**ASP.NET MVC 5.x Tutorial**

(http://www.asp.net/mvc/overview/getting-started/introduction/)  
Autor: Rick Anderson, October 17, 2013

**Inhalt**

[Getting Started 3](#_Toc417213736)

[Adding a Controller 8](#_Toc417213737)

[Adding a View 16](#_Toc417213738)

[Changing Views and Layout Pages 20](#_Toc417213739)

[Passing Data from the Controller to the View 25](#_Toc417213740)

[Adding a Model 29](#_Toc417213741)

[Creating a Connection String and Working with SQL Server LocalDB 33](#_Toc417213742)

[Creating a Connection String and Working with SQL Server LocalDB 33](#_Toc417213743)

[SQL Server Express LocalDB 33](#_Toc417213744)

[Accessing Your Model's Data from a Controller 36](#_Toc417213745)

[Creating a Movie 39](#_Toc417213746)

[Examining the Generated Code 40](#_Toc417213747)

[Strongly Typed Models and the @model Keyword 41](#_Toc417213748)

[Working with SQL Server LocalDB 44](#_Toc417213749)

[Examining the Edit Methods and Edit View 49](#_Toc417213750)

[Search 61](#_Toc417213751)

[Adding a Search Method and Search View 61](#_Toc417213752)

[Updating the Index Form 61](#_Toc417213753)

[Adding Search by Genre 68](#_Toc417213754)

[Adding Markup to the Index View to Support Search by Genre 69](#_Toc417213755)

[Adding a New Field 72](#_Toc417213756)

[Setting up Code First Migrations for Model Changes 72](#_Toc417213757)

[Adding a Rating Property to the Movie Model 78](#_Toc417213758)

[Adding Validation 85](#_Toc417213759)

[Keeping Things DRY 85](#_Toc417213760)

[Adding Validation Rules to the Movie Model 85](#_Toc417213761)

[Validation Error UI in ASP.NET MVC 89](#_Toc417213762)

[How Validation Occurs in the Create View and Create Action Method 91](#_Toc417213763)

[Using DataType Attributes 96](#_Toc417213764)

[Examining the Details and Delete Methods 98](#_Toc417213765)

[Examining the Details and Delete Methods 98](#_Toc417213766)

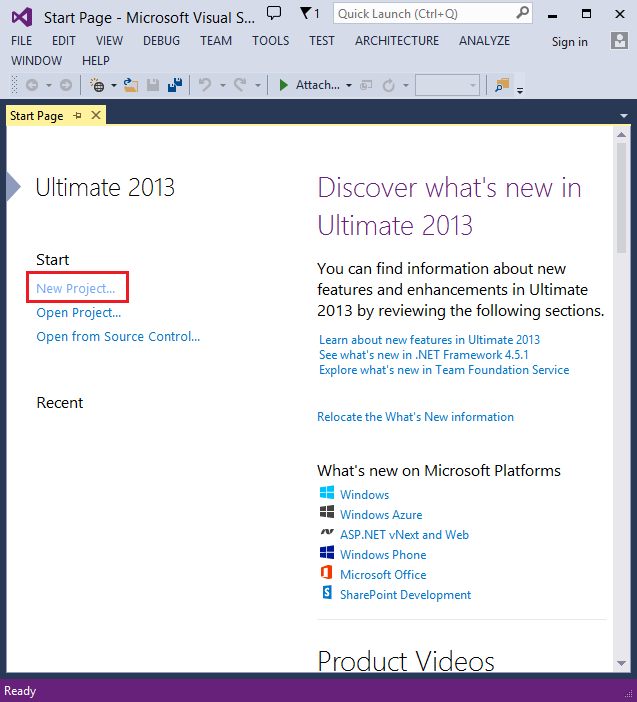
[Summary 101](#_Toc417213768)

[Next Steps 101](#_Toc417213769)

Getting Started with ASP.NET MVC 5

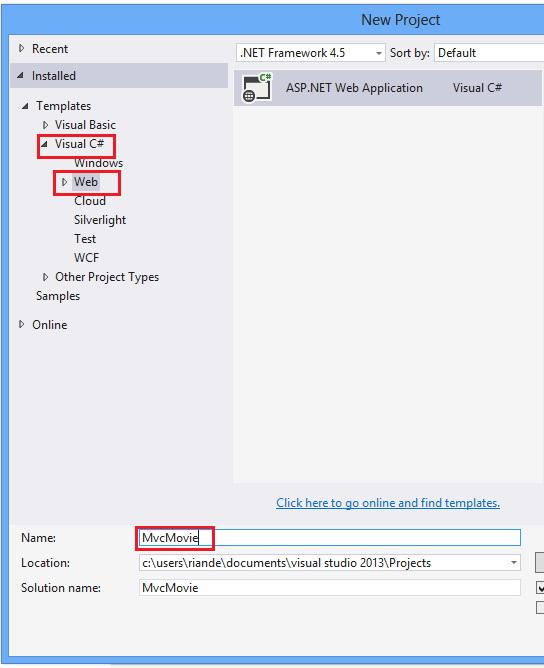
# Getting Started

Start by installing and running [Visual Studio Express 2013 for Web](http://www.microsoft.com/visualstudio/eng/2013-downloads#d-2013-express) or [Visual Studio 2013](http://www.microsoft.com/visualstudio/eng/2013-downloads).

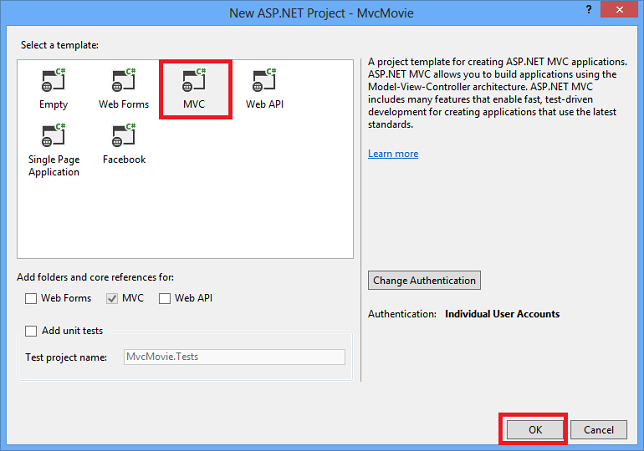
Visual Studio is an IDE, or integrated development environment. Just like you use Microsoft Word to write documents, you'll use an IDE to create applications. In Visual Studio there's a toolbar along the top showing various options available to you. There's also a menu that provides another way to perform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


Creating Your First Application

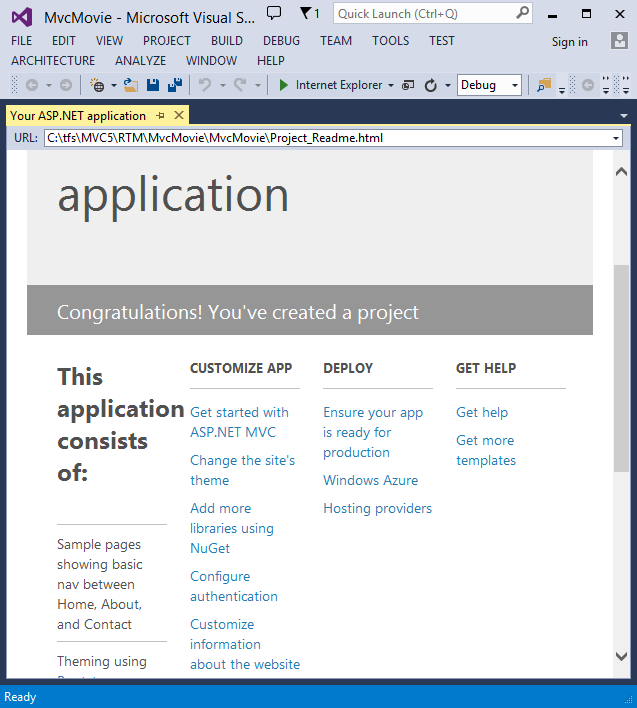
Click **New Project**, then select Visual C# on the left, then **Web** and then select **ASP.NET  Web Application**. Name your project "MvcMovie" and then click **OK**.



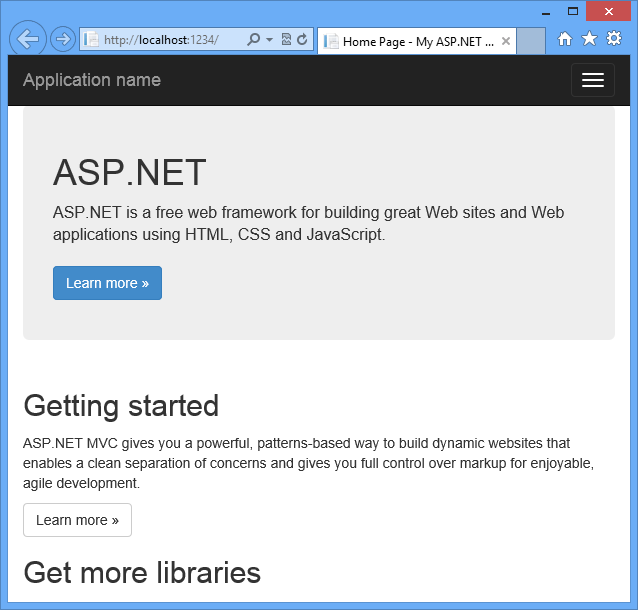
In the **New ASP.NET Project** dialog, click **MVC** and then click **OK**.



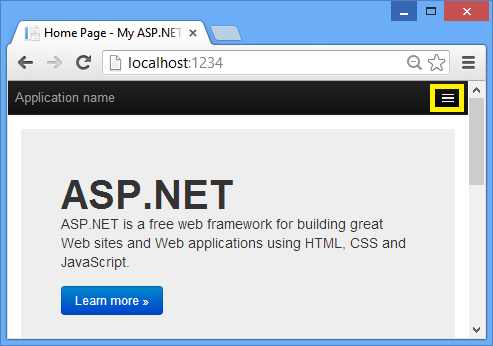
Visual Studio used a default template for the ASP.NET MVC project you just created, so you have a working application right now without doing anything! This is a simple "Hello World!" project, and it's a good place to start your application.

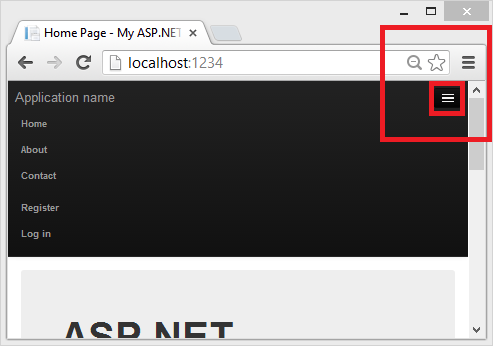


Click F5 to start debugging. F5 causes Visual Studio to start [**IIS Express**](http://www.iis.net/learn/extensions/introduction-to-iis-express/iis-express-overview) and run your web app. Visual Studio then launches a browser and opens the application's home page. Notice that the address bar of the browser says localhost:port# and not something like example.com. That's because localhost always points to your own local computer, which in this case is running the application you just built. When Visual Studio runs a web project, a random port is used for the web server. In the image below, the port number is 1234. When you run the application, you'll see a different port number.



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

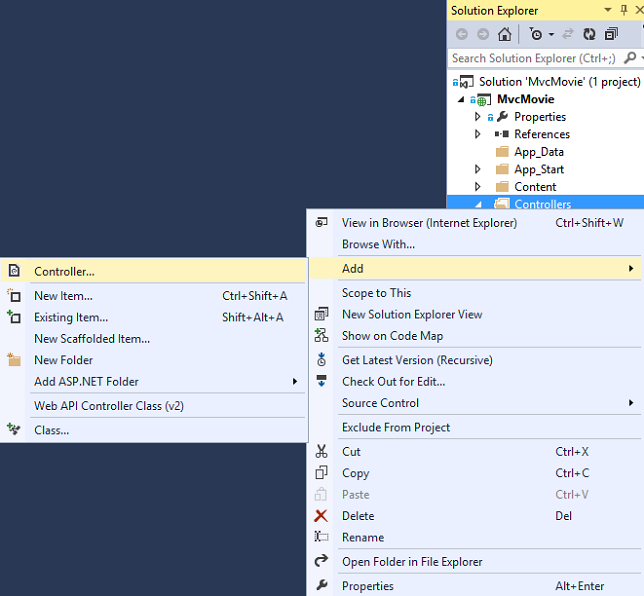




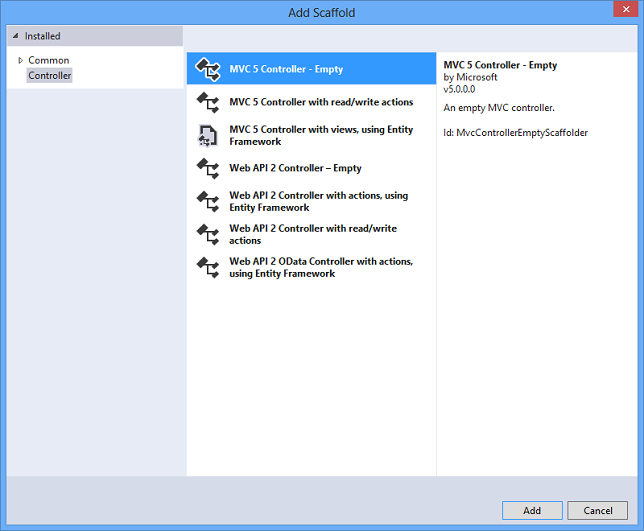
The application also provides support to register and log in. The next step is to change how this application works and learn a little bit about ASP.NET MVC. Close the ASP.NET MVC application and let's change some code.

# Adding a Controller

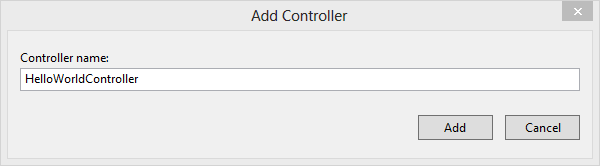
Let's begin by creating a controller class. In **Solution Explorer**, right-click the Controllers folder  and then click**Add**,  then **Controller**.



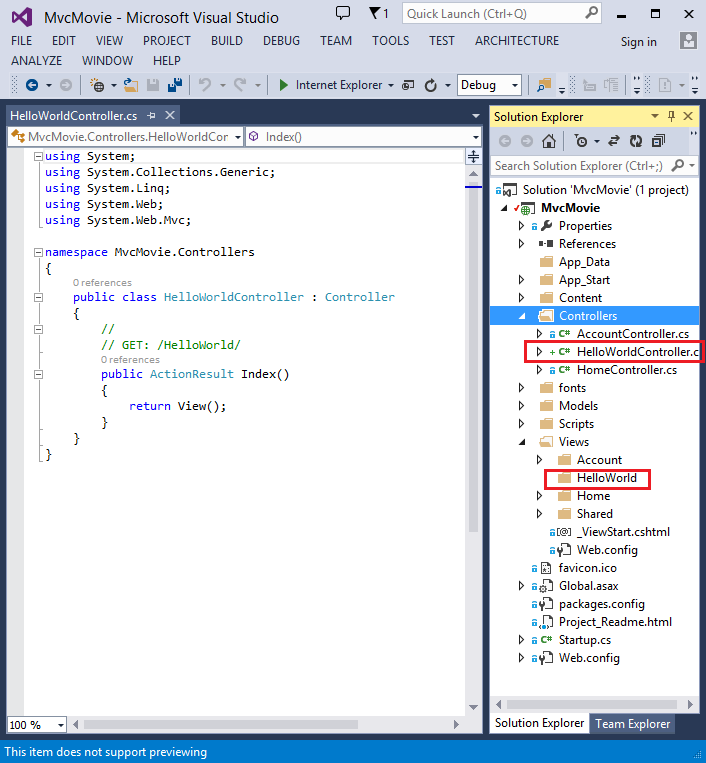
In the **Add Scaffold** dialog box, click **MVC 5  Controller - Empty**, and then click **Add**.



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

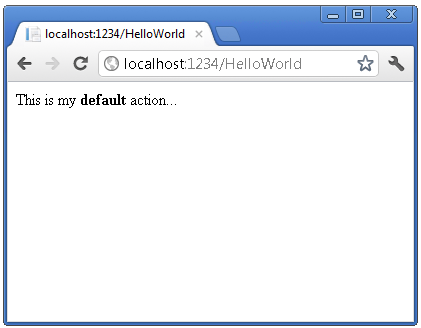


Notice in **Solution  Explorer** that a new file  has been created named HelloWorldController.cs and a new folder Views\HelloWorld. The controller is open in the IDE.





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



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

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

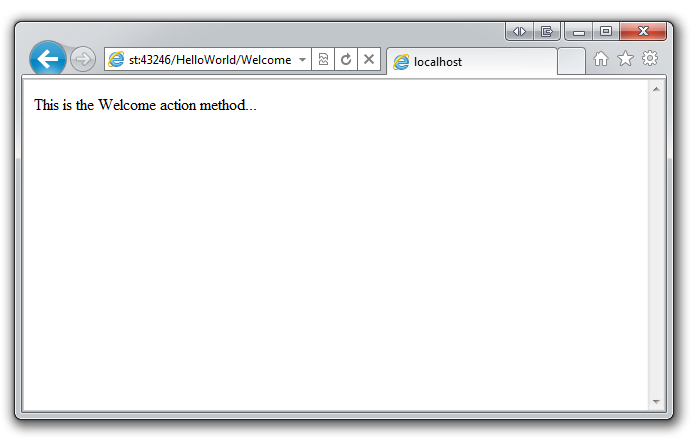
You set the format for routing in the App\_Start/RouteConfig.cs  file.



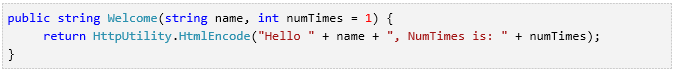
When you run the application and don't supply any URL segments, it defaults to  the "Home" controller and the "Index" action method specified in the defaults  section of the code above.

The first part of the URL determines the controller class to execute. So /HelloWorld maps  to the HelloWorldController class.  The second part of the URL determines the action method on the class to execute.  So /HelloWorld/Index would  cause the Index method of the HelloWorldController class to execute. Notice that we only had to browse to /HelloWorld and  the Index method  was used by default. This is because a method named Index is the default method that will be called on a controller if one is not explicitly specified. The third part of the URL segment ( Parameters) is for route data. We'll see route data later on in this  tutorial.

Browse to http://localhost:xxxx/HelloWorld/Welcome.  The Welcome method  runs and returns the string "This is the Welcome action method...". The  default MVC mapping is /[Controller]/[ActionName]/[Parameters].  For this URL, the controller is HelloWorld and Welcome is  the action method. You haven't used the [Parameters] part  of the URL yet.

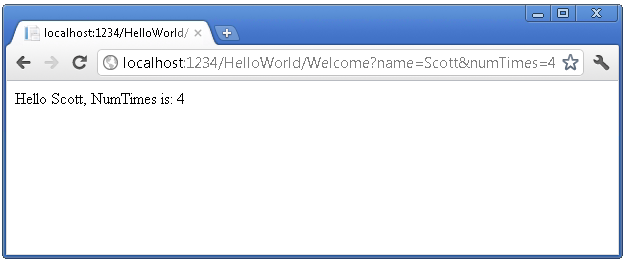


Let's modify the example slightly so that you can pass some parameter information  from the URL to the controller (for example, /HelloWorld/Welcome?name=Scott&numtimes=4).  Change your Welcome method  to parameters as shown below. Note that the code uses the C# optional-parameter feature to indicate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



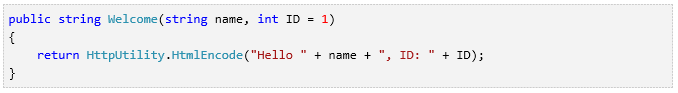
Security Note: The code above uses [HttpServerUtility.HtmlEncode](http://msdn.microsoft.com/en-us/library/w3te6wfz.aspx) to protect the application from malicious input (namely JavaScript).  For more information see [How to: Protect Against Script Exploits in a Web Application by Applying HTML Encoding to Strings](http://msdn.microsoft.com/en-us/library/a2a4yykt(v=vs.100).aspx).

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](http://odetocode.com/Blogs/scott/archive/2009/04/27/6-tips-for-asp-net-mvc-model-binding.aspx) automatically maps the named parameters from  the query string in the address bar to parameters in your method.

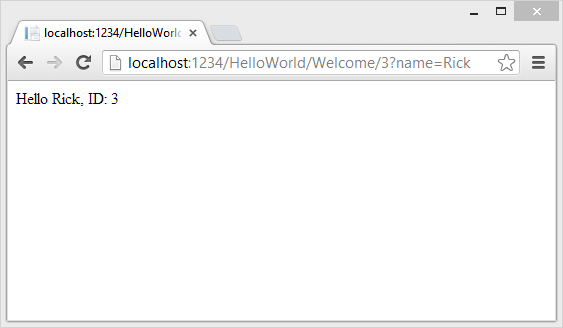


In the sample above, the URL segment ( Parameters) is not used, the name and numTimes parameters are passed as [query strings](http://en.wikipedia.org/wiki/Query_string). The ? (question mark) in the above URL is a separator, and the query strings follow. The & character separates query strings.

Replace the Welcome method with the following code:



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



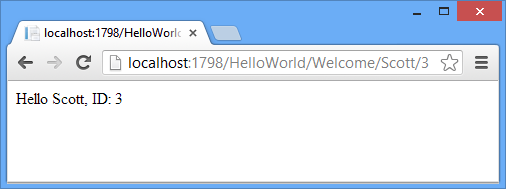
This time the third URL segment  matched the route parameter ID. The Welcome action method contains a parameter  (ID) that matched the URL specification in the RegisterRoutes method.



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:



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



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

In these examples the controller has been doing the "VC" portion  of MVC — that is, the view and controller work. The controller is returning HTML  directly. Ordinarily you don't want controllers returning HTML directly, since  that becomes very cumbersome to code. Instead we'll typically use a separate  view template file to help generate the HTML response. Let's look next at how  we can do this.

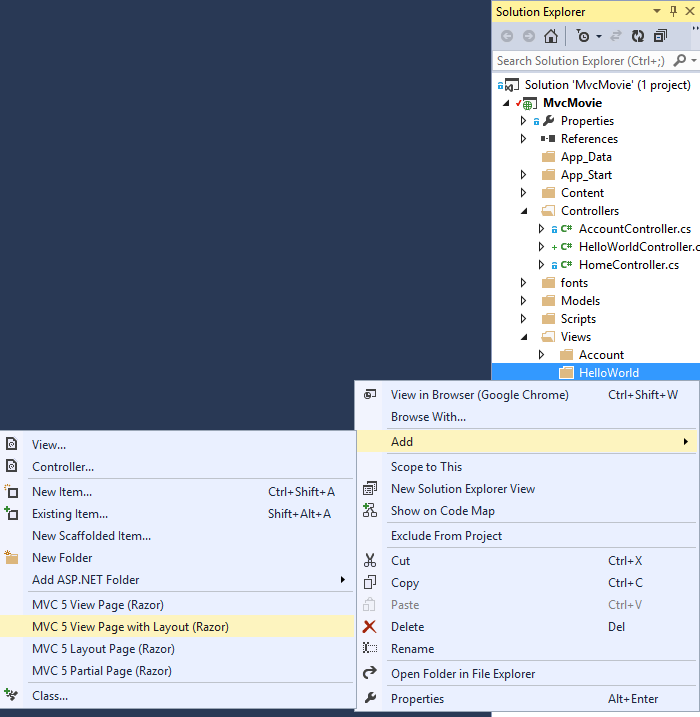
# Adding a View

You'll create a view template file using the [Razor view engine](http://www.asp.net/web-pages/tutorials/basics/2-introduction-to-asp-net-web-programming-using-the-razor-syntax). 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:

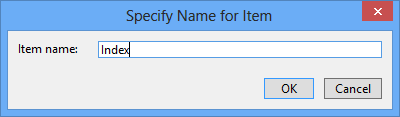


The Index method above uses a view template to generate an HTML response to the browser. Controller methods (also known as [action methods](http://rachelappel.com/asp.net-mvc-actionresults-explained)), such as the Index method above, generally return an [ActionResult](http://msdn.microsoft.com/en-us/library/system.web.mvc.actionresult.aspx) (or a class derived from [ActionResult](http://msdn.microsoft.com/en-us/library/system.web.mvc.actionresult.aspx)), not primitive types like string.

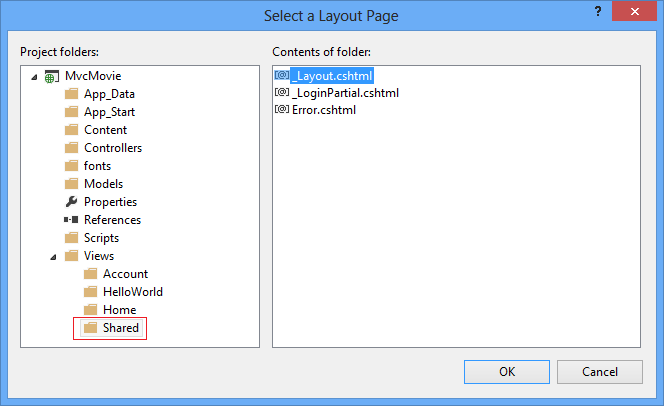
Right click the *Views\HelloWorld* folder and click **Add**, then click **MVC 5 View Page with (Layout Razor)**.

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

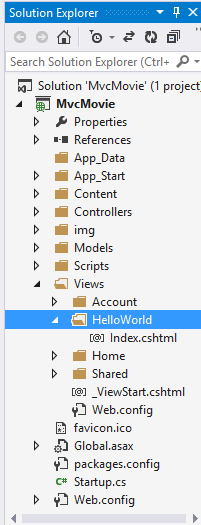


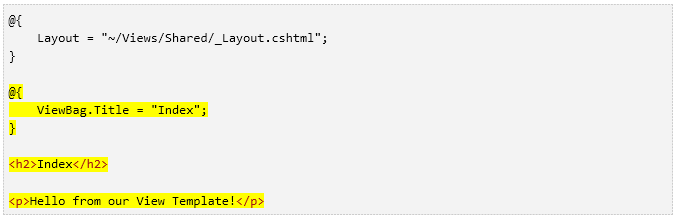
In the Select a Layout Page dialog, accept the default \_Layout.cshtml and click OK.



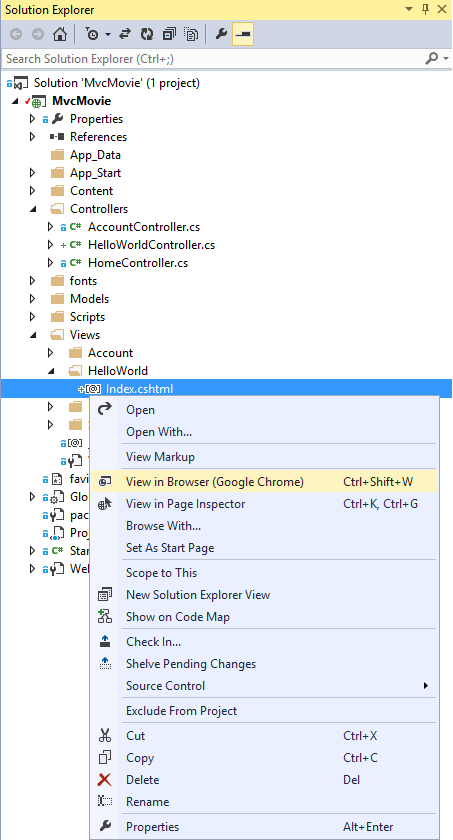
In the dialog above, the *Views\Shared* folder is selected in the left pane. If you had a custom layout file in another folder, you could select it. We'll talk about the layout file later in the tutorial

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



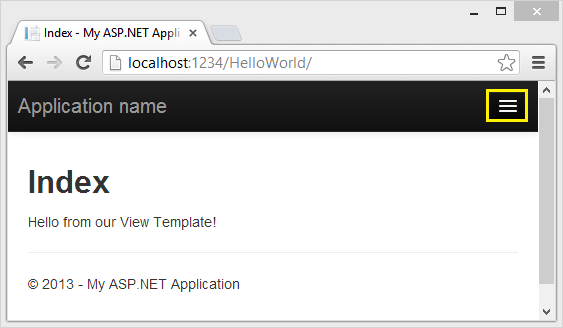


Right click the *Index.cshtml* file and select **View in Browser**.



You can also right click the *Index.cshtml* file and select **View in Page Inspector.** See the [Page Inspector tutorial](http://www.asp.net/mvc/tutorials/mvc-4/using-page-inspector-in-aspnet-mvc) for more information.

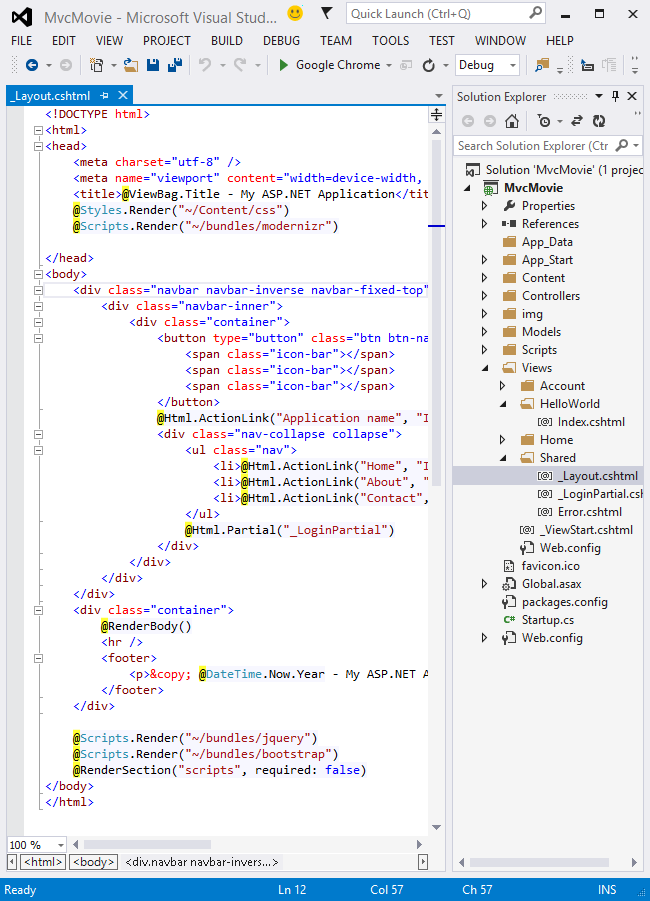
Alternatively, run the application and browse to the HelloWorld controller (*http://localhost:xxxx/HelloWorld*). The Index method in your controller didn't do much work; it simply ran the statement return View(), which specified that the method should use a view template file to render a response to the browser. Because you didn't explicitly specify the name of the view template file to use, ASP.NET MVC defaulted to using the *Index.cshtml* view file in the *\Views\HelloWorld* folder. The image below shows the string "Hello from our View Template!" hard-coded in the view.



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

## Changing Views and Layout Pages

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

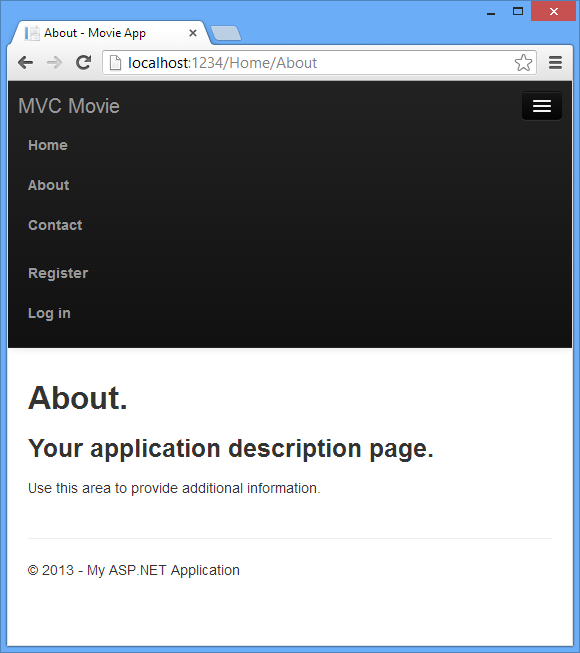


Layout templates allow you to specify the HTML container layout of your site in one place and then apply it across multiple pages in your site. Find the @RenderBody() line. [RenderBody](http://msdn.microsoft.com/en-us/gg618478) is a placeholder where all the view-specific pages you create show up, "wrapped" 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](http://msdn.microsoft.com/en-us/library/dd504972(v=vs.108).aspx) in the layout template from "Application name" to "MVC Movie" and the controller from Home to Movies. The complete layout file is shown below:

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>  <meta charset="utf-8" />  <meta name="viewport" content="width=device-width, initial-scale=1.0">  <title>@ViewBag.Title - Movie App</title>  @Styles.Render("~/Content/css")  @Scripts.Render("~/bundles/modernizr")  </head>  <body>  <div class="navbar navbar-inverse navbar-fixed-top">  <div class="container">  <div class="navbar-header">  <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".navbar-collapse">  <span class="icon-bar"></span>  <span class="icon-bar"></span>  <span class="icon-bar"></span>  </button>  @Html.ActionLink("MVC Movie", "Index", "Movies", null, new { @class = "navbar-brand" })  </div>  <div class="navbar-collapse collapse">  <ul class="nav navbar-nav">  <li>@Html.ActionLink("Home", "Index", "Home")</li>  <li>@Html.ActionLink("About", "About", "Home")</li>  <li>@Html.ActionLink("Contact", "Contact", "Home")</li>  </ul>  @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>  @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:



The Razor code above is explicted setting the layout page. Examine the Views\\_ViewStart.cshtml file, it contains the exact same Razor markup. The[Views\\_ViewStart.cshtml](http://weblogs.asp.net/scottgu/archive/2010/10/22/asp-net-mvc-3-layouts.aspx) 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.



You can use the Layout property to set a different layout view, or set it to null so no layout file will be used.

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

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



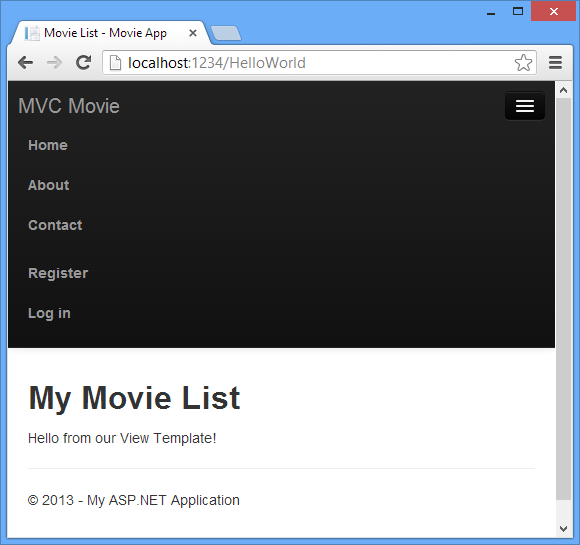
To indicate the HTML title to display, the code above sets a Title property of the [ViewBag](http://msdn.microsoft.com/en-us/library/system.web.mvc.controllerbase.viewbag(v=vs.108).aspx) object (which is in the *Index.cshtml* view template). Notice that the layout template  ( *Views\Shared\\_Layout.cshtml* ) uses this value in the <title> element as part of the <head> section of the HTML that we modified previously.



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

Run the application. Notice that the browser title, the primary heading, and the secondary headings have changed. (If you don't see changes in the browser, you might be viewing cached content. Press Ctrl+F5 in your browser to force the response from the server to be loaded.) The browser title is created with the  ViewBag.Title we set in the *Index.cshtml* view template  and the additional  "- Movie App" added in the layout file.

Also notice how the content in the *Index.cshtml* view template was merged with the *\_Layout.cshtml* view template and a single HTML response was sent to the browser. Layout templates make it really easy to make changes that apply across all of the pages in your application.



Our little bit of "data" (in this case the "Hello from our View Template!" message) is hard-coded, though. The MVC application has a "V" (view) and you've got a "C" (controller), but no "M" (model) yet. Shortly, we'll walk through how create a database and retrieve model data from it.

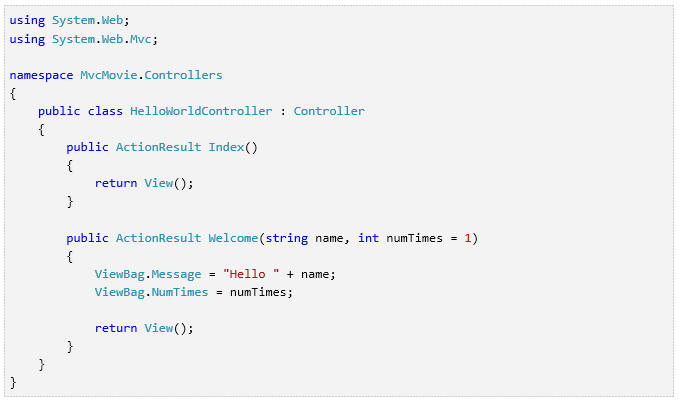
## Passing Data from the Controller to the View

Before we go to a database and talk about models, though, let's first talk about passing information from the controller to a view. Controller classes are invoked in response to an incoming URL request. A controller class is where you write the code that handles the incoming browser requests, retrieves data from a database, and ultimately decides what type of response to send back to the browser. View templates can then be used from a controller to generate and format an HTML response to the browser.

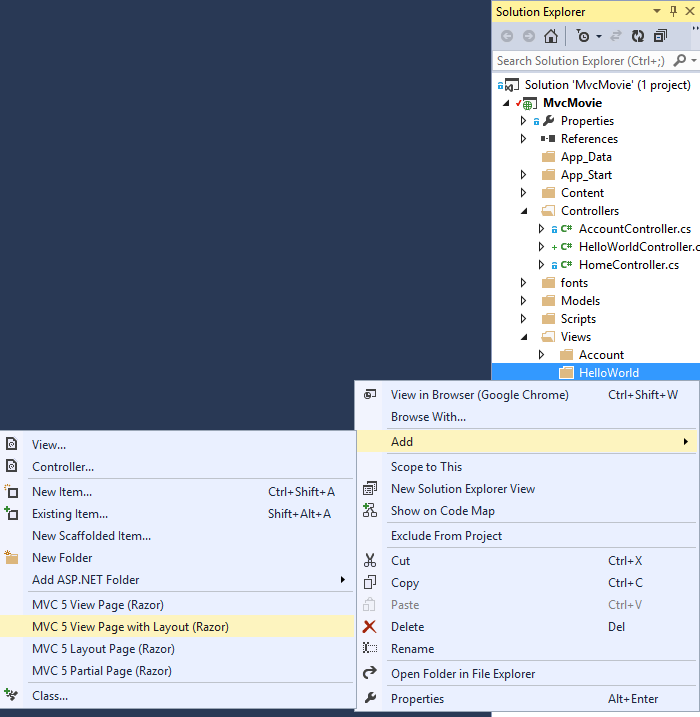
Controllers are responsible for providing whatever data or objects are required in order for a view template to render a response to the browser. A best practice: **A view template should never perform business logic or interact with a database directly**. Instead, a view template should work only with the data that's provided to it by the controller. Maintaining this "separation of concerns" helps keep your code clean, testable and more maintainable.

Currently, the Welcome action method in the HelloWorldController class takes a name and a numTimes parameter and then outputs the values directly to the browser. Rather than have the controller render this response as a string, let’s change the controller to use a view template instead. The view template will generate a dynamic response, which means that you need to pass appropriate bits of data from the controller to the view in order to generate the response. You can do this by having the controller put the dynamic data (parameters) that the view template needs in a ViewBag object that the view template can then access.

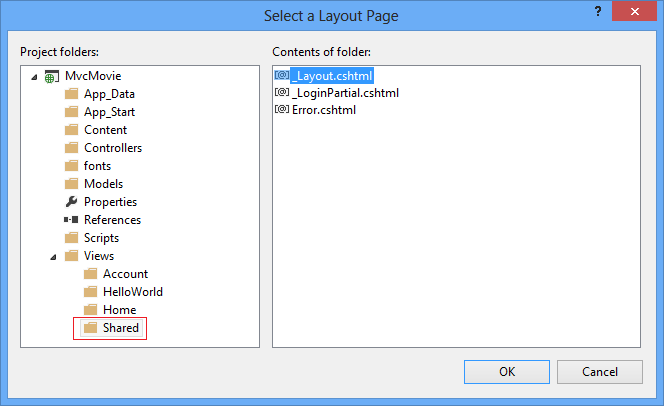
Return to the *HelloWorldController.cs* file and change the Welcome method to add a Message and NumTimes value to the [ViewBag](http://rachelappel.com/when-to-use-viewbag-viewdata-or-tempdata-in-asp.net-mvc-3-applications) object. ViewBag is a dynamic object, which means you can put whatever you want in to it; the ViewBag object has no defined properties until you put something inside it. The [ASP.NET MVC model binding system](http://odetocode.com/Blogs/scott/archive/2009/04/27/6-tips-for-asp-net-mvc-model-binding.aspx) 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:



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

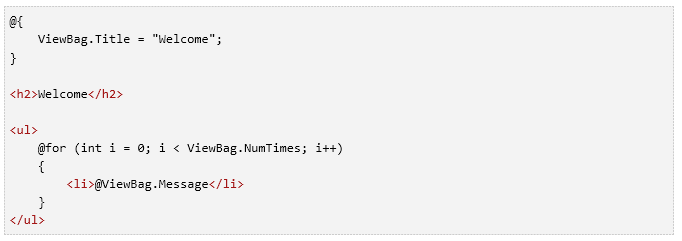
   
  
In the **Specify Name for Item** dialog box, enter Welcome, and then click **OK**.

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



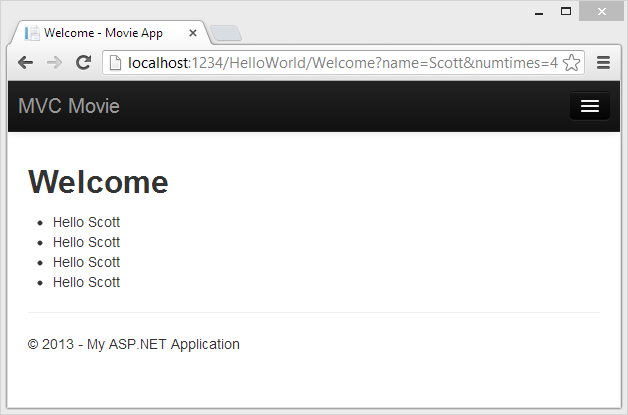
The MvcMovie\Views\HelloWorld\Welcome.cshtml file is created.

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

Run the application and browse to the following URL:

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

Now data is taken from the URL and passed to the controller using the [model binder](http://odetocode.com/Blogs/scott/archive/2009/04/27/6-tips-for-asp-net-mvc-model-binding.aspx). 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](http://blogs.msdn.com/b/rickandy/archive/2011/01/28/dynamic-v-strongly-typed-views.aspx) for more information.

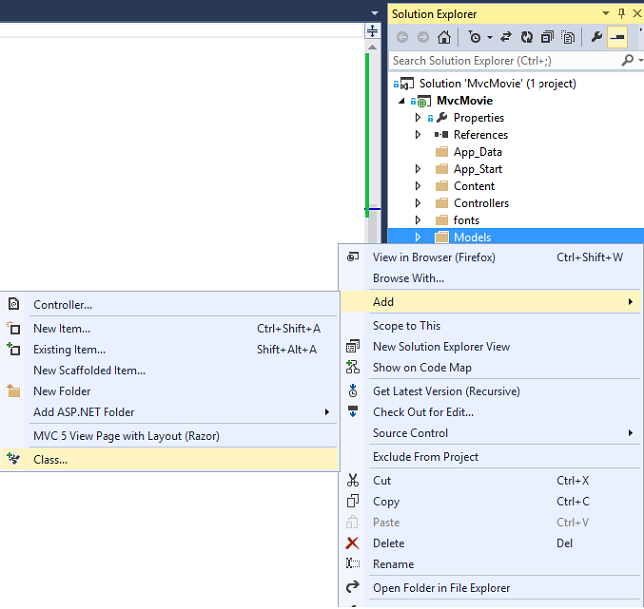
Well, that was a kind of an "M" for model, but not the database kind. Let's take what we've learned and create a database of movies.

# Adding a Model

You’ll use a .NET Framework data-access technology known as the [Entity Framework](http://msdn.microsoft.com/en-us/library/bb399572(VS.110).aspx) 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](http://www.asp.net/visual-studio/overview/2013/aspnet-scaffolding-overview) tutorial, which covers the database first approach.

Adding Model Classes

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



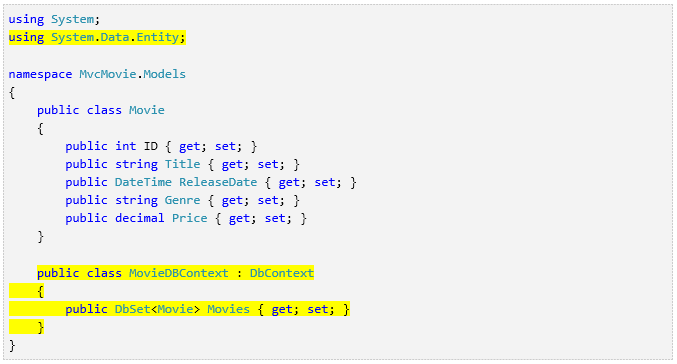
Enter the *class*  name "Movie".

Add the following five properties to the Movie class:



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.

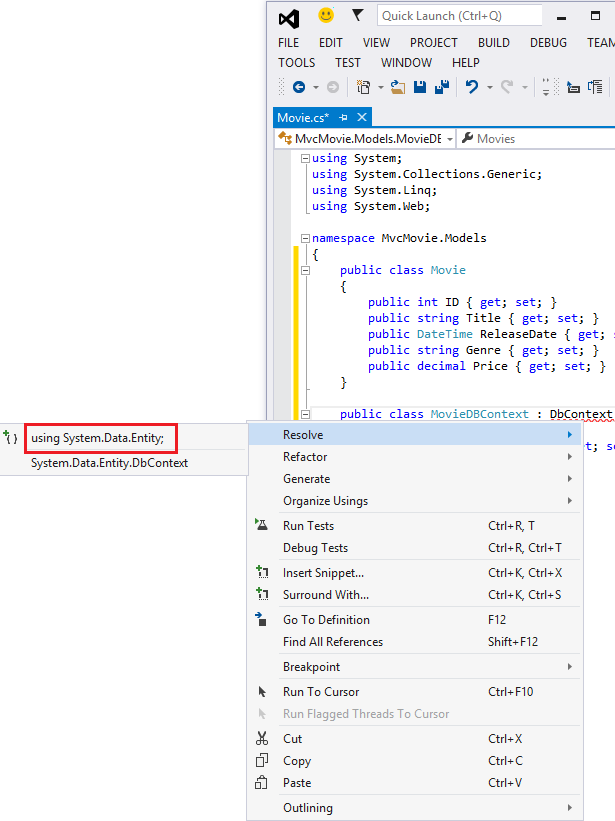
In the same file, add the following MovieDBContext class:



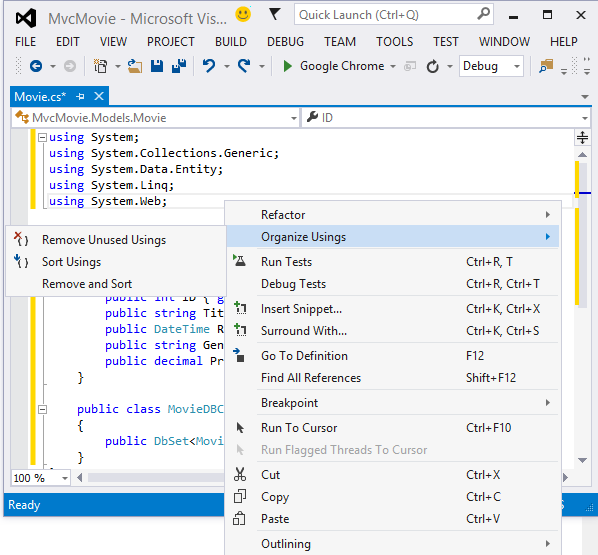
The MovieDBContext class represents the Entity Framework movie database context, which handles fetching, storing, and updating Movie class instances in a database. The MovieDBContext derives from the [DbContext](http://msdn.microsoft.com/en-us/library/system.data.entity.dbcontext(v=VS.103).aspx) base 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:

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**.



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.

# Creating a Connection String and Working with SQL Server LocalDB

## 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](http://msdn.microsoft.com/en-us/library/hh510202.aspx). In this section we'll explicitly add  a connection string in the *Web.config* file of the application.

## SQL Server Express LocalDB

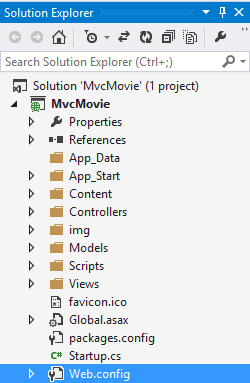
[LocalDB](http://blogs.msdn.com/b/sqlexpress/archive/2011/07/12/introducing-localdb-a-better-sql-express.aspx) is a lightweight version of the SQL Server Express Database Engine that starts on demand and runs in user mode. LocalDB runs in a special execution mode of SQL Server Express that enables you 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](http://msdn.microsoft.com/en-us/library/jj653752.aspx).

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



Find  the <connectionStrings>  element:



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" /> |

The two connection strings are very similar. The first connection string is  named DefaultConnection and is used for the membership database to control who can access the application. The connection string you've added specifies a LocalDB database named *Movie.mdf* located in the *App\_Data*  folder.  We won't use the membership database in this tutorial, for more information on membership, authentication and security, see my tutorial [Create an ASP.NET MVC app with auth and SQL DB and deploy to Azure App Service](http://www.windowsazure.com/en-us/develop/net/tutorials/web-site-with-sql-database/).

The name of the connection string must match the name of the [DbContext](http://msdn.microsoft.com/en-us/library/system.data.entity.dbcontext(v=vs.103).aspx) class.



You don't actually need to add the MovieDBContext connection string. If you don't specify a connection string, Entity Framework will create a LocalDB database in the users directory with the fully qualified name of the [DbContext](http://msdn.microsoft.com/en-us/library/system.data.entity.dbcontext(v=vs.103).aspx) 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*.

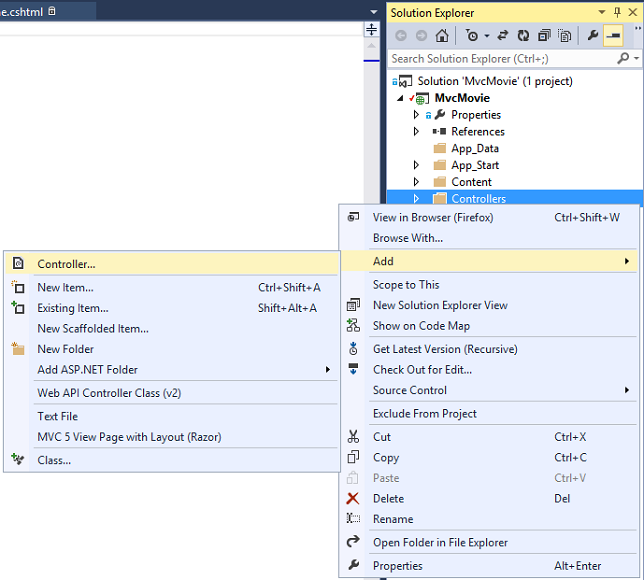
Next, you'll build a new MoviesController class that you can use  to display the movie data and allow users to create new movie listings.

# Accessing Your Model's Data from a Controller

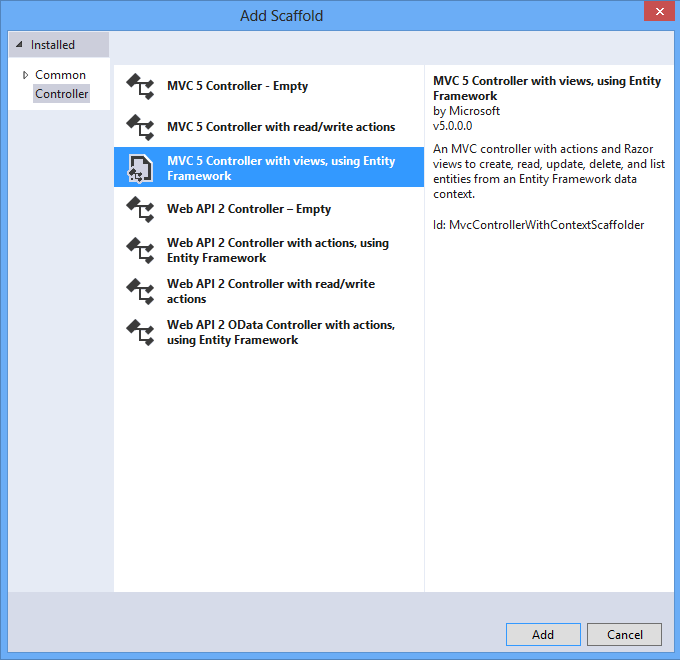
In this section, you'll create a new MoviesController class and write code that retrieves the movie data and displays it in the browser using a view template.

**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**.

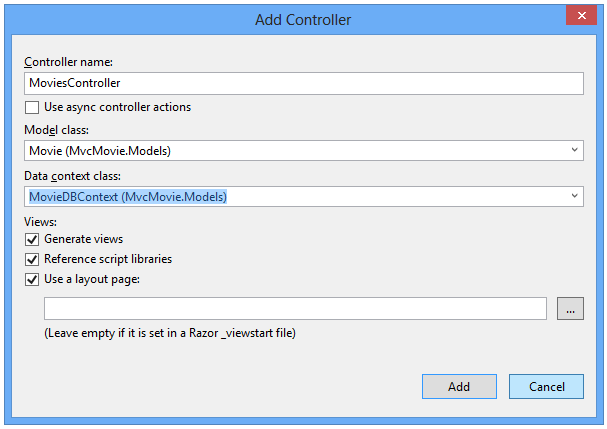


In the **Add Scaffold** dialog box, click **MVC 5  Controller with views, using Entity Framework**, and then click **Add**.



* + 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.

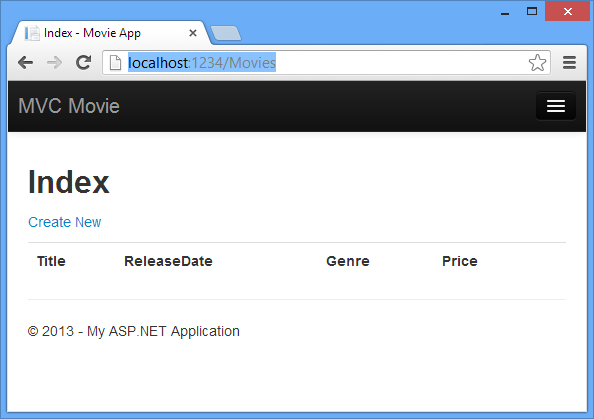


Click **Add**. (If you get an error, you probably didn't build the application before starting adding the controller.) Visual Studio creates the following files and folders:

* 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.

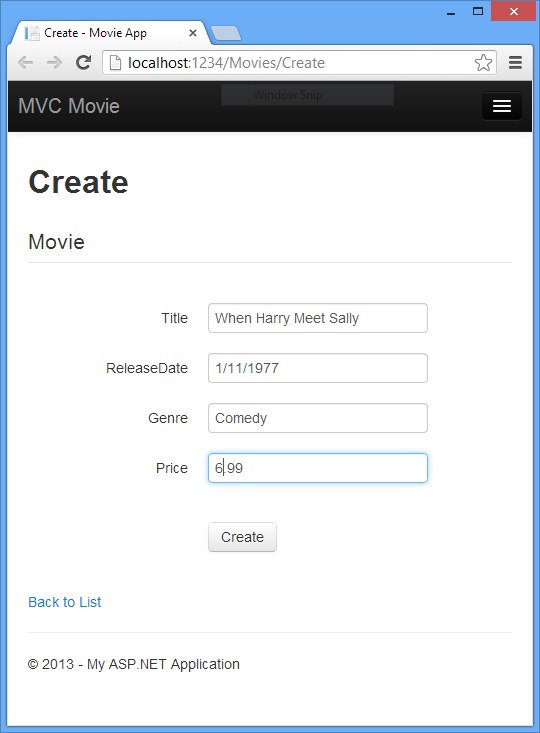
Visual Studio automatically created the [CRUD](http://en.wikipedia.org/wiki/Create,_read,_update_and_delete)  (create, read, update, and delete) action methods and views for you (the automatic creation of CRUD action methods and views is known as scaffolding). You now have a fully functional web application that lets you create, list, edit, and delete movie entries.

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



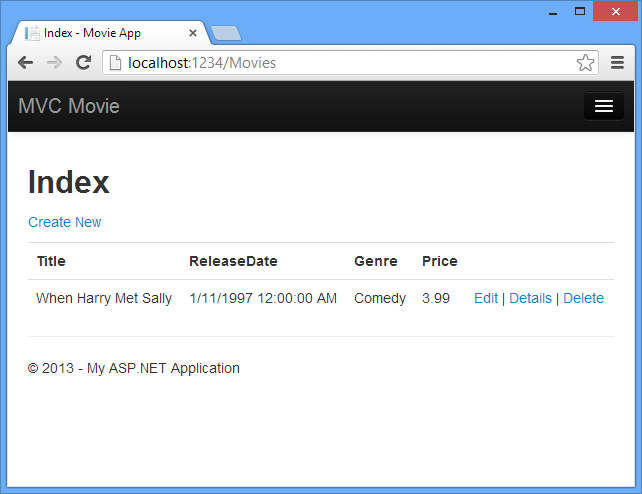
## Creating a Movie

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



**Note:** You may not be able to enter decimal points or commas in the Price field. To support jQuery validation for non-English locales that use a comma (",") for a decimal point, and non US-English date formats, you must include globalize.js and your specific cultures/globalize.cultures.js file(from <https://github.com/jquery/globalize> ) and JavaScript to use Globalize.parseFloat. I'll show how to do this in the next tutorial. For now, just enter whole numbers like 10.

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



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

## Examining the Generated Code

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



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.



## 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](http://msdn.microsoft.com/en-us/library/hcw1s69b(v=vs.120).aspx) 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.



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:



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

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:



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](http://msdn.microsoft.com/en-us/library/system.web.mvc.html.displaynameextensions.displaynamefor(v=vs.108).aspx) and  [DisplayFor](http://msdn.microsoft.com/en-us/library/system.web.mvc.html.displayextensions.displayfor(VS.98).aspx) HTML Helpers with the strongly typed Model object.  The Create and  Edit methods and  view templates also pass a movie model object.

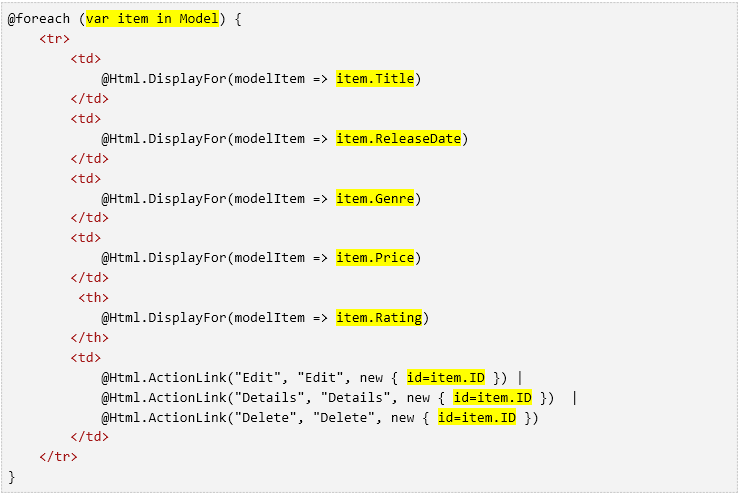
Examine the Index.cshtml view template and the Index method in the MoviesController.cs file. Notice how the code creates a [List](http://msdn.microsoft.com/en-us/library/6sh2ey19.aspx)  object when it calls the View helper method in the Index  action method. The code then passes this Movies list from the Index action method to the view:



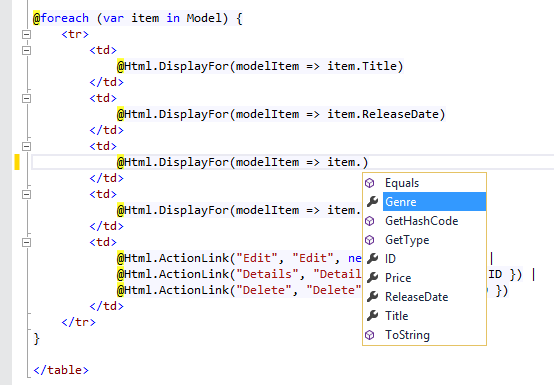
When you created  the movie controller, Visual Studio automatically included the following @model statement at the top of the Index.cshtml file:



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:

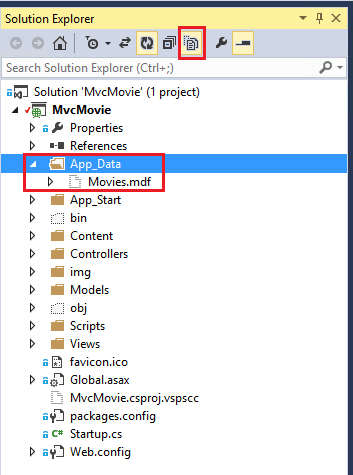


Because the Model object is strongly typed (as an IEnumerable<Movie> object), each item object in the loop is typed as Movie. Among other benefits, this means that you get  compile-time checking of the code and full IntelliSense support in the code editor:

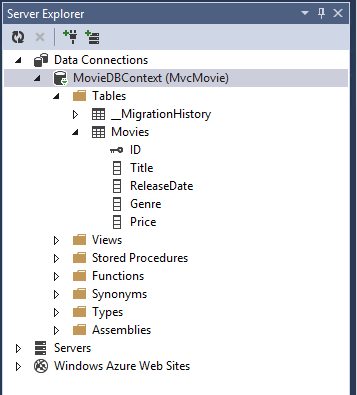


## Working with SQL Server LocalDB

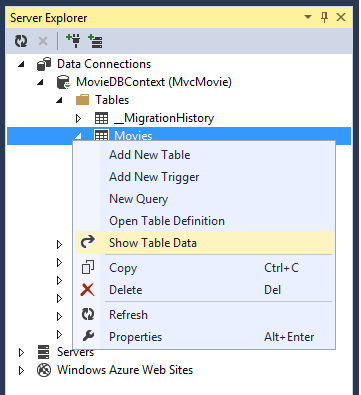
Entity Framework Code First detected that the database connection string that was provided pointed to a Movies database that didn’t exist yet, so Code First created the database automatically. You can verify that it's been  created by looking in the App\_Data folder. If you don't see the Movies.mdf file,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

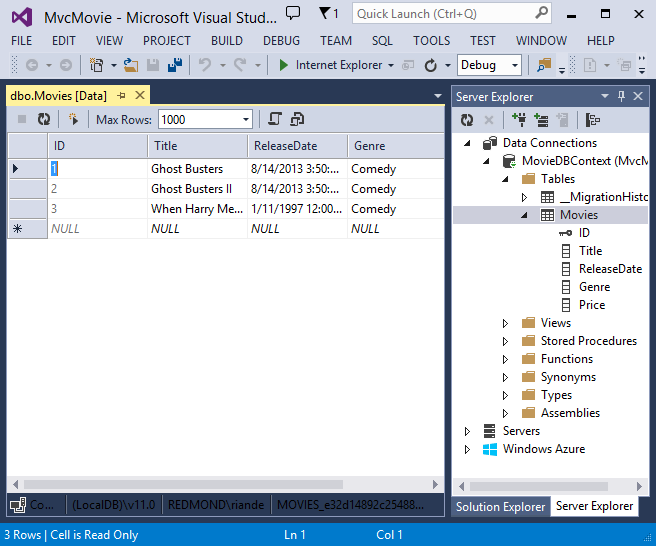


Double-click Movies.mdf to open **SERVER EXPLORER**, then expand the **Tables** folder to see the Movies table. Note the key icon next to ID. By default, EF will make a property named ID the primary key. For more information on EF and MVC, see Tom Dykstra's excellent tutorial on [MVC and EF](http://www.asp.net/mvc/tutorials/getting-started-with-ef-using-mvc/creating-an-entity-framework-data-model-for-an-asp-net-mvc-application).

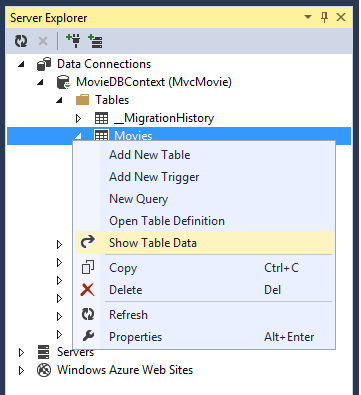


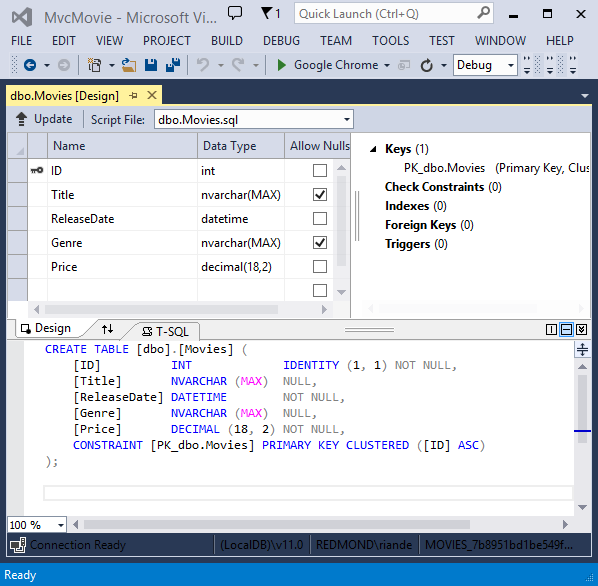
Right-click the Movies table and select **Show Table Data**  to see the data you created.





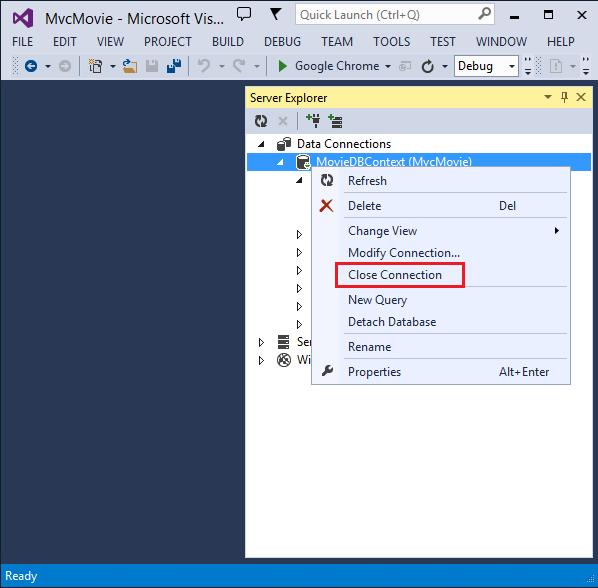
Right-click the Movies table and select **Open Table Definition** to see the table structure that Entity Framework Code First  created for you.





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

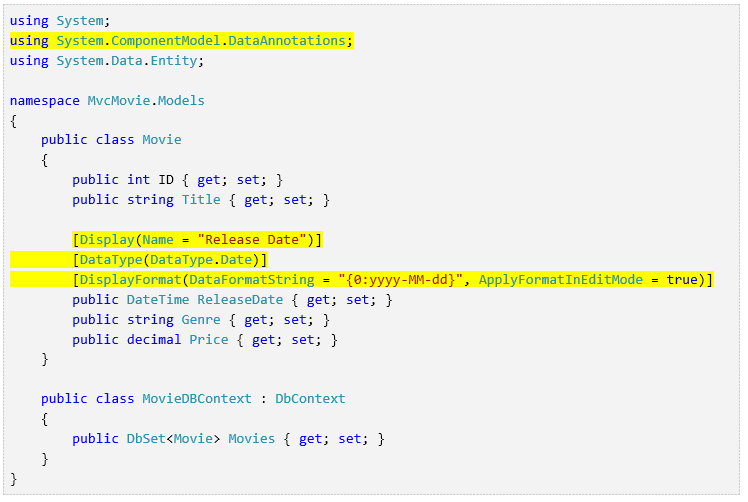
When you're finished, close the connection by right clicking MovieDBContext and selecting **Close Connection**. (If you don't close the  connection, you might get an error the next time you run the project).



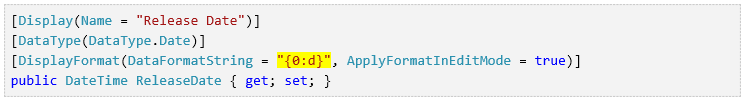
You now have a database and pages to display, edit, update and delete data. In the next tutorial, we'll examine the rest of the scaffolded code and add  a SearchIndex method and a SearchIndex view that lets you search for movies in this database. For more information on using Entity Framework with MVC, see [Creating an Entity Framework Data Model for an ASP.NET MVC Application](http://www.asp.net/mvc/tutorials/getting-started-with-ef-using-mvc/creating-an-entity-framework-data-model-for-an-asp-net-mvc-application).

# 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:

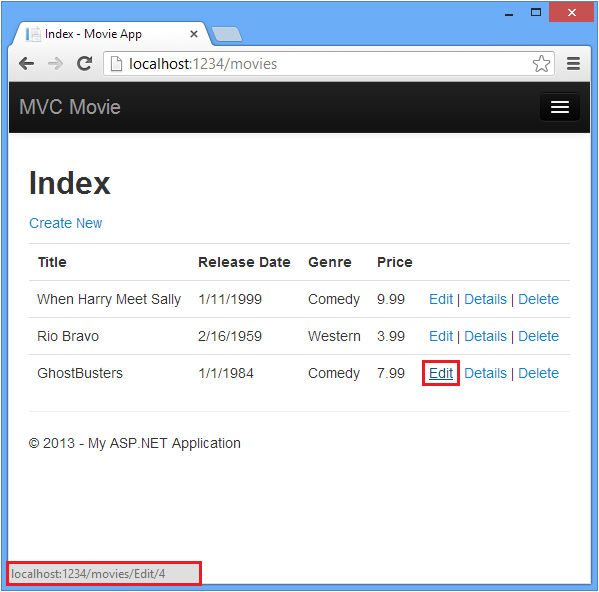


You can also make the date culture specific like this:

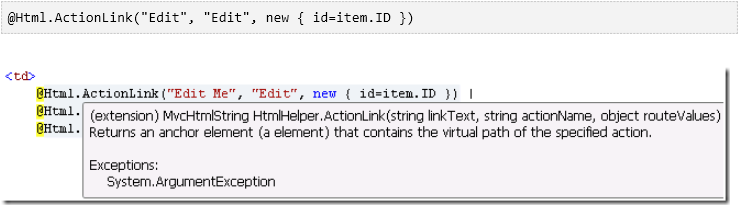


We'll cover [DataAnnotations](http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.aspx) in the next tutorial. The [Display](http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.displayattribute.aspx) attribute specifies what to display for the name of a field (in this case "Release Date" instead of "ReleaseDate"). The [DataType](http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.datatypeattribute.aspx) 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](http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.displayformatattribute.aspx) 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](http://msdn.microsoft.com/en-us/library/system.web.mvc.html.linkextensions.actionlink(v=vs.108).aspx) method in the Views\Movies\Index.cshtml view:

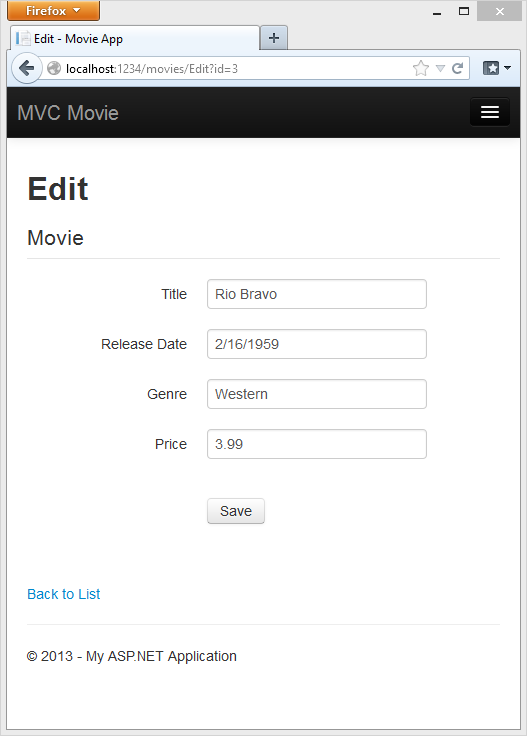


The Html object is a helper that's exposed using a property on the [System.Web.Mvc.WebViewPage](http://msdn.microsoft.com/en-us/library/gg402107(VS.98).aspx) base class. The [ActionLink](http://msdn.microsoft.com/en-us/library/system.web.mvc.html.linkextensions.actionlink.aspx) method of the helper makes it easy to dynamically generate HTML hyperlinks that link to action methods on controllers. The first argument to the ActionLink method is the link text to render (for example, <a>Edit Me</a>). The second argument is the name of the action method to invoke (In this case, the Edit action). The final argument is an [anonymous object](http://weblogs.asp.net/scottgu/archive/2007/05/15/new-orcas-language-feature-anonymous-types.aspx) 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](http://www.asp.net/mvc/tutorials/older-versions/controllers-and-routing/asp-net-mvc-routing-overview-cs) method is used to route HTTP requests to the correct controller and action method and supply the optional ID parameter.  The [MapRoute](http://www.asp.net/mvc/tutorials/older-versions/controllers-and-routing/asp-net-mvc-routing-overview-cs) method is also used by the [HtmlHelpers](http://msdn.microsoft.com/en-us/library/system.web.mvc.htmlhelper(v=vs.108).aspx) such as [ActionLink](http://msdn.microsoft.com/en-us/library/system.web.mvc.html.linkextensions.actionlink.aspx) to generate URLs given the controller, action method and any route data.



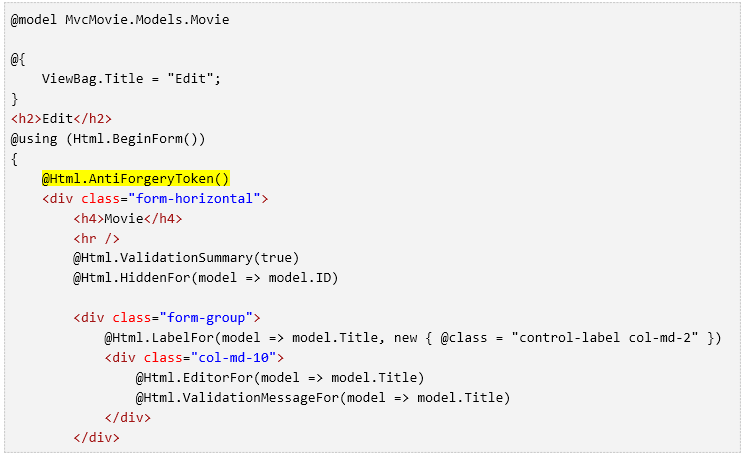
You can also pass action method parameters using a query string. For example, the URL *http://localhost:1234/Movies/Edit?ID=3* also passes the parameter ID of 3 to the Edit action method of the Movies controller.



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



Notice the second Edit action method is preceded by the [HttpPost](http://msdn.microsoft.com/en-us/library/system.web.mvc.httppostattribute.aspx) attribute. This attribute specifies that that overload of the Edit method can be invoked only for POST requests. You could apply the [HttpGet](http://msdn.microsoft.com/en-us/library/system.web.mvc.httpgetattribute.aspx) 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](http://msdn.microsoft.com/en-us/library/system.web.mvc.bindattribute(v=vs.108).aspx) 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](http://go.microsoft.com/fwlink/?LinkId=317598). In the simple model used in this tutorial, we will be binding all the data in the model. The [ValidateAntiForgeryToken](http://msdn.microsoft.com/en-us/library/system.web.mvc.validateantiforgerytokenattribute(v=vs.108).aspx) attribute is used to prevent forgery of a request and is paired up with @Html.[AntiForgeryToken](http://msdn.microsoft.com/en-us/library/system.web.mvc.htmlhelper.antiforgerytoken(v=vs.108).aspx)() in the edit view file (Views\Movies\Edit.cshtml), a portion is shown below:



@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](http://www.asp.net/mvc/overview/security/xsrfcsrf-prevention-in-aspnet-mvc-and-web-pages).

The HttpGet Edit method takes the movie ID parameter, looks up the movie using the Entity Framework [Find](http://msdn.microsoft.com/en-us/library/gg696418(v=vs.103).aspx) method, and returns the selected movie to the Edit view.  If a movie cannot be found,  [HttpNotFound](http://msdn.microsoft.com/en-us/library/gg453938(VS.98).aspx) is returned. When the scaffolding system created the Edit view, it examined the Movie class and created code to render <label> and <input> elements for each property of the class. The following example shows the Edit view that was generated by the visual studio scaffolding system:



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](http://msdn.microsoft.com/en-us/library/gg401864(VS.98).aspx) helper displays the name of the field ("Title", "ReleaseDate", "Genre", or "Price"). The [Html.EditorFor](http://msdn.microsoft.com/en-us/library/system.web.mvc.html.editorextensions.editorfor(VS.98).aspx) helper renders an HTML <input> element. The [Html.ValidationMessageFor](http://msdn.microsoft.com/en-us/library/system.web.mvc.html.validationextensions.validationmessagefor(VS.98).aspx) helper displays any validation messages associated with that property.

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

|  |
| --- |
| <form action="/movies/Edit/4" method="post">  <input name="\_\_RequestVerificationToken" type="hidden" value="UxY6bkQyJCXO3Kn5AXg-6TXxOj6yVBi9tghHaQ5Lq\_qwKvcojNXEEfcbn-FGh\_0vuw4tS\_BRk7QQQHlJp8AP4\_X4orVNoQnp2cd8kXhykS01" /> <fieldset class="form-horizontal">  <legend>Movie</legend>  <input data-val="true" data-val-number="The field ID must be a number." data-val-required="The ID field is required." id="ID" name="ID" type="hidden" value="4" />  <div class="control-group">  <label class="control-label" for="Title">Title</label>  <div class="controls">  <input class="text-box single-line" id="Title" name="Title" type="text" value="GhostBusters" />  <span class="field-validation-valid help-inline" data-valmsg-for="Title" data-valmsg-replace="true"></span>  </div>  </div>  <div class="control-group">  <label class="control-label" for="ReleaseDate">Release Date</label>  <div class="controls">  <input class="text-box single-line" data-val="true" data-val-date="The field Release Date must be a date." data-val-required="The Release Date field is required." id="ReleaseDate" name="ReleaseDate" type="date" value="1/1/1984" />  <span class="field-validation-valid help-inline" data-valmsg-for="ReleaseDate" data-valmsg-replace="true"></span>  </div>  </div>  <div class="control-group">  <label class="control-label" for="Genre">Genre</label>  <div class="controls">  <input class="text-box single-line" id="Genre" name="Genre" type="text" value="Comedy" />  <span class="field-validation-valid help-inline" data-valmsg-for="Genre" data-valmsg-replace="true"></span>  </div>  </div>  <div class="control-group">  <label class="control-label" for="Price">Price</label>  <div class="controls">  <input class="text-box single-line" data-val="true" data-val-number="The field Price must be a number." data-val-required="The Price field is required." id="Price" name="Price" type="text" value="7.99" />  <span class="field-validation-valid help-inline" data-valmsg-for="Price" data-valmsg-replace="true"></span>  </div>  </div>  <div class="form-actions no-color">  <input type="submit" value="Save" class="btn" />  </div>  </fieldset>  </form> |

The <input> elements are in an HTML <form> element whose action attribute is set to post to the /Movies/Edit URL. The form data will be posted to the server when the **Save** button is clicked. The second line shows the hidden  [XSRF](http://www.asp.net/mvc/overview/security/xsrfcsrf-prevention-in-aspnet-mvc-and-web-pages) token generated by the @Html.AntiForgeryToken() call.

Processing the POST Request

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

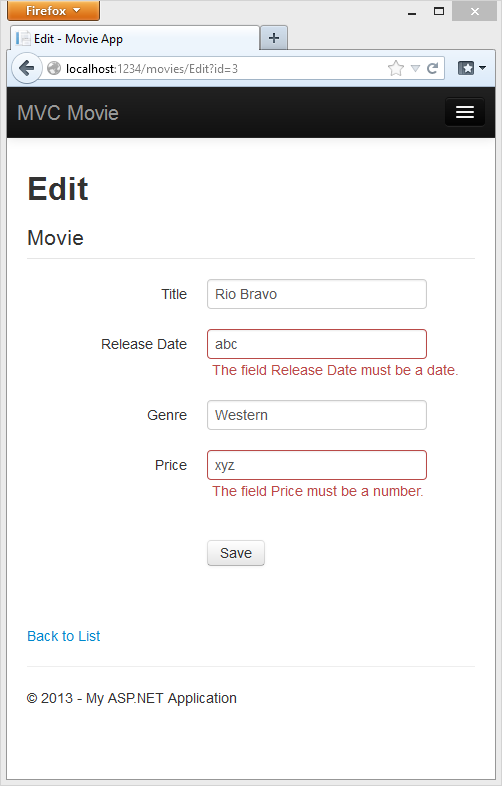


The [ValidateAntiForgeryToken](http://msdn.microsoft.com/en-us/library/system.web.mvc.validateantiforgerytokenattribute(v=vs.108).aspx) attribute validates the  [XSRF](http://www.asp.net/mvc/overview/security/xsrfcsrf-prevention-in-aspnet-mvc-and-web-pages) token generated by the @Html.AntiForgeryToken() call in the view.

The [ASP.NET MVC model binder](http://msdn.microsoft.com/en-us/library/dd410405.aspx) takes the posted form values and creates a Movie object that's passed as the movie parameter. The ModelState.IsValid method verifies that the data submitted in the form can be used to modify (edit or update) a Movie object. If the data is valid, the movie data is saved to the Movies collection of the db(MovieDBContext instance). The  new movie data is saved to the database by calling the SaveChanges method of MovieDBContext. After saving the data, the code redirects the user to the Index action method of the MoviesController class, which displays the movie collection, including the changes just made.

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

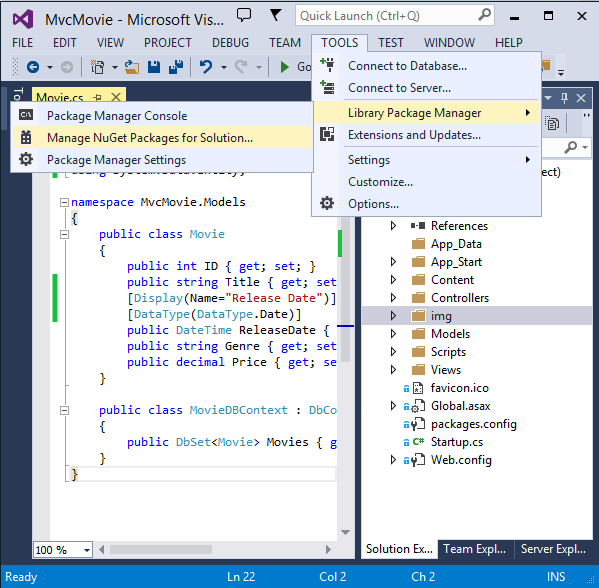
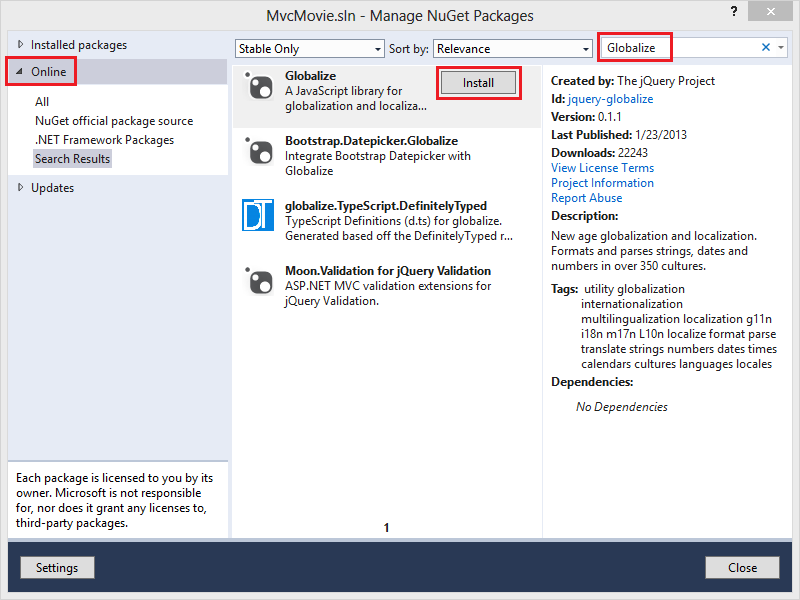
 The [Html.ValidationMessageFor](http://msdn.microsoft.com/en-us/library/system.web.mvc.html.validationextensions.validationmessagefor(v=vs.108).aspx) helpers in the Edit.cshtml view template take care of displaying appropriate error messages.



All the HttpGet methods follow a similar pattern. They get a movie object (or list of objects, in the case of Index), and pass the model to the view. The Create method passes an empty movie object to the Create view. All the methods that create, edit, delete, or otherwise modify data do so in the HttpPost overload of the method. Modifying data in an HTTP GET method is a security risk, as described in the blog post entry [ASP.NET MVC Tip #46 – Don’t use Delete Links because they create Security Holes](http://stephenwalther.com/blog/archive/2009/01/21/asp.net-mvc-tip-46-ndash-donrsquot-use-delete-links-because.aspx). Modifying data in a GET method also violates HTTP best practices and the architectural [REST](http://en.wikipedia.org/wiki/Representational_State_Transfer) pattern, which specifies that GET requests should not change the state of your application. In other words, performing a GET operation should be a safe operation that has no side effects and doesn't modify your persisted data.

If you are using a US-English computer, you can skip this section and go to the next tutorial. You can download the Globalize version of this tutorial [here](http://archive.msdn.microsoft.com/Project/Download/FileDownload.aspx?ProjectName=aspnetmvcsamples&DownloadId=16475). For an excellent two part tutorial on Internationalization, see [Nadeem's ASP.NET MVC 5 Internationalization](http://afana.me/post/aspnet-mvc-internationalization.aspx).

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

1. From the Tools menu click Library Package Manager, and then click Manage NuGet Packages for Solution.  
     
   
2. On the left pane, select **Online.** (See the image below.)
3. In the **Search Installed packages** input box, enter Globalize. **Click **Install**. The Scripts\jquery.globalize\globalize.js  file will be added to your project. The Scripts\jquery.globalize\cultures\ folder will contain many culture JavaScript files. Note, it can take five minutes to install this package.

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

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

To avoid repeating this code in every Edit view, you can move it to the layout file.  To optimize the script download, see my tutorial [Bundling and Minification](http://www.asp.net/mvc/tutorials/mvc-4/bundling-and-minification).

For more information see [ASP.NET MVC 3 Internationalization](http://afana.me/post/aspnet-mvc-internationalization.aspx) and [ASP.NET MVC 3 Internationalization - Part 2 (NerdDinner)](http://afana.me/post/aspnet-mvc-internationalization-part-2.aspx).

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



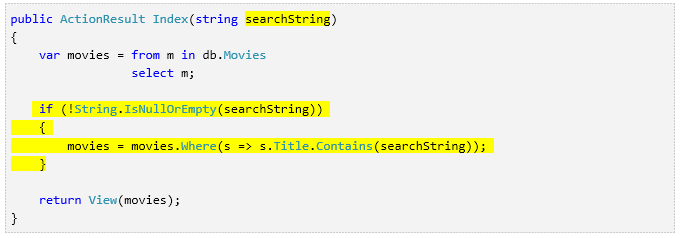
# Search

## Adding a Search Method and Search View

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

## Updating the Index Form

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

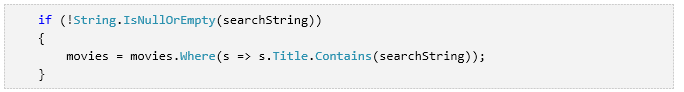


The first line of the Index method creates the following [LINQ](http://msdn.microsoft.com/en-us/library/bb397926.aspx) query  to select the movies:



The query is defined at this point, but hasn't yet been run against the database.

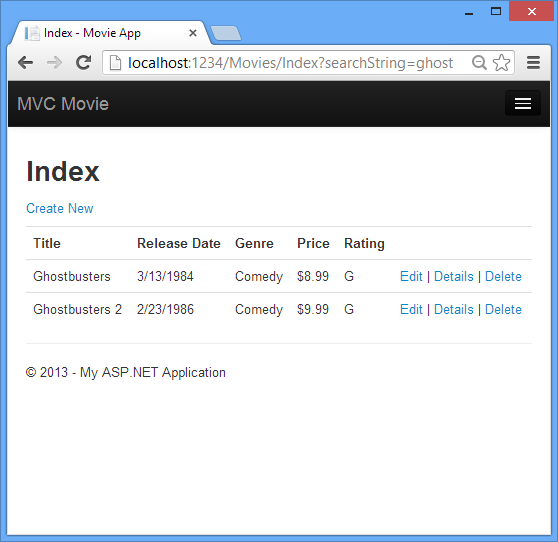
If the searchString parameter contains a string, the movies query is modified to filter on the value of the search string, using the  following code:



The s => s.Title code above is a [Lambda  Expression](http://msdn.microsoft.com/en-us/library/bb397687.aspx). Lambdas are used in method-based [LINQ](http://msdn.microsoft.com/en-us/library/bb397926.aspx) queries  as arguments to standard query operator methods such as the [Where](http://msdn.microsoft.com/en-us/library/system.linq.enumerable.where.aspx) 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](http://msdn.microsoft.com/en-us/library/bb342261.aspx)  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](http://msdn.microsoft.com/en-us/library/bb738633.aspx). Note: The [Contains](http://msdn.microsoft.com/en-us/library/bb155125.aspx)  method is run on the database, not the c# code above. On the database, [Contains](http://msdn.microsoft.com/en-us/library/bb155125.aspx) maps to [SQL LIKE](http://msdn.microsoft.com/en-us/library/ms179859.aspx),  which is case insensitive.

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

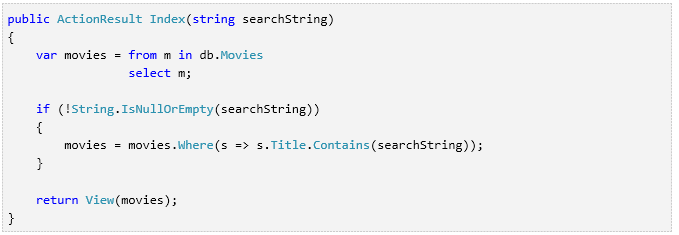
Run the application and navigate to /Movies/Index. Append a  query string such as ?searchString=ghost to the URL. The filtered  movies are displayed.



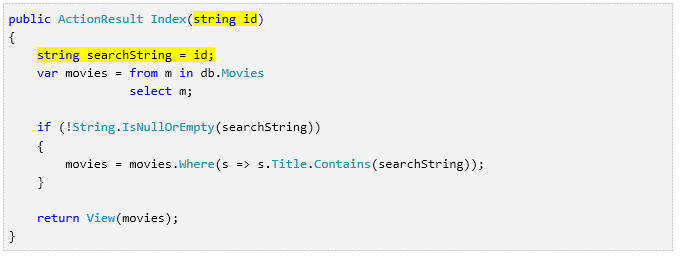
If you change the signature of the Index method to have a  parameter named id, the id parameter will match the {id} placeholder for the default routes set in the App\_Start\RouteConfig.cs  file.

{controller}/{action}/{id}

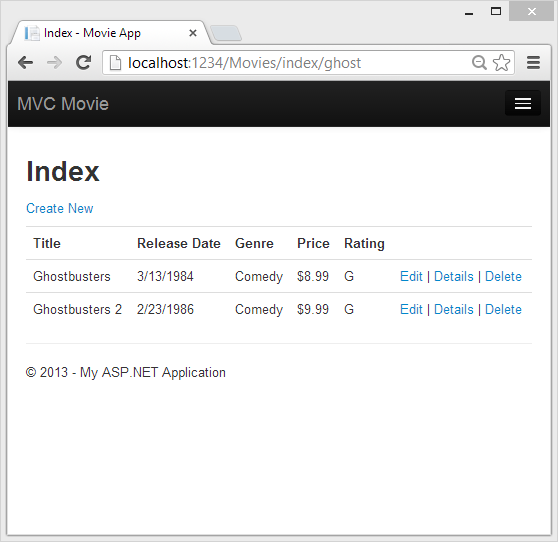
The original Index method looks like this::



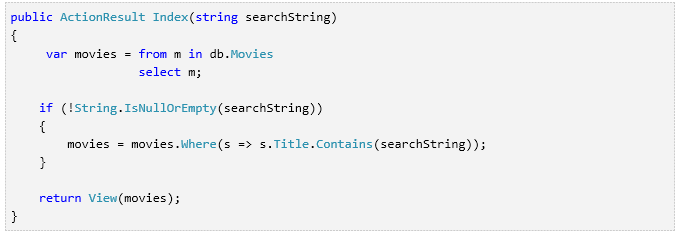
The modified Index method would look as follows:



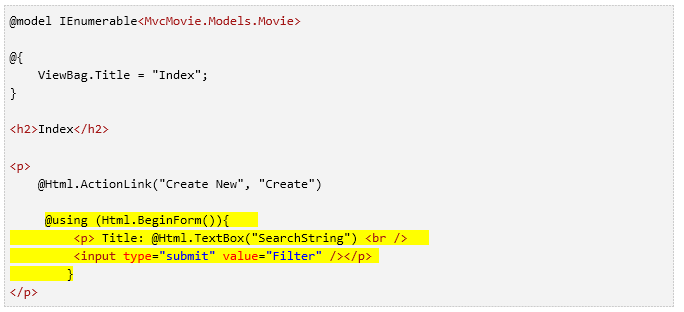
You can now pass the search title as route data (a URL segment) instead of as  a query string value.



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

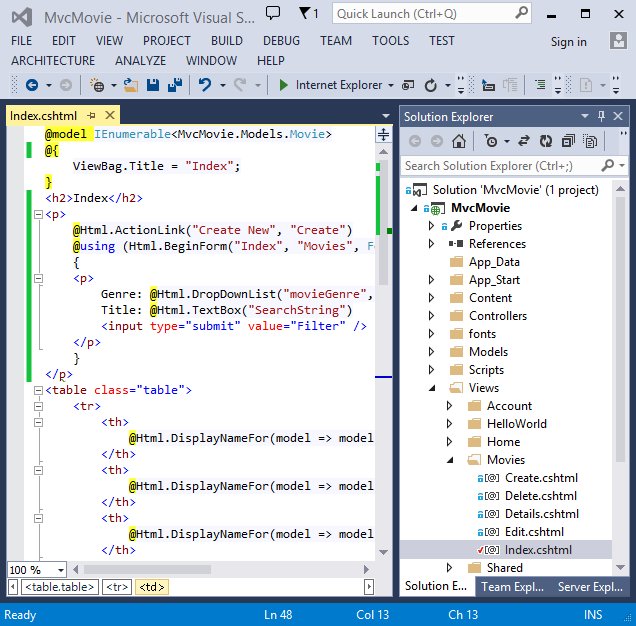


Open the Views\Movies\Index.cshtml file, and just after @Html.ActionLink("Create New", "Create"), add the  form markup highlighted below:

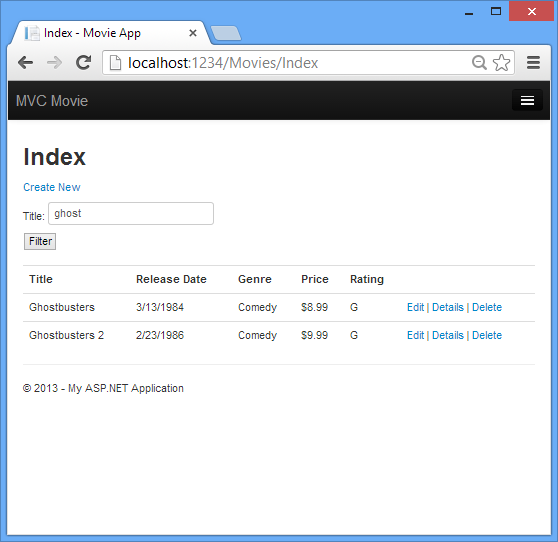


The Html.[BeginForm](http://msdn.microsoft.com/en-us/library/dd505244(v=vs.108).aspx) helper creates an opening <[form](http://www.javascript-coder.com/html-form/html-form-tag.phtml)>  tag. The Html.BeginForm helper causes the form to post to itself  when the user submits the form by clicking the **Filter** button.

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

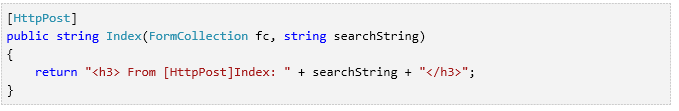


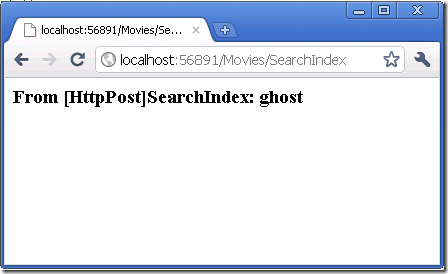
With the Index view open in Visual Studio (as shown in the image above), tap Ctr F5 or F5 to run the application and then try searching for a movie.



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

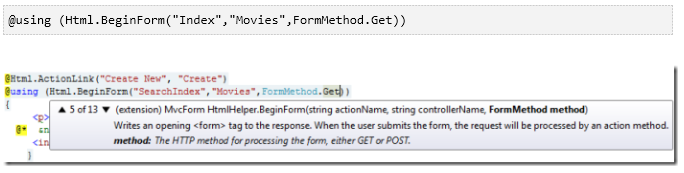
You could add the following HttpPost Index method. In that  case, the action invoker would match the HttpPost Index  method, and the HttpPost Index method would run as shown in  the image below.



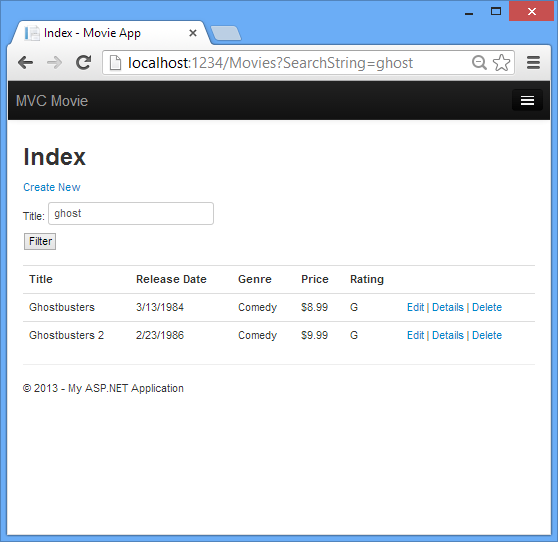


However, even if you add this HttpPost version of the Index method, there's a limitation in how this has all been  implemented. Imagine that you want to bookmark a particular search or you want  to send a link to friends that they can click in order to see the same filtered  list of movies. Notice that the URL for the HTTP POST request is the same as the URL for the GET request (localhost:xxxxx/Movies/Index) -- there's no  search information in the URL itself. Right now, the search string information is sent to the server as a form field value. This means you can't capture that search information to bookmark or send to friends in a URL.

The solution is to use an overload of [BeginForm](http://msdn.microsoft.com/en-us/library/dd460344(v=vs.108).aspx) 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:



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



## Adding Search by Genre

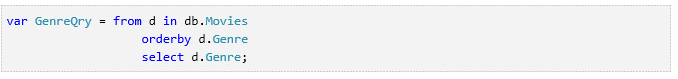
If you added the HttpPost version of the Index  method, delete it now.

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



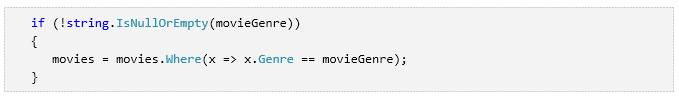
This version of the Index method takes an additional  parameter, namely movieGenre. The first few lines of code create a List object to hold movie genres from the database.

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



The code uses the [AddRange](http://msdn.microsoft.com/en-us/library/z883w3dc.aspx)  method of the generic [List](http://msdn.microsoft.com/en-us/library/6sh2ey19.aspx)  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](http://msdn.microsoft.com/en-us/library/system.web.mvc.selectlist(v=vs.108).aspx) 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.

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

## Adding Markup to the Index View to Support Search by Genre

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



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:

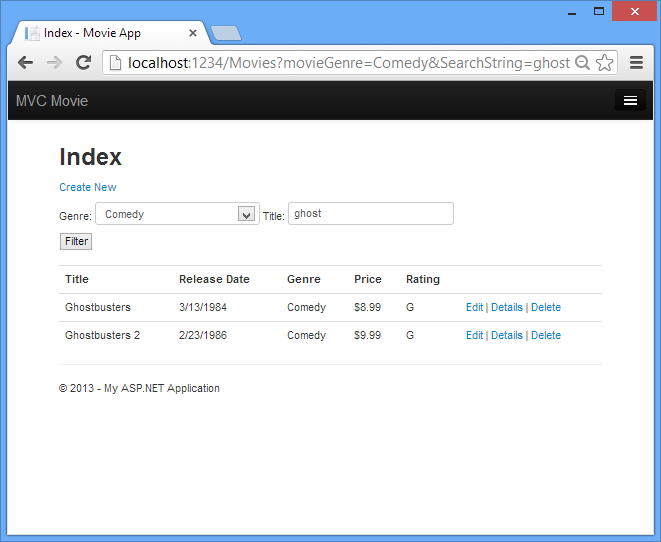


The parameter "All" provides the item in the list to be preselected. Had we used the following code:

@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 slection, 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.

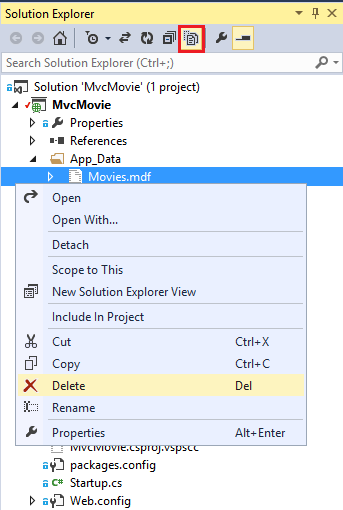
# Adding a New Field

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.

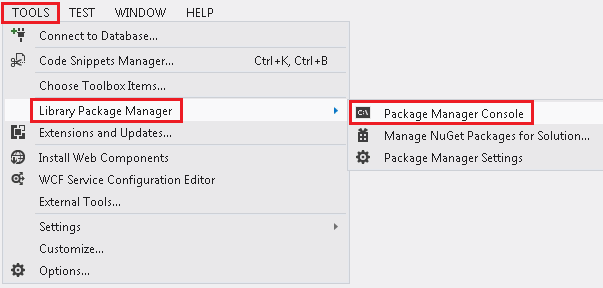
## Setting up Code First Migrations for Model Changes

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.



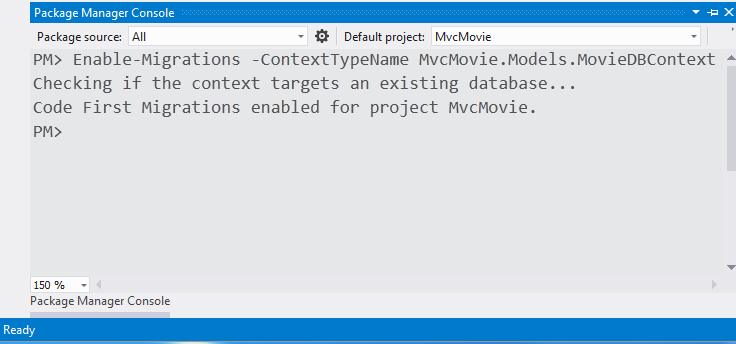
Build the application to make sure there are no errors.

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



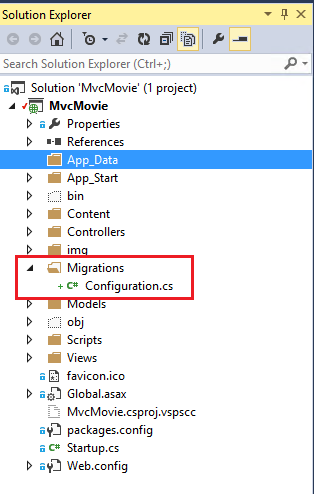
In the **Package Manager Console** window at the PM> prompt enter

Enable-Migrations -ContextTypeName MvcMovie.Models.MovieDBContext



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

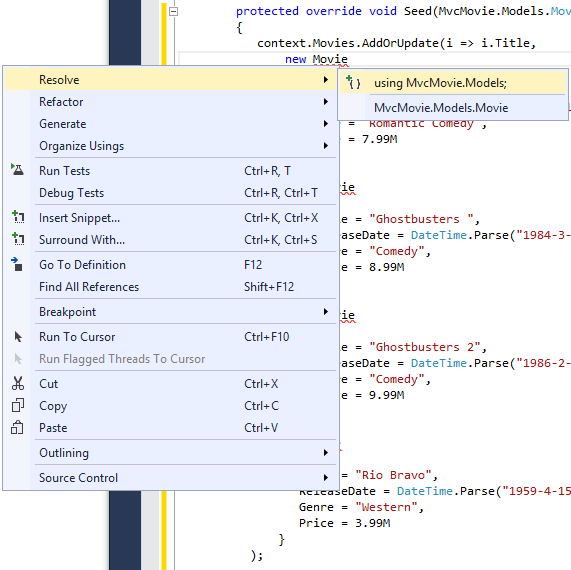
Movie-Datenbank:  
Enable-Migrations -ContextTypeName Wifi.Media.Backend.Repositories.MovieRepositories.DatabaseContext.MovieContext -StartUpProjectName “MovieRepositories”



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



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

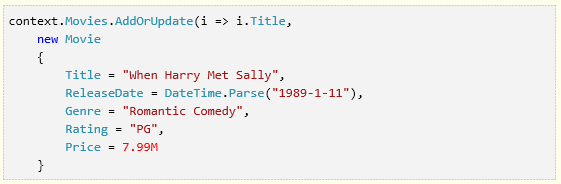


Doing so adds the following using statement:



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](http://msdn.microsoft.com/en-us/library/system.data.entity.migrations.idbsetextensions.addorupdate(v=vs.103).aspx) method in the following code performs an "upsert" operation:



 Because the [Seed](http://msdn.microsoft.com/en-us/library/hh829453(v=vs.103).aspx) 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](http://en.wikipedia.org/wiki/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](http://msdn.microsoft.com/en-us/library/system.data.entity.migrations.idbsetextensions.addorupdate(v=vs.103).aspx) 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:

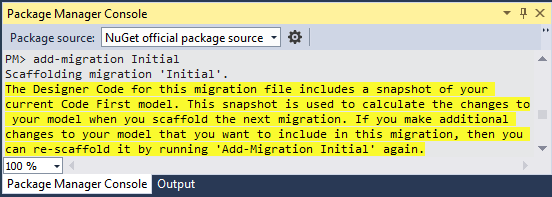


This code assumes that titiles are unique. If you manually add a duplicate title, you'll get the following exception the next time you perform a migration.   
  
     Sequence contains more than one element  
  
For more information about the [AddOrUpdate](http://msdn.microsoft.com/en-us/library/system.data.entity.migrations.idbsetextensions.addorupdate(v=vs.103).aspx) method, see [Take care with EF 4.3 AddOrUpdate Method](http://thedatafarm.com/blog/data-access/take-care-with-ef-4-3-addorupdate-method/)..

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

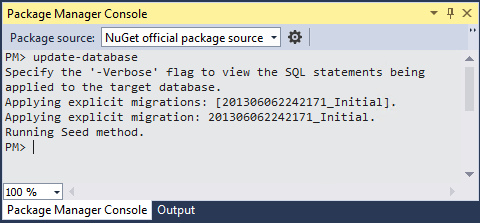
The next step is to create a DbMigration class for the initial migration. This migration creates a new database, that's why you deleted the movie.mdf file in a previous step.

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



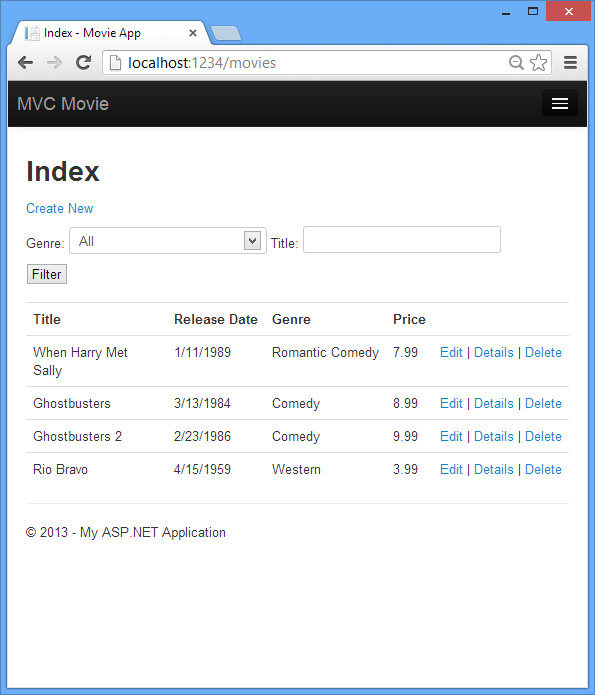
Code First Migrations creates another class file in the Migrations folder (with the name {DateStamp}\_Initial.cs ), and this class contains code that creates the database schema. The migration filename is pre-fixed with a timestamp to help with ordering.  Examine the {DateStamp}\_Initial.cs  file, it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

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



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

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



## Adding a Rating Property to the Movie Model

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



The complete Movie class now looks like the following code:



Build the application (Ctrl+Shift+B).

Because you've added a new field to  the Movie class, you also need to update the 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:



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:

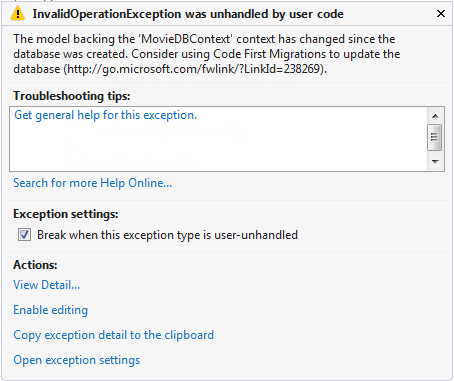


Next, open the \Views\Movies\Create.cshtml file and add the Rating field with the following highlighed markup. This renders a text box so that you can specify a rating when a new movie is created.

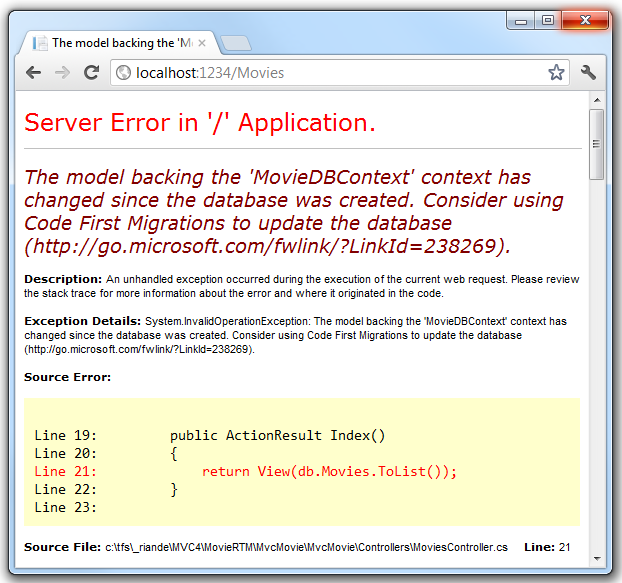


You've now updated the application code to support the new Rating property.

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



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



You're seeing this error because the updated Movie model class in the application is now different than the schema of the Movie table of the existing database. (There's no Rating column in the database table.)

There are a few approaches to resolving the error:

1. Have the Entity Framework automatically drop and re-create the database based on the new model class schema. This approach is very convenient early in the development cycle when you are doing active development on a test database; it allows you to quickly evolve the model and database schema together. The downside, though, is that you lose existing data in the database — so you don't want to use this approach on a production database! Using an initializer to automatically seed a database with test data is often a productive way to develope an application. For more information on Entity Framework database initializers, see Tom Dykstra's fantastic [ASP.NET MVC/Entity Framework tutorial](http://www.asp.net/mvc/tutorials/getting-started-with-ef-using-mvc/creating-an-entity-framework-data-model-for-an-asp-net-mvc-application).
2. Explicitly modify the schema of the existing database so that it matches the model classes. The advantage of this approach is that you keep your data. You can make this change either manually or by creating a database change script.
3. Use Code First Migrations to update the database schema.

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

Update the 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.



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

add-migration Rating

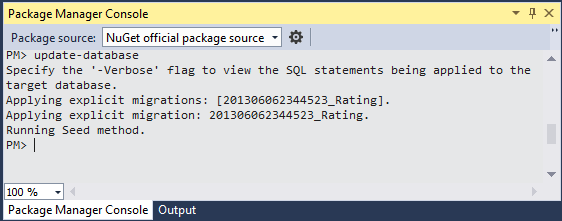
The add-migration command tells the migration framework to examine the current movie model with the current movie DB schema and create the necessary code to migrate the DB to the new model. The name Rating is arbitrary and is used to name the migration file. It's helpful to use a meaningful name for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

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

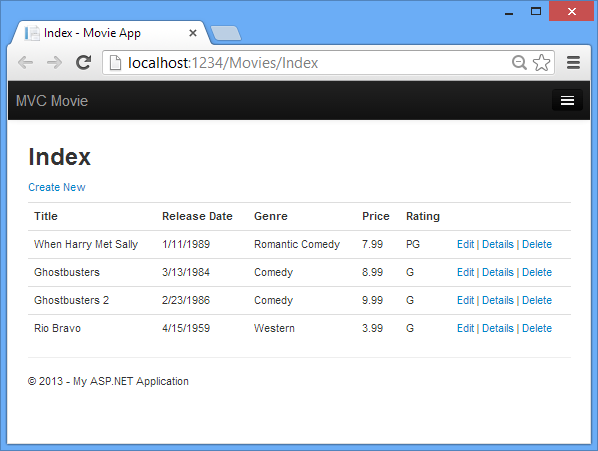


Build the solution, and then enter the update-database command in the **Package Manager Console** window.

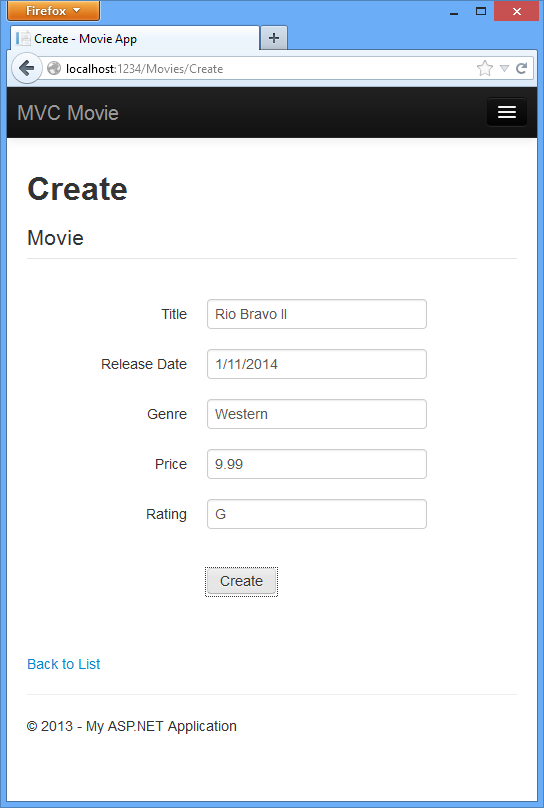
The following image shows the output in the **Package Manager Console** window (The date stamp prepending Rating will be different.)



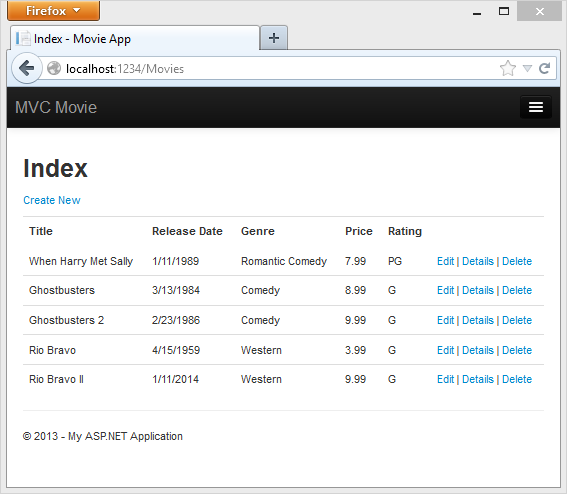
Re-run the application and navigate to the /Movies URL. You can see the new Rating field.



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



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



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

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

You could enter the "update-database" command in the **Package Manager Console** window again and no migration code would run, because the schema matches the model. However, running "update-database" will run the Seed method again, and if you changed any of the Seed data, the changes will be lost because the Seed method upserts data. You can read more about the Seed method in Tom Dykstra's popular [ASP.NET MVC/Entity Framework tutorial](http://www.asp.net/mvc/tutorials/getting-started-with-ef-using-mvc/creating-an-entity-framework-data-model-for-an-asp-net-mvc-application).

In this section you saw how you can modify model objects and keep the database in sync with the changes. You also learned a way to populate a newly created database with sample data so you can try out scenarios. This was just a quick introduction to Code First, see [Creating an Entity Framework Data Model for an ASP.NET MVC Application](http://www.asp.net/mvc/tutorials/getting-started-with-ef-using-mvc/creating-an-entity-framework-data-model-for-an-asp-net-mvc-application) for a more complete tutorial on the subject. Next, let's look at how you can add richer validation logic to the model classes and enable some business rules to be enforced.

# 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](http://en.wikipedia.org/wiki/Don't_repeat_yourself) ("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

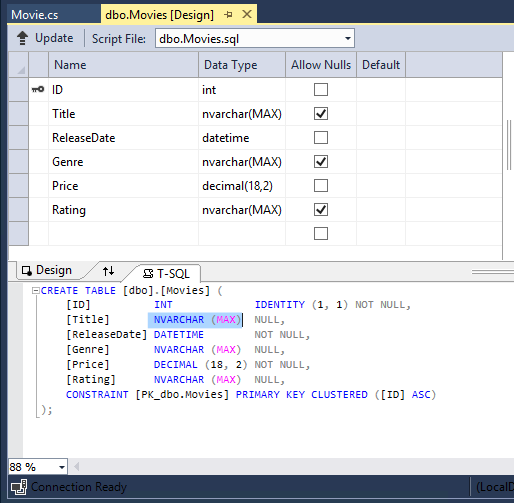
You'll begin by adding some validation logic to the Movie class.

Open the Movie.cs file. Notice the [System.ComponentModel.DataAnnotations](http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.aspx) 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](http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.datatype.aspx) that help with formatting and don't provide any validation.)

Now update the Movie class to take advantage of the built-in [Required](http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.requiredattribute.aspx), [StringLength](http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.stringlengthattribute.aspx),  [RegularExpression](http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.regularexpressionattribute.aspx), and [Range](http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.rangeattribute.aspx) validation attributes. Replace the Movie  class with the following:



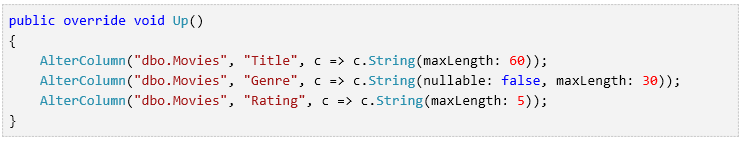
The [StringLength](http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.stringlengthattribute.aspx) attribute sets the maximum length of the string,  and it sets this limitation on the database, therefore the database schema will  change. Right click on the **Movies** table in **Server  explorer** and click **Open Table Definition**:



In the image above, you can see all the string fields are set to  [NVARCHAR  (MAX)](http://technet.microsoft.com/en-us/library/ms186939.aspx).  We will use migrations to update the schema. Build the solution, and then open the **Package Manager Console**  window and enter the following commands:

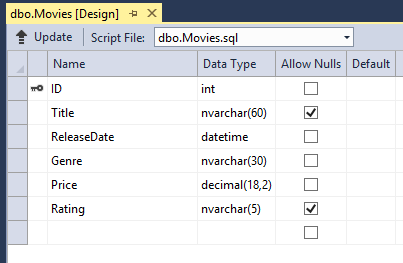
add-migration DataAnnotations  
update-database

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:



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

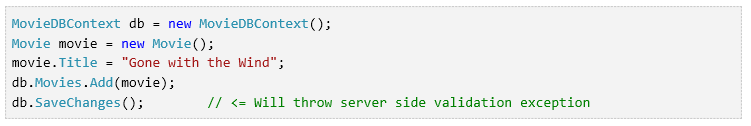
Examine the Movie schema:



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

The validation attributes specify behavior that you want to enforce on the model  properties they are applied to. The [Required](http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.requiredattribute(VS.110).aspx)  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](http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.regularexpressionattribute.aspx) 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](http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.rangeattribute.aspx)   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](http://msdn.microsoft.com/en-us/library/system.data.entity.validation.dbentityvalidationexception(v=vs.103).aspx) exception when the SaveChanges method is  called, because several required Movie property values are missing:



The code above throws the following exception:

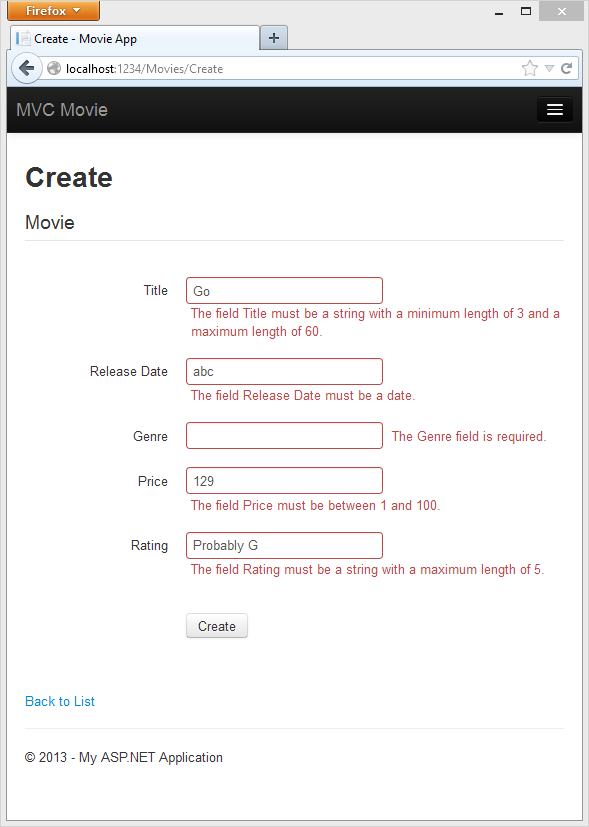
Validation failed for one or more entities. See  'EntityValidationErrors' property for more details.

Having validation rules automatically enforced by the .NET Framework helps  make your application more robust. It also ensures that you can't forget to  validate something and inadvertently let bad data into the database.

## Validation Error UI in ASP.NET MVC

Run the application and navigate to the /Movies URL.

Click the **Create New** link to add a new movie. Fill out the  form with some invalid values. As soon as jQuery client side  validation detects the error, it displays an error message.

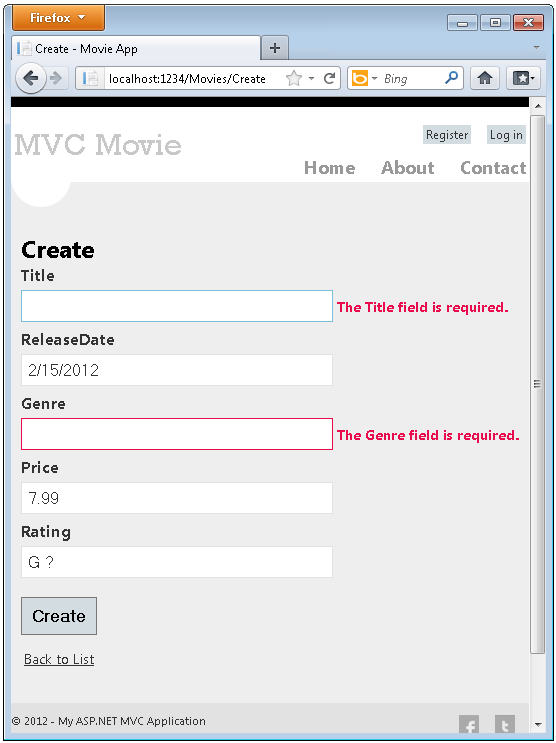


**Note** to support jQuery validation for non-English locales  that use a comma (",") for a decimal point, you must include the NuGet  globalize as described previously in this tutorial.

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

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

The form data is not sent to the  server until there are no client side validation errors. You can verify this by  putting a break point in the HTTP Post method, by using the [fiddler tool](http://fiddler2.com/fiddler2/), or the IE  [F12 developer tools](http://msdn.microsoft.com/en-us/ie/aa740478).



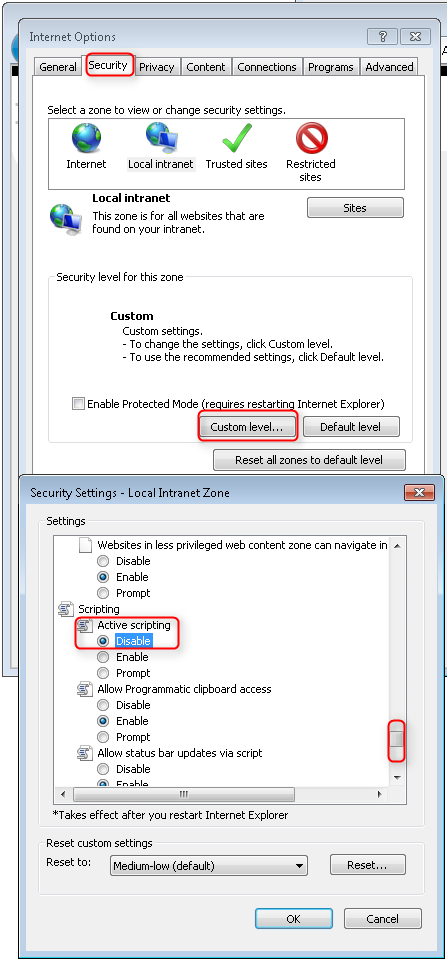
## How Validation Occurs in the Create View and Create Action Method

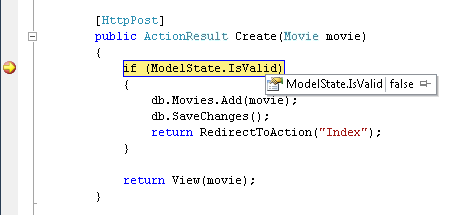
You might wonder how the validation UI was generated without any updates to  the code in the controller or views. The next listing shows what the Create methods in the MovieController class look like.  They're unchanged from how you created them earlier in this tutorial.



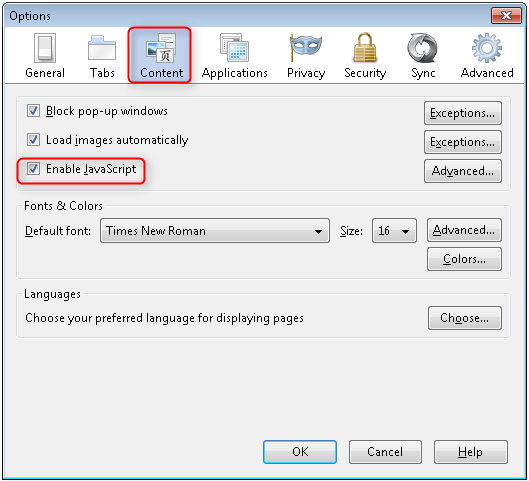
The first (HTTP GET) Create action method displays the initial Create form. The second  ([HttpPost]) version handles  the form post. The second Create method (The HttpPost version) calls ModelState.IsValid to check whether the movie has any validation errors.  Calling this method evaluates any validation 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](http://msdn.microsoft.com/en-us/library/system.web.mvc.modelstatedictionary.isvalid.aspx) to check whether the movie has any validation errors.

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

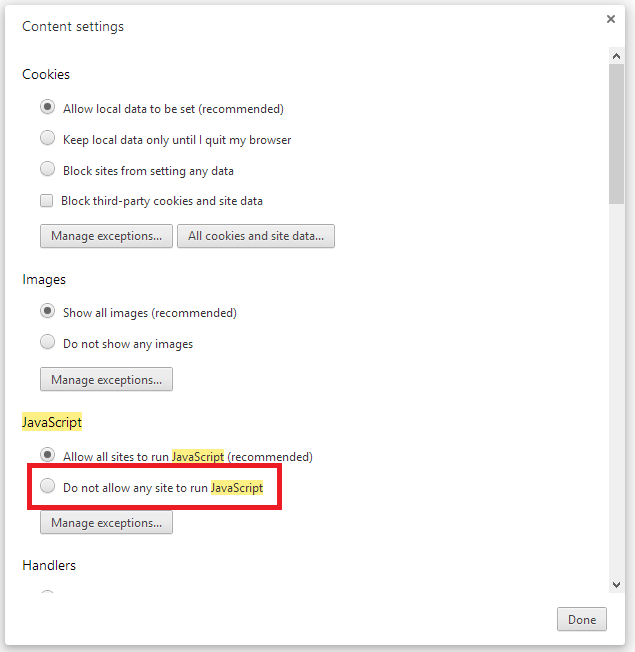




The following image shows how to disable JavaScript in the FireFox browser.



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



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

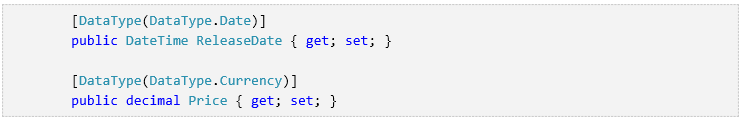
Notice how the code uses an [Html.EditorFor](http://msdn.microsoft.com/en-us/library/ee402949(v=vs.108).aspx) helper to output the [<input>](http://www.w3schools.com/tags/tag_input.asp) element for each Movie property. Next to this  helper is a call to the [Html.ValidationMessageFor](http://msdn.microsoft.com/en-us/library/system.web.mvc.html.validationextensions.validationmessagefor(v=vs.108).aspx) 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](http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.aspx) namespace provides  formatting attributes in addition to the built-in set of validation attributes.  We've already applied  a [DataType](http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.datatype.aspx) enumeration value to the release date and to the price  fields. The following code shows the ReleaseDate and Price  properties with the appropriate [DataType](http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.datatype.aspx) attribute.



The  [DataType](http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.datatypeattribute.aspx) attributes only provide hints for the view engine to format  the data (and supply attributes such as <a> for URL's and  <a  href="mailto:EmailAddress.com"> for email. You can use the [RegularExpression](http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.regularexpressionattribute.aspx) attribute to validate the format of the data. The [DataType](http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.datatypeattribute.aspx) 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](http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.datatype.aspx) 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](http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.datatype.aspx),  and a date selector can be provided for  [DataType.Date](http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.datatype.aspx) in browsers that support  [HTML5](http://html5.org/). The [DataType](http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.datatypeattribute.aspx) attributes emits HTML 5  [data-](http://ejohn.org/blog/html-5-data-attributes/)  (pronounced data dash) attributes that HTML 5  browsers can understand. The [DataType](http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.datatypeattribute.aspx) 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](http://msdn.microsoft.com/en-us/library/vstudio/system.globalization.cultureinfo(v=vs.110).aspx).

The [DisplayFormat](http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.displayformatattribute.aspx) attribute is used to explicitly specify the  date format:



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](http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.displayformatattribute.aspx) attribute by itself, but it's generally a good idea to use the [DataType](http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.datatypeattribute.aspx)  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](http://msdn.microsoft.com/en-us/library/vstudio/wyzd2bce.aspx).

The [DataType](http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.datatypeattribute.aspx) attribute can enable MVC to choose the right field template to  render the data (the [DisplayFormat](http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.displayformatattribute.aspx) if used by itself uses the string template). For more information, see Brad  Wilson's [ASP.NET MVC 2 Templates](http://bradwilson.typepad.com/blog/2009/10/aspnet-mvc-2-templates-part-1-introduction.html). (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](http://stackoverflow.com/questions/12633471/mvc4-datatype-date-editorfor-wont-display-date-value-in-chrome-fine-in-ie).

Note: jQuery validation does not work with the [Range](http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.rangeattribute.aspx) attribute and [DateTime](http://msdn.microsoft.com/en-us/library/system.datetime.aspx). For example, the following code will always display a client  side validation error, even when the date is in the specified range:



You will need to disable  jQuery date validation to use the  [Range](http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.rangeattribute.aspx) attribute with [DateTime](http://msdn.microsoft.com/en-us/library/system.datetime.aspx). It's generally not a good practice to compile hard dates in  your models, so using the [Range](http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.rangeattribute.aspx) attribute and [DateTime](http://msdn.microsoft.com/en-us/library/system.datetime.aspx) is discouraged.

The following code shows combining attributes on one line:



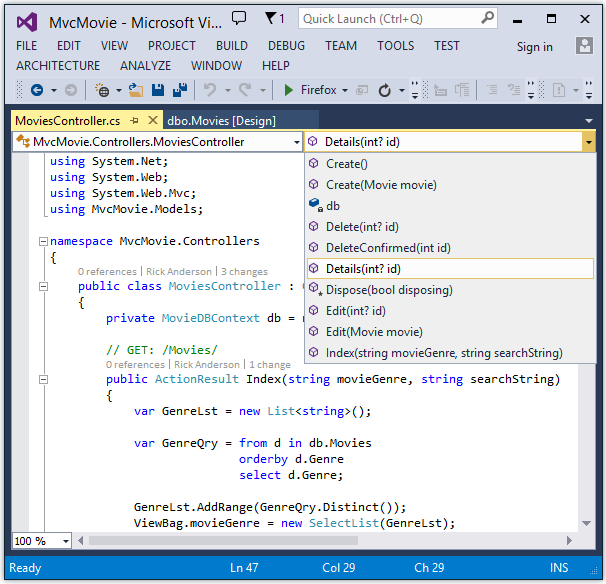
In the next part of the series, we'll review the application and make some  improvements to the automatically generated Details and Delete methods.

# Examining the Details and Delete Methods

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

## Examining the Details and Delete Methods

Open the Movie controller and examine the Details method.





The MVC scaffolding engine that created this action method adds a comment showing a HTTP request that invokes the method. In this case it's a GET request with three URL segments, the Movies controller, the Details method and a ID value.

Code First makes it easy to search for data using the [Find](http://msdn.microsoft.com/en-us/library/system.data.entity.dbset.find(v=vs.103).aspx) 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.

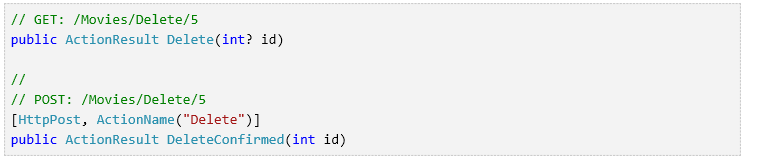
Examine the Delete and DeleteConfirmed methods.



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](http://stephenwalther.com/blog/archive/2009/01/21/asp.net-mvc-tip-46-ndash-donrsquot-use-delete-links-because.aspx).

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



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.)

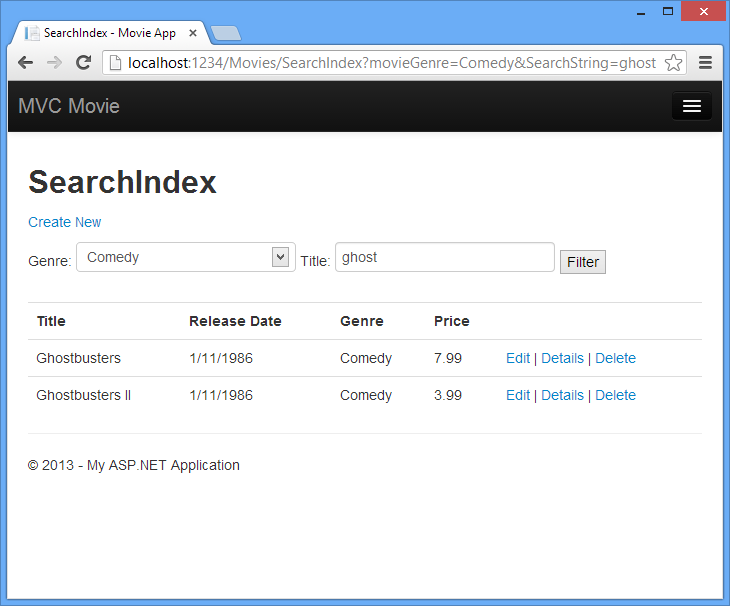
To sort this out, you can do a couple of things. One is to give the methods different names. That's what the scaffolding mechanism did in the preceding example. However, this introduces a small problem: ASP.NET maps segments of a URL to action methods by name, and if you rename a method, routing normally 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](http://msdn.microsoft.com/en-us/library/system.web.mvc.formcollection.aspx) that is passed to the POST method, and then simply don't use the parameter:



## Summary

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



## Next Steps

After you have built and tested a web application, the next step is to make it available to other people to use over the Internet. To do that, you have to deploy it to a web hosting provider. Microsoft offers free web hosting for up to 10 web sites in a [free Azure trial account](http://www.windowsazure.com/en-us/pricing/free-trial/?WT.mc_id=A443DD604). I suggest you next follow my tutorial [Deploy a Secure ASP.NET MVC app with Membership, OAuth, and SQL Database to Azure](http://www.windowsazure.com/en-us/develop/net/tutorials/web-site-with-sql-database/) . An excellent tutorial is Tom Dykstra's intermediate-level [Creating an Entity Framework Data Model for an ASP.NET MVC Application](http://www.asp.net/entity-framework/tutorials/creating-an-entity-framework-data-model-for-an-asp-net-mvc-application). [Stackoverflow](http://stackoverflow.com/help) and the [ASP.NET MVC forums](http://forums.asp.net/1146.aspx) are a great places to ask questions. Follow [me](https://twitter.com/RickAndMSFT) on twitter so you can get updates on my latest tutorials.