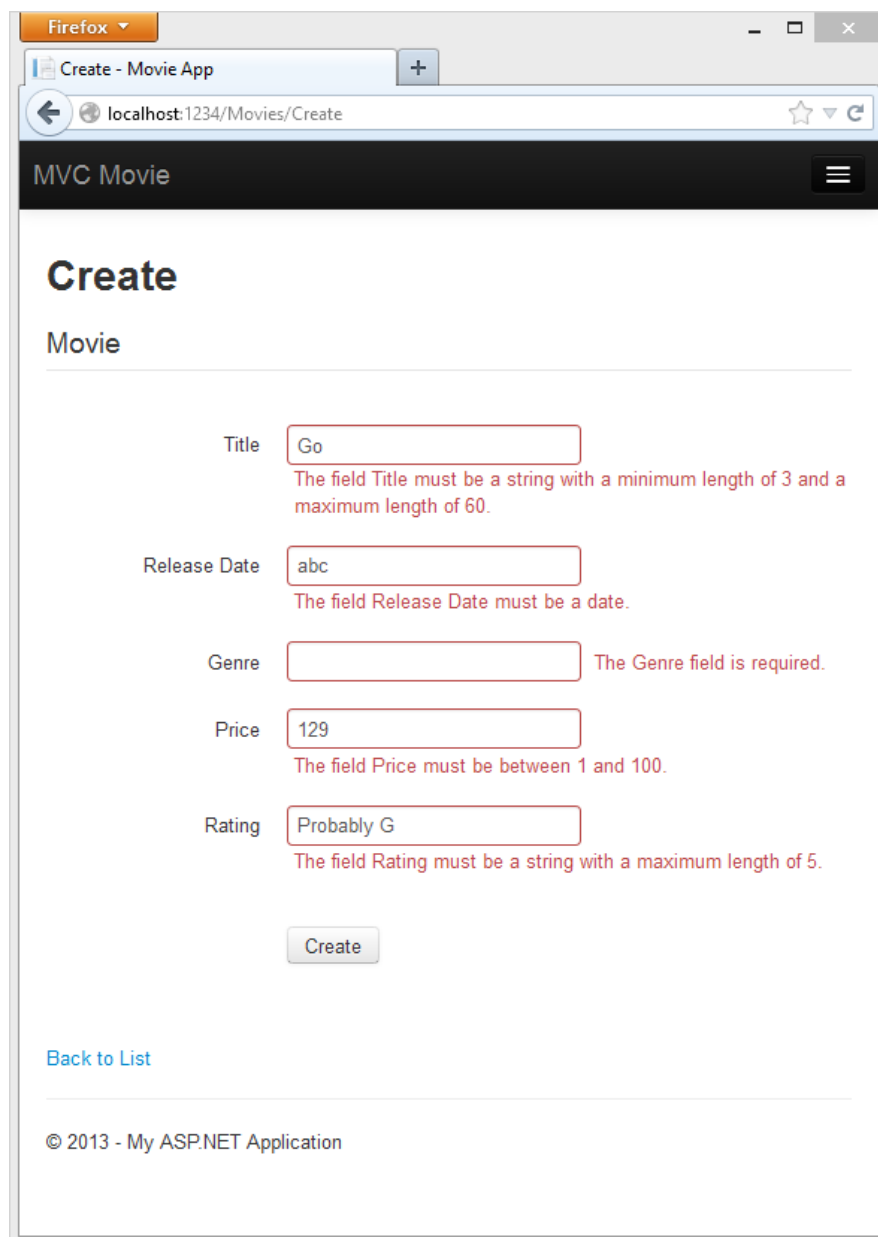
Validation failed for one or more entities. See 'EntityValidationErrors' property for more details.

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.



The screenshot shows a web browser window titled 'Create - Movie App' with the address bar at 'localhost:1234/Movies/Create'. The page has a dark header with 'MVC Movie' and a hamburger menu icon. The main content area is titled 'Create Movie' and contains a form with the following fields and validation messages:

- Title:** Input field contains 'Go'. Error message: 'The field Title must be a string with a minimum length of 3 and a maximum length of 60.'
- Release Date:** Input field contains 'abc'. Error message: 'The field Release Date must be a date.'
- Genre:** Empty input field. Error message: 'The Genre field is required.'
- Price:** Input field contains '129'. Error message: 'The field Price must be between 1 and 100.'
- Rating:** Input field contains 'Probably G'. Error message: 'The field Rating must be a string with a maximum length of 5.'

At the bottom of the form is a 'Create' button. Below the form is a link 'Back to List' and a footer '© 2013 - My ASP.NET Application'.

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

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](#).

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

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");

    }

}
```

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

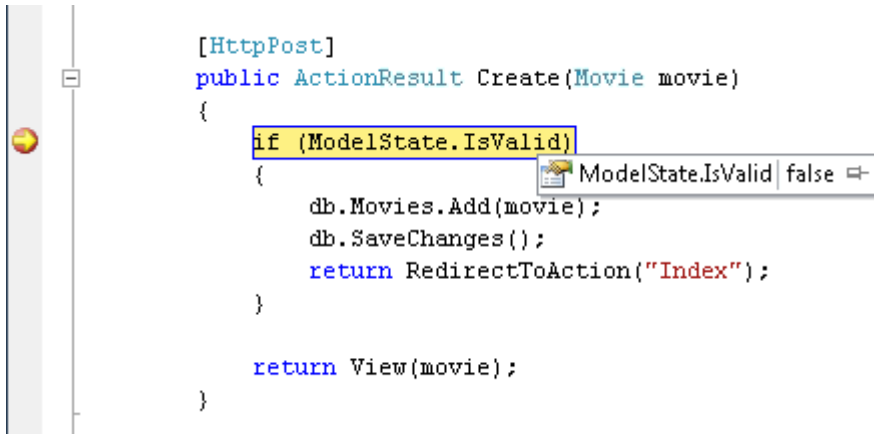
```
return View(movie);
```

```
}
```

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.

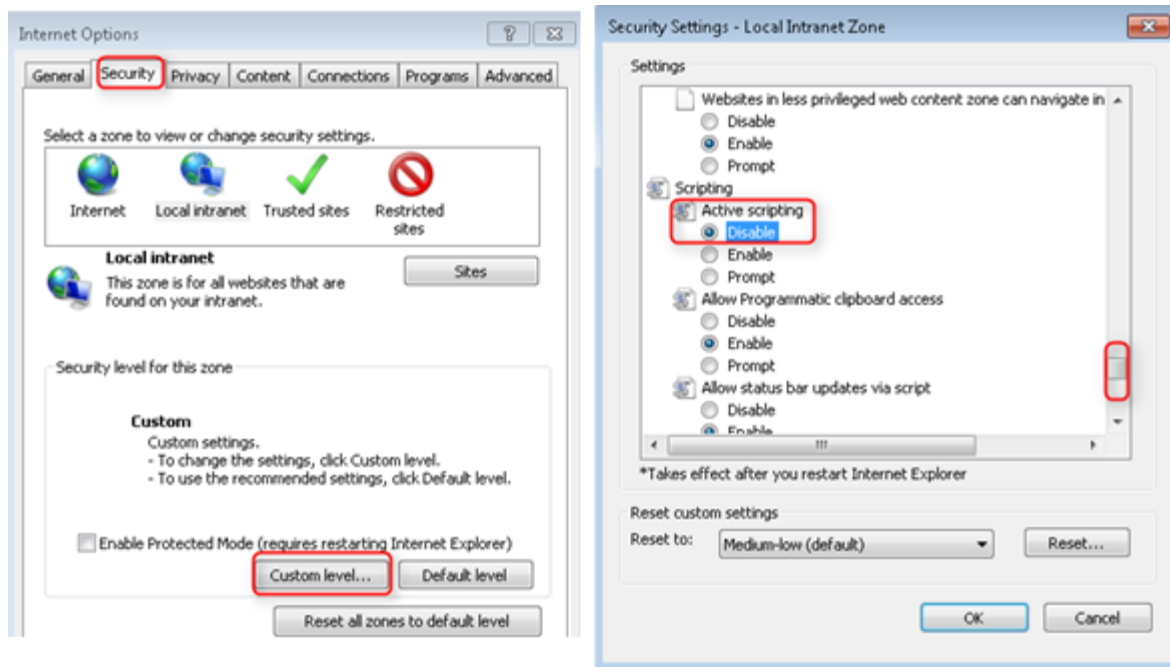


```
[HttpPost]
public ActionResult Create(Movie movie)
{
    if (ModelState.IsValid)
    {
        db.Movies.Add(movie);
        db.SaveChanges();
        return RedirectToAction("Index");
    }

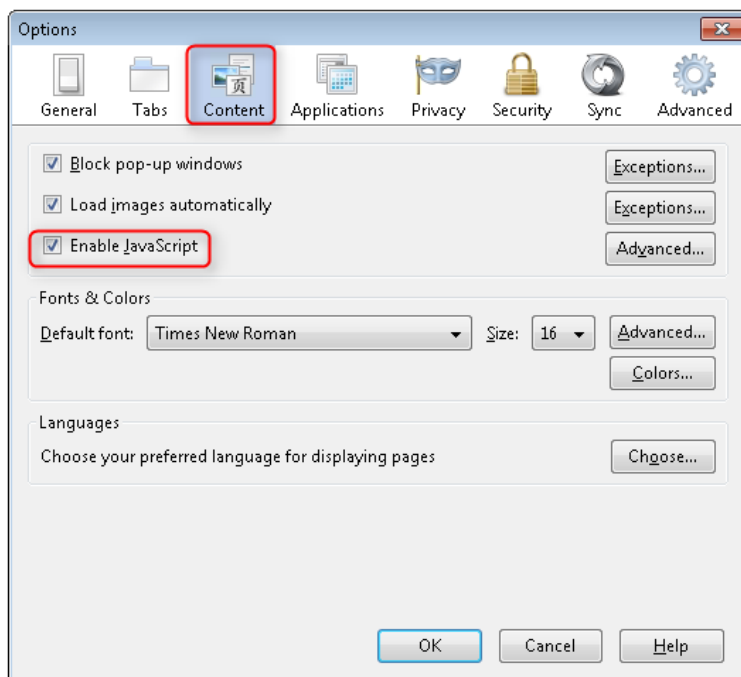
    return View(movie);
}
```

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

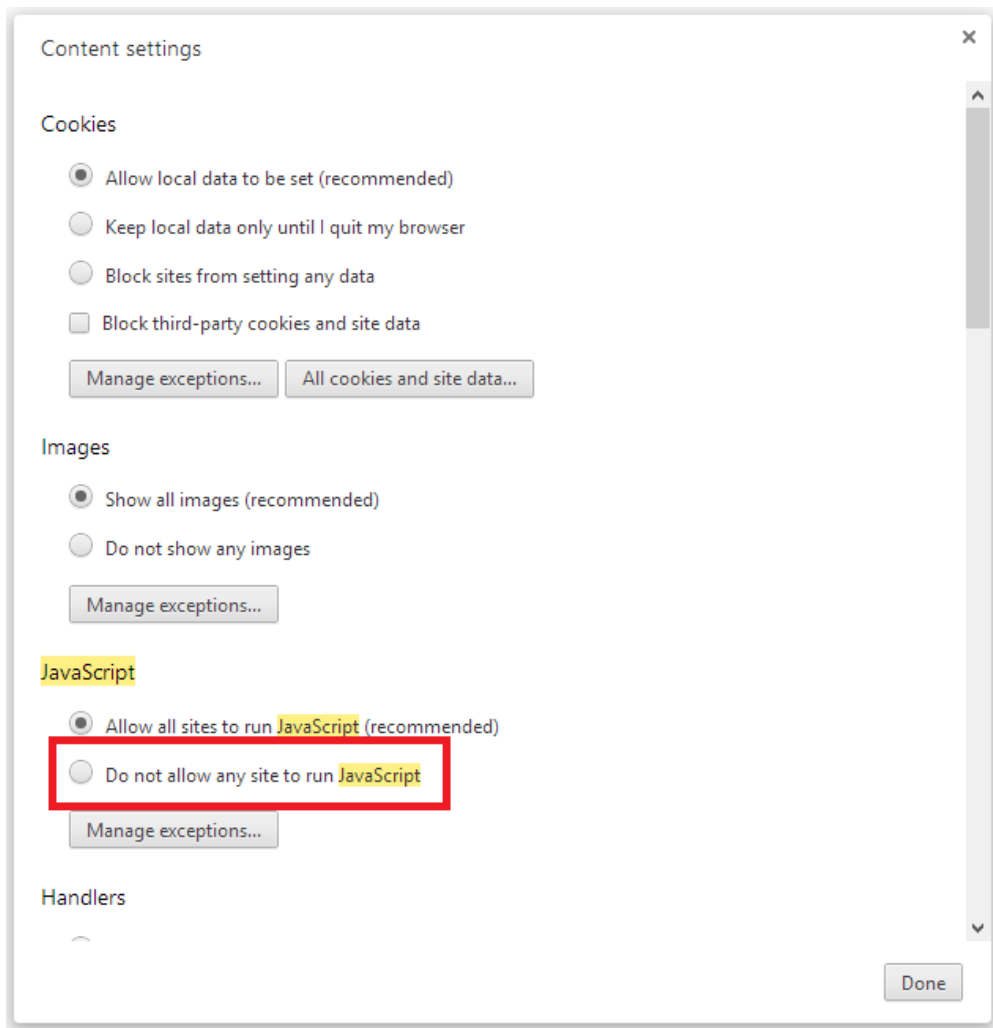
If you disable JavaScript in your browser, then submit the form with errors, the break point will be hit. You still get full validation without JavaScript. The following image shows how to disable JavaScript in Internet Explorer.



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



Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..
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>

    </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>
```

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

```
}

<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 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)]
```

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

```
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 `` 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's [CultureInfo](#).

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 your [locale](#).
- The [DataType](#) attribute can enable MVC to choose the right field template to render the data (the [DisplayFormat](#) if used by itself uses the string template). For more information, see Brad Wilson's [ASP.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 the [Range](#) attribute and [DateTime](#). For example, the following code will always display a client side validation error, even when the date is in the specified range:

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

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

The following code shows combining attributes on one line:

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

```
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; }

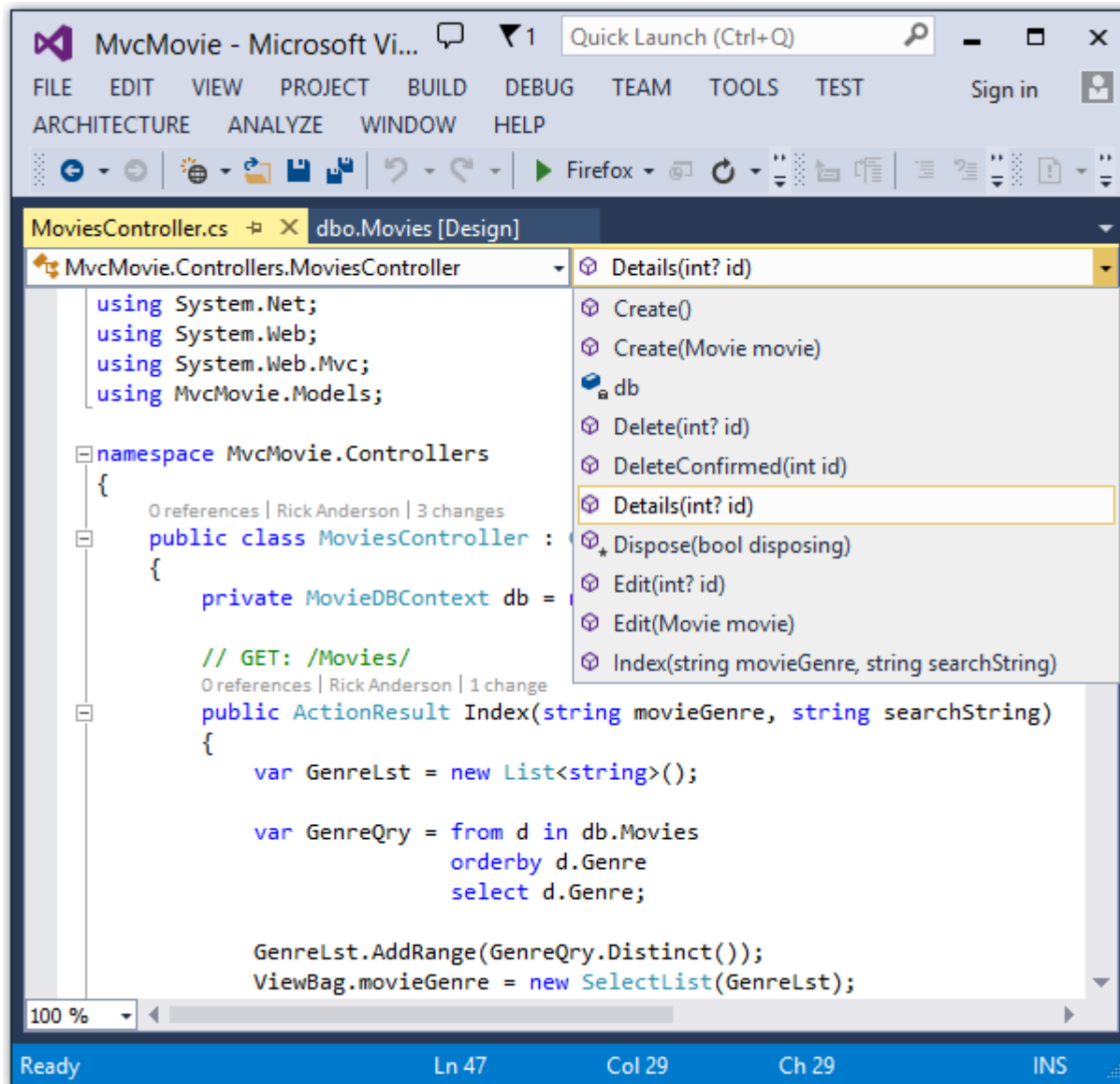
}
```

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

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

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);

    }
}
```

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

```
}

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 an **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);

    }

}
```

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

```
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](#).

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

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();

    }

}
```


Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

```
db.Movies.Remove(movie);

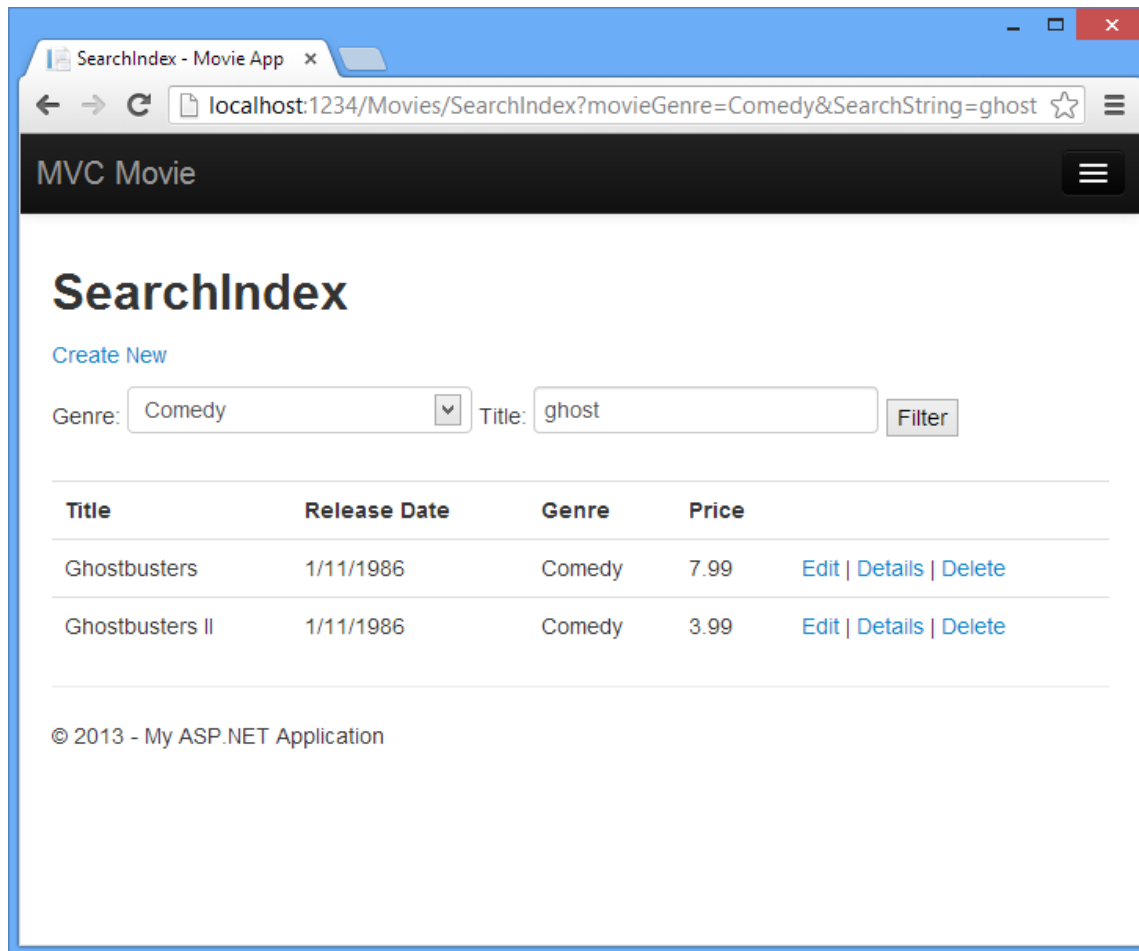
db.SaveChanges();

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. (CRUD – an industry term...)



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.

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

Introduction to ASP.NET Web Programming Using the Razor Syntax (C#)

Razor: The Top 8 Programming Tips

This section lists a few tips that you absolutely need to know as you start writing ASP.NET server code using the Razor syntax.

Note The Razor syntax is based on the C# programming language, and that's the language that's used most often with ASP.NET Web Pages. However, the Razor syntax also supports the Visual Basic language, and everything you see you can also do in Visual Basic. For details, see the appendix [Visual Basic Language and Syntax](#).

You can find more details about most of these programming techniques later in the article.

1. You add code to a page using the @ character

The @ character starts inline expressions, single statement blocks, and multi-statement blocks:

```
<!-- Single statement blocks -->

@{ var total = 7; }

@{ var myMessage = "Hello World"; }

<!-- Inline expressions -->

<p>The value of your account is: @total </p>

<p>The value of myMessage is: @myMessage</p>

<!-- Multi-statement block -->

@{

    var greeting = "Welcome to our site!";

    var weekDay = DateTime.Now.DayOfWeek;

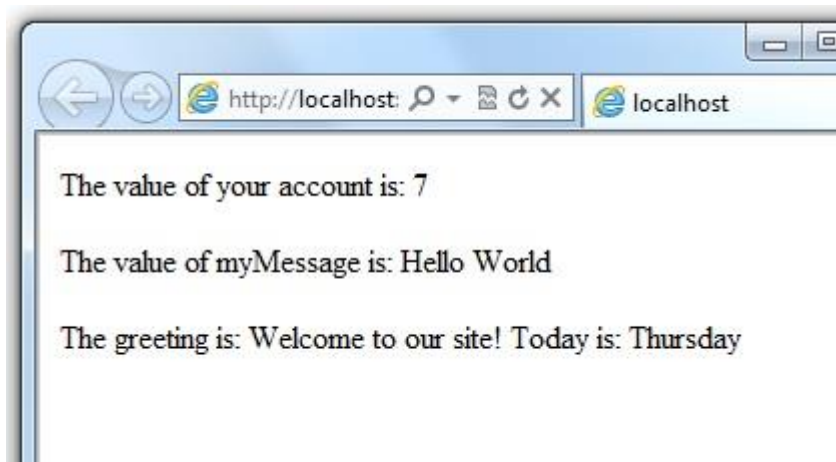
    var greetingMessage = greeting + " Today is: " + weekDay;

}

<p>The greeting is: @greetingMessage</p>
```

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

This is what these statements look like when the page runs in a browser:



HTML Encoding

When you display content in a page using the `@` character, as in the preceding examples, ASP.NET HTML-encodes the output. This replaces reserved HTML characters (such as `<` and `>` and `&`) with codes that enable the characters to be displayed as characters in a web page instead of being interpreted as HTML tags or entities. Without HTML encoding, the output from your server code might not display correctly, and could expose a page to security risks.

If your goal is to output HTML markup that renders tags as markup (for example `<p></p>` for a paragraph or `` to emphasize text), see the section [Combining Text, Markup, and Code in Code Blocks](#) later in this article.

You can read more about HTML encoding in [Working with Forms](#).

2. You enclose code blocks in braces

A *code block* includes one or more code statements and is enclosed in braces.

```
<!-- Single statement block. -->
```

```
@{ var theMonth = DateTime.Now.Month; }
```

```
<p>The numeric value of the current month: @theMonth</p>
```

```
<!-- Multi-statement block. -->
```

```
@{
```

```
    var outsideTemp = 79;
```

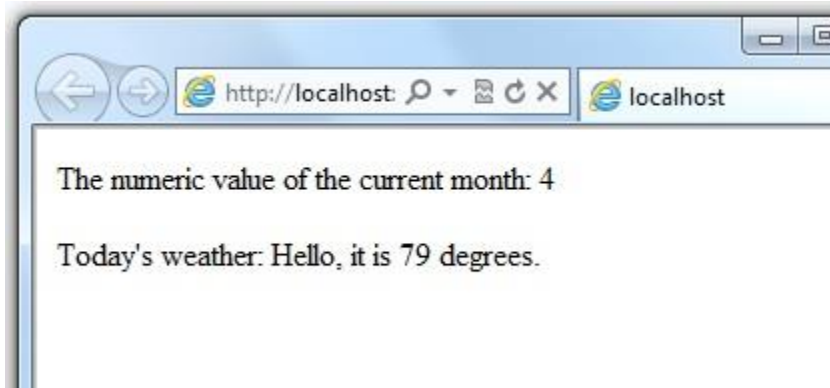
```
    var weatherMessage = "Hello, it is " + outsideTemp + " degrees.";
```

```
}
```

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

```
<p>Today's weather: @weatherMessage</p>
```

The result displayed in a browser:



3. Inside a block, you end each code statement with a semicolon

Inside a code block, each complete code statement must end with a semicolon. Inline expressions don't end with a semicolon.

```
<!-- Single-statement block -->
```

```
@{ var theMonth = DateTime.Now.Month; }
```

```
<!-- Multi-statement block -->
```

```
@{
```

```
    var outsideTemp = 79;
```

```
    var weatherMessage = "Hello, it is " + outsideTemp + " degrees.";
```

```
}
```

```
<!-- Inline expression, so no semicolon -->
```

```
<p>Today's weather: @weatherMessage</p>
```

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

4. You use variables to store values

You can store values in a *variable*, including strings, numbers, and dates, etc. You create a new variable using the **var** keyword. You can insert variable values directly in a page using **@**.

```
<!-- Storing a string -->

@{ var welcomeMessage = "Welcome, new members!"; }

<p>@welcomeMessage</p>

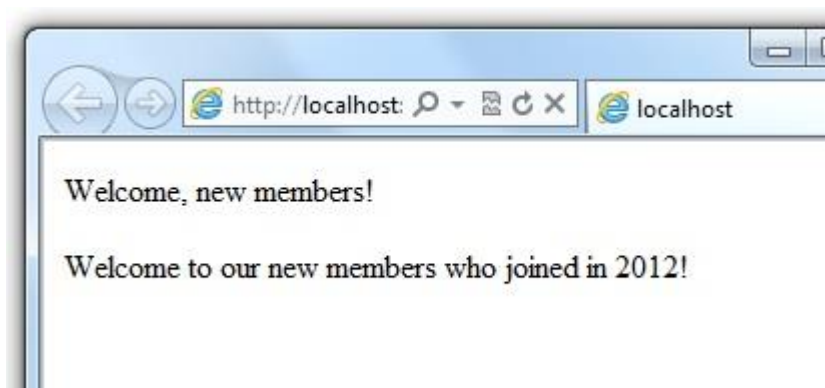
<!-- Storing a date -->

@{ var year = DateTime.Now.Year; }

<!-- Displaying a variable -->

<p>Welcome to our new members who joined in @year!</p>
```

The result displayed in a browser:



5. You enclose literal string values in double quotation marks

A *string* is a sequence of characters that are treated as text. To specify a string, you enclose it in double quotation marks:

```
@{ var myString = "This is a string literal"; }
```

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

If the string that you want to display contains a backslash character (\) or double quotation marks ("), use a *verbatim string literal* that's prefixed with the @ operator. (In C#, the \ character has special meaning unless you use a verbatim string literal.)

```
<!-- Embedding a backslash in a string -->
```

```
@{ var myFilePath = @"C:\MyFolder\"; }
```

```
<p>The path is: @myFilePath</p>
```

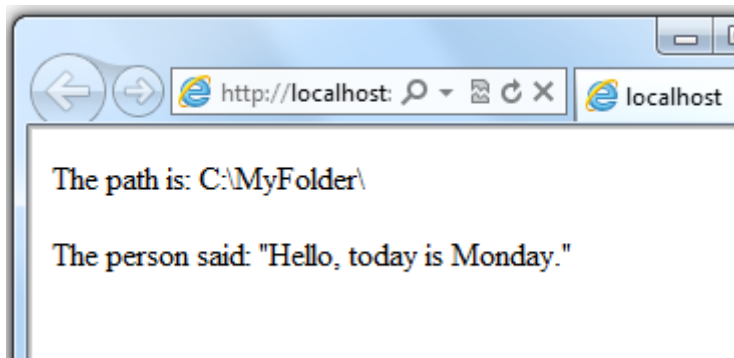
To embed double quotation marks, use a verbatim string literal and repeat the quotation marks:

```
<!-- Embedding double quotation marks in a string -->
```

```
@{ var myQuote = @"The person said: ""Hello, today is Monday."""; }
```

```
<p>@myQuote</p>
```

Here's the result of using both of these examples in a page:



Note Notice that the @ character is used both to mark verbatim string literals in C# and to mark code in ASP.NET pages.

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

6. Code is case sensitive

In C#, keywords (like **var**, **true**, and **if**) and variable names are case sensitive. The following lines of code create two different variables, **lastName** and **LastName**.

```
@{  
  
    var lastName = "Smith";  
  
    var LastName = "Jones";  
  
}
```

If you declare a variable as **var lastName = "Smith";** and if you try to reference that variable in your page as **@LastName**, an error results because **LastName** won't be recognized.

Note In Visual Basic, keywords and variables are *not* case sensitive.

7. Much of your coding involves objects

An *object* represents a thing that you can program with — a page, a text box, a file, an image, a web request, an email message, a customer record (database row), etc. Objects have properties that describe their characteristics and that you can read or change — a text box object has a **Text** property (among others), a request object has a **Url** property, an email message has a **From** property, and a customer object has a **FirstName** property. Objects also have methods that are the "verbs" they can perform. Examples include a file object's **Save** method, an image object's **Rotate** method, and an email object's **Send** method.

You'll often work with the **Request** object, which gives you information like the values of text boxes (form fields) on the page, what type of browser made the request, the URL of the page, the user identity, etc. The following example shows how to access properties of the **Request** object and how to call the **MapPath** method of the **Request** object, which gives you the absolute path of the page on the server:

```
<table border="1">  
  
<tr>  
  
    <td>Requested URL</td>  
  
    <td>Relative Path</td>  
  
    <td>Full Path</td>  
  
    <td>HTTP Request Type</td>  
  
</tr>
```

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

```
<tr>

    <td>@Request.Url</td>

    <td>@Request.FilePath</td>

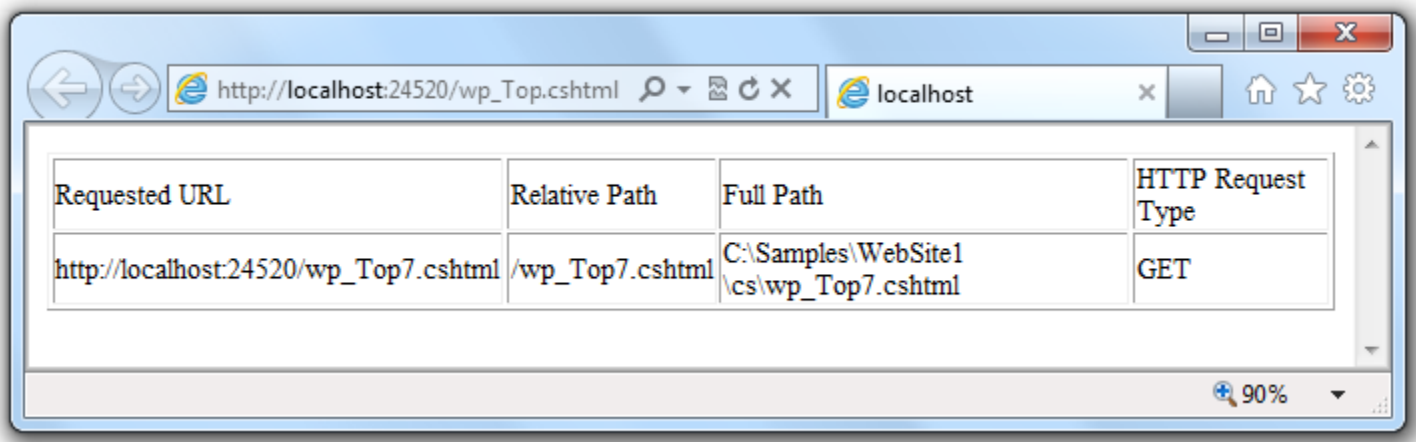
    <td>@Request.MapPath(Request.FilePath)</td>

    <td>@Request.RequestType</td>

</tr>

</table>
```

The result displayed in a browser:



8. You can write code that makes decisions

A key feature of dynamic web pages is that you can determine what to do based on conditions. The most common way to do this is with the **if** statement (and optional **else** statement).

```
@{

    var result = "";

    if(IsPost)

    {
```


Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

```
        result = "This page was posted using the Submit button.";

    }

    else

    {

        result = "This was the first request for this page.";

    }

}

<!DOCTYPE html>

<html>

    <head>

        <title></title>

    </head>

    <body>

        <form method="POST" action="" >

            <input type="Submit" name="Submit" value="Submit"/>

            <p>@result</p>

        </form>

    </body>

</html>

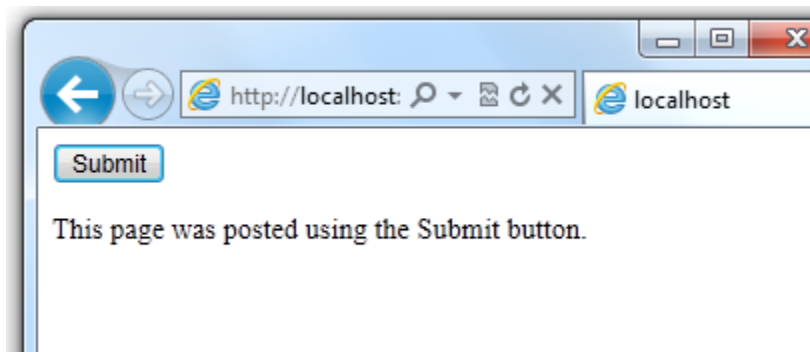
</body>
```

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

</html>

The statement `if(IsPost)` is a shorthand way of writing `if(IsPost == true)`. Along with `if` statements, there are a variety of ways to test conditions, repeat blocks of code, and so on, which are described later in this article.

The result displayed in a browser (after clicking **Submit**):



HTTP GET and POST Methods and the IsPost Property

The protocol used for web pages (HTTP) supports a very limited number of methods (verbs) that are used to make requests to the server. The two most common ones are GET, which is used to read a page, and POST, which is used to submit a page. In general, the first time a user requests a page, the page is requested using GET. If the user fills in a form and then clicks a submit button, the browser makes a POST request to the server.

In web programming, it's often useful to know whether a page is being requested as a GET or as a POST so that you know how to process the page. In ASP.NET Web Pages, you can use the `IsPost` property to see whether a request is a GET or a POST. If the request is a POST, the `IsPost` property will return true, and you can do things like read the values of text boxes on a form. Many examples you'll see show you how to process the page differently depending on the value of `IsPost`.

A Simple Code Example

This procedure shows you how to create a page that illustrates basic programming techniques. In the example, you create a page that lets users enter two numbers, then it adds them and displays the result.

1. In your editor, create a new file and name it *AddNumbers.cshtml*.
2. Copy the following code and markup into the page, replacing anything already in the page.

```
@{  
  
    var total = 0;  
  
    var totalMessage = "";  
  
    if(IsPost) {  
  
        // Retrieve the numbers that the user entered.
```

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

```
var num1 = Request["text1"];

var num2 = Request["text2"];

// Convert the entered strings into integers numbers and add.

total = num1.AsInt() + num2.AsInt();

totalMessage = "Total = " + total;

}

}

<!DOCTYPE html>

<html lang="en">

<head>

<title>Add Numbers</title>

<meta charset="utf-8" />

<style type="text/css">

body {background-color: beige; font-family: Verdana, Arial;

margin: 50px; }

form {padding: 10px; border-style: solid; width: 250px;}

</style>

</head>

<body>

<p>Enter two whole numbers and then click <strong>Add</strong>.</p>
```

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

```
<form action="" method="post">

    <p><label for="text1">First Number:</label>

    <input type="text" name="text1" />

    </p>

    <p><label for="text2">Second Number:</label>

    <input type="text" name="text2" />

    </p>

    <p><input type="submit" value="Add" /></p>

</form>

<p>@totalMessage</p>

</body>

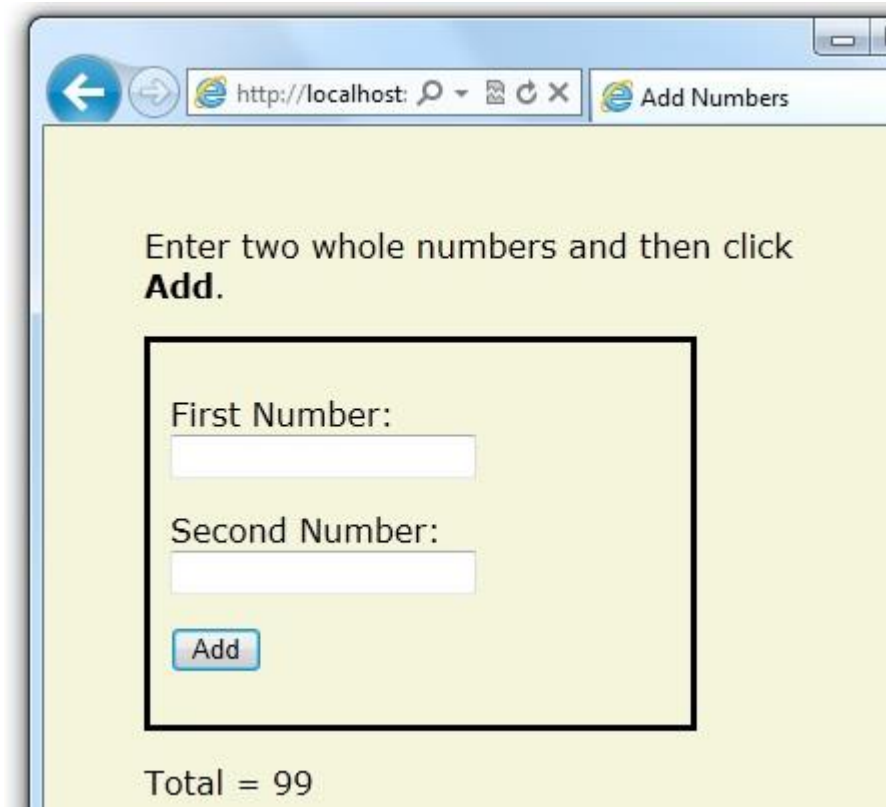
</html>
```

Here are some things for you to note:

- The @ character starts the first block of code in the page, and it precedes the `totalMessage` variable that's embedded near the bottom of the page.
- The block at the top of the page is enclosed in braces.
- In the block at the top, all lines end with a semicolon.
- The variables `total`, `num1`, `num2`, and `totalMessage` store several numbers and a string.
- The literal string value assigned to the `totalMessage` variable is in double quotation marks.
- Because the code is case-sensitive, when the `totalMessage` variable is used near the bottom of the page, its name must match the variable at the top exactly.
- The expression `num1.AsInt() + num2.AsInt()` shows how to work with objects and methods. The `AsInt` method on each variable converts the string entered by a user to a number (an integer) so that you can perform arithmetic on it.

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

- The `<form>` tag includes a `method="post"` attribute. This specifies that when the user clicks **Add**, the page will be sent to the server using the HTTP POST method. When the page is submitted, the `if(IsPost)` test evaluates to true and the conditional code runs, displaying the result of adding the numbers.
3. Save the page and run it in a browser. (Make sure the page is selected in the **Files** workspace before you run it.) Enter two whole numbers and then click the **Add** button.



Basic Programming Concepts

This article provides you with an overview of ASP.NET web programming. It isn't an exhaustive examination, just a quick tour through the programming concepts you'll use most often. Even so, it covers almost everything you'll need to get started with ASP.NET Web Pages.

But first, a little technical background.

The Razor Syntax, Server Code, and ASP.NET

Razor syntax is a simple programming syntax for embedding server-based code in a web page. In a web page that uses the Razor syntax, there are two kinds of content: client content and server code. Client content is the stuff you're used to in web pages: HTML markup (elements), style information such as CSS, maybe some client script such as JavaScript, and plain text.

Razor syntax lets you add server code to this client content. If there's server code in the page, the server runs that code first, before it sends the page to the browser. By running on the server, the code can perform tasks that can be a lot more complex to do using client content alone, like accessing server-based databases. Most importantly, server code can dynamically create client content — it can generate HTML markup or other content on the fly and then send it to the browser along with any static HTML that the page might contain. From the browser's perspective, client content that's generated by your server code is no different than any other client content. As you've already seen, the server code that's required is quite simple.

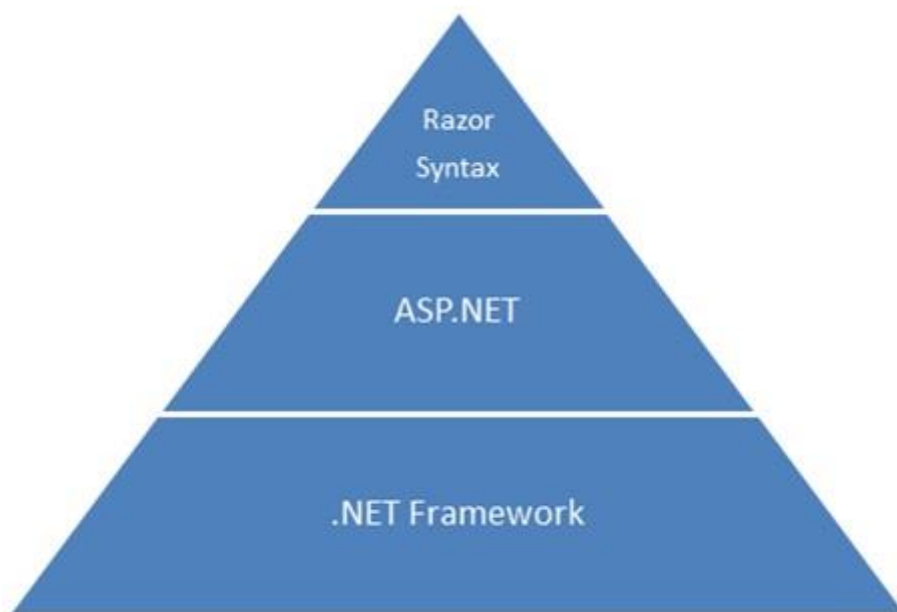
Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

ASP.NET web pages that include the Razor syntax have a special file extension (*.cshtml* or *.vbhtml*). The server recognizes these extensions, runs the code that's marked with Razor syntax, and then sends the page to the browser.

Where does ASP.NET fit in?

Razor syntax is based on a technology from Microsoft called ASP.NET, which in turn is based on the Microsoft .NET Framework. The .NET Framework is a big, comprehensive programming framework from Microsoft for developing virtually any type of computer application. ASP.NET is the part of the .NET Framework that's specifically designed for creating web applications. Developers have used ASP.NET to create many of the largest and highest-traffic websites in the world. (Any time you see the file-name extension *.aspx* as part of the URL in a site, you'll know that the site was written using ASP.NET.)

The Razor syntax gives you all the power of ASP.NET, but using a simplified syntax that's easier to learn if you're a beginner and that makes you more productive if you're an expert. Even though this syntax is simple to use, its family relationship to ASP.NET and the .NET Framework means that as your websites become more sophisticated, you have the power of the larger frameworks available to you.



Classes and Instances

ASP.NET server code uses objects, which are in turn built on the idea of classes. The class is the definition or template for an object. For example, an application might contain a **Customer** class that defines the properties and methods that any customer object needs.

When the application needs to work with actual customer information, it creates an instance of (or *instantiates*) a customer object. Each individual customer is a separate instance of the **Customer** class. Every instance supports the same properties and methods, but the property values for each instance are typically different, because each customer object is unique. In one customer object, the **LastName** property might be "Smith"; in another customer object, the **LastName** property might be "Jones."

Similarly, any individual web page in your site is a **Page** object that's an instance of the **Page** class. A button on the page is a **Button** object that is an instance of the **Button** class, and so on. Each instance has its own characteristics, but they all are based on what's specified in the object's class definition.

Basic Syntax

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

Earlier you saw a basic example of how to create an ASP.NET Web Pages page, and how you can add server code to HTML markup. Here you'll learn the basics of writing ASP.NET server code using the Razor syntax — that is, the programming language rules.

If you're experienced with programming (especially if you've used C, C++, C#, Visual Basic, or JavaScript), much of what you read here will be familiar. You'll probably need to familiarize yourself only with how server code is added to markup in *.cshtml* files.

Combining Text, Markup, and Code in Code Blocks

In server code blocks, you often want to output text or markup (or both) to the page. If a server code block contains text that's not code and that instead should be rendered as is, **ASP.NET needs to be able to distinguish that text from code**. There are several ways to do this.

- Enclose the text in an HTML element like `<p></p>` or ``:

```
@if(IsPost) {

    // This line has all content between matched <p> tags.

    <p>Hello, the time is @DateTime.Now and this page is a postback!</p>

} else {

    // All content between matched tags, followed by server code.

    <p>Hello <em>stranger</em>, today is: <br /> </p> @DateTime.Now

}
```

The HTML element can include text, additional HTML elements, and server-code expressions. When ASP.NET sees the opening HTML tag (for example, `<p>`), it renders everything including the element and its content as is to the browser, resolving server-code expressions as it goes.

- Use the `@:` operator or the `<text>` element. The `@:` outputs a single line of content containing plain text or unmatched HTML tags; the `<text>` element encloses multiple lines to output. These options are useful when you don't want to render an HTML element as part of the output.

```
@if(IsPost) {

    // Plain text followed by an unmatched HTML tag and server code.

    @: The time is: <br /> @DateTime.Now

    <br/>
```

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

```
// Server code and then plain text, matched tags, and more text.

@DateTime.Now @:is the <em>current</em> time.

}
```

If you want to output multiple lines of text or unmatched HTML tags, you can precede each line with `@:`, or you can enclose the line in a `<text>` element. Like the `@:` operator, `<text>` tags are used by ASP.NET to identify text content and are never rendered in the page output.

```
@if(IsPost) {

    // Repeat the previous example, but use <text> tags.

    <text>

    The time is: <br /> @DateTime.Now

    <br/>

    @DateTime.Now is the <em>current</em> time.

    </text>

}

@{

    var minTemp = 75;

    <text>It is the month of @DateTime.Now.ToString("MMMM"), and

    it's a <em>great</em> day! <br /><p>You can go swimming if it's at

    least @minTemp degrees. </p></text>

}
```


Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

The first example repeats the previous example but uses a single pair of `<text>` tags to enclose the text to render. In the second example, the `<text>` and `</text>` tags enclose three lines, all of which have some uncontained text and unmatched HTML tags (`
`), along with server code and matched HTML tags. Again, you could also precede each line individually with the `@:` operator; either way works.

Note When you output text as shown in this section — using an HTML element, the `@:` operator, or the `<text>` element — ASP.NET doesn't HTML-encode the output. (As noted earlier, ASP.NET does encode the output of server code expressions and server code blocks that are preceded by `@`, except in the special cases noted in this section.)

Whitespace

Extra spaces in a statement (and outside of a string literal) don't affect the statement:

```
@{ var lastName = "Smith"; }
```

A line break in a statement has no effect on the statement, and you can wrap statements for readability. The following statements are the same:

```
@{ var theName =
```

```
"Smith"; }
```

```
@{
```

```
var
```

```
personName
```

```
=
```

```
"Smith"
```

```
;
```

```
}
```

However, you can't wrap a line in the middle of a string literal. The following example doesn't work:

```
@{ var test = "This is a long
```

```
string"; } // Does not work!
```

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

To combine a long string that wraps to multiple lines like the above code, there are two options. You can use the concatenation operator (+), which you'll see later in this article. You can also use the @ character to create a verbatim string literal, as you saw earlier in this article. You can break verbatim string literals across lines:

```
@{ var longString = @"This is a  
  
long  
  
string";  
  
}
```

Code (and Markup) Comments

Comments let you leave notes for yourself or others. They also allow you to disable (*comment out*) a section of code or markup that you don't want to run but want to keep in your page for the time being.

There's different commenting syntax for Razor code and for HTML markup. As with all Razor code, Razor comments are processed (and then removed) on the server before the page is sent to the browser. Therefore, the Razor commenting syntax lets you put comments into the code (or even into the markup) that you can see when you edit the file, but that users don't see, even in the page source.

For ASP.NET Razor comments, you start the comment with @* and end it with *@. The comment can be on one line or multiple lines:

```
@* A one-line code comment. *@  
  
@*  
  
This is a multiline code comment.  
  
It can continue for any number of lines.  
  
*@
```

Here is a comment within a code block:

```
@{  
  
    @* This is a comment. *@  
  
    var theVar = 17;  
  
}
```

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

```
}
```

Here is the same block of code, with the line of code commented out so that it won't run:

```
@{  
  
    @* This is a comment. *@  
  
    @* var theVar = 17; *@  
  
}
```

Inside a code block, as an alternative to using Razor comment syntax, you can use the commenting syntax of the programming language you're using, such as C#:

```
@{  
  
    // This is a comment.  
  
    var myVar = 17;  
  
    /* This is a multi-line comment  
    that uses C# commenting syntax. */  
  
}
```

In C#, single-line comments are preceded by the `//` characters, and multi-line comments begin with `/*` and end with `*/`. (As with Razor comments, C# comments are not rendered to the browser.)

For markup, as you probably know, you can create an HTML comment:

```
<!-- This is a comment. -->
```

HTML comments start with `<!--` characters and end with `-->`. You can use HTML comments to surround not only text, but also any HTML markup that you may want to keep in the page but don't want to render. This HTML comment will hide the entire content of the tags and the text they contain:

```
<!-- <p>This is my paragraph.</p> -->
```

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

Unlike Razor comments, HTML comments *are* rendered to the page and the user can see them by viewing the page source.

Variables

A variable is a named object that you use to store data. You can name variables anything, but the name must begin with an alphabetic character and it cannot contain whitespace or reserved characters.

Variables and Data Types

A variable can have a specific data type, which indicates what kind of data is stored in the variable. You can have string variables that store string values (like "Hello world"), integer variables that store whole-number values (like 3 or 79), and date variables that store date values in a variety of formats (like 4/12/2012 or March 2009). And there are many other data types you can use.

However, you generally don't have to specify a type for a variable. Most of the time, ASP.NET can figure out the type based on how the data in the variable is being used. (Occasionally you must specify a type; you'll see examples where this is true.)

You declare a variable using the **var** keyword (if you don't want to specify a type) or by using the name of the type:

```
@{  
  
    // Assigning a string to a variable.  
  
    var greeting = "Welcome!";  
  
    // Assigning a number to a variable.  
  
    var theCount = 3;  
  
    // Assigning an expression to a variable.  
  
    var monthlyTotal = theCount + 5;  
  
    // Assigning a date value to a variable.  
  
    var today = DateTime.Today;  
  
    // Assigning the current page's URL to a variable.  
  
    var myPath = this.Request.Url;  
  
    // Declaring variables using explicit data types.  
  
    string name = "Joe";
```

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

```
int count = 5;

DateTime tomorrow = DateTime.Now.AddDays(1);

}
```

The following example shows some typical uses of variables in a web page:

```
@{

    // Embedding the value of a variable into HTML markup.

    <p>@greeting, friends!</p>

    // Using variables as part of an inline expression.

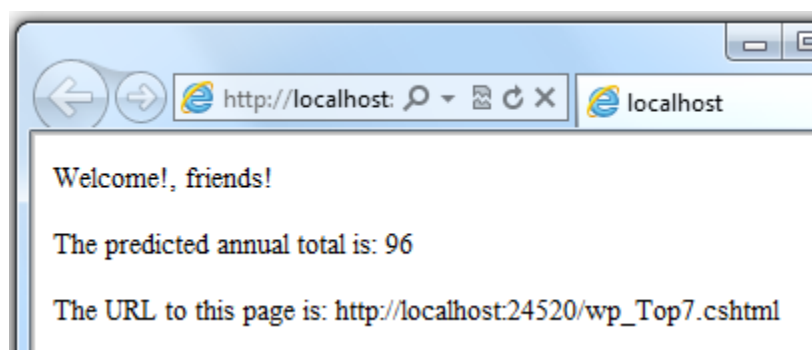
    <p>The predicted annual total is: @( monthlyTotal * 12)</p>

    // Displaying the page URL with a variable.

    <p>The URL to this page is: @myPath</p>

}
```

If you combine the previous examples in a page, you see this displayed in a browser:



Converting and Testing Data Types

Although ASP.NET can usually determine a data type automatically, sometimes it can't. Therefore, you might need to help ASP.NET out by performing an explicit conversion. Even if you don't have to convert types, sometimes it's helpful to test to see what type of data you might be working with.

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

The most common case is that you have to convert a string to another type, such as to an integer or date. The following example shows a typical case where you must convert a string to a number.

```
@{  
  
    var total = 0;  
  
    if(IsPost) {  
  
        // Retrieve the numbers that the user entered.  
  
        var num1 = Request["text1"];  
  
        var num2 = Request["text2"];  
  
        // Convert the entered strings into integers numbers and add.  
  
        total = num1.AsInt() + num2.AsInt();  
  
    }  
  
}
```

As a rule, user input comes to you as strings. Even if you've prompted users to enter a number, and even if they've entered a digit, when user input is submitted and you read it in code, the data is in string format. Therefore, you must convert the string to a number. In the example, if you try to perform arithmetic on the values without converting them, the following error results, because ASP.NET cannot add two strings:

Cannot implicitly convert type 'string' to 'int'.

To convert the values to integers, you call the **AsInt** method. If the conversion is successful, you can then add the numbers.

The following table lists some common conversion and test methods for variables.

Method	Description	Example
AsInt() , IsInt()	Converts a string that represents a whole number (like "593") to an integer.	<pre>var myIntNumber = 0; var myStringNum = "539";</pre>

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

		<pre>if(myStringNum.IsInt()==true){ myIntNumber = myStringNum.AsInt(); }</pre>
AsBool(), IsBool()	Converts a string like "true" or "false" to a Boolean type.	<pre>var myStringBool = "True"; var myVar = myStringBool.AsBool();</pre>
AsFloat(), IsFloat()	Converts a string that has a decimal value like "1.3" or "7.439" to a floating-point number.	<pre>var myStringFloat = "41.432895"; var myFloatNum = myStringFloat.AsFloat();</pre>
AsDecimal(), IsDecimal()	Converts a string that has a decimal value like "1.3" or "7.439" to a decimal number. (In ASP.NET, a decimal number is more precise than a floating-point number.)	<pre>var myStringDec = "10317.425"; var myDecNum = myStringDec.AsDecimal();</pre>
AsDateTime(), IsDateTime()	Converts a string that represents a date and time value to the ASP.NET DateTime type.	<pre>var myDateString = "12/27/2012"; var newDate = myDateString.AsDateTime();</pre>
ToString()	Converts any other data type to a string.	<pre>int num1 = 17; int num2 = 76;</pre>

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

		<pre>// myString is set to 1776 string myString = num1.ToString() + num2.ToString();</pre>
--	--	--

Operators

An operator is a keyword or character that tells ASP.NET what kind of command to perform in an expression. The C# language (and the Razor syntax that's based on it) supports many operators, but you only need to recognize a few to get started. The following table summarizes the most common operators.

Operator	Description	Examples
<pre>+</pre> <pre>-</pre> <pre>*</pre> <pre>/</pre>	Math operators used in numerical expressions.	<pre>@(5 + 13)</pre> <pre>@{ var netWorth = 150000; }</pre> <pre>@{ var newTotal = netWorth * 2; }</pre> <pre>@(newTotal / 2)</pre>
<pre>=</pre>	Assignment. Assigns the value on the right side of a statement to the object on the left side.	<pre>var age = 17;</pre>
<pre>==</pre>	Equality. Returns true if the values are equal. (Notice the distinction between the = operator and the == operator.)	<pre>var myNum = 15;</pre> <pre>if (myNum == 15) {</pre> <pre> // Do something.</pre> <pre>}</pre>

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

!=	Inequality. Returns true if the values are not equal.	<pre>var theNum = 13; if (theNum != 15) { // Do something. }</pre>
< > <= >=	Less-than, greater-than, less-than-or-equal, and greater-than-or-equal.	<pre>if (2 < 3) { // Do something. } var currentCount = 12; if(currentCount >= 12) { // Do something. }</pre>
+	Concatenation, which is used to join strings. ASP.NET knows the difference between this operator and the addition operator based on the data type of the expression.	<pre>// The displayed result is "abcdef". @"abc" + "def")</pre>
+= -=	The increment and decrement operators, which add and subtract 1 (respectively) from a variable.	<pre>int theCount = 0; theCount += 1; // Adds 1 to count</pre>

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

.	Dot. Used to distinguish objects and their properties and methods.	<pre>var myUrl = Request.Url; var count = Request["Count"].AsInt();</pre>
()	Parentheses. Used to group expressions and to pass parameters to methods.	<pre>@(3 + 7) @Request.MapPath(Request.FilePath);</pre>
[]	Brackets. Used for accessing values in arrays or collections.	<pre>var income = Request["AnnualIncome"];</pre>
!	Not. Reverses a true value to false and vice versa. Typically used as a shorthand way to test for false (that is, for not true).	<pre>bool taskCompleted = false; // Processing. if(!taskCompleted) { // Continue processing }</pre>
&& 	Logical AND and OR, which are used to link conditions together.	<pre>bool myTaskCompleted = false; int totalCount = 0; // Processing. if(!myTaskCompleted && totalCount < 12) {</pre>

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

		<pre>// Continue processing. }</pre>
--	--	---

Working with File and Folder Paths in Code

You'll often work with file and folder paths in your code. Here is an example of physical folder structure for a website as it might appear on your development computer:

```
C:\WebSites\MyWebSite  
  default.cshtml  
  datafile.txt  
  \images  
    Logo.jpg  
  \styles  
    Styles.css
```

Here are some essential details about URLs and paths:

- A URL begins with either a domain name (*http://www.example.com*) or a server name (*http://localhost*, *http://mycomputer*).
- A URL corresponds to a physical path on a host computer. For example, *http://myserver* might correspond to the folder *C:\websites\mywebsite* on the server.
- A virtual path is shorthand to represent paths in code without having to specify the full path. It includes the portion of a URL that follows the domain or server name. When you use virtual paths, you can move your code to a different domain or server without having to update the paths.

Here's an example to help you understand the differences:

Complete URL	<i>http://mycompanyserver/humanresources/CompanyPolicy.htm</i>
Server name	<i>mycompanyserver</i>
Virtual path	<i>/humanresources/CompanyPolicy.htm</i>
Physical path	<i>C:\mywebsites\humanresources\CompanyPolicy.htm</i>

The virtual root is /, just like the root of your C: drive is \. (Virtual folder paths always use forward slashes.) The virtual path of a folder doesn't have to have the same name as the physical folder; it can be an alias. (On production servers, the virtual path rarely matches an exact physical path.)

When you work with files and folders in code, sometimes you need to reference the physical path and sometimes a virtual path, depending on what objects you're working with. ASP.NET gives you these tools for working with file and folder paths in code: the **Server.MapPath** method, and the **~** operator and **Href** method.

Converting virtual to physical paths: the **Server.MapPath** method

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

The `Server.MapPath` method converts a virtual path (like `/default.cshtml`) to an absolute physical path (like `C:\WebSites\MyWebSiteFolder\default.cshtml`). You use this method any time you need a complete physical path. A typical example is when you're reading or writing a text file or image file on the web server.

You typically don't know the absolute physical path of your site on a hosting site's server, so this method can convert the path you do know — the virtual path — to the corresponding path on the server for you. You pass the virtual path to a file or folder to the method, and it returns the physical path:

```
@{  
  
    var dataFilePath = "~/dataFile.txt";  
  
}  
  
<!-- Displays a physical path C:\Websites\MyWebSite\datafile.txt -->  
  
<p>@Server.MapPath(dataFilePath)</p>
```

Referencing the virtual root: the `~` operator and `Href` method

In a `.cshtml` or `.vbhtml` file, you can reference the virtual root path using the `~` operator. This is very handy because you can move pages around in a site, and any links they contain to other pages won't be broken. It's also handy in case you ever move your website to a different location. Here are some examples:

```
@{  
  
    var myImagesFolder = "~/images";  
  
    var myStyleSheet = "~/styles/StyleSheet.css";  
  
}
```

If the website is `http://myserver/myapp`, here's how ASP.NET will treat these paths when the page runs:

- `myImagesFolder`: `http://myserver/myapp/images`
- `myStyleSheet`: `http://myserver/myapp/styles/Stylesheet.css`

(You won't actually see these paths as the values of the variable, but ASP.NET will treat the paths as if that's what they were.)

In **ASP.NET Web Pages 2**, you can use the `~` operator both in server code (as above) and in markup, like this:

```
<!-- Examples of using the ~ operator in markup in ASP.NET Web Pages 2 -->
```

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

<!-- (Using the ~ operator like this in markup is not supported in ASP.NET

Web Pages 1.0) -->

Home

In markup, you use the ~ operator to create paths to resources like image files, other web pages, and CSS files. When the page runs, ASP.NET looks through the page (both code and markup) and resolves all the ~ references to the appropriate path.

In **ASP.NET Web Pages 1**, you can use the ~ operator in server code blocks, like the first example above. But in order to use it in markup, you have to put the ~ operator inside a call to the **Href** method. (ASP.NET does not parse through the markup looking for the ~ operator.)

For example, you can use the **Href** method in HTML markup for attributes of **** elements, **<link>** elements, and **<a>** elements. Notice that the **Href** method is preceded by **@** to mark it as server code. Also notice that the **Href** method is inside the double quotation marks that enclose attribute values.

<!-- Examples of using the Href method in ASP.NET Web Pages 1.0 to include

the ~ operator in markup. -->

Home

<!-- This code creates the path "../images/Logo.jpg" in the src attribute. -->

<!-- This creates a link to the CSS file using their server variable. -->

<link rel="stylesheet" type="text/css" href="@Href(myStyleSheet)" />

Conditional Logic and Loops

ASP.NET server code lets you perform tasks based on conditions and write code that repeats statements a specific number of times (that is, code that runs a loop).

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

Testing Conditions

To test a simple condition you use the **if** statement, which returns true or false based on a test you specify:

```
@{  
  
    var showToday = true;  
  
    if(showToday)  
  
    {  
  
        @DateTime.Today;  
  
    }  
  
}
```

The **if** keyword starts a block. The actual test (condition) is in parentheses and returns true or false. The statements that run if the test is true are enclosed in braces. An **if** statement can include an **else** block that specifies statements to run if the condition is false:

```
@{  
  
    var showToday = false;  
  
    if(showToday)  
  
    {  
  
        @DateTime.Today;  
  
    }  
  
    else  
  
    {  
  
        <text>Sorry!</text>  
  
    }  
  
}
```

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

```
}  
  
}
```

You can add multiple conditions using an **else if** block:

```
@{  
  
    var theBalance = 4.99;  
  
    if(theBalance == 0)  
  
    {  
  
        <p>You have a zero balance.</p>  
  
    }  
  
    else if (theBalance > 0 && theBalance <= 5)  
  
    {  
  
        <p>Your balance of $@theBalance is very low.</p>  
  
    }  
  
    else  
  
    {  
  
        <p>Your balance is: $@theBalance</p>  
  
    }  
  
}
```

In this example, if the first condition in the if block is not true, the **else if** condition is checked. If that condition is met, the statements in the **else if** block are executed. If none of the conditions are met, the statements in the **else** block are executed. You can add any number of else if blocks, and then close with an **else** block as the "everything else" condition.

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

To test a large number of conditions, use a **switch** block:

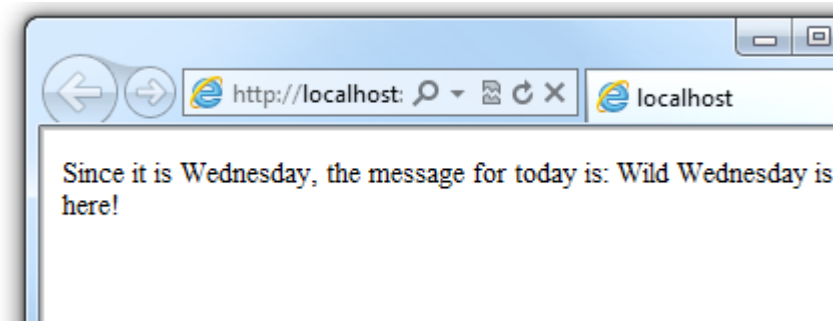
```
@{  
  
    var weekday = "Wednesday";  
  
    var greeting = "";  
  
    switch(weekday)  
  
    {  
  
        case "Monday":  
  
            greeting = "Ok, it's a marvelous Monday";  
  
            break;  
  
        case "Tuesday":  
  
            greeting = "It's a tremendous Tuesday";  
  
            break;  
  
        case "Wednesday":  
  
            greeting = "Wild Wednesday is here!";  
  
            break;  
  
        default:  
  
            greeting = "It's some other day, oh well.";  
  
            break;  
  
    }  
  
    <p>Since it is @weekday, the message for today is: @greeting</p>
```



```
}
```

The value to test is in parentheses (in the example, the **weekday** variable). Each individual test uses a **case** statement that ends with a colon (:). If the value of a **case** statement matches the test value, the code in that case block is executed. You close each case statement with a **break** statement. (If you forget to include break in each **case** block, the code from the next **case** statement will run also.) A **switch** block often has a **default** statement as the last case for an "everything else" option that runs if none of the other cases are true.

The result of the last two conditional blocks displayed in a browser:



Looping Code

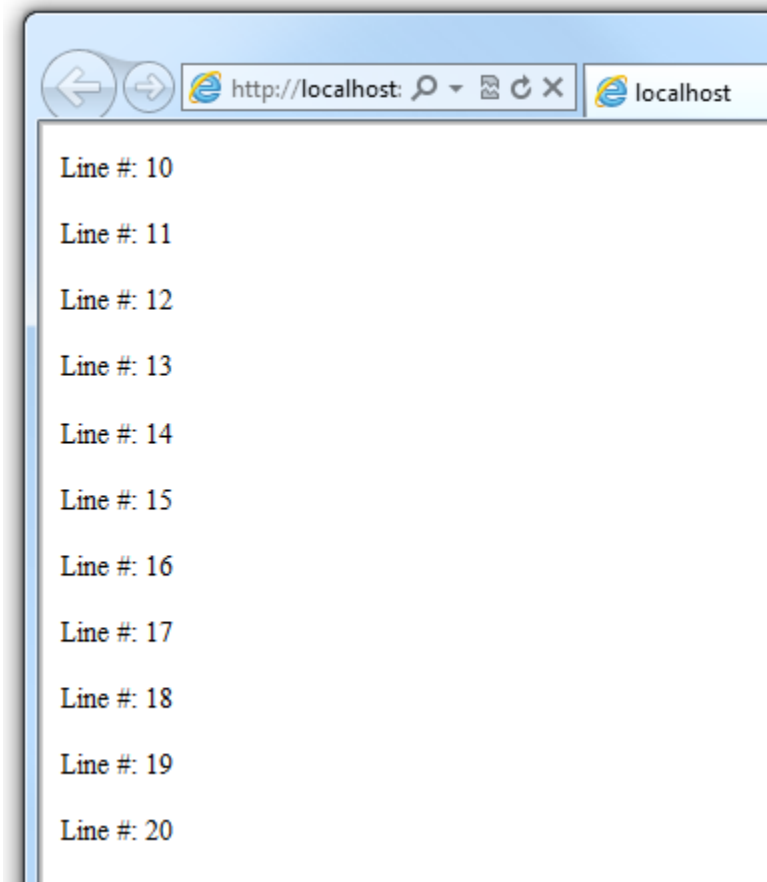
You often need to run the same statements repeatedly. You do this by looping. For example, you often run the same statements for each item in a collection of data. If you know exactly how many times you want to loop, you can use a **for** loop. This kind of loop is especially useful for counting up or counting down:

```
@for(var i = 10; i < 21; i++)  
  
{  
  
    <p>Line #: @i</p>  
  
}
```

The loop begins with the **for** keyword, followed by three statements in parentheses, each terminated with a semicolon.

- Inside the parentheses, the first statement (**var i=10;**) creates a counter and initializes it to 10. You don't have to name the counter **i** — you can use any variable. When the **for** loop runs, the counter is automatically incremented.
- The second statement (**i < 21;**) sets the condition for how far you want to count. In this case, you want it to go to a maximum of 20 (that is, keep going while the counter is less than 21).
- The third statement (**i++**) uses an increment operator, which simply specifies that the counter should have 1 added to it each time the loop runs.

Inside the braces is the code that will run for each iteration of the loop. The markup creates a new paragraph (**<p>** element) each time and adds a line to the output, displaying the value of **i** (the counter). When you run this page, the example creates 11 lines displaying the output, with the text in each line indicating the item number.



If you're working with a collection or array, you often use a **foreach** loop. A collection is a group of similar objects, and the **foreach** loop lets you carry out a task on each item in the collection. This type of loop is convenient for collections, because unlike a **for** loop, you don't have to increment the counter or set a limit. Instead, the **foreach** loop code simply proceeds through the collection until it's finished.

For example, the following code returns the items in the **Request.ServerVariables** collection, which is an object that contains information about your web server. It uses a **foreach** loop to display the name of each item by creating a new **** element in an HTML bulleted list.

```
<ul>

@foreach (var myItem in Request.ServerVariables)

{

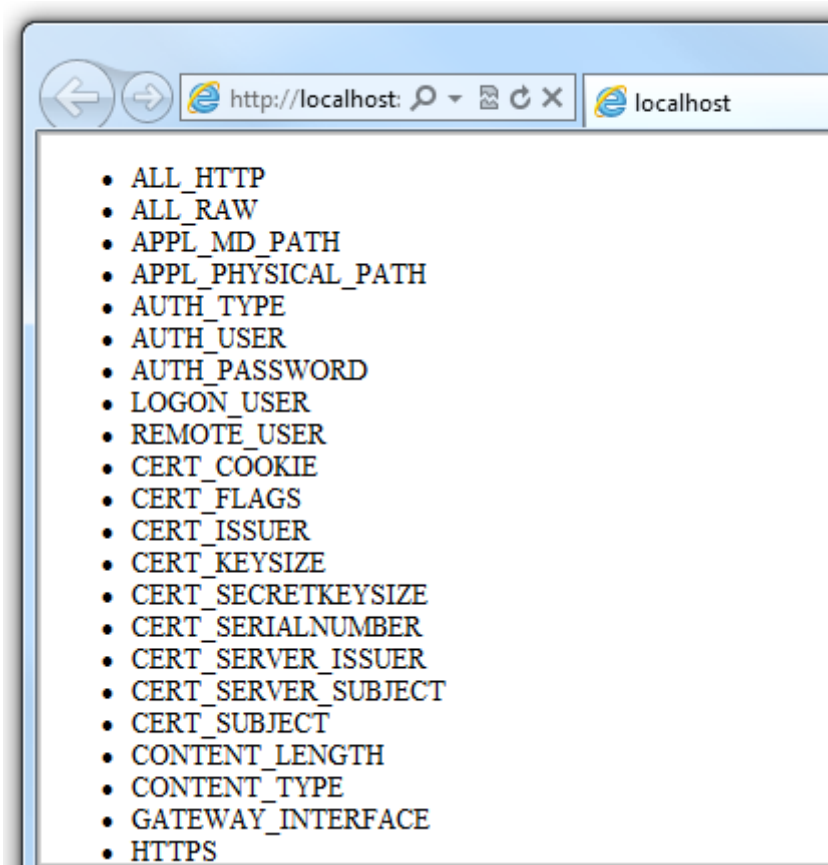
    <li>@myItem</li>

}

</ul>
```

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

The **foreach** keyword is followed by parentheses where you declare a variable that represents a single item in the collection (in the example, **var item**), followed by the **in** keyword, followed by the collection you want to loop through. In the body of the **foreach** loop, you can access the current item using the variable that you declared earlier.



To create a more general-purpose loop, use the **while** statement:

```
@{  
  
    var countNum = 0;  
  
    while (countNum < 50)  
  
    {  
  
        countNum += 1;  
  
        <p>Line #@countNum: </p>  
  
    }  
}
```

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

```
}
```

A **while** loop begins with the **while** keyword, followed by parentheses where you specify how long the loop continues (here, for as long as **countNum** is less than 50), then the block to repeat. Loops typically increment (add to) or decrement (subtract from) a variable or object used for counting. In the example, the **+=** operator adds 1 to **countNum** each time the loop runs. (To decrement a variable in a loop that counts down, you would use the decrement operator **-=**).

Objects and Collections

Nearly everything in an ASP.NET website is an object, including the web page itself. This section discusses some important objects you'll work with frequently in your code.

Page Objects

The most basic object in ASP.NET is the page. You can access properties of the page object directly without any qualifying object. The following code gets the page's file path, using the **Request** object of the page:

```
@{  
  
    var path = Request.FilePath;  
  
}
```

To make it clear that you're referencing properties and methods on the current page object, you can optionally use the keyword **this** to represent the page object in your code. Here is the previous code example, with **this** added to represent the page:

```
@{  
  
    var path = this.Request.FilePath;  
  
}
```

You can use properties of the **Page** object to get a lot of information, such as:

- **Request**. As you've already seen, this is a collection of information about the current request, including what type of browser made the request, the URL of the page, the user identity, etc.
- **Response**. This is a collection of information about the response (page) that will be sent to the browser when the server code has finished running. For example, you can use this property to write information into the response.

```
@{  
  
    // Access the page's Request object to retrieve the Url.
```

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

```
var pageUrl = this.Request.Url;

}

<a href="@pageUrl">My page</a>
```

Collection Objects (Arrays and Dictionaries)

A *collection* is a group of objects of the same type, such as a collection of **Customer** objects from a database. ASP.NET contains many built-in collections, like the **Request.Files** collection.

You'll often work with data in collections. Two common collection types are the *array* and the *dictionary*. An array is useful when you want to store a collection of similar items but don't want to create a separate variable to hold each item:

```
@* Array block 1: Declaring a new array using braces. *@

@{

    <h3>Team Members</h3>

    string[] teamMembers = {"Matt", "Joanne", "Robert", "Nancy"};

    foreach (var person in teamMembers)

    {

        <p>@person</p>

    }

}
```

With arrays, you declare a specific data type, such as **string**, **int**, or **DateTime**. To indicate that the variable can contain an array, you add brackets to the declaration (such as **string[]** or **int[]**). You can access items in an array using their position (index) or by using the **foreach** statement. Array indexes are zero-based — that is, the first item is at position 0, and the second item is at position 1, and so on.

```
@{

    string[] teamMembers = {"Matt", "Joanne", "Robert", "Nancy"};
```

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

<p>The number of names in the teamMembers array: @teamMembers.Length </p>

<p>Robert is now in position: @Array.IndexOf(teamMembers, "Robert")</p>

<p>The array item at position 2 (zero-based) is @teamMembers[2]</p>

<h3>Current order of team members in the list</h3>

foreach (var name in teamMembers)

{

<p>@name</p>

}

<h3>Reversed order of team members in the list</h3>

Array.Reverse(teamMembers);

foreach (var reversedItem in teamMembers)

{

<p>@reversedItem</p>

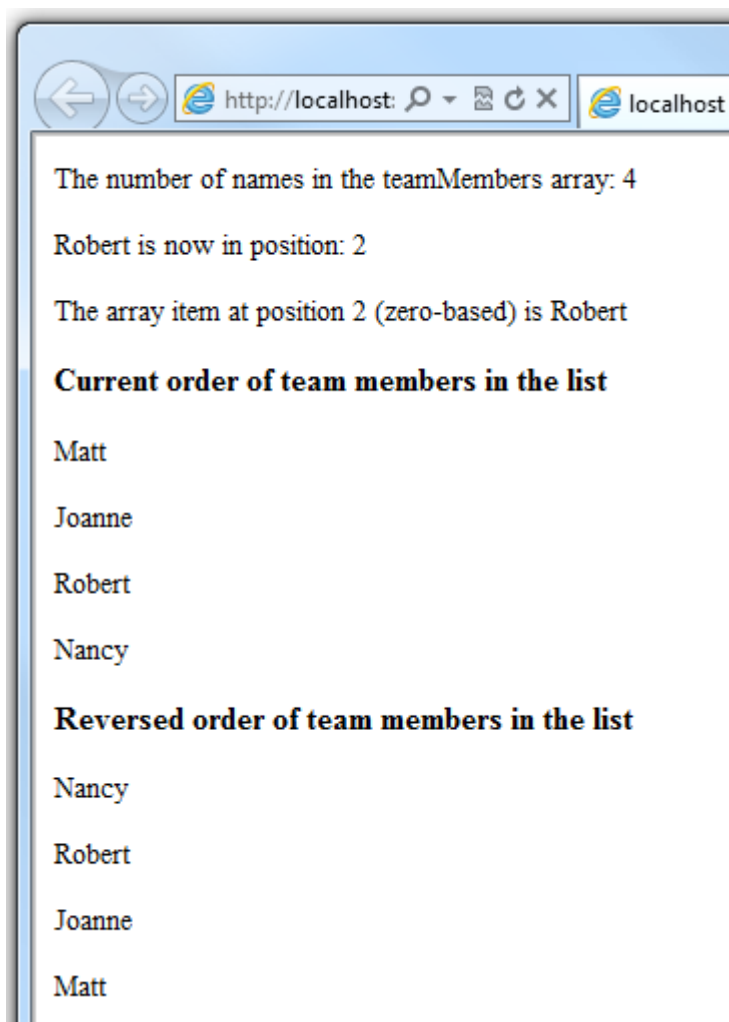
}

}

You can determine the number of items in an array by getting its **Length** property. To get the position of a specific item in the array (to search the array), use the **Array.IndexOf** method. You can also do things like reverse the contents of an array (the **Array.Reverse** method) or sort the contents (the **Array.Sort** method).

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

The output of the string array code displayed in a browser:



A dictionary is a collection of key/value pairs, where you provide the key (or name) to set or retrieve the corresponding value:

```
@{  
  
    var myScores = new Dictionary<string, int>();  
  
    myScores.Add("test1", 71);  
  
    myScores.Add("test2", 82);  
  
    myScores.Add("test3", 100);  
  
    myScores.Add("test4", 59);  
  
}
```

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

```
<p>My score on test 3 is: @myScores["test3"]%</p>
```

```
@(myScores["test4"] = 79)
```

```
<p>My corrected score on test 4 is: @myScores["test4"]%</p>
```

To create a dictionary, you use the **new** keyword to indicate that you're creating a new dictionary object. You can assign a dictionary to a variable using the **var** keyword. You indicate the data types of the items in the dictionary using angle brackets (**< >**). At the end of the declaration, you must add a pair of parentheses, because this is actually a method that creates a new dictionary.

To add items to the dictionary, you can call the **Add** method of the dictionary variable (**myScores** in this case), and then specify a key and a value. Alternatively, you can use square brackets to indicate the key and do a simple assignment, as in the following example:

```
myScores["test4"] = 79;
```

To get a value from the dictionary, you specify the key in brackets:

```
var testScoreThree = myScores["test3"];
```

Calling Methods with Parameters

As you read earlier in this article, the objects that you program with can have methods. For example, a **Database** object might have a **Database.Connect** method. Many methods also have one or more parameters. A *parameter* is a value that you pass to a method to enable the method to complete its task. For example, look at a declaration for the **Request.MapPath** method, which takes three parameters:

```
public string MapPath(string virtualPath, string baseVirtualDir,  
  
    bool allowCrossAppMapping);
```

(The line has been wrapped to make it more readable. Remember that you can put line breaks almost any place except inside strings that are enclosed in quotation marks.)

This method returns the physical path on the server that corresponds to a specified virtual path. The three parameters for the method are **virtualPath**, **baseVirtualDir**, and **allowCrossAppMapping**. (Notice that in the declaration, the parameters are listed with the data types of the data that they'll accept.) When you call this method, you must supply values for all three parameters.

The Razor syntax gives you two options for passing parameters to a method: *positional parameters* and *named parameters*. To call a method using positional parameters, you pass the parameters in a strict order that's specified in the method declaration. (You would typically know this order by reading documentation for the method.) You must follow the order, and you can't skip any of the parameters — if necessary, you pass an empty string ("") or **null** for a positional parameter that you don't have a value for.

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

The following example assumes you have a folder named *scripts* on your website. The code calls the `Request.MapPath` method and passes values for the three parameters in the correct order. It then displays the resulting mapped path.

```
@{  
  
    // Pass parameters to a method using positional parameters.  
  
    var myPathPositional = Request.MapPath("/scripts", "/", true);  
  
}  
  
<p>@myPathPositional</p>
```

When a method has many parameters, you can keep your code more readable by using named parameters. To call a method using named parameters, you specify the parameter name followed by a colon (:), and then the value. The advantage of named parameters is that you can pass them in any order you want. (A disadvantage is that the method call is not as compact.)

The following example calls the same method as above, but uses named parameters to supply the values:

```
@{  
  
    // Pass parameters to a method using named parameters.  
  
    var myPathNamed = Request.MapPath(baseVirtualDir: "/",  
  
        allowCrossAppMapping: true, virtualPath: "/scripts");  
  
}  
  
<p>@myPathNamed</p>
```

As you can see, the parameters are passed in a different order. However, if you run the previous example and this example, they'll return the same value.

Handling Errors

Try-Catch Statements

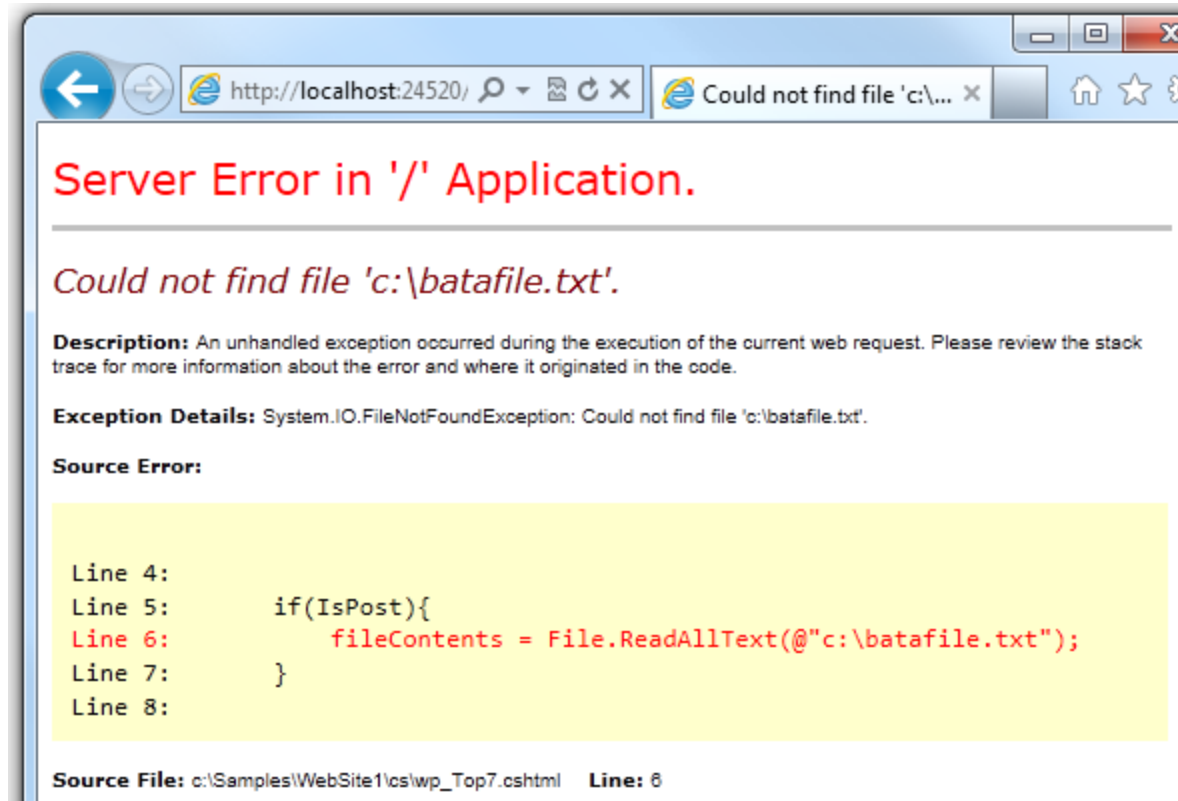
You'll often have statements in your code that might fail for reasons outside your control. For example:

- If your code tries to create or access a file, all sorts of errors might occur. The file you want might not exist, it might be locked, the code might not have permissions, and so on.

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

- Similarly, if your code tries to update records in a database, there can be permissions issues, the connection to the database might be dropped, the data to save might be invalid, and so on.

In programming terms, these situations are called *exceptions*. If your code encounters an exception, it generates (throws) an error message that's, at best, annoying to users:



In situations where your code might encounter exceptions, and in order to avoid error messages of this type, you can use **try/catch** statements. In the **try** statement, you run the code that you're checking. In one or more **catch** statements, you can look for specific errors (specific types of exceptions) that might have occurred. You can include as many **catch** statements as you need to look for errors that you are anticipating.

Note We recommend that you avoid using the **Response.Redirect** method in **try/catch** statements, because it can cause an exception in your page.

The following example shows a page that creates a text file on the first request and then displays a button that lets the user open the file. The example deliberately uses a bad file name so that it will cause an exception. The code includes **catch** statements for two possible exceptions: **FileNotFoundException**, which occurs if the file name is bad, and **DirectoryNotFoundException**, which occurs if ASP.NET can't even find the folder. (You can uncomment a statement in the example in order to see how it runs when everything works properly.)

If your code didn't handle the exception, you would see an error page like the previous screen shot.

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..
However, the **try/catch** section helps prevent the user from seeing these types of errors.

```
@{

    var dataFilePath = "~/dataFile.txt";

    var fileContents = "";

    var physicalPath = Server.MapPath(dataFilePath);

    var userMessage = "Hello world, the time is " + DateTime.Now;

    var userErrMsg = "";

    var errMsg = "";

    if(IsPost)

    {

        // When the user clicks the "Open File" button and posts

        // the page, try to open the created file for reading.

        try {

            // This code fails because of faulty path to the file.

            fileContents = File.ReadAllText(@"c:\batafile.txt");

            // This code works. To eliminate error on page,

            // comment the above line of code and uncomment this one.

            //fileContents = File.ReadAllText(physicalPath);

        }

        catch (FileNotFoundException ex) {
```

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

```
// You can use the exception object for debugging, logging, etc.

errMsg = ex.Message;

// Create a friendly error message for users.

userErrMsg = "A file could not be opened, please contact "

    + "your system administrator.";

}

catch (DirectoryNotFoundException ex) {

    // Similar to previous exception.

    errMsg = ex.Message;

    userErrMsg = "A directory was not found, please contact "

        + "your system administrator.";

}

}

else

{

    // The first time the page is requested, create the text file.

    File.WriteAllText(physicalPath, userMessage);

}

}
```

<!DOCTYPE html>

Getting Started with ASP.NET MVC 5 & Entity Framework – Hello World/Movie App..

```
<html lang="en">

<head>

  <meta charset="utf-8" />

  <title>Try-Catch Statements</title>

</head>

<body>

  <form method="POST" action="" >

    <input type="Submit" name="Submit" value="Open File"/>

  </form>

  <p>@fileContents</p>

  <p>@userErrMsg</p>

</body>

</html>
```