Getting started with MVC3

By Scott Hanselman

Contents

Intro to ASP.NET MVC 3

This tutorial will teach you the basics of building an ASP.NET MVC Web application using Microsoft Visual Web Developer Express, which is a free version of Microsoft Visual Studio. Before you start, make sure you have the following installed using the Web Platform Installer.

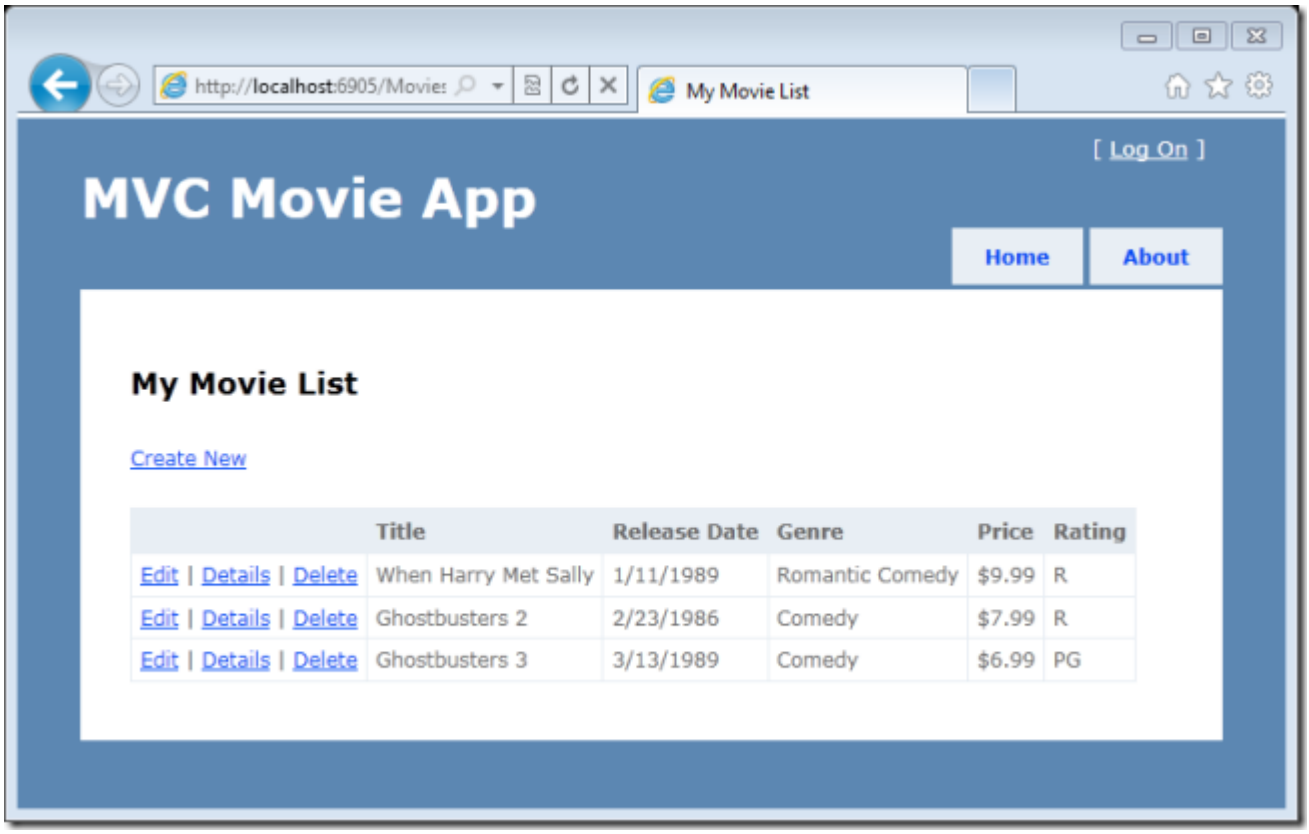
- Visual Studio Web Developer Express with SQL Express
- ASP.NET MVC 3
- SQL Management Studio

This tutorial will teach you the basics of building an ASP.NET MVC Web application using Microsoft Visual Web Developer 2010 Express, which is a free version of Microsoft Visual Studio. Before you start, make sure you've installed the prerequisites listed above. You can install all of them using the Web Platform Installer.

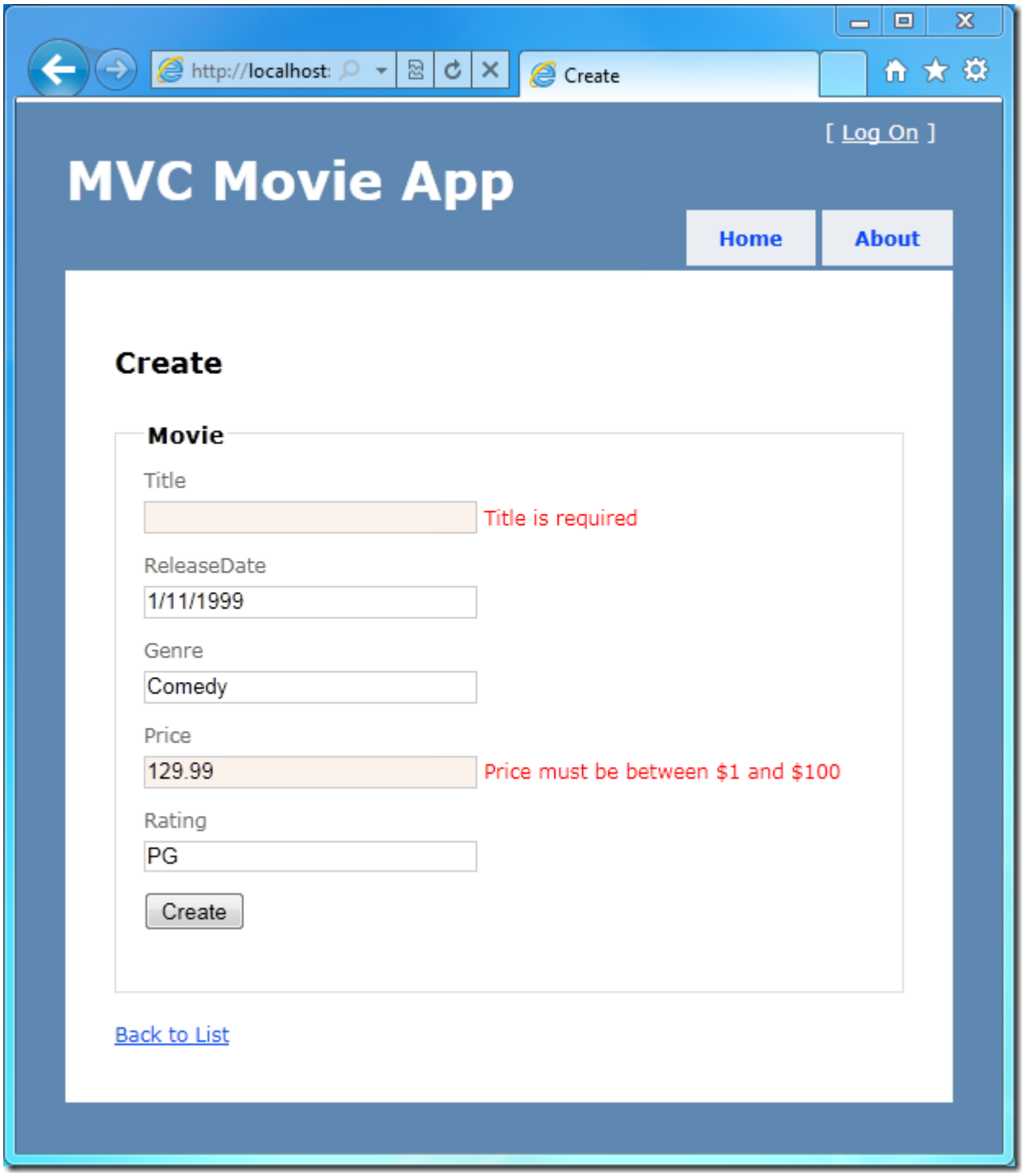
A Visual Web Developer project with C# source code is available to accompany this topic. Download the C# version here. If you prefer Visual Basic, switch to the Visual Basic version of this tutorial.

What You'll Build

You'll implement a simple movie-listing application that supports creating, editing and listing movies from a database. Below are two screenshots of the application you'll build. It includes a page that displays a list of movies from a database:



The application also lets you add, edit, and delete movies as well as see details about individual ones. All data-entry scenarios include validation to ensure that the data stored in the database is correct.


http://localhost:
Create
[Log On]
MVC Movie App
Home
About
Create
Movie
Title
Title is required
ReleaseDate
1/11/1999
Genre
Comedy
Price
129.99
Price must be between $1 and $100
Rating
PG
Create
Back to List

Skills You'll Learn

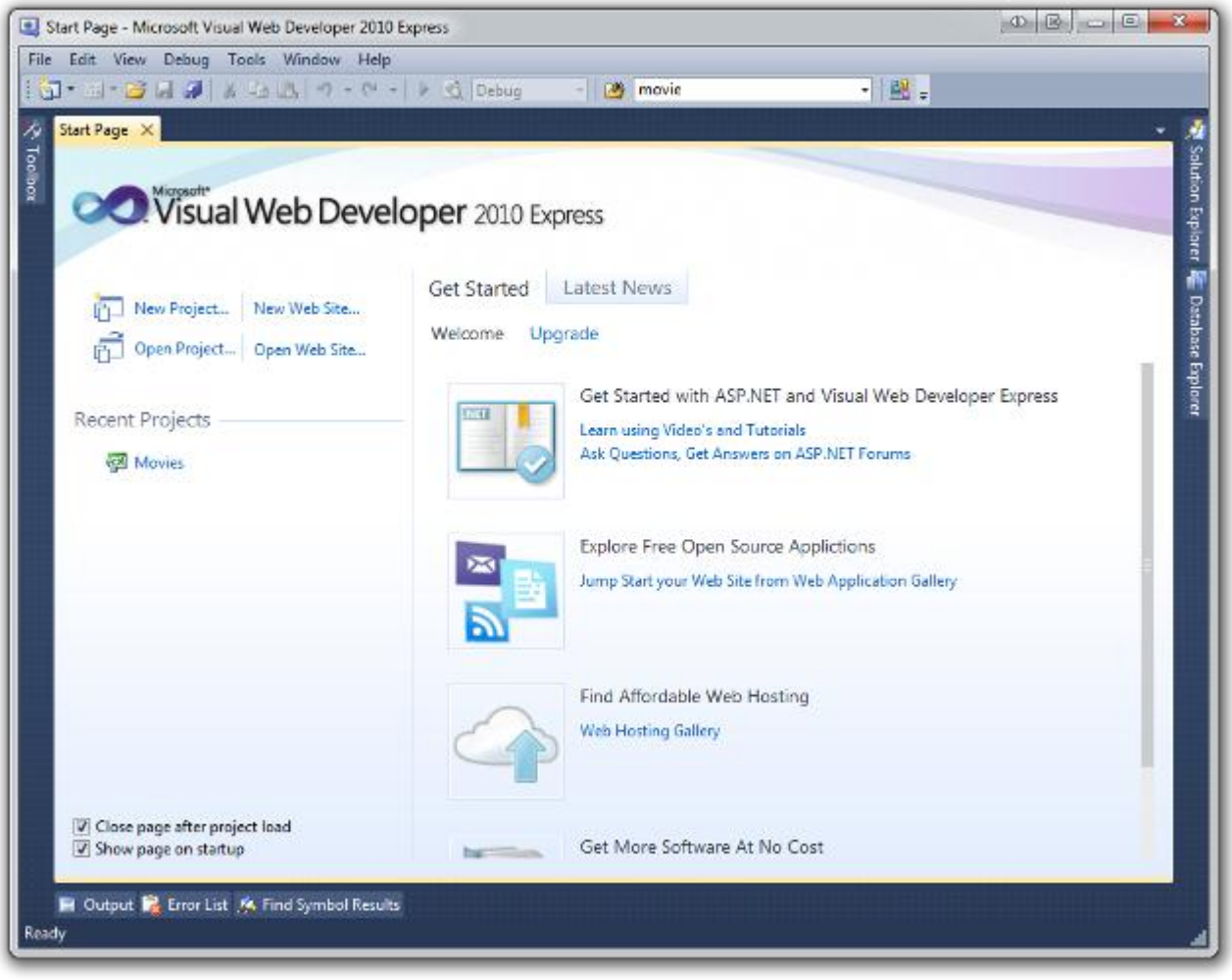
Here's what you'll learn:

- How to create a new ASP.NET MVC project.
- How to create ASP.NET MVC controllers and views.
- How to create a new database using the Entity Framework code-first paradigm.
- How to retrieve and display data.
- How to edit data and enable data validation.

Getting Started

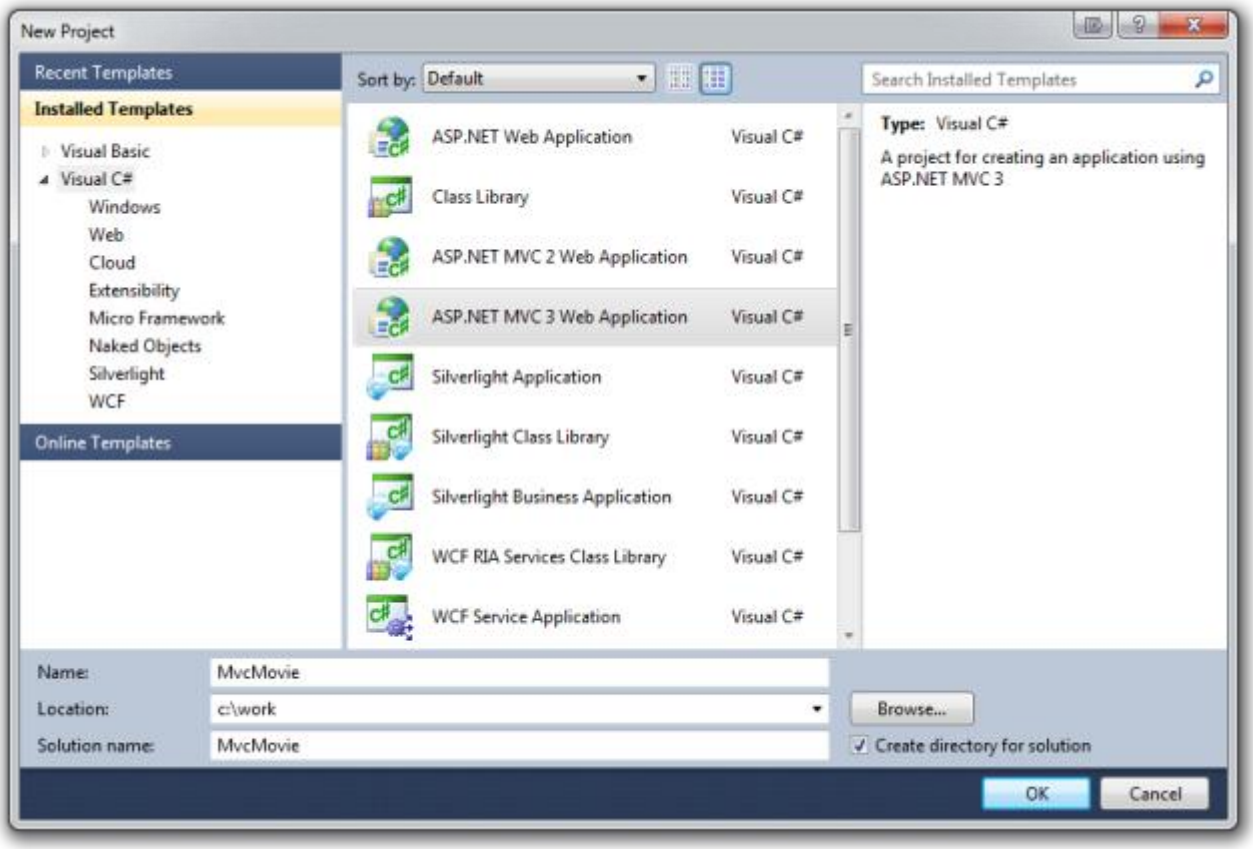
Start by running Visual Web Developer 2010 Express ("Visual Web Developer" for short) and select **New Project** from the **Start** page.

Visual Web Developer 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 Web Developer 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**.)

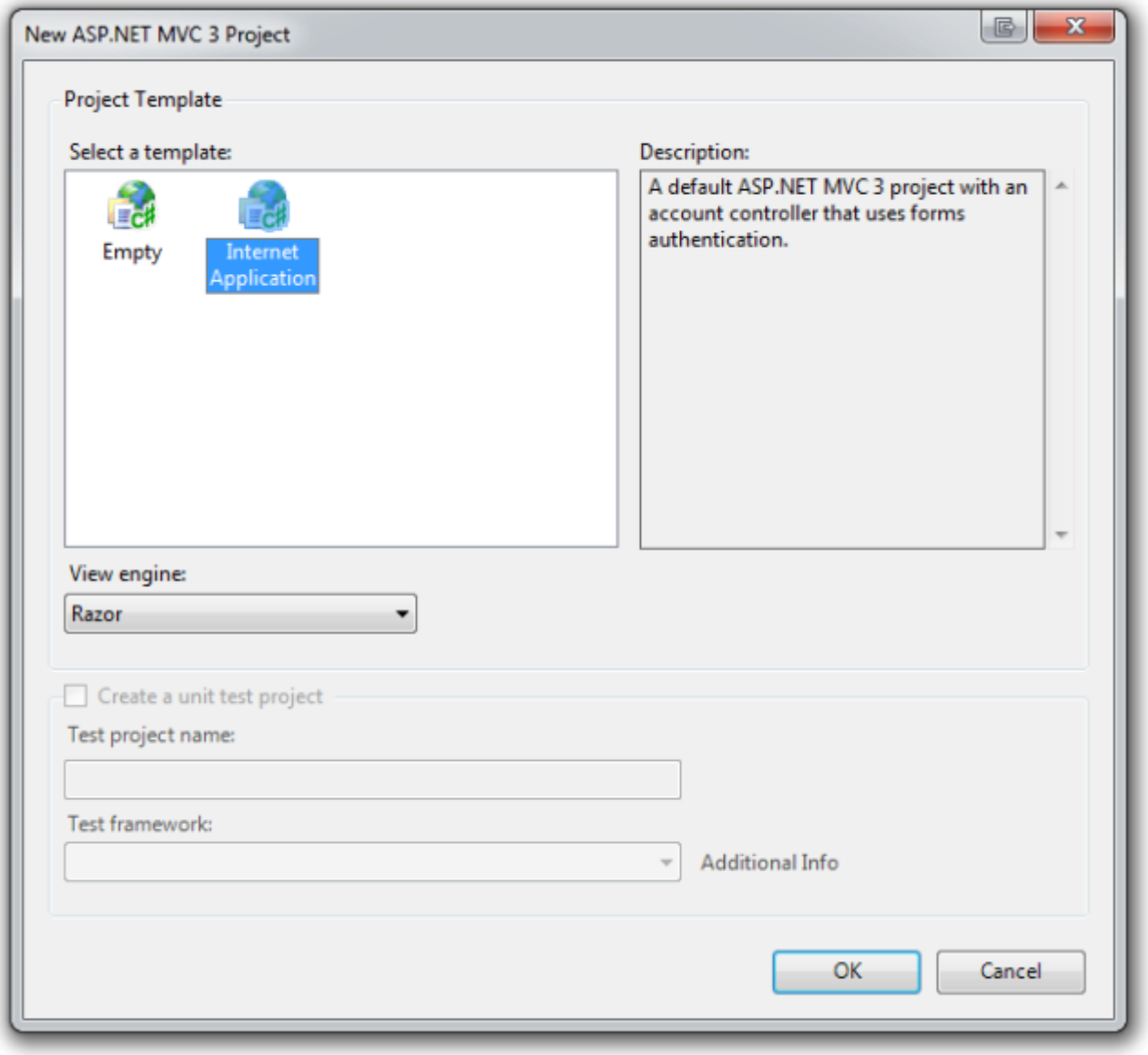


Creating Your First Application

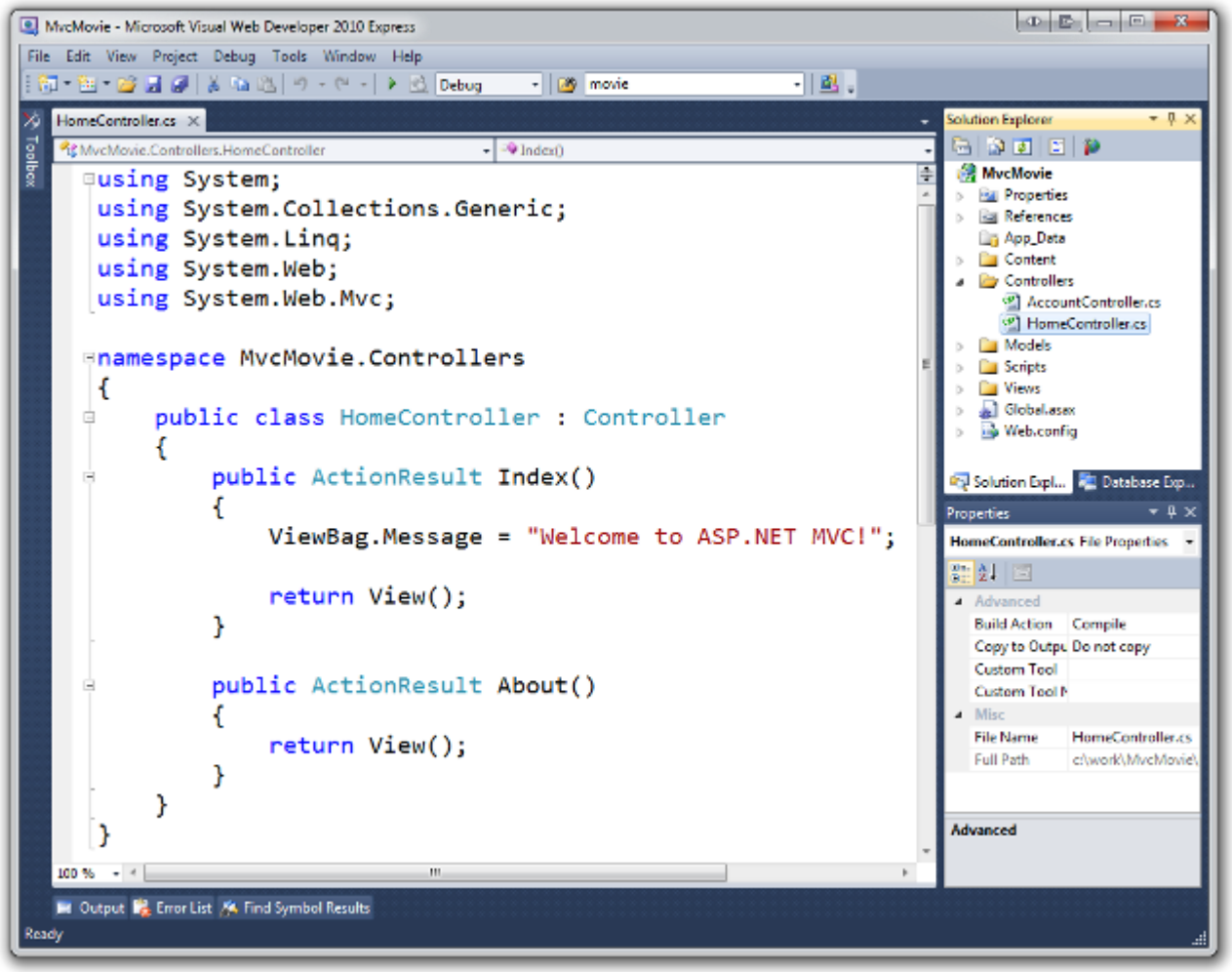
You can create applications using either Visual Basic or Visual C# as the programming language. For now, select Visual C# on the left and then select **ASP.NET MVC 3 Web Application**. Name your project "MvcMovie" and then click **OK**.



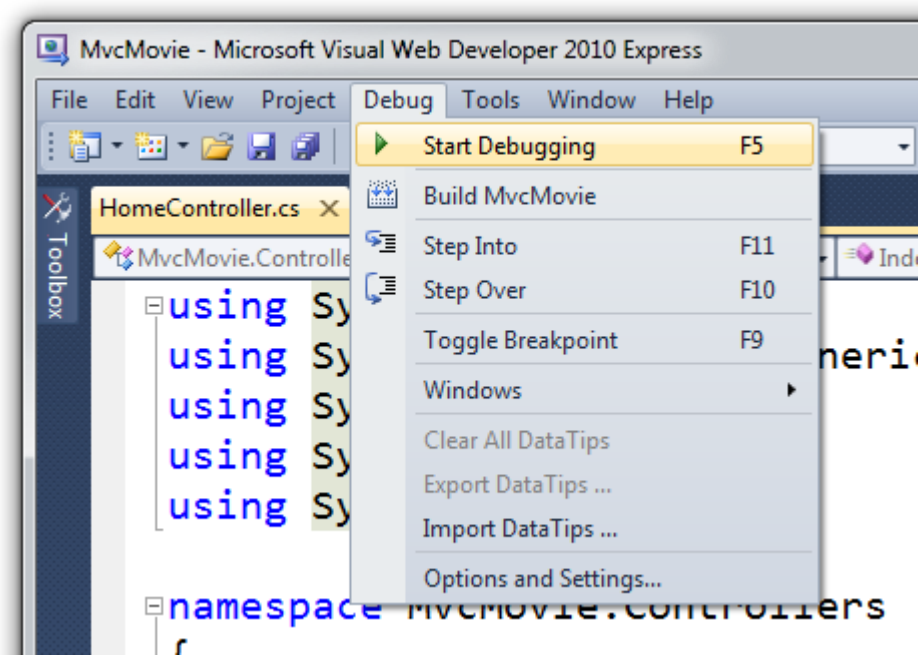
In the **New ASP.NET MVC 3 Project** dialog box, select **Internet Application**. Leave **Razor** as the default view engine.



Click **OK**. Visual Web Developer 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.

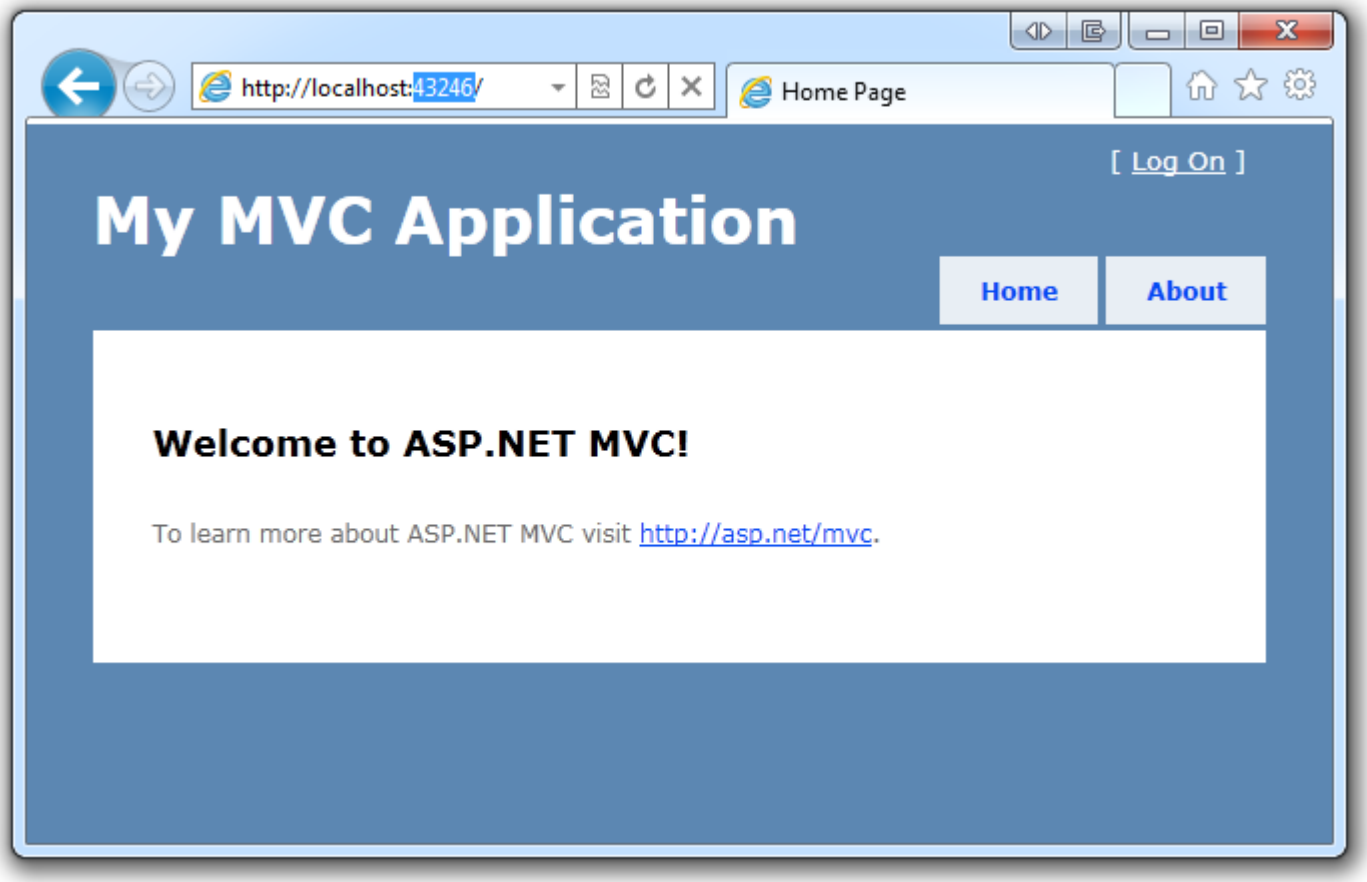


From the **Debug** menu, select **Start Debugging**.



Notice that the keyboard shortcut to start debugging is F5.

F5 causes Visual Web Developer to start a development web server and run your web application. Visual Web Developer then launches a browser and opens the application's home page. Notice that the address bar of the browser says `localhost` 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 Web Developer runs a web project, a random port is used for the web server. In the image below, the random port number is 43246. When you run the application, you'll probably see a different port number.



Right out of the box this default template gives you two pages to visit and a basic login page. Let's change how this application works and learn a little bit about ASP.NET MVC in the process. Close your browser and let's change some code.

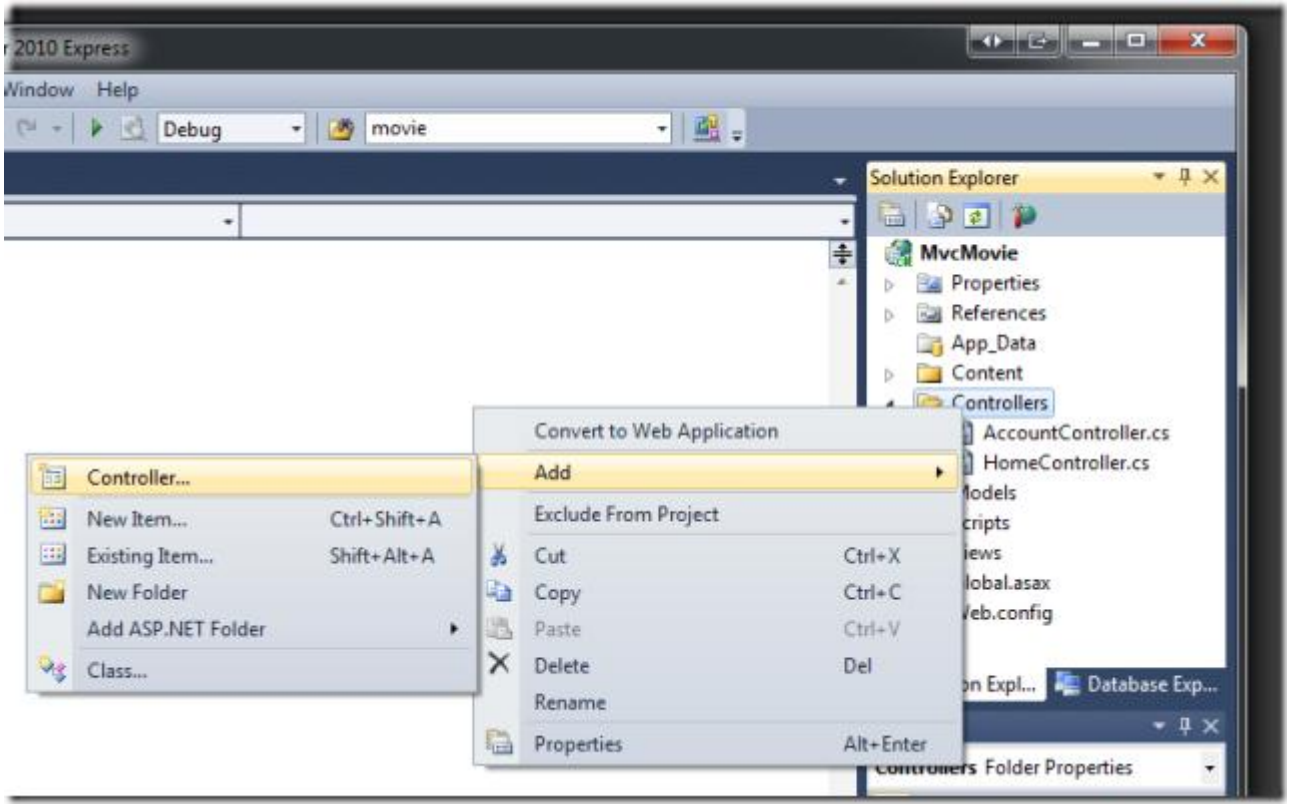
Adding a Controller

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

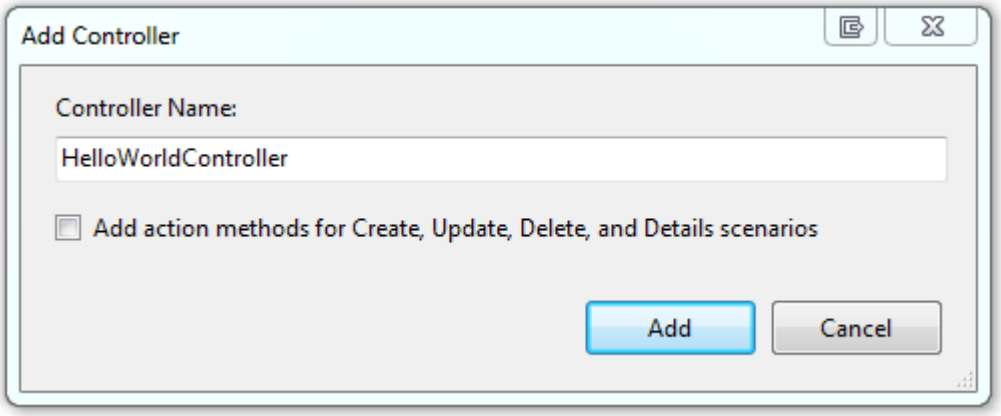
- Controllers: Classes that handle incoming requests to the application, retrieve model data, and then specify view templates that return a response to the client.
- Models: Classes that represent the data of the application and that use validation logic to enforce business rules for that data.
- Views: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

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

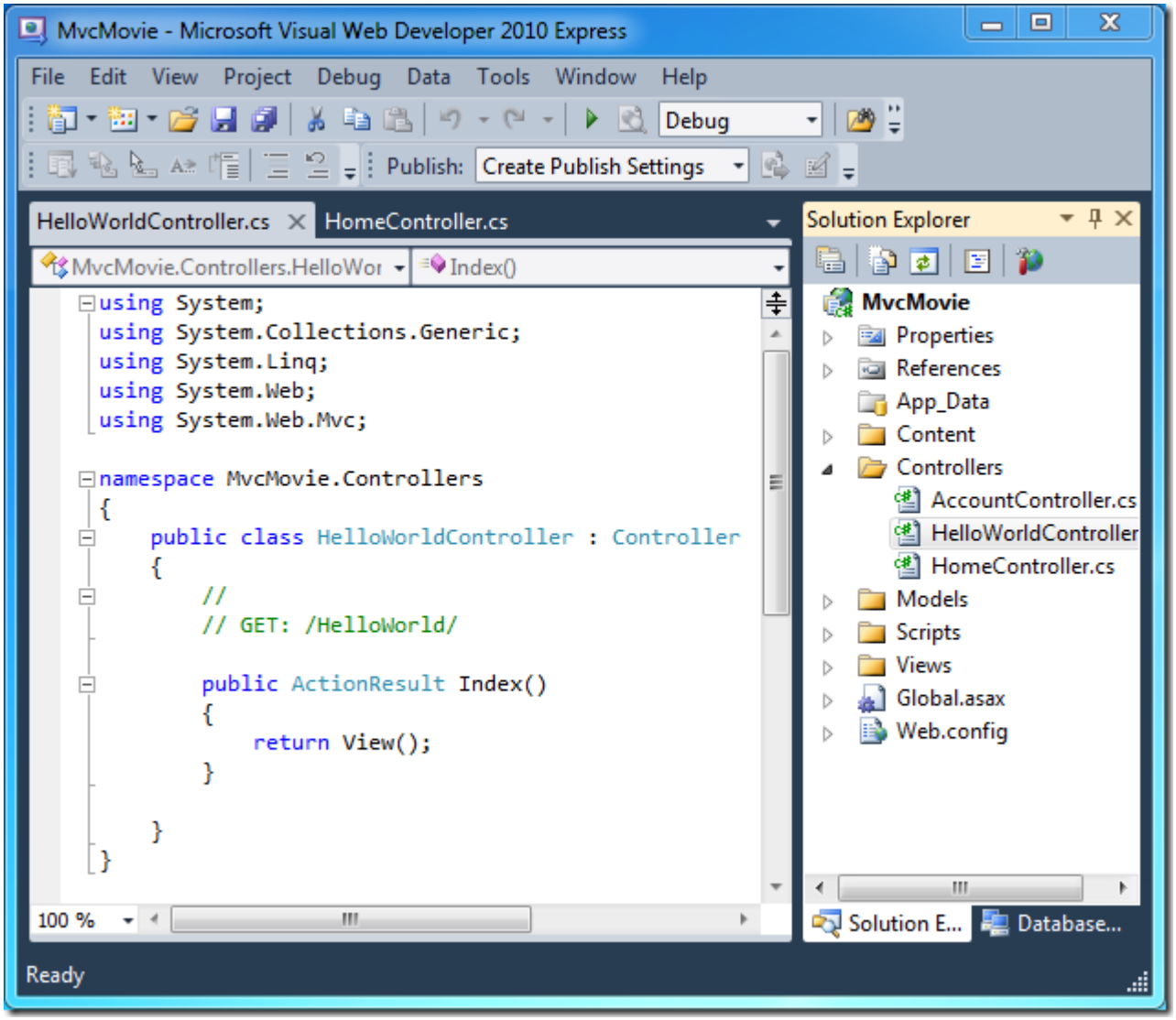
Let's begin by creating a controller class. In **Solution Explorer**, right-click the *Controllers* folder and then select **Add Controller**.



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



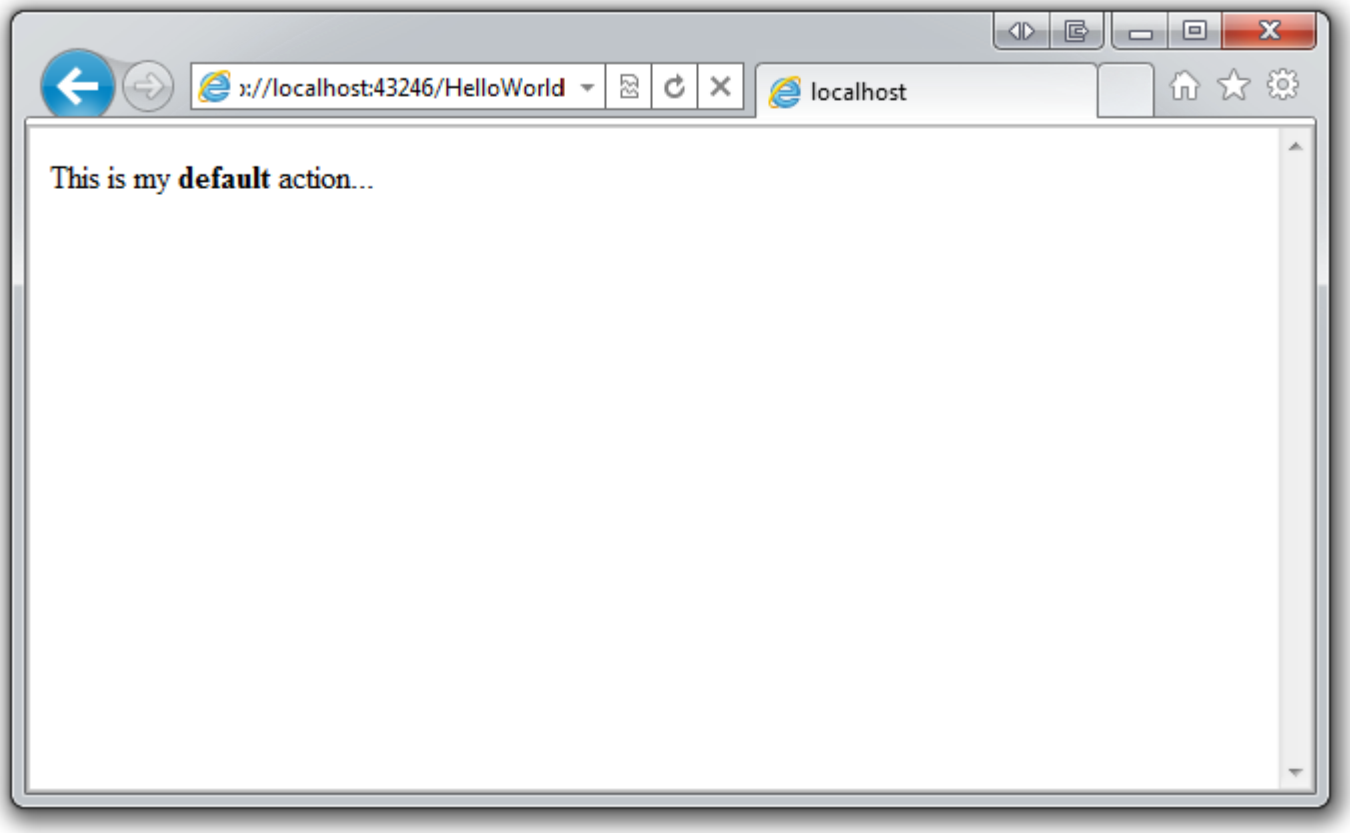
Notice in **Solution Explorer** that a new file has been created named *HelloWorldController.cs*. The file is open in the IDE.



Inside the `public class HelloWorldController` block, create two methods that look like the following code. The controller will return a string of HTML as an example.

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

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

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

        //
        // GET: /HelloWorld/Welcome/

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

Your controller is named `HelloWorldController` and the first method above 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, on my computer, it's *http://localhost:43246/HelloWorld.*) The page in the browser will look like the screenshot below. In the method above, the code returned a string directly. You told the system to just return some HTML, and it did!

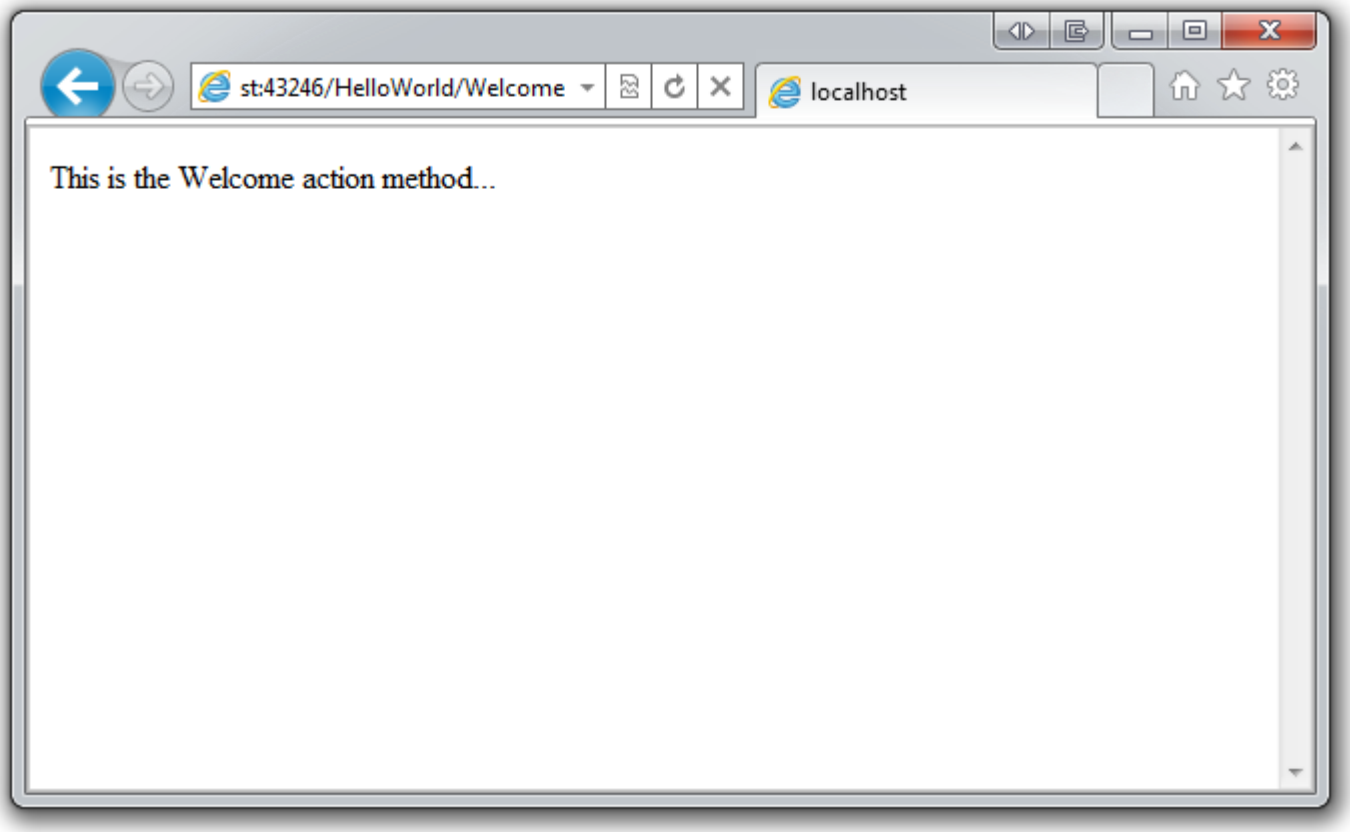


ASP.NET MVC invokes different controller classes (and different action methods within them) depending on the incoming URL. The default mapping logic used by ASP.NET MVC uses a format like this to determine what code to invoke:

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

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.

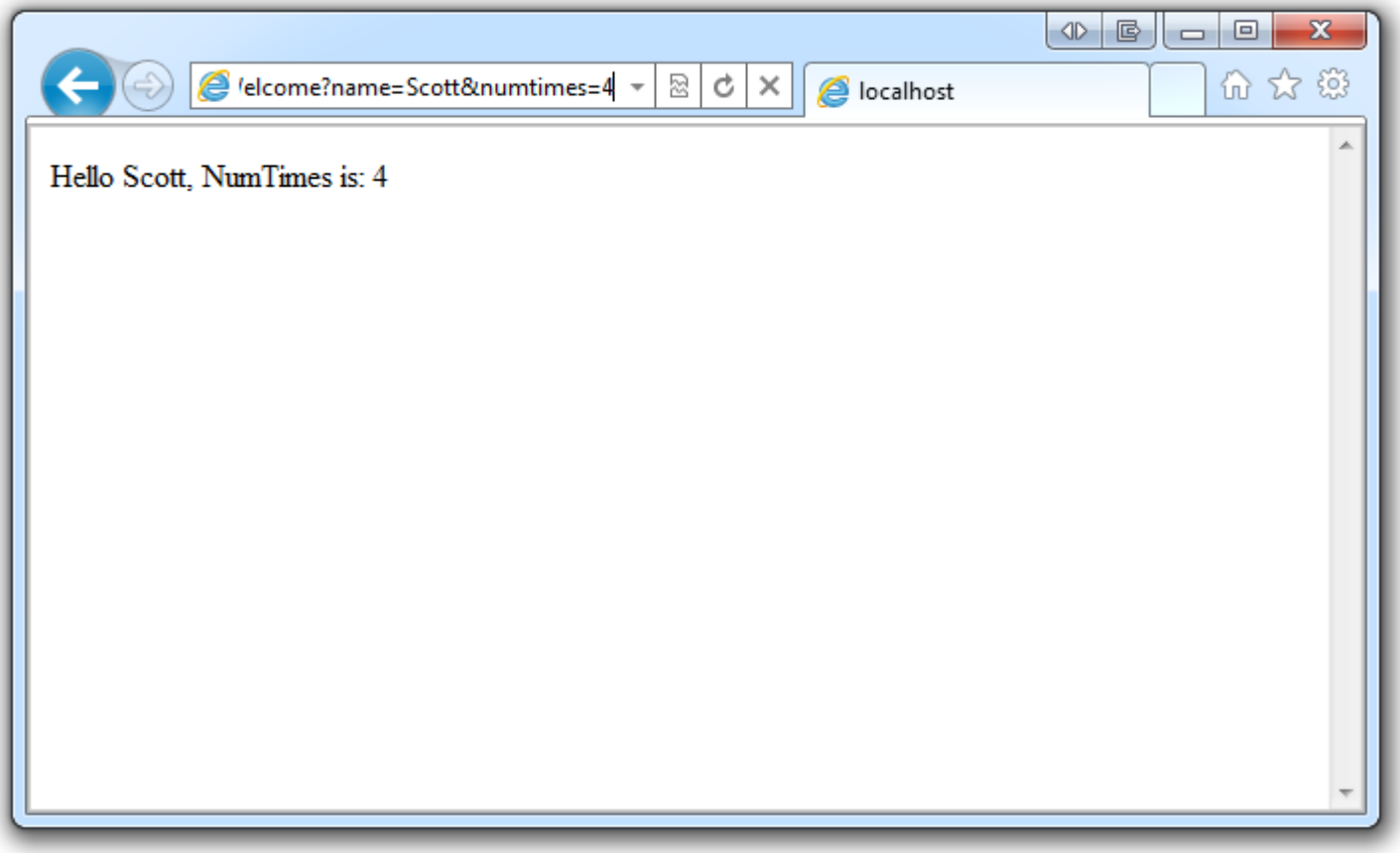
Browse to *http://localhost:xxxx/HelloWorld/Welcome*. The `Welcome` method runs and returns the string "This is the Welcome action method...". The default MVC mapping is `/[Controller]/[ActionName]/[Parameters]`. For this URL, the controller is `HelloWorld` and `Welcome` is the action method. You haven't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



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

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

Run your application and browse to *http://localhost:xxxx/HelloWorld/Welcome?name=Scott&numtimes=4*
You can try different values for `name` and `numtimes` in the URL. The system automatically maps the named parameters from your query string in the address bar to parameters in your method.



In both 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

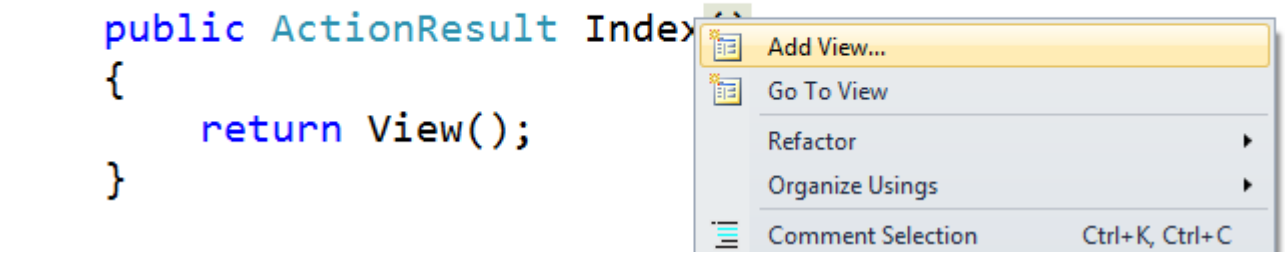
In this section we're going to modify the `HelloWorldController` class to use view template files to cleanly encapsulate the process of generating HTML responses to a client.

We will create our view template files using the new Razor view engine introduced with ASP.NET MVC 3. 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.

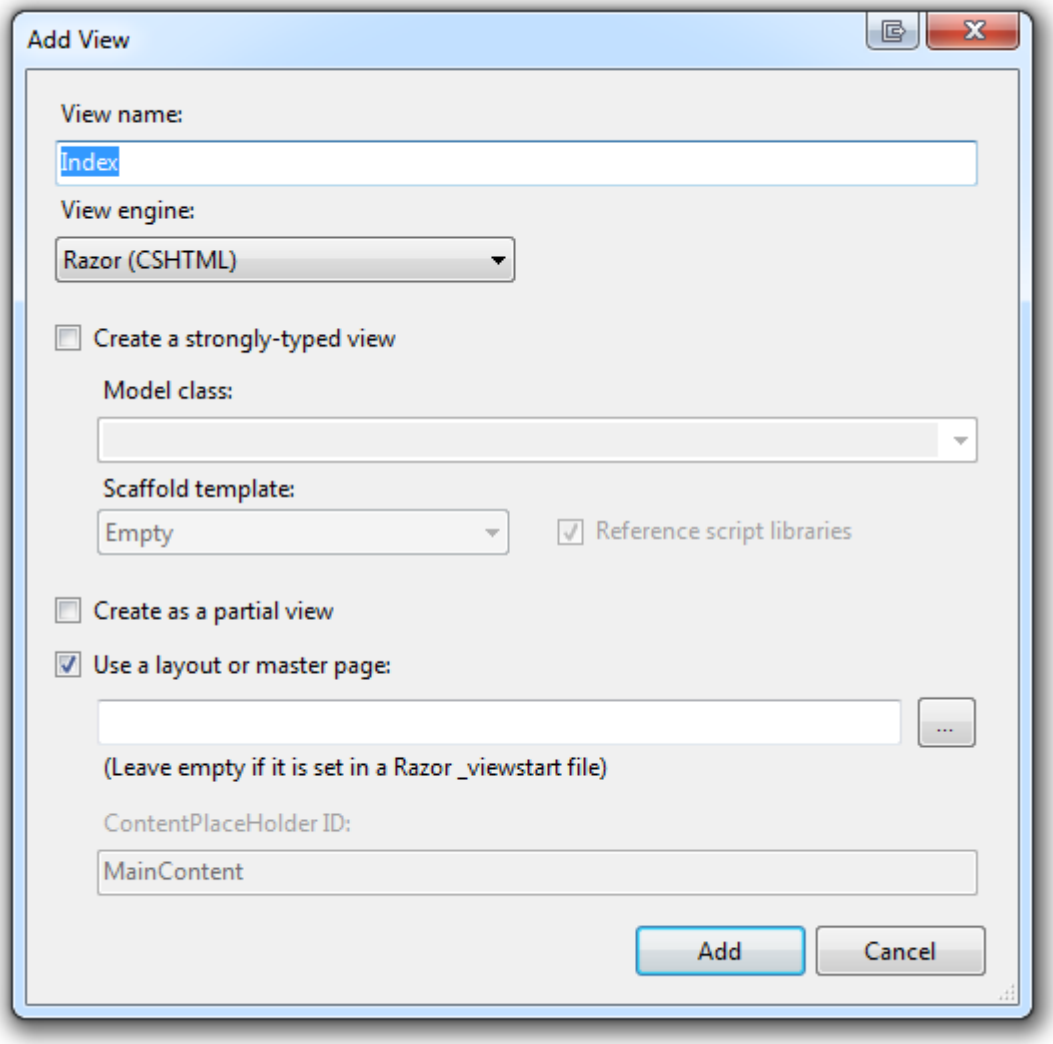
Let's start by using a view template with the `Index` method in the `HelloWorldController` class. Currently the `Index` method returns a string with a message that is hard-coded in the controller class. Change the `Index` method to return a `View` object, as shown in the following:

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

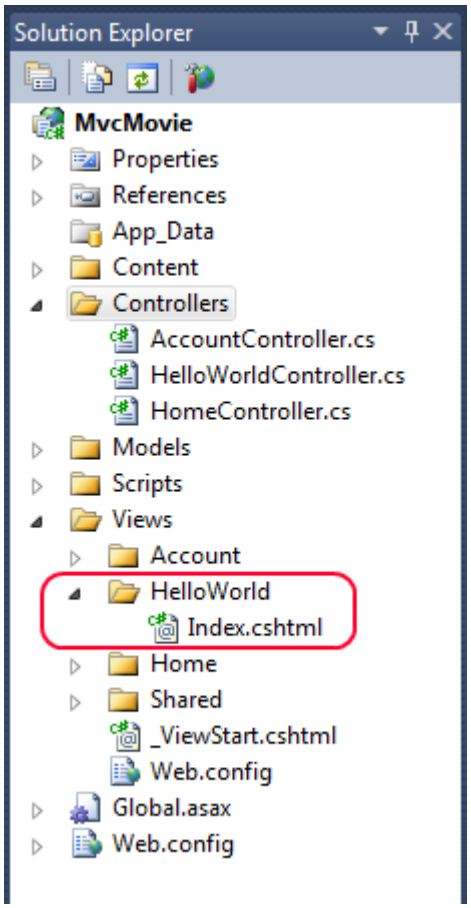
This code indicates that we want to use a view template to generate an HTML response to the browser. Let's add a view template to our project that we can we can use with the`Index` method. To do this, right-click inside the `Index` method and click **Add View**. The **Add View** dialog box appears.



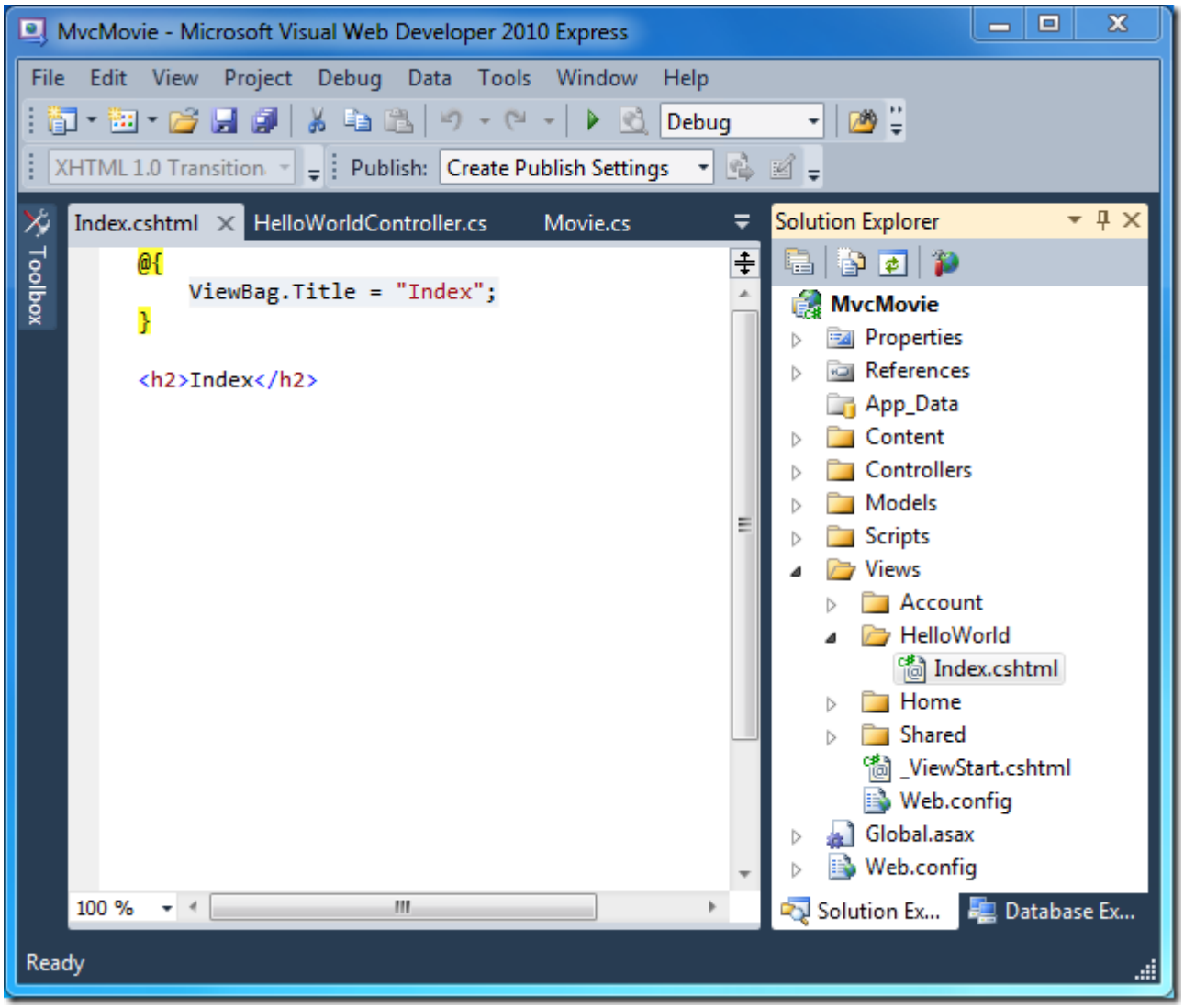
Leave the defaults the way they are and click the **Add** button.



The *MvcMovie**Views**HelloWorld* folder and the *MvcMovie**Views**HelloWorld**Index.cshtml* file are created. You can see them in **Solution Explorer**:



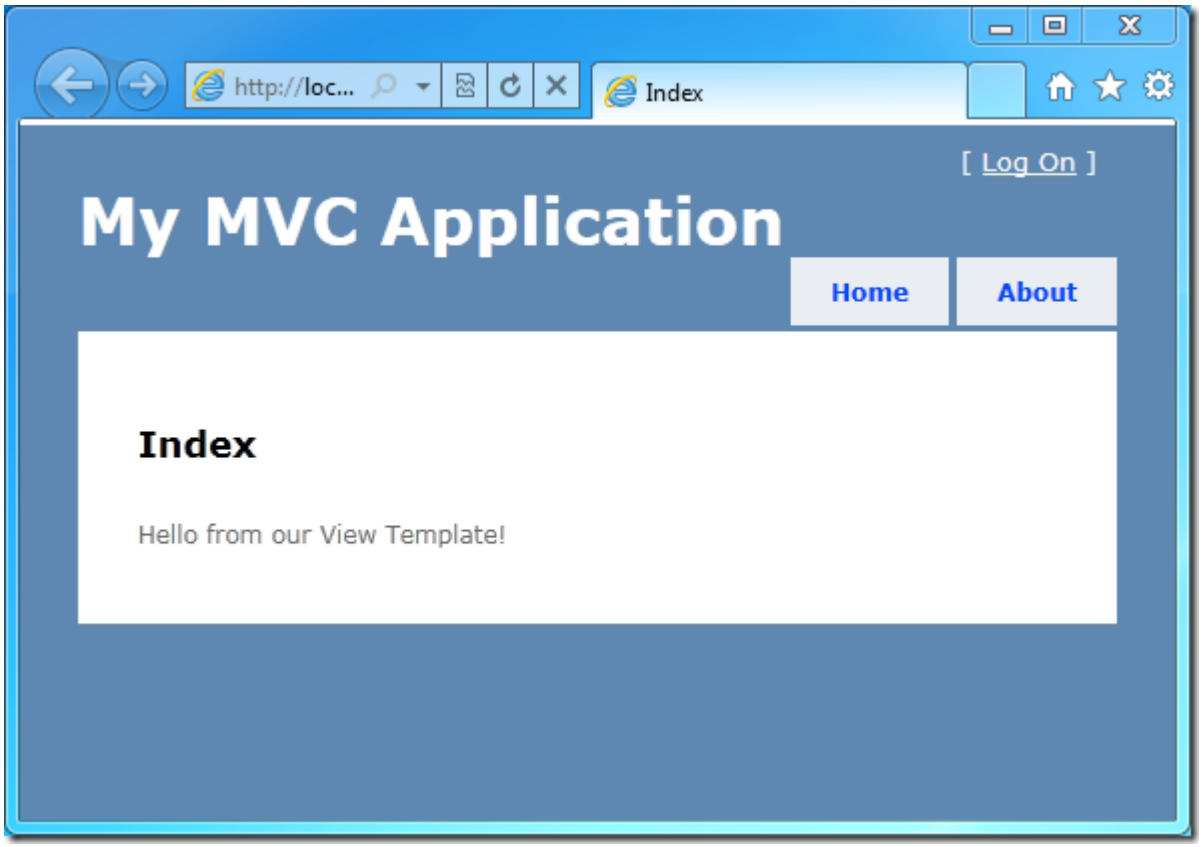
This shows the *Index.cshtml* file that was created:



Add some HTML under the `<h2>` tag. The modified *MvcMovie\Views\HelloWorld\Index.cshtml* file is shown below.

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

<h2>Index</h2>

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

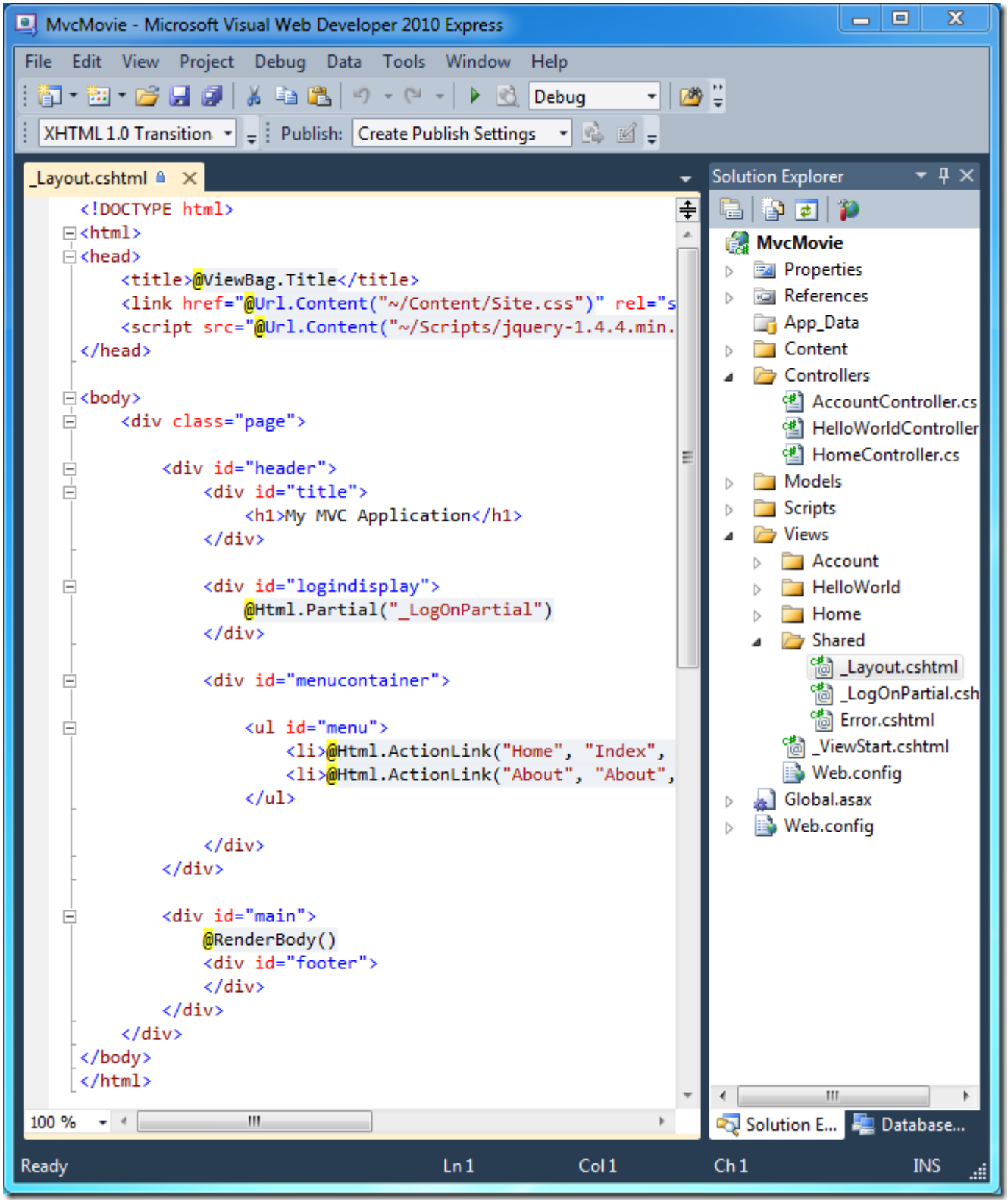
Run the application and browse to the "hello world" controller (*http://localhost:xxxx/HelloWorld*). The `Index` method in your controller didn't do much work; it simply ran the statement `return View();`, which indicated that we wanted to use a view template file to render a response to the browser. Because we did not explicitly specify the name of the view template file to use, ASP.NET MVC defaulted to using the *Index.cshtml* view file within the *\Views\HelloWorld* folder. The image below shows the string hard-coded in the view.



Looks pretty good. However, notice that the browser's title bar says "Index" and the big title on the page says "My MVC Application." Let's change those.

Changing Views and Layout Pages

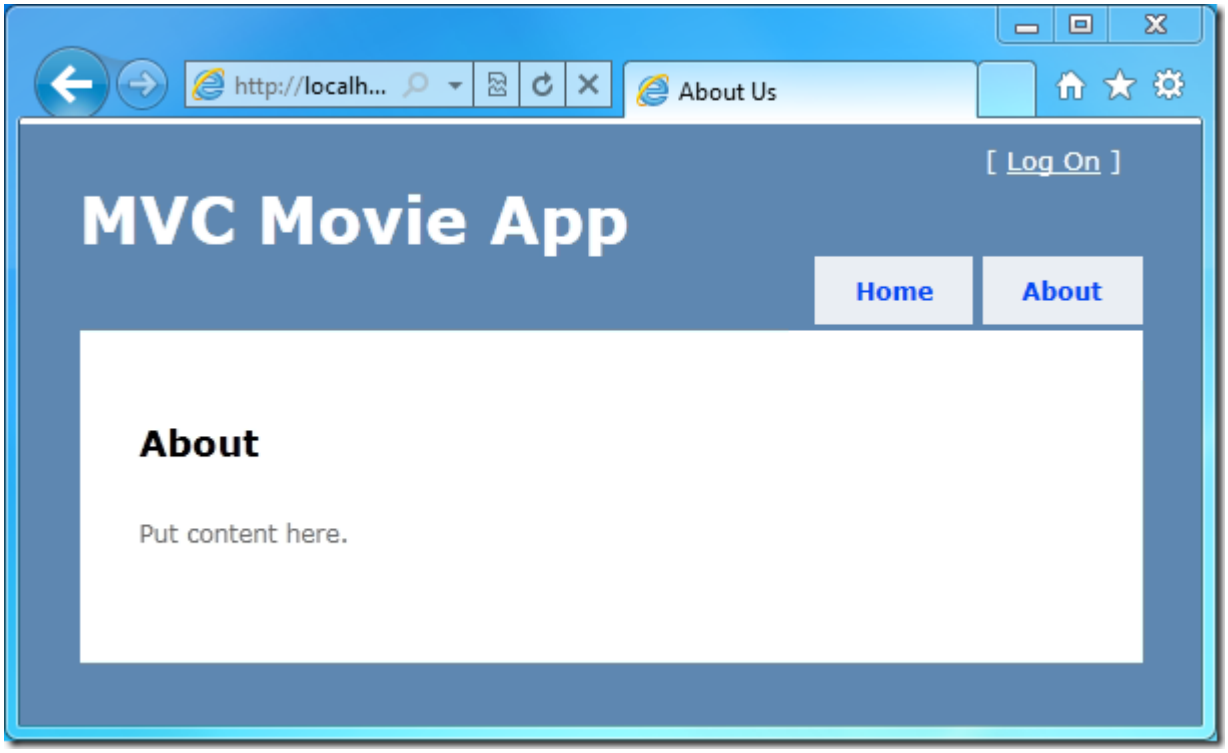
First, let's change the "My MVC Application" title at the top of the page. That text is common to every page. It actually is implemented in only one place in the project, even though it appears on every page in the application. Go to the */Views/Shared* folder in**Solution Explorer** and open the *_Layout.cshtml* file. This file is called a *layout page* and it's the shared "shell" 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. Note the `@RenderBody()` line of code near the bottom of the file.`RenderBody` is a placeholder where all the view-specific pages you create show up, "wrapped" in the layout page. Change the title heading in the layout template from "My MVC Application" to "MVC Movie App".

```
   <div id="title">
      <h1>MVC Movie App</h1>
  </div>
```

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



The complete *_Layout.cshtml* file is shown below:

```
<!DOCTYPE html>
<html>
<head>
    <title>@ViewBag.Title</title>
    <link href="@Url.Content("~/Content/Site.css")" rel="stylesheet" type="text/css"
/>
    <script src="@Url.Content("~/Scripts/jquery-1.4.4.min.js")"
type="text/javascript"></script>
</head>

<body>
    <div class="page">

        <div id="header">
            <div id="title">
                <h1>MVC Movie App</h1>
            </div>
```

```
            <div id="logindisplay">
                @Html.Partial("_LogOnPartial")
            </div>

            <div id="menucontainer">

                <ul id="menu">
                    <li>@Html.ActionLink("Home", "Index", "Home")</li>
                    <li>@Html.ActionLink("About", "About", "Home")</li>
                </ul>

            </div>
        </div>

        <div id="main">
            @RenderBody()
            <div id="footer">
            </div>
        </div>
    </div>
</body>
</html>
```

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

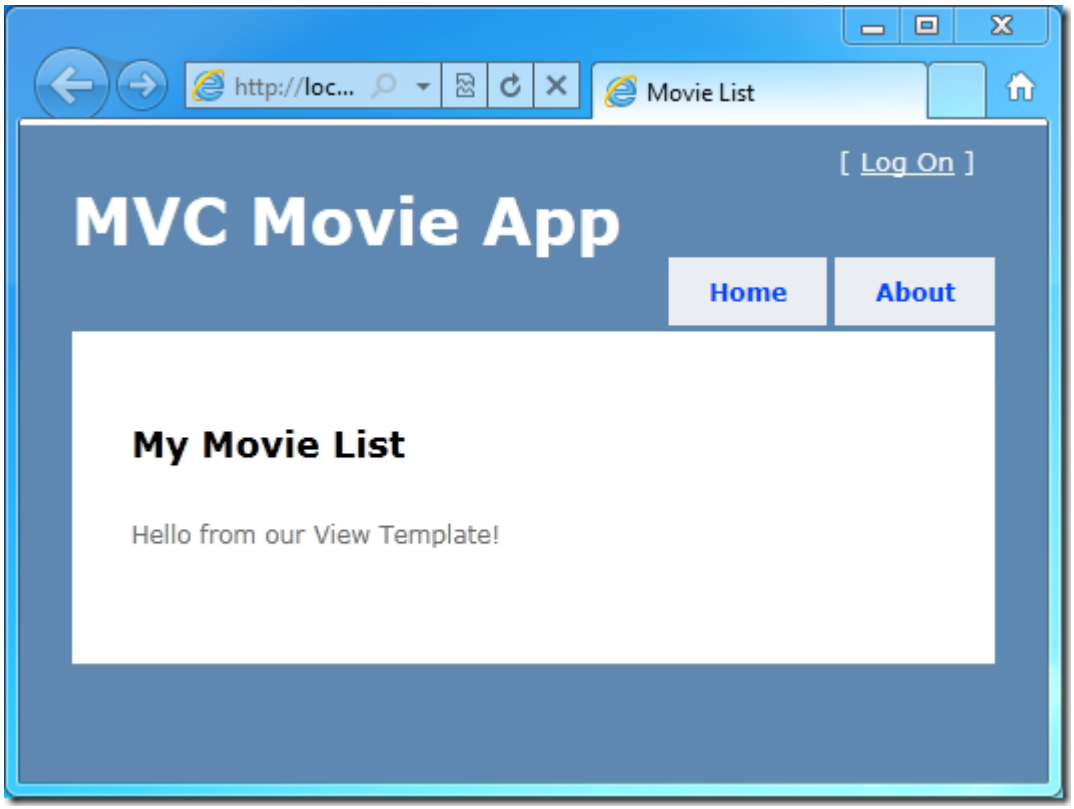
Open *MvcMovie**Views**HelloWorld**Index.cshtml*. There are two places to make a change: first, the text that appears in the title of the browser, and then in the secondary header (the `<h2>` element). We'll make them slightly different so you can see which bit of code changes which part of the app.

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

<h2>My Movie List</h2>

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

To indicate the HTML title we want to display, we're setting a `Title` property of the `ViewBag` object in the code above (which is in the*Index.cshtml* view template). If you look back at the source code of the layout template, you'll notice that the template uses this value in the `<title>` element as part of the `<head>` section of the HTML. Using this approach, you can easily pass other parameters between your view template and your layout file.

Run the application and browse to *http://localhost:xx/HelloWorld*. Notice that the browser title, the primary heading, and the secondary headings have changed. (If you don't see changes in the browser, you might be viewing cached content. Press Ctrl+F5 in your browser to force the response from the server to be loaded.)

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. Our MVC application has a "V" (view) and we'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 parameters, 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 back to the browser.

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

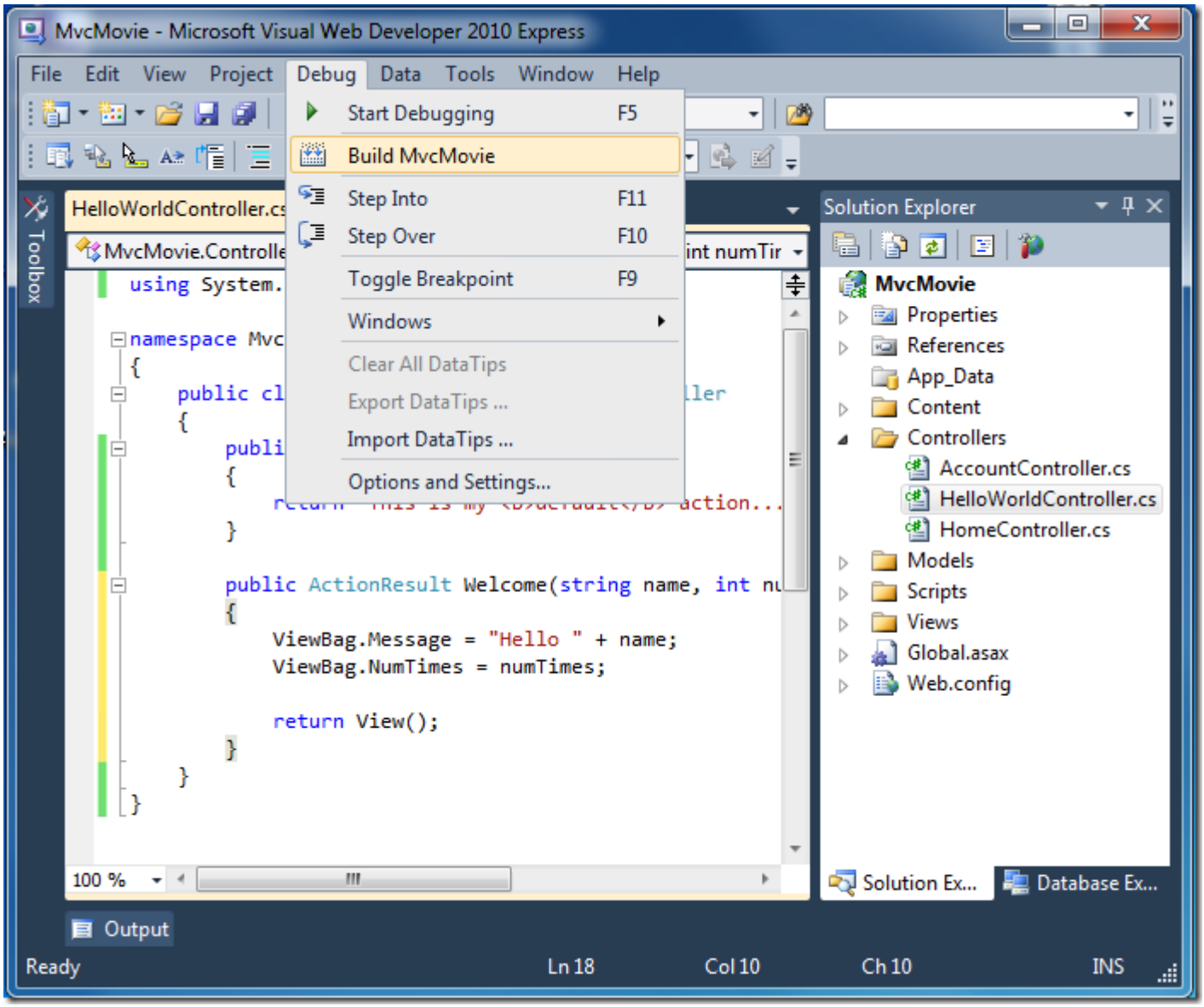
Currently, the `Welcome` action method in the `HelloWorldController` class takes a `name` and a `numTimes` parameter and then outputs the values directly to the browser. Rather than have the controller render this response as a string, let’s change it to use a view template instead. The view template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

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

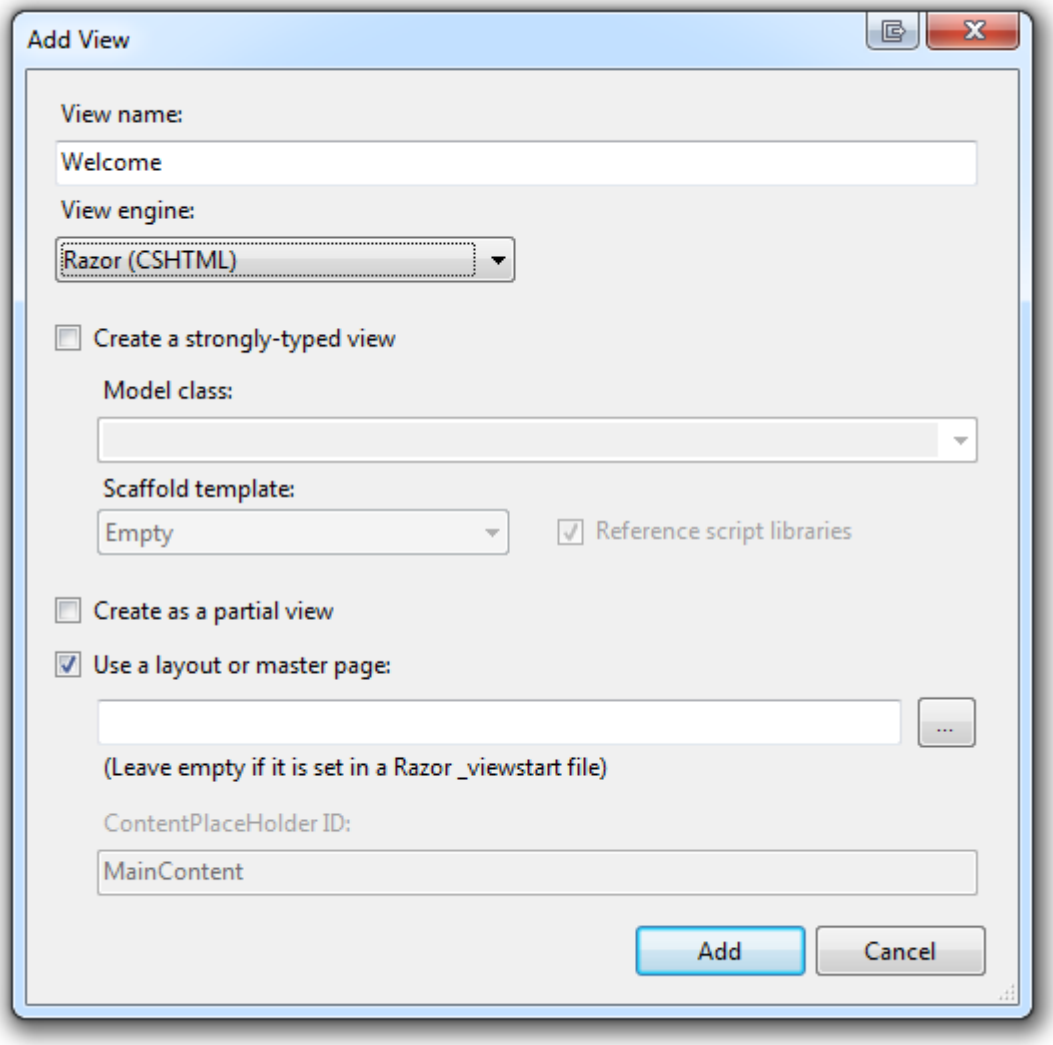
```
using System.Web;
using System.Web.Mvc;

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

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

            return View();
        }
    }
}
```

Now the `ViewBag` object contains data that will be passed to the view automatically.

Next, we need a Welcome view template! In the **Debug** menu, select **Build MvcMovie** to make sure the project is compiled.



Then right-click inside the `Welcome` method and click **Add View**. Here's what the **Add View** dialog box looks like:



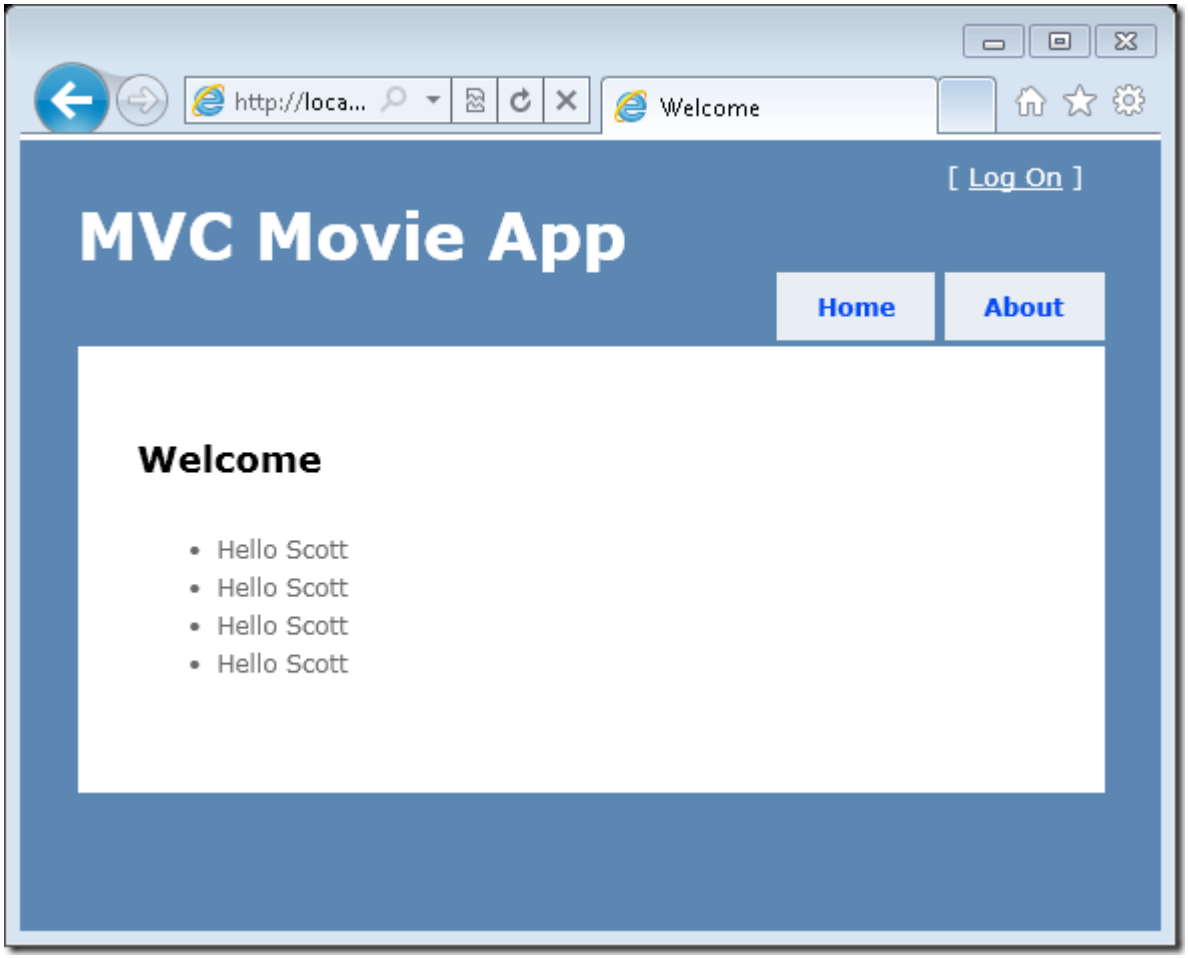
Click **Add**, and then add the following code under the `<h2>` element in the new *Welcome.cshtml* file. We'll make a loop and say "Hello" as many times as the user says we should!

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

<h2>Welcome</h2>

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

Run the application and browse to the following URL:

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

Now data is taken from the URL and passed to the controller automatically. The controller packages up the data into a `ViewBag` object and passes that object to the view. The view than displays the data as HTML to the user.



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

Adding a Model

In this section we'll add some classes for managing movies in a database. These classes will be the "model" part of our MVC application.

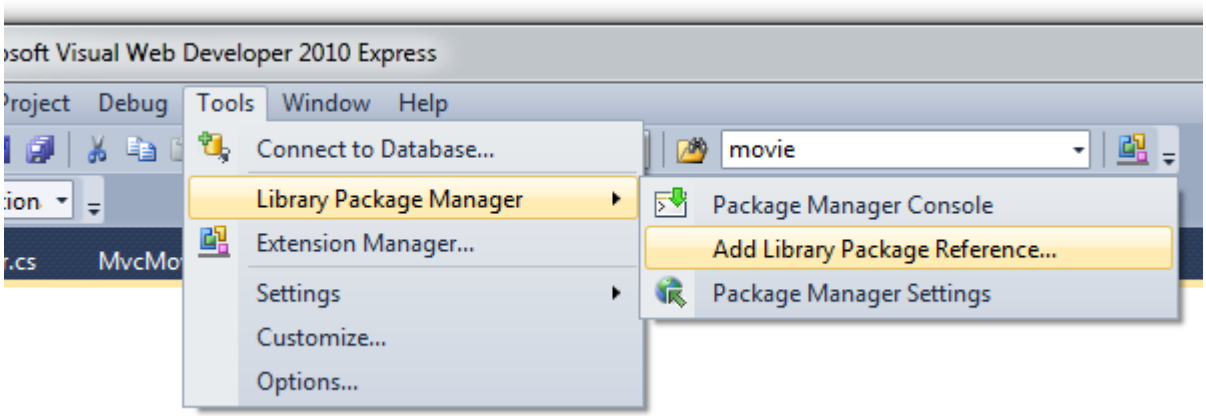
We'll use a .NET Framework data-access technology known as the Entity Framework to define and work with these model classes. The Entity Framework (often referred to as "EF") supports a development paradigm called *code-first*. Code-first allows you to create model objects by writing simple classes. (These are also known as POCO classes, from "plain-old CLR objects.") You can then have the database created on the fly from your classes, which enables a very clean and rapid development workflow.

Using NuGet to Install EFCodeFirst

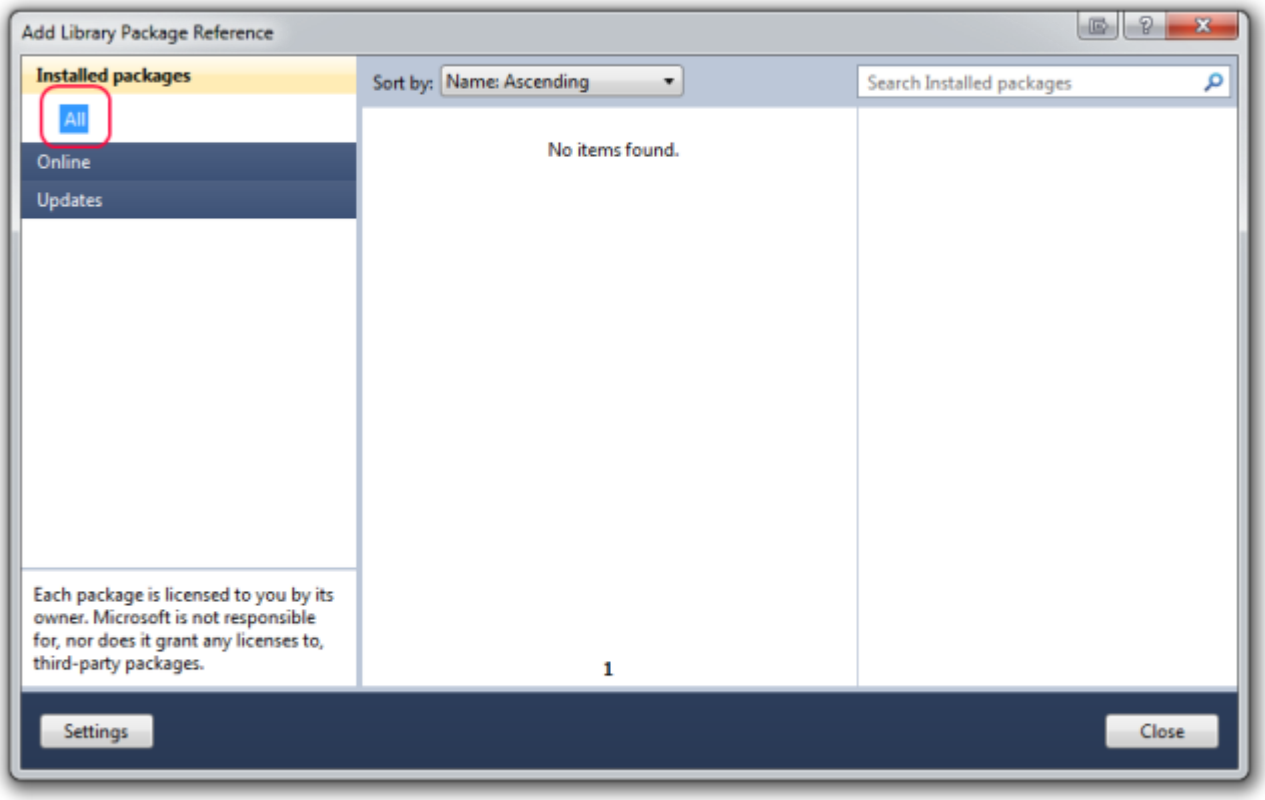
We'll start by using the NuGet package manager (automatically installed by ASP.NET MVC 3) to add the EFCodeFirst library to the `MvcMovie` project. This library lets us use the code-first approach.

Warning:
You must stop debugging before you access the NuGet package manager. If you access NuGet while you are still debugging, you'll get the error message and the ***Add Library Package Reference...*** *menu item will disappear. You must then exit Visual Web Developer and restart the project.*

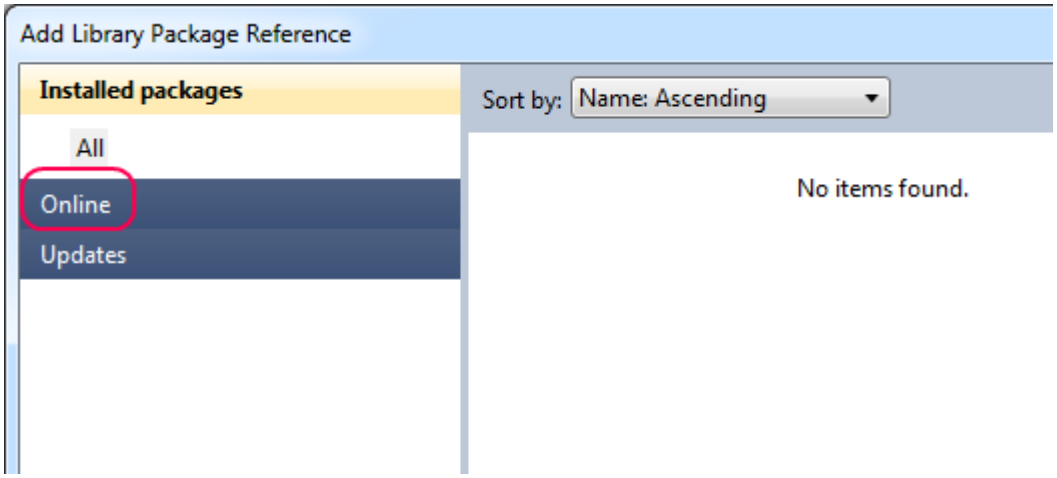
From the **Tools** menu, select **Library Package Manager** and then **Add Library Package Reference**.



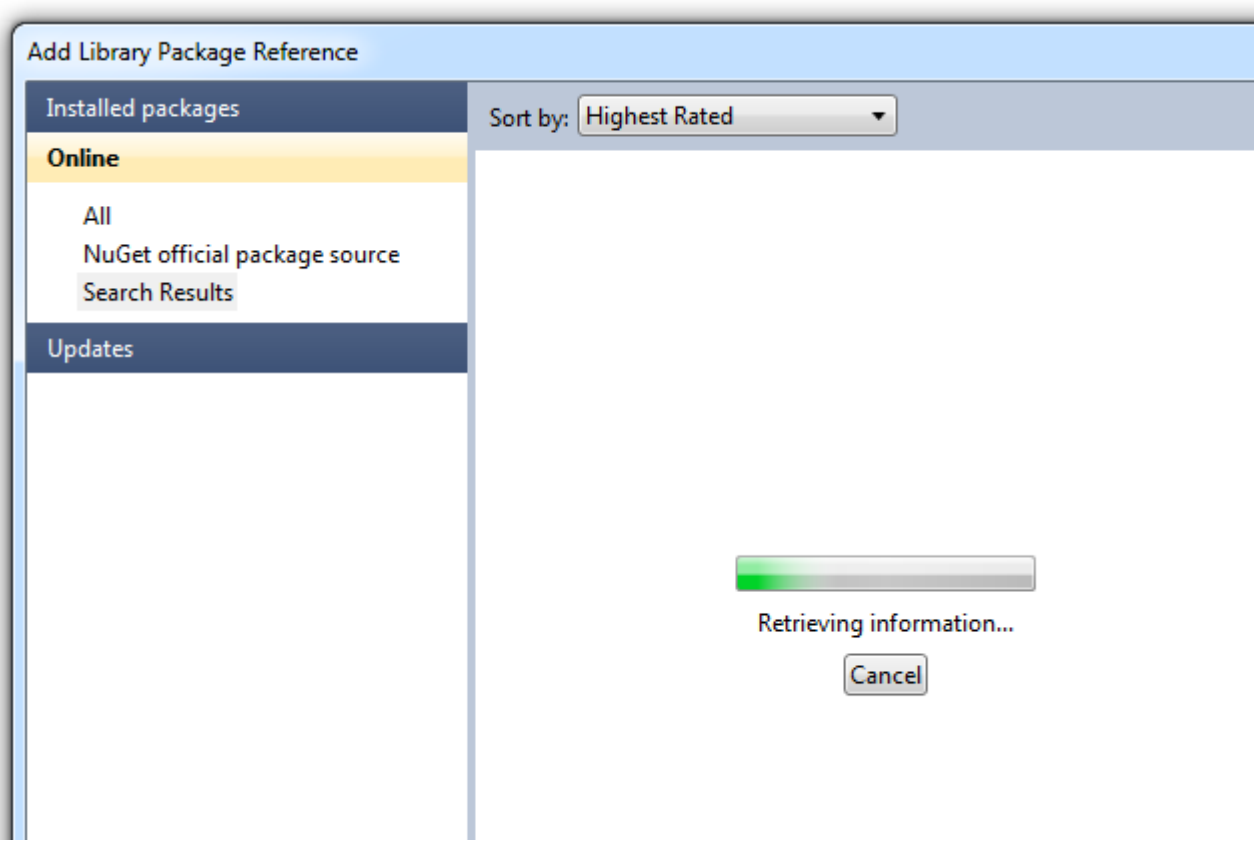
The **Add Library Package Reference** dialog box appears.



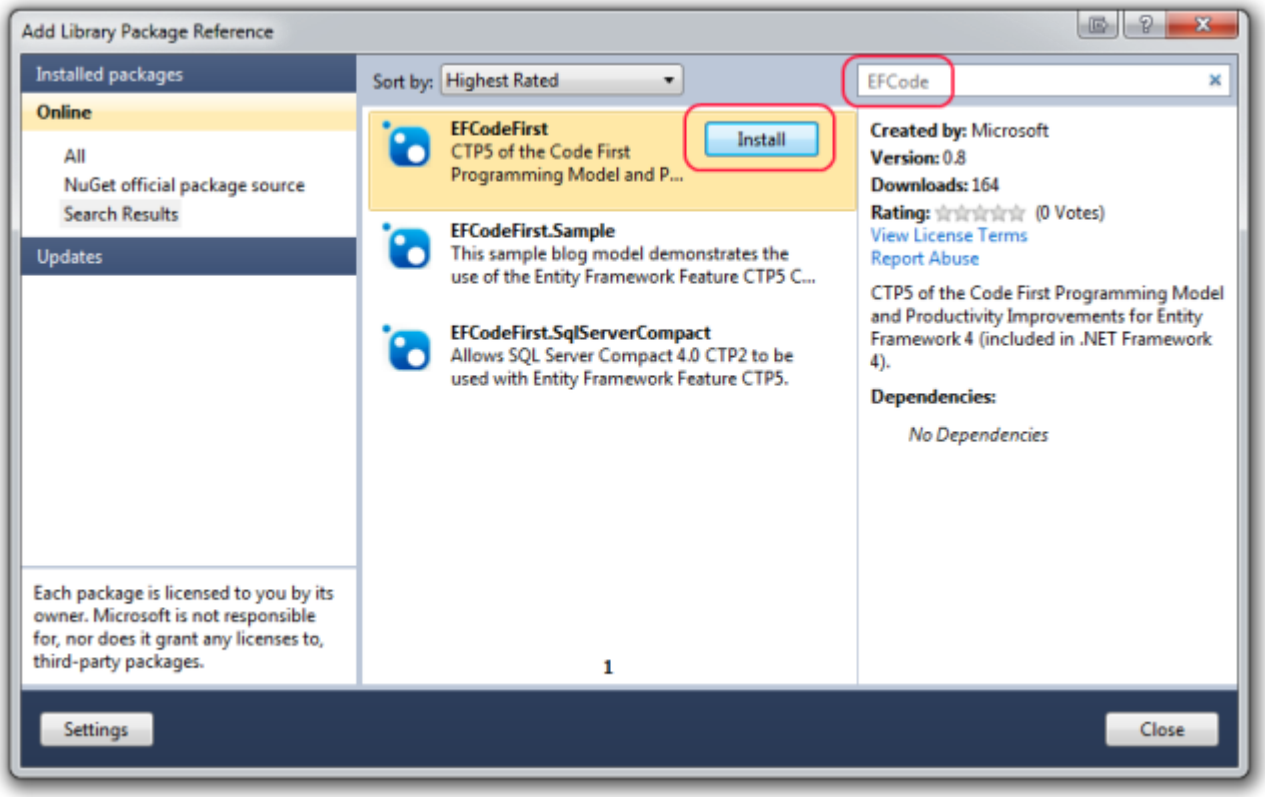
By default, **All** is selected in the left pane. Because no packages are installed, the center pane shows **No items found**. Click **Online** in the left pane.



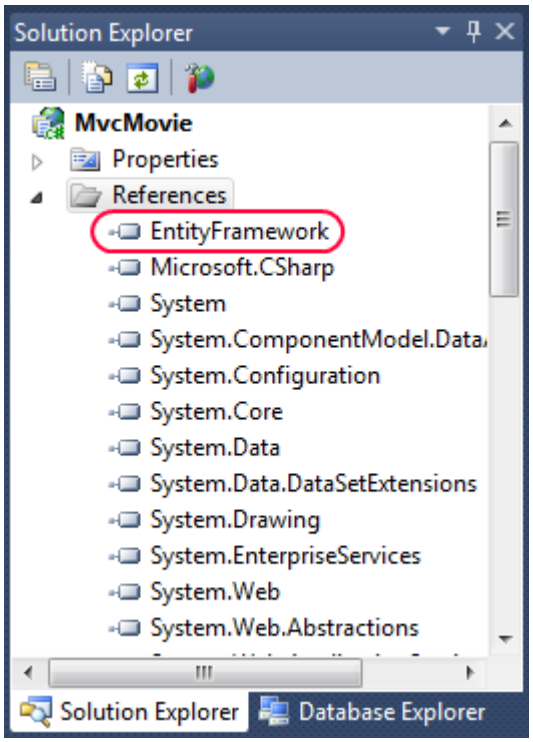
NuGet queries the server for all available packages.



There are hundreds of packages available. We're interested in the EFCodeFirst package. In the search box, enter "EFCode". In the search results, select the **EFCodeFirst** package and click the **Install** button.

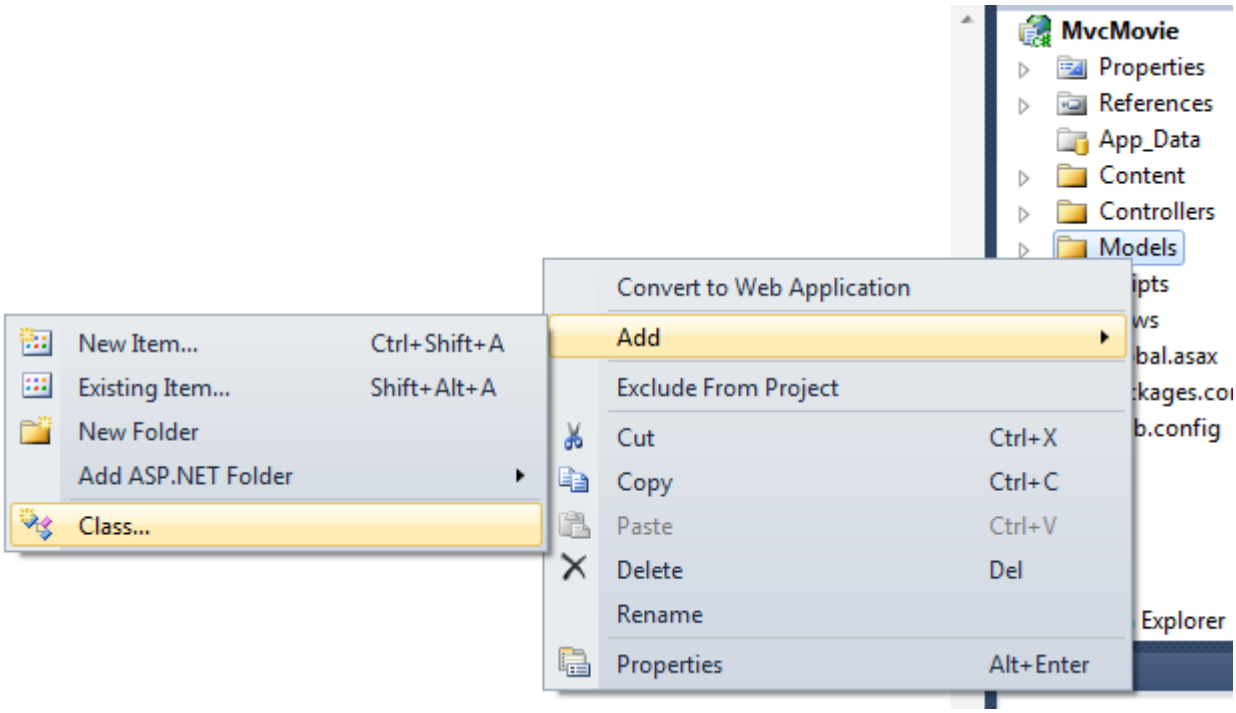


After the package installs, click **Close**. The installation process downloaded the EFCodeFirst library and added it to the `MvcMovie` project. The EFCodeFirst library is contained in the *EntityFramework* assembly.

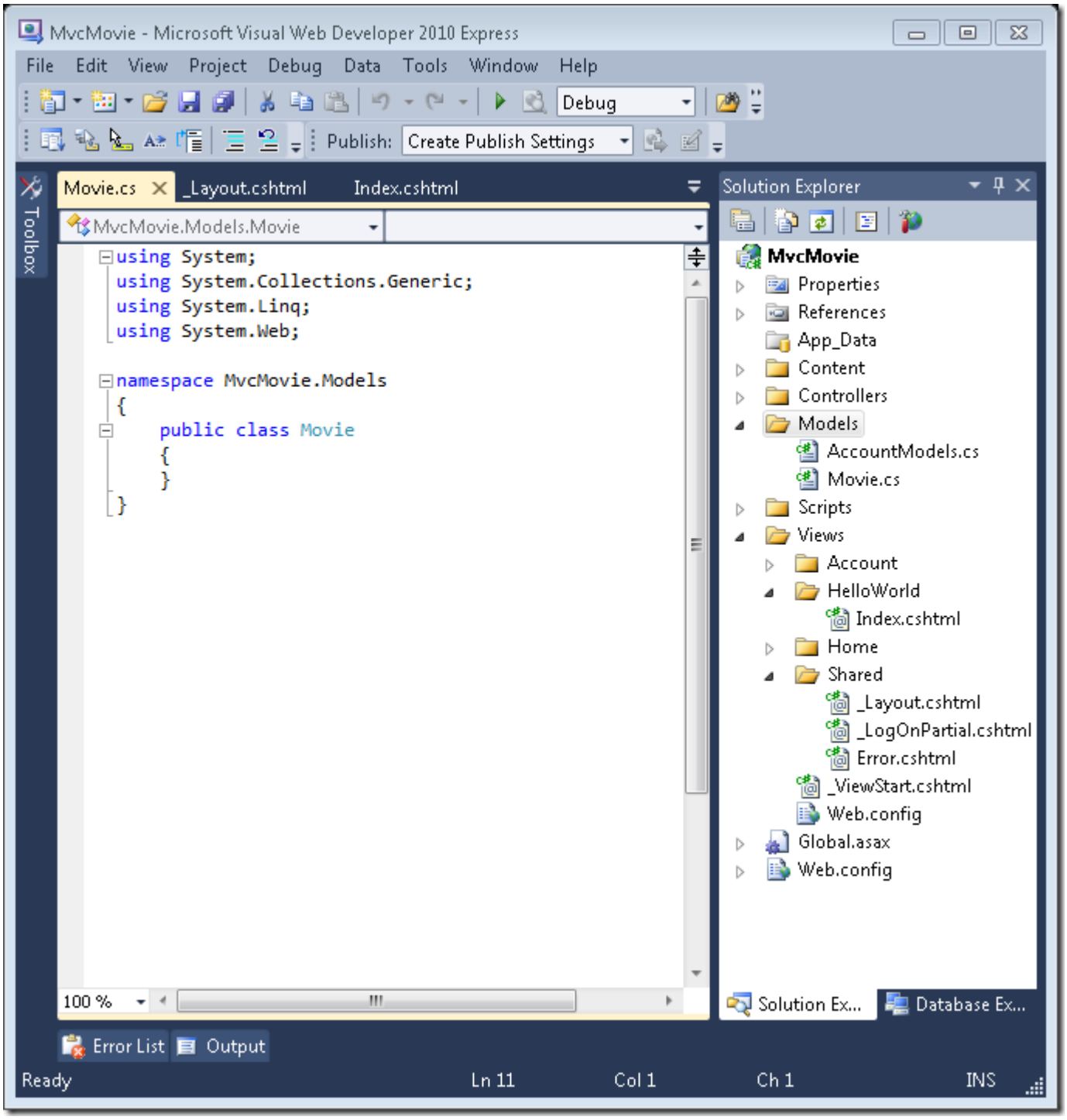


Adding Model Classes

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



Name the class "Movie".



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

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

We'll use the `Movie` class above 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:

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

The `MovieDBContext` class represents the Entity Framework movie database context, which handles fetching, storing, and updating `Movie` class instances in a database. The `MovieDBContext` derives from the `DbContext` base class provided by the Entity Framework. In order to be able to reference the `DbContext` class, you need to add the following `using` statement at the top of the file:

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

The complete *Movie.cs* file is shown below.

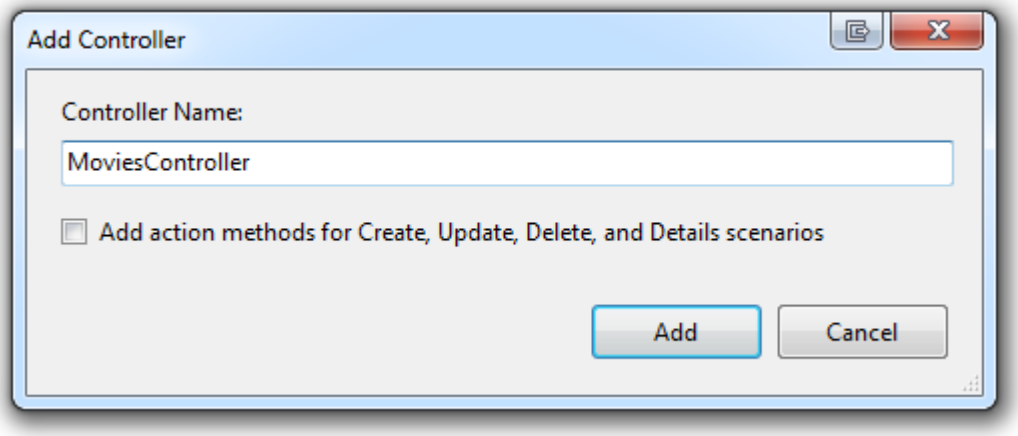
```
using System;
using System.Data.Entity;

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

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

This small amount of code is everything you need to write in order to represent and store the movie data in a database. Next, let's build a new `MoviesController` class that we 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.

Right-click the *Controllers* folder and make a new `MoviesController` class.



This creates a new *MoviesController.cs* file in the project's *Controllers* folder. Let's update the `Index` action method in the `MoviesController` class so that it retrieves the list of movies.

Note that in order to be able to reference the `MovieDBContext` class we created earlier, you need to add the following two `using` statements at the top of the file:

```
using MvcMovie.Models;
using System.Linq;
```

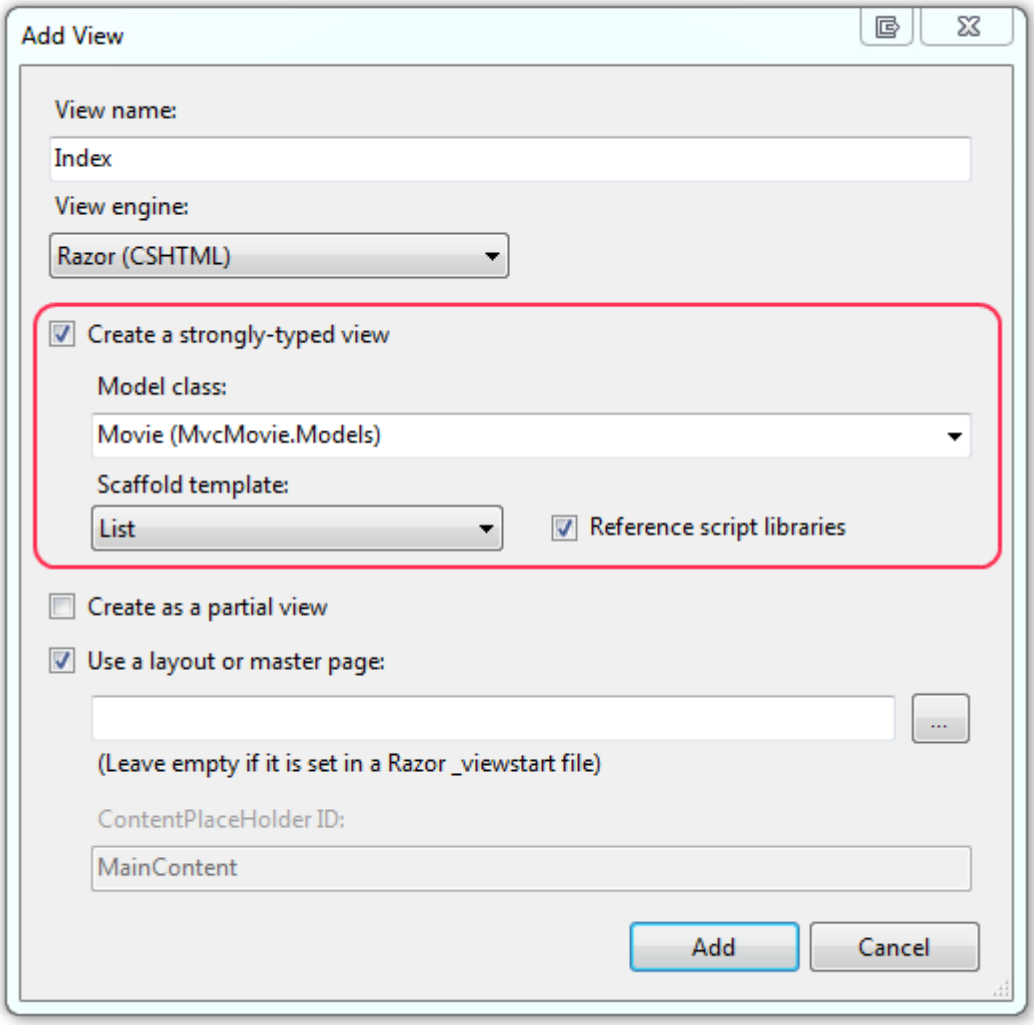
The code for the `MoviesController` class looks like:

```
using MvcMovie.Models;
using System.Linq;
using System;
using System.Web.Mvc;

namespace MvcMovie.Controllers
{
    public class MoviesController : Controller
    {
        MovieDBContext db = new MovieDBContext();

        public ActionResult Index()
        {
            var movies = from m in db.Movies
                         where m.ReleaseDate > new DateTime(1984, 6, 1)
                         select m;

            return View(movies.ToList());
        }
    }
}
```

The code is performing a LINQ query to retrieve only movies that were released after the summer of 1984. We'll need a view template to render this list of movies, so right-click inside the method and select **Add View** to create it.

In the **Add View** dialog box, we'll indicate that we're passing a `Movie` class to the view template. Unlike the previous times when we used the **Add View** dialog box and chose to create an empty template, this time we'll indicate that we want Visual Web Developer to automatically *scaffold* a view template for us, meaning that the new view template will contain some default content. To do this, select**List** in the **Scaffold template** drop-down list.

Remember that after you've created a new class, you'll need to compile your application before the class shows up in the **Add View** dialog box.



Click **Add**. Visual Web Developer automatically generates the code for a view that displays a list of movies. This is a good time to change the `<h2>` heading to something like "My Movie List" like you did earlier with the "Hello World" view.

The code below show a portion of the `Index` view for the movie controller. In this section, change the format string for the release date from `{0:g}` to `{0:d}` (that is, from general date to short date). Change the format string for the `Price` property from `{0:F}` to `{0:c}` (from float to currency).

In addition, in the table header, change the column name from "ReleaseDate" to "Release Date" (two words).

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

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

<h2>My Movie List</h2>

<p>
    @Html.ActionLink("Create New", "Create")
</p>
<table>
    <tr>
        <th></th>
        <th>
            Title
        </th>
        <th>
            Release Date
        </th>
        <th>
            Genre
        </th>
        <th>
            Price
        </th>
    </tr>

@foreach (var item in Model) {
    <tr>
        <td>
            @Html.ActionLink("Edit", "Edit", new { id=item.ID }) |
            @Html.ActionLink("Details", "Details", new { id=item.ID }) |
            @Html.ActionLink("Delete", "Delete", new { id=item.ID })
        </td>
        <td>
            @item.Title
        </td>
        <td>
            @String.Format("{0:d}", item.ReleaseDate)
        </td>
        <td>
            @item.Genre
        </td>
        <td>
            @String.Format("{0:c}", item.Price)
        </td>
    </tr>
}
</table>
```

Strongly typed Models and the @model keyword

Earlier in this tutorial, we covered how a Controller can pass data/objects to a view template using the ViewBag. The ViewBag is a dynamic object, and provides a convenient, late-bound, way to pass bits of information to a view.

ASP.NET MVC also provides the ability to pass data/objects to a view template using a strongly-typed approach. This strongly-typed approach enables richer editor intellisense, and better compile-time checking of your code. We are using this approach above with our `MoviesController` and *Index.cshtml* view template.

Notice how we are passing an argument when calling the `View()` helper method within our `Index` action:

```
public class MoviesController : Controller
{
    MovieDBContext db = new MovieDBContext();

    public ActionResult Index()
    {
        var movies = from m in db.Movies
                     where m.ReleaseDate > new DateTime(1984, 6, 1)
                     select m;

        return View(movies.ToList());
    }
}
```

This line of code indicates that we are passing a list of Movies from our Controller to our View:

```
return View(movies.ToList());
```

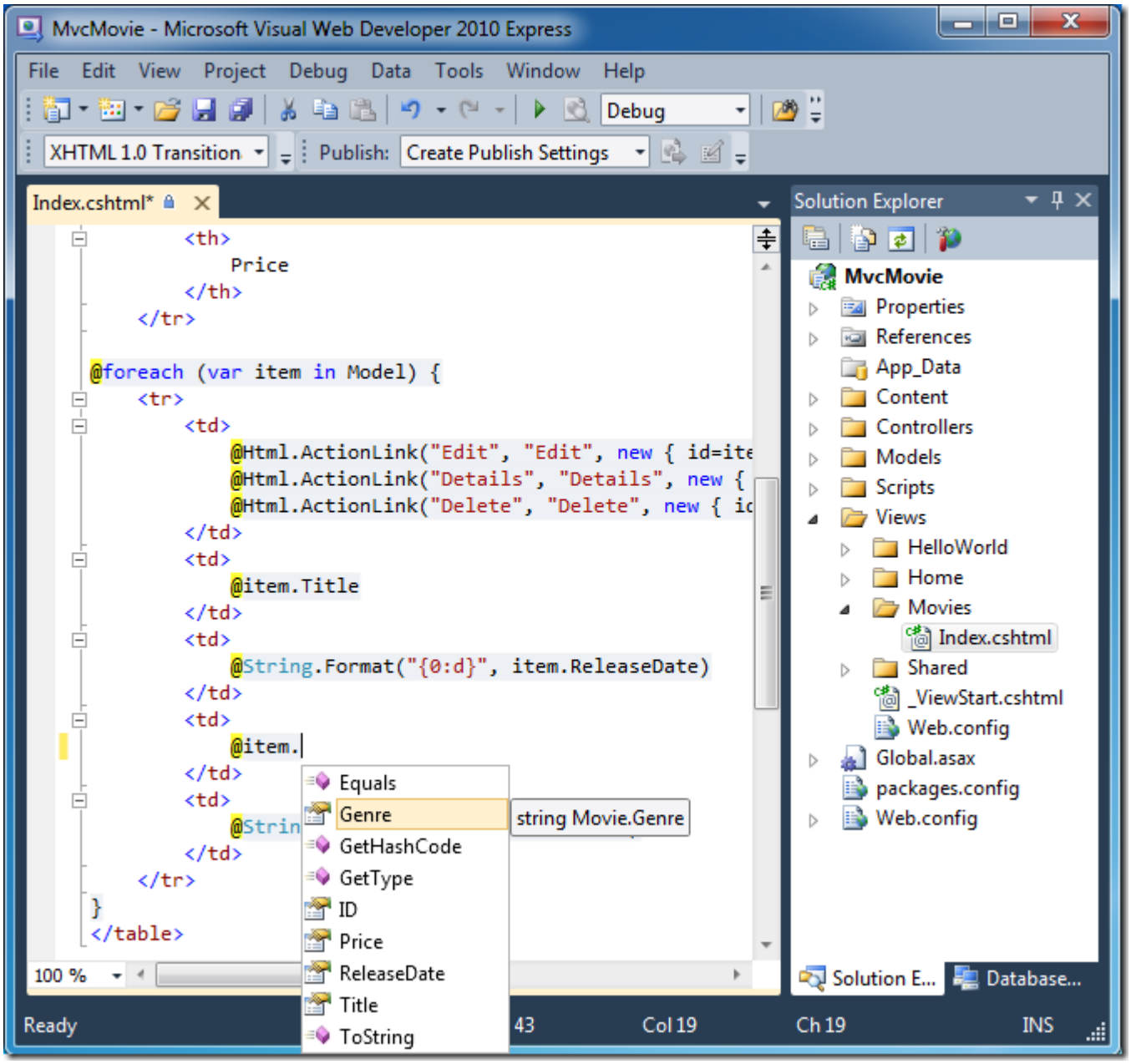
View templates can indicate the type of object they expect to be passed using a `@model` statement at the top of the view template file. Remember how when we created our *Index.cshtml* view template we checked the "Create a strongly-typed view" checkbox within the "Add View" dialog, and told it that we were passing a list of Movies? This caused Visual Web Developer to automatically emit the following `@model` statement at the top of our *Index.cshtml* file when it scaffolded our view:

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

This `@model` directive allows us to access the list of Movies that our Controller passed us using a "Model" object that is strongly-typed. For example, within our *Index.cshtml* template we are looping over the movies by doing a foreach statement on this strongly-typed Model:

```
@foreach (var item in Model) {
    <tr>
        <td>
            @Html.ActionLink("Edit", "Edit", new { id=item.ID }) |
            @Html.ActionLink("Details", "Details", new { id=item.ID }) |
            @Html.ActionLink("Delete", "Delete", new { id=item.ID })
        </td>
        <td>
            @item.Title
        </td>
        <td>
            @String.Format("{0:d}", item.ReleaseDate)
        </td>
        <td>
            @item.Genre
        </td>
        <td>
            @String.Format("{0:c}", item.Price)
        </td>
    </tr>
}
```

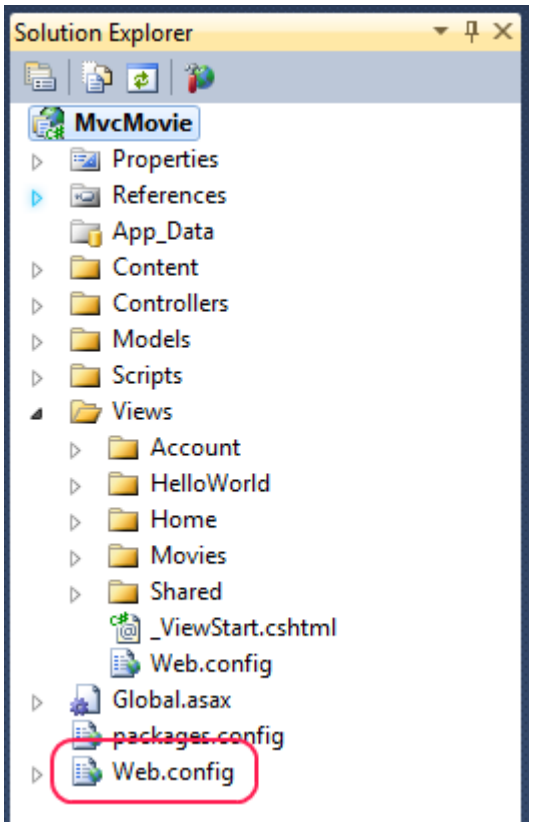
Because our "Model" is strongly-typed (as an `IEnumerable<Movie>`), each "item" within the loop is strongly- typed as a "Movie". Among other benefits, this means that we get compile-time checking of our code, and full Intellisense support within the code editor:


MvcMovie - Microsoft Visual Web Developer 2010 Express
File Edit View Project Debug Data Tools Window Help
Debug
XHTML 1.0 Transition
Publish: Create Publish Settings
Index.cshtml*
<th>
Price
</th>
</tr>
@foreach (var item in Model) {
<tr>
<td>
@Html.ActionLink("Edit", "Edit", new { id=ite
@Html.ActionLink("Details", "Details", new {
@Html.ActionLink("Delete", "Delete", new { ic
</td>
<td>
@item.Title
</td>
<td>
@String.Format("{0:d}", item.ReleaseDate)
</td>
<td>
@item.
</td>
<td>
@Strin
</td>
</tr>
}
</table>
Equals
Genre
GetHashCode
GetType
ID
Price
ReleaseDate
Title
ToString
string Movie.Genre
100 %
Solution Explorer
MvcMovie
Properties
References
App_Data
Content
Controllers
Models
Scripts
Views
HelloWorld
Home
Movies
Index.cshtml
Shared
_ViewStart.cshtml
Web.config
Global.asax
packages.config
Web.config
Solution E...
Database...
Ready
43
Col 19
Ch 19
INS

Creating a Connection String and Working with SQL Server Express

The `MovieDBContext` class we created in the previous section 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? We'll do that by adding connection information in the *Web.config* file of our application.

Open the application root *Web.config* file. (Not the *Web.config* file in the *Views* folder.) The image below show both *Web.config* files; open the *Web.config* file circled in red.



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

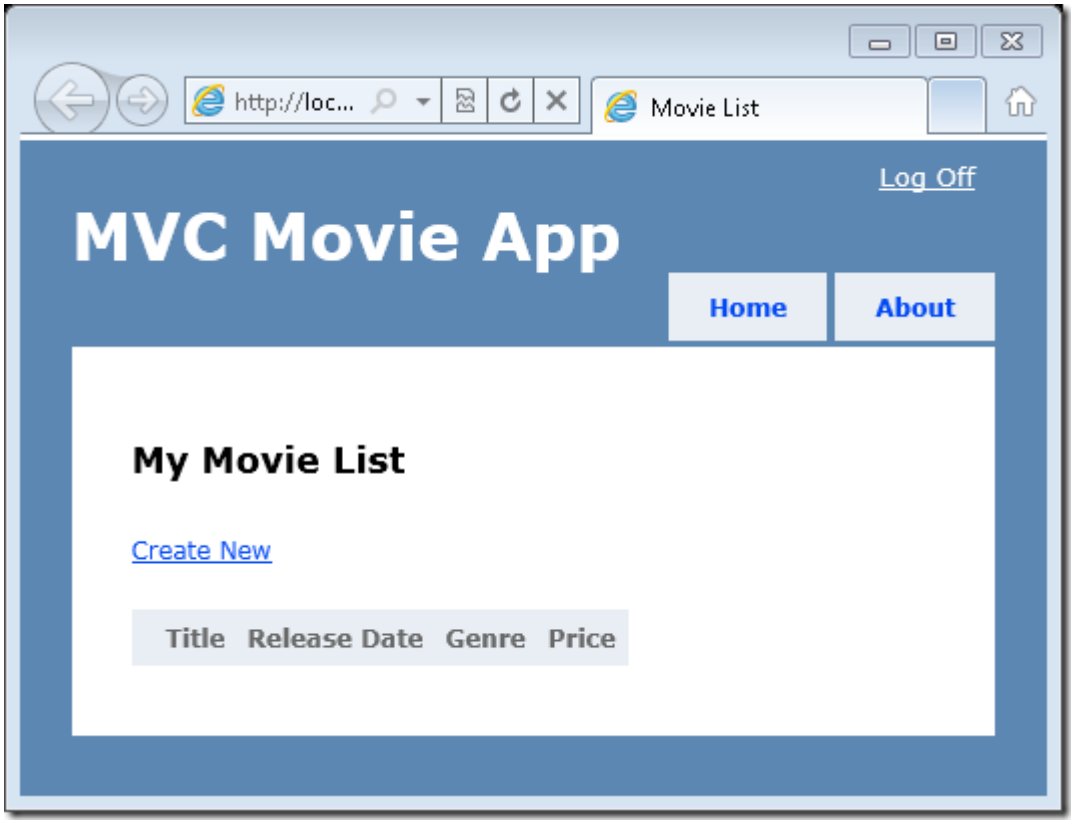
```
    <add name="MovieDBContext"
     connectionString="Server=.\SQLEXPRESS;
     Database=Movies;Trusted_Connection=true"
     providerName="System.Data.SqlClient" />
```

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

```
<configuration>
  <connectionStrings>
    <add name="ApplicationServices"
         connectionString="data source=.\SQLEXPRESS;Integrated
Security=SSPI;AttachDBFilename=|DataDirectory|aspnetdb.mdf;User Instance=true"
         providerName="System.Data.SqlClient" />
    <add name="MovieDBContext"
         connectionString="Server=.\SQLEXPRESS;
         Database=Movies;Trusted_Connection=true"
         providerName="System.Data.SqlClient" />
  </connectionStrings>
```

The value of the `connectionString` attribute indicates that we want to use a `Movies` database that's managed by a local instance of SQL Server Express. When you installed Visual Web Developer Express, the installation process automatically installed SQL Server Express on your computer as well, which means you have everything necessary for the database to work.

Run the application and browse to the `Movies` controller by appending*/Movies* to the URL in the address bar of your browser. An empty list of movies is displayed.



EF code-first detected that the database connection-string we provided pointed to a “Movies” database that didn’t yet exist. And so it helpfully created one for us automatically. You can verify that it's been created by looking in the *C:\Program Files\Microsoft SQL \MSSQL10.SQLEXPRESS\MSSQL\DATA* folder.

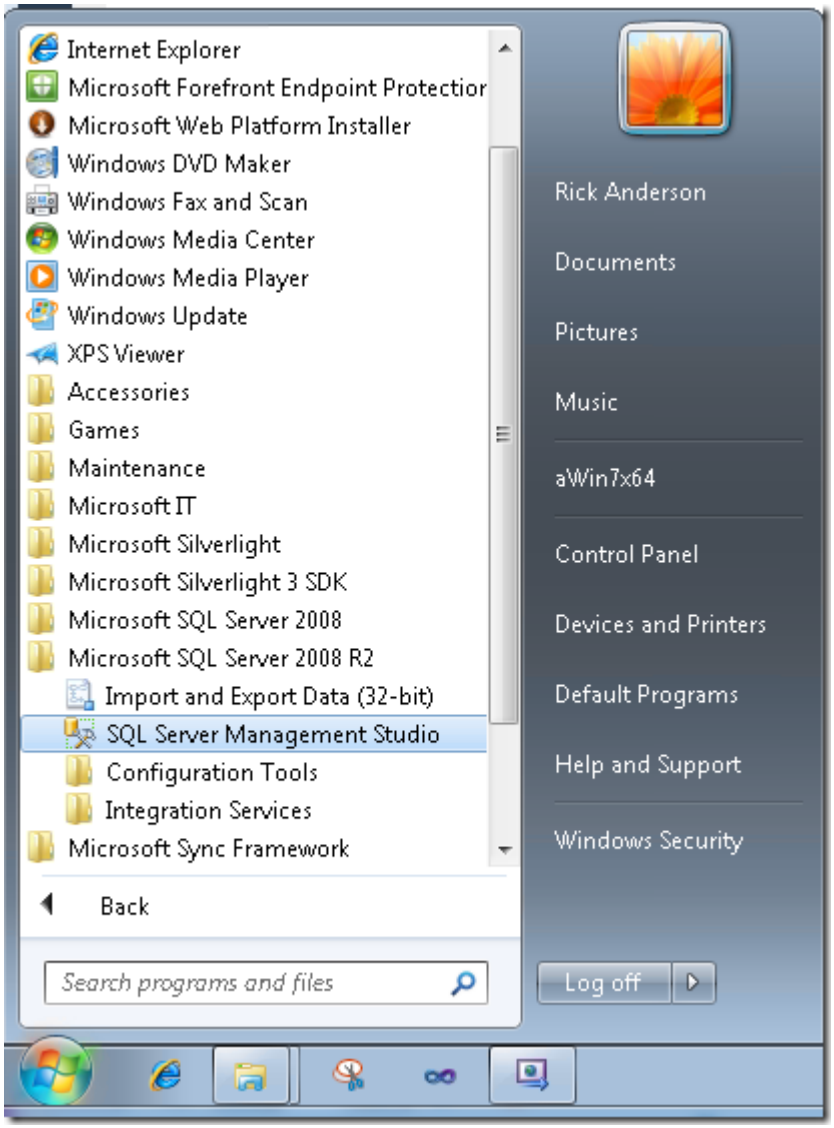
Remember that in the previous part of the tutorial, we created a `Movie` model using the code below:

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

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

    public class MovieDBContext : DbContext
    {
```

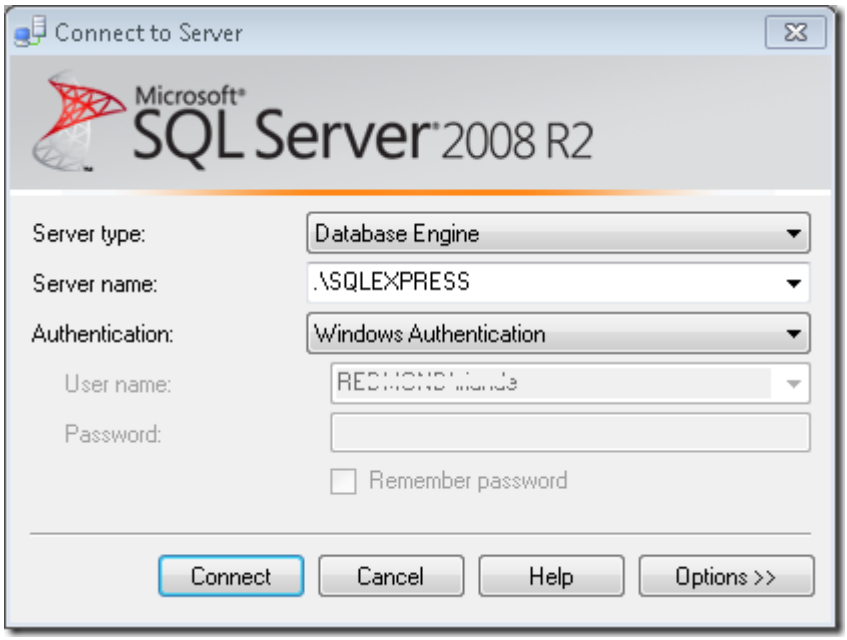
```
        public DbSet<Movie> Movies { get; set; }
    }
}
```

As you just saw, when you first accessed the `MovieDBContext` instance using the `MoviesController` code above, the Entity Framework automatically created an empty `Movies` database for you. It mapped the `Movies` properties of the `MovieDBContext` class to a new `Movies` table that it created in the database. Each row in the table is mapped to a `Movie` instance and each column in the `Movies` table is mapped to a property on the`Movie` class.

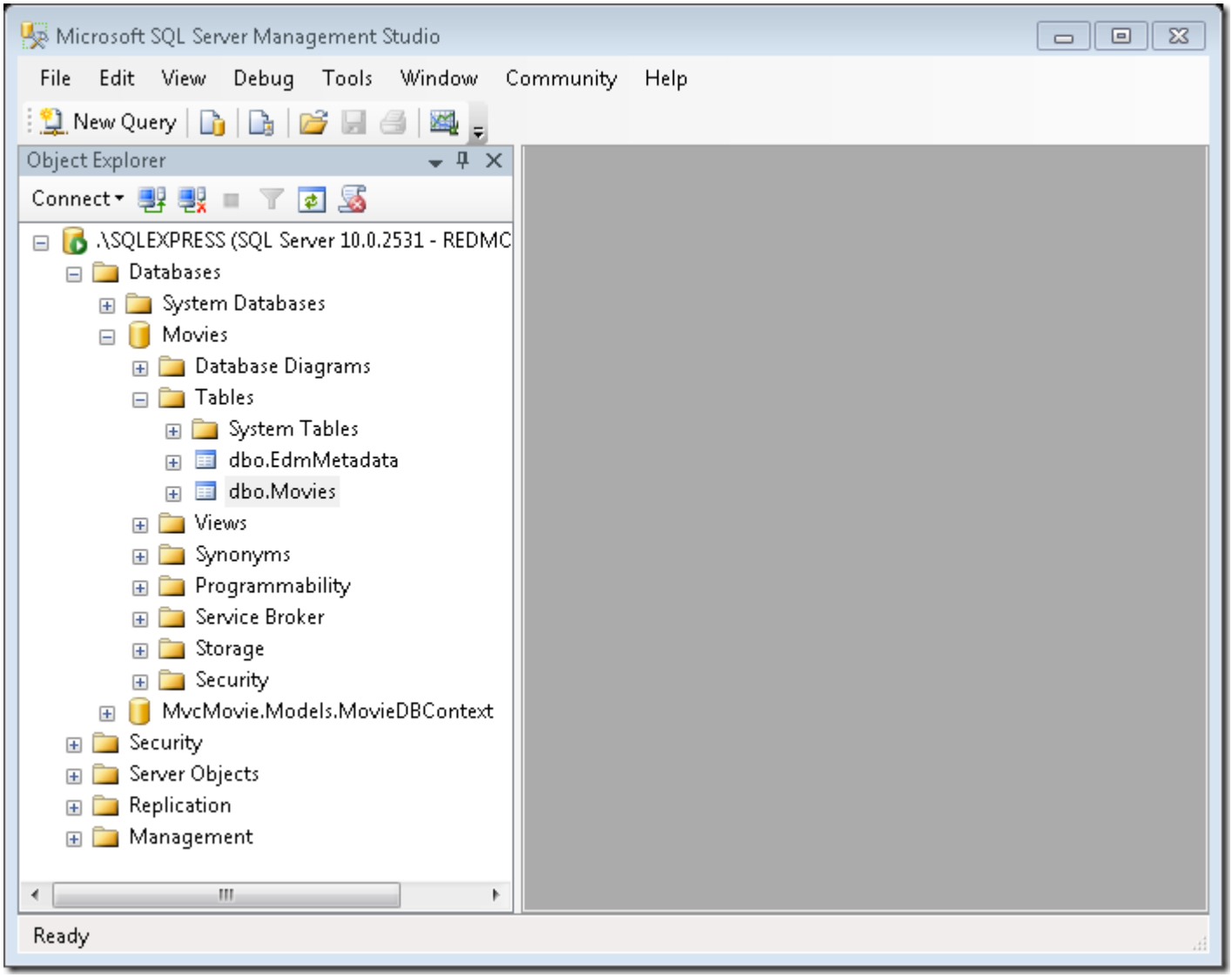
You can use the SQL Server Management Studio tool to see the database schema that was created using the model. Start SQL Server Management Studio.



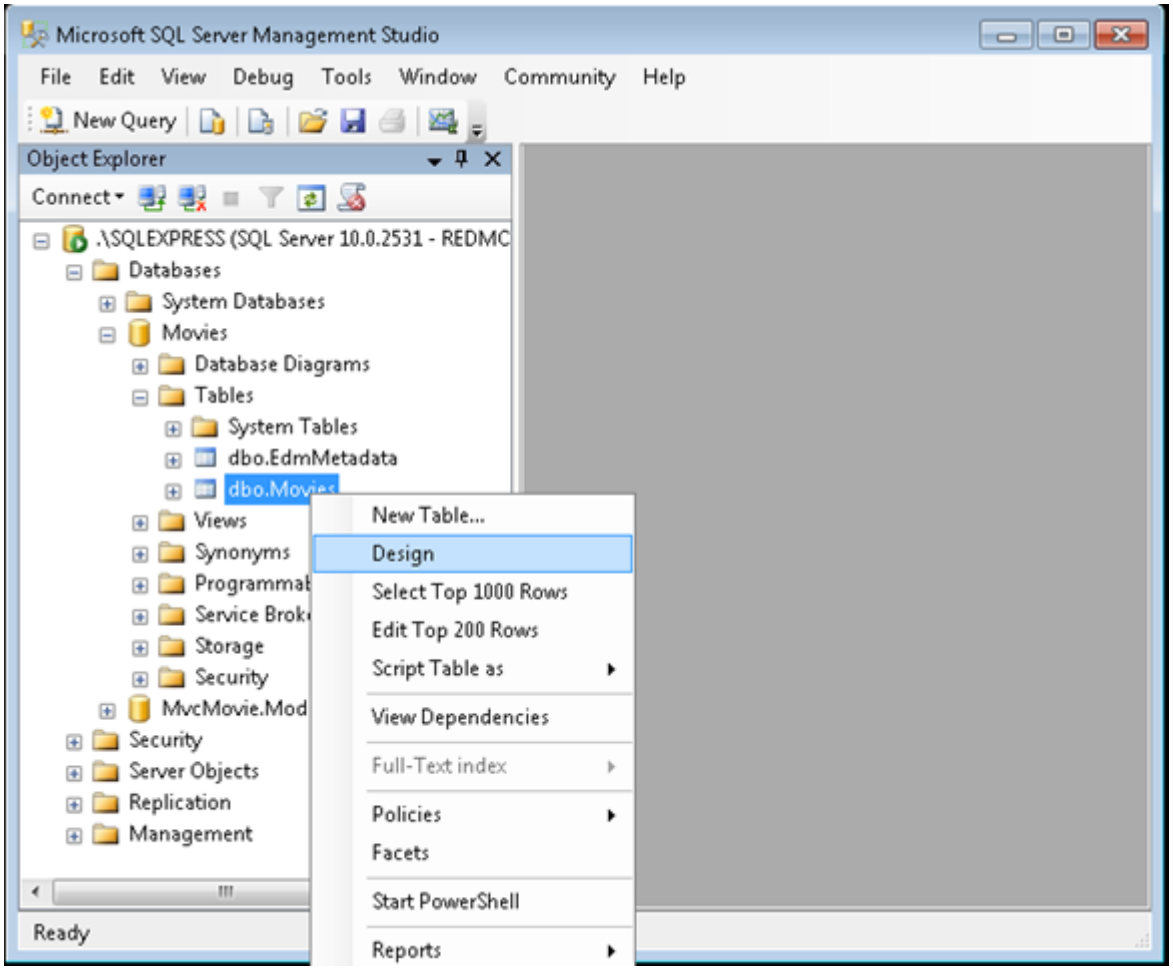
The **Connect to Server** dialog box is displayed. In the **Server name** box, enter the following name: .*SQLEXPRESS*



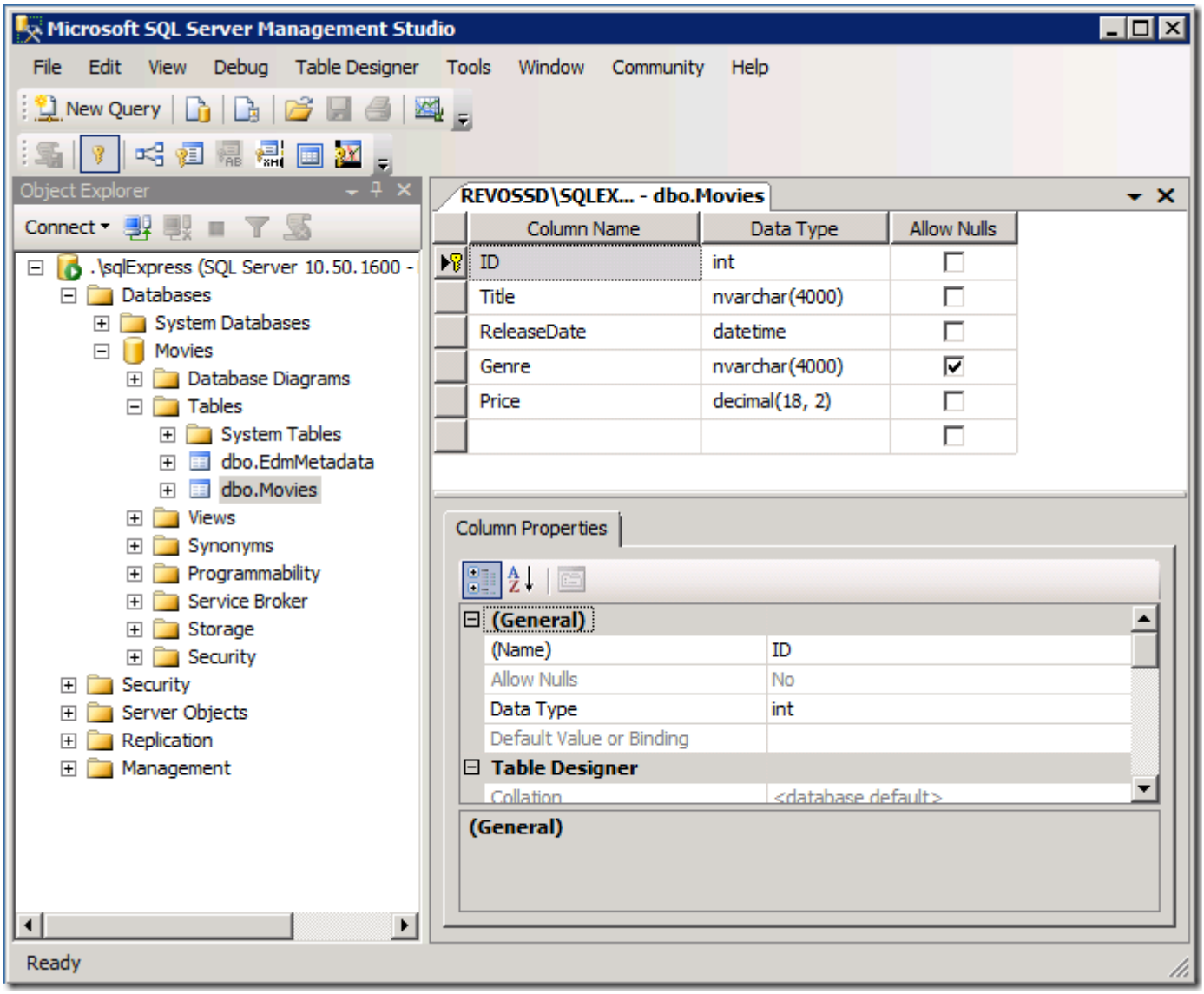
Click **Connect**. The `Movies` database is displayed in the **Object Explorer** pane.



Right-click the `Movies` table and select **Design**.



You see the database schema.



Notice how the schema of the `Movies` database maps to the `Movie` class you created earlier. Entity Framework code-first automatically created this database schema for you based on your `Movie` class.

You now have the database and a simple listing page to display content from it. In the next tutorial, we'll add a `Create` method and a Create view that lets you add movies to this database.

Adding a Create Method and Create View

In this section we are going to add support for creating and saving new movies in the database. We'll implement the /Movies/Create URL to enable this. It will show an HTML <form> with appropriate input elements that a user can fill out to enter a new movie. When a user submits the form, we'll retrieve the values they posted and save them in the database.

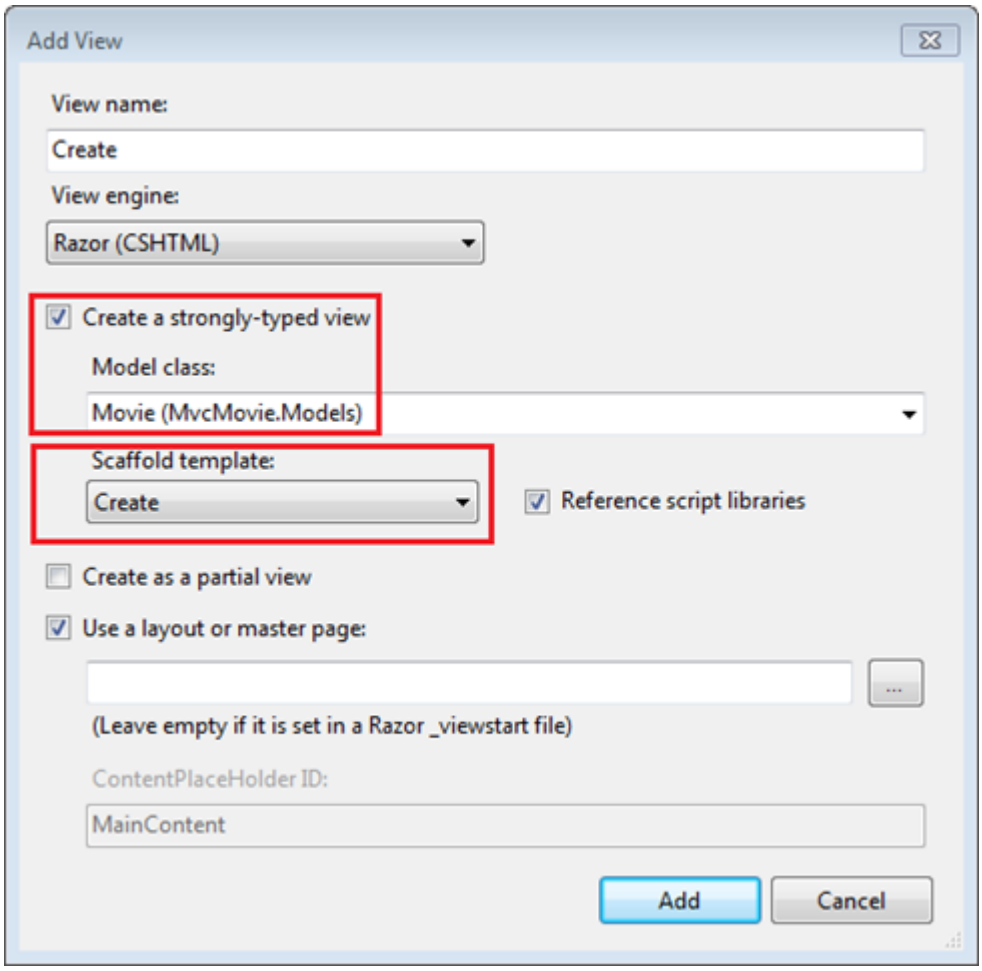
Displaying the Create Form

We'll start by adding a `Create` action method to our existing `MoviesController` class. It will return back a view that contains an HTML form:

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

Now let's implement the Create view that we'll use to display the form to the user. Right-click inside the `Create` method and select **Add View** from the context menu to create the view template for the movie form.

Specify that you're going to pass a `Movie` object to the view template as its model class. In the **Scaffold template** list, choose **Create**, then click **Add**.



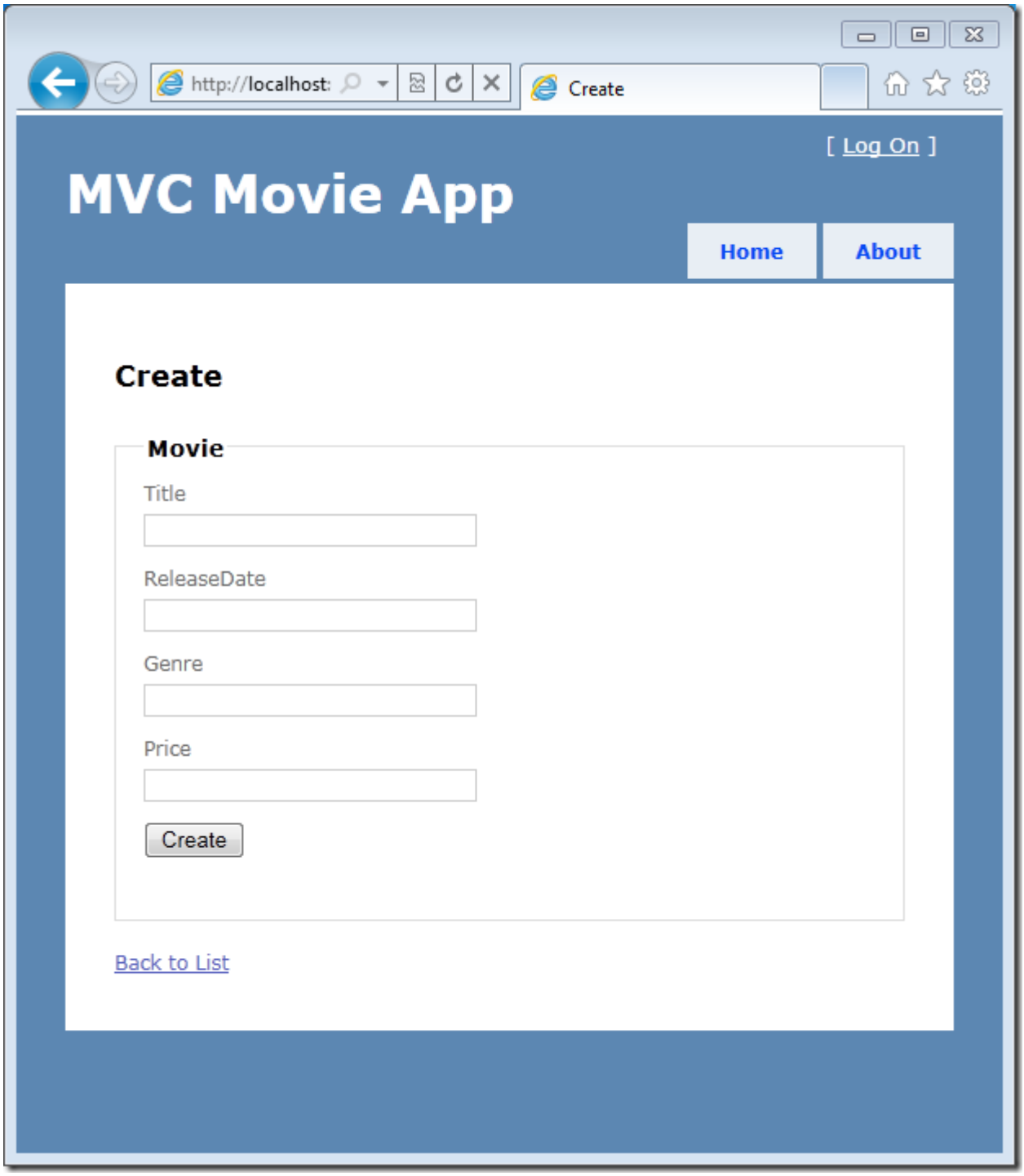
After you click the **Add** button, the *Views**Movies**Create.cshtml* view template is created. Because you selected**Create** in the **Scaffold template** list, Visual Web Developer automatically generated (scaffolded) some default content in the view. The scaffolding created an HTML form and a place for validation error messages. It examined the `Movie` class and created code to render `<label>` and `<input>` elements for each property of the class. The listing below shows theCreate view that was generated:

```
@model MvcMovie.Models.Movie

@{
    ViewBag.Title = "Create";
}

<h2>Create</h2>

<script src="@Url.Content("~/Scripts/jquery.validate.min.js")"
type="text/javascript"></script>
<script src="@Url.Content("~/Scripts/jquery.validate.unobtrusive.min.js")"
type="text/javascript"></script>

@using (Html.BeginForm()) {
    @Html.ValidationSummary(true)
    <fieldset>
        <legend>Movie</legend>

        <div class="editor-label">
            @Html.LabelFor(model => model.Title)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Title)
            @Html.ValidationMessageFor(model => model.Title)
        </div>

        <div class="editor-label">
            @Html.LabelFor(model => model.ReleaseDate)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.ReleaseDate)
            @Html.ValidationMessageFor(model => model.ReleaseDate)
        </div>

        <div class="editor-label">
            @Html.LabelFor(model => model.Genre)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Genre)
            @Html.ValidationMessageFor(model => model.Genre)
        </div>

        <div class="editor-label">
            @Html.LabelFor(model => model.Price)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Price)
            @Html.ValidationMessageFor(model => model.Price)
        </div>

        <p>
            <input type="submit" value="Create" />
        </p>
    </fieldset>
```

```
}

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

The scaffolded code uses several *HTML helper methods* to help streamline the HTML markup. TheHtml.LabelFor helper displays the name of the field ("Title", "ReleaseDate", "Genre", or "Price"). TheHtml.EditorFor helper displays an HTML `<input>` element where the user can enter a value. TheHtml.ValidationMessageFor helper displays any validation messages associated with that property. Notice how our view template has a `@model MvcMovie.Models.Movie` statement at the top of the file – this strongly-types the “Model” of our view template to be a Movie.

Run the application and navigate to the */Movies/Create* URL. You'll see an HTML form like the following:



Right-click within the browser and choose the “View Source” option. The HTML in the page looks like the following (the menu template was excluded for clarity):

```
<!DOCTYPE html>
<html>
<head>
    <title>Create</title>
    <link href="/Content/Site.css" rel="stylesheet" type="text/css" />
    <script src="/Scripts/jquery-1.4.4.min.js" type="text/javascript"></script>
</head>

<body>

<h2>Create</h2>

<script src="/Scripts/jquery.validate.min.js" type="text/javascript"></script>
<script src="/Scripts/jquery.validate.unobtrusive.min.js"
type="text/javascript"></script>

<form action="/Movies/Create" method="post">    <fieldset>
        <legend>Movie</legend>

        <div class="editor-label">
            <label for="Title">Title</label>
        </div>
        <div class="editor-field">
            <input class="text-box single-line" data-val="true" data-val-
required="Title is required" id="Title" name="Title" type="text" value="" />
            <span class="field-validation-valid" data-valmsg-for="Title" data-
valmsg-replace="true"></span>
        </div>

        <div class="editor-label">
            <label for="ReleaseDate">ReleaseDate</label>
        </div>
        <div class="editor-field">
            <input class="text-box single-line" data-val="true" data-val-
required="The ReleaseDate field is required." id="ReleaseDate" name="ReleaseDate"
type="text" value="" />
            <span class="field-validation-valid" data-valmsg-for="ReleaseDate" data-
valmsg-replace="true"></span>
        </div>

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

        <div class="editor-label">
            <label for="Price">Price</label>
        </div>
        <div class="editor-field">
            <input class="text-box single-line" data-val="true" data-val-number="The
field Price must be a number." data-val-range="Price must be between $1 and $100"
data-val-range-max="100" data-val-range-min="1" data-val-required="The Price field
is required." id="Price" name="Price" type="text" value="" />
```

```
            <span class="field-validation-valid" data-valmsg-for="Price" data-
valmsg-replace="true"></span>
        </div>


        <p>
            <input type="submit" value="Create" />
        </p>
    </fieldset>
</form>
<div>
    <a href="/">Back to List</a>
</div>


            <div id="footer">
            </div>
        </div>
    </div>
</body>
</html>
```

The `<input>` elements are in an HTML `<form>` element whose `action` attribute is set to post to the */Movies/Create* URL. The form data will be posted to the server when the **Create** button is clicked.

Processing the HTTP-POST

We've implemented the code necessary to show our create form. Our next step will be to write the code to handle what happens when the form is posted back to the server. We'll want to take the posted values and save them as a new Movie in our database.

To do this, we'll add a second `Create` action method to the MoviesController class. This second `Create` action method will have an `[HttpPost]` attribute on it – indicating that we want to use it to handle POST requests to the `/Movies/Create` URL. All non-POST requests (in effect, GET requests) to the */Movies/Create* URL will instead be handled by the first `Create` action method, which simply displays the empty form.

The following shows the code for both `Create` action methods in the `MoviesController` class:

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

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

Earlier you saw how ASP.NET MVC can automatically pass querystring parameters from a URL (for example, `/HelloWorld/Welcome?name=Scott&numTimes=5`) as method parameters to an action method. In addition to passing querystring parameters, ASP.NET MVC can also pass posted form parameters this way.

Form posted parameters can be passed as individual parameters to an action method. For example, the ASP.NET MVC framework can pass in our form posted values as parameters to the POST Create action method as shown below:

```
 [HttpPost]
 public ActionResult Create(string title, DateTime releaseDate, string genre,
decimal price)
 {
```

The form posted values can also be mapped to a complex object with properties (like our `Movie` class) and passed as a single parameter to an action method. This is the approach we are taking within our `HTTP-POST Create` action method. Notice below how it accepts a single `Movie` object as a parameter:

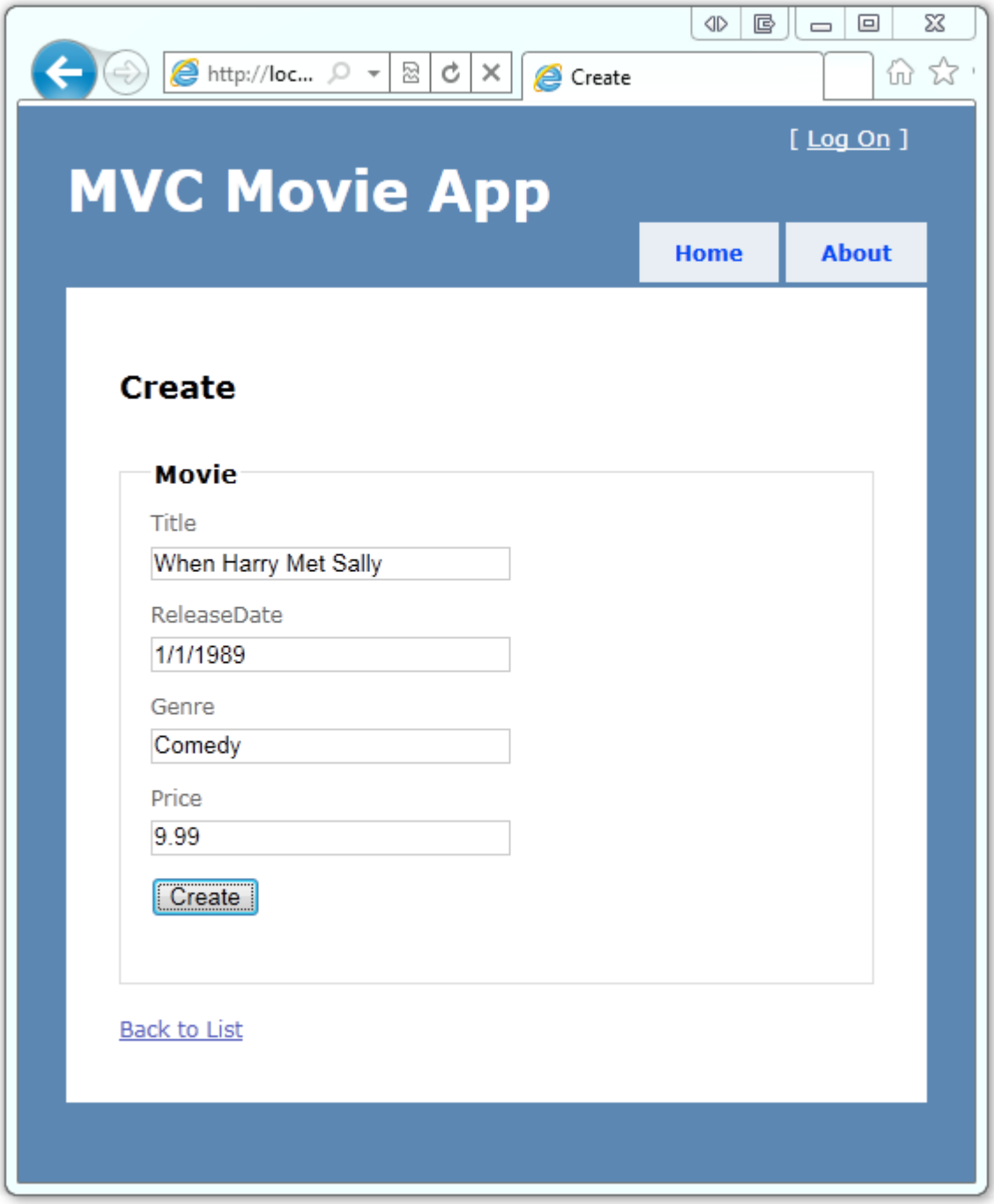
```
[HttpPost]
public ActionResult Create(Movie newMovie)
{
    if (ModelState.IsValid)
    {
        db.Movies.Add(newMovie);
        db.SaveChanges();
        return RedirectToAction("Index");
    }
    else
    {
        return View(newMovie);
    }
}
```

The `ModelState.IsValid` check in the code above verifies that the data submitted in the form can be used to create a `Movie` object . If the data is valid, our code adds the posted Movie to the `Movies` collection of the `MoviesDBContext` instance . The code then saves the new movie to the database by calling the `SaveChanges()` method on our `MoviesDbContext`, which persists changes to the database. After saving the data, the code redirects the user to the `Index` action method of the `MoviesController` class, which causes the new movie to be displayed in the listing of movies.

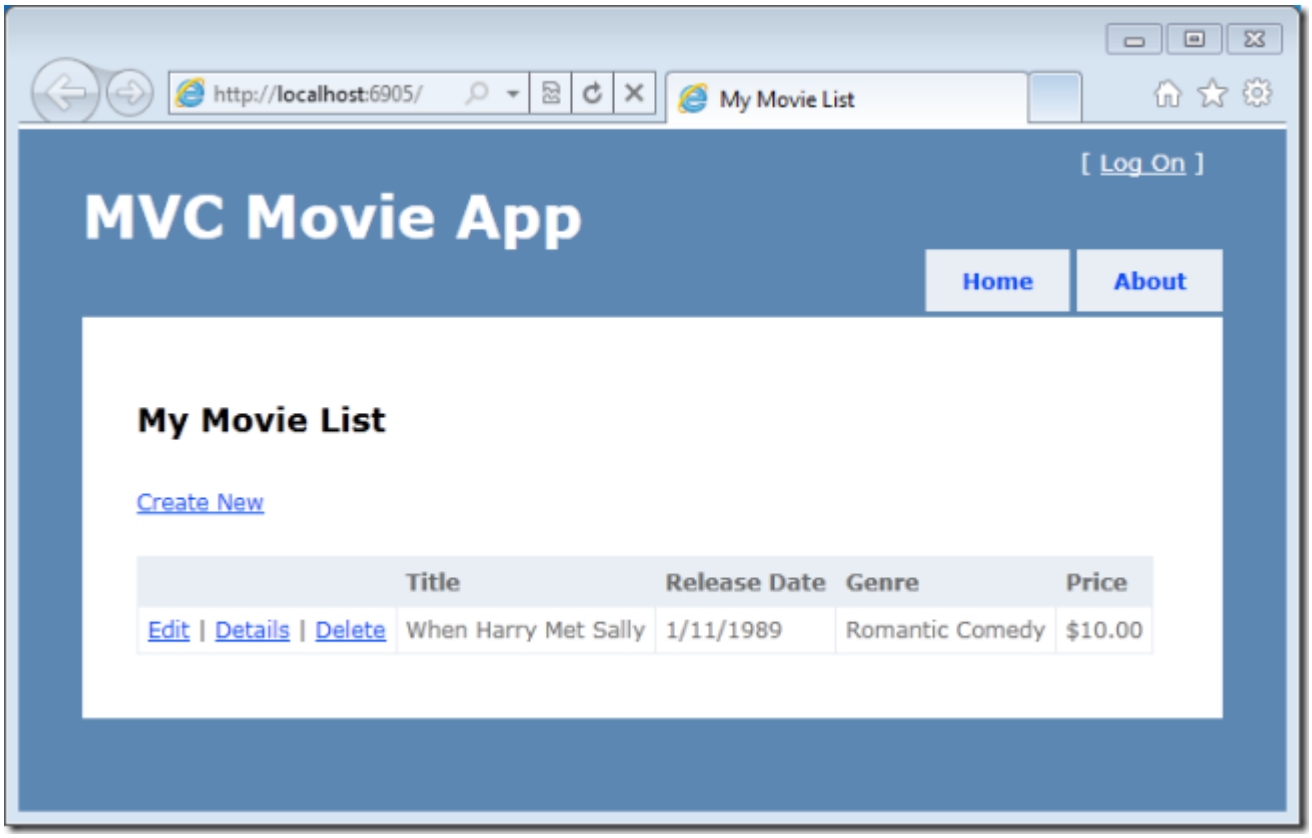
If the posted values are not valid, they are redisplayed in the form. The Html.ValidationMessageFor helpers we are using in the Create.cshtml view template take care of displaying appropriate error messages for any posted values that were not valid.

Creating a Movie

Run the application and navigate to the */Movies/Create* URL. Enter some details about a movie and then click the **Create** button.



Clicking the **Create** button will cause our form to post back to the server, and the movie will be saved in the database. We're then redirected to the */Movies* URL, where we can see the newly created movie in the listing.



You might have noticed that the price displayed in the list is $10, not the $9.99 we entered. That's because the default precision of the `Decimal` type in our database currently doesn't allow decimal point values. We'll fix this when we make some tweaks to the model in the next section.

We now have the beginning of an application that can create and display data from a database. Below is what our MoviesController class looks like:

```
using System;
using System.Linq;
using System.Web.Mvc;
using MvcMovie.Models;

namespace MvcMovie.Controllers
{
    public class MoviesController : Controller
    {
        MovieDBContext db = new MovieDBContext();

        //
        // GET: /Movies/

        public ActionResult Index()
        {
            var movies = from m in db.Movies
                         where m.ReleaseDate > new DateTime(1984, 6, 1)
                         select m;

            return View(movies.ToList());
        }

        //
```

```
        // GET: /Movies/Create

        public ActionResult Create()
        {
            return View();
        }

        //
        // POST: /Movies/Create

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

Note about locales

If you normally work with a locale other than English, you need to include the correct locale-specific jQuery scripts, which you can download from the following URL:

http://plugins.jquery.com/node/8/release

For example, for German locales, you need to download the following file:

jquery.validate_17\jquery-validate\localization\methods_de.js

You then need to include the correct script reference in the *Movies\Create.cshtml* file, as in the following example:

```
<script src="@Url.Content("~/Scripts/methods_de.js")"
type="text/javascript"></script>
```

You don't need to include the localized messages script (*messages_de.js* for German locales), because ASP.NET MVC and the `DataAnnotations` types use their own localized messages.

In addition to including the localized methods script, the current locale on the server must be set to the target locale so that any server-side messages (used for both client-side and server-side validation) will be used correctly.

If you use a non-English locale, you should also use the correct locale-specific character for the decimal delimiter in the price. (In German, for example, you would enter a price as "9,99".) On some non-English keyboards, the dot character (.) does not produce the English dot character used for decimal places; in those cases, if you use a dot character for the decimal point, you might see the following error:

```
The field Price must be a number.
```

In the next section, we will look at how we can add an additional property to our Movie model, and customize the precision of our Price column within the database.

Adding a New Field to the Movie Model and Table

In this section we are going to make some changes to our Model classes and walkthrough how we can evolve the schema of our database to match them.

Adding a Rating Property to our Movie Model

Let's begin by adding an additional "Rating" property to our existing Movie class. Open the *Movie.cs* file and add a `Rating` property to the `Movie` class within it:

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

The complete Movie class should now look like the code below:

```
public class Movie
{
    public int      ID          { get; set; }
    public string   Title       { get; set; }
    public DateTime ReleaseDate { get; set; }
    public string   Genre       { get; set; }
    public decimal  Price       { get; set; }
    public string   Rating      { get; set; }
}
```

Recompile the application using the **Debug->Build Movie** menu command.

Now that we've updated our Model, let's also update our *\Views\Movies\Index.cshtml* and *\Views\Movies\Create.cshtml* view templates to support the new Rating property.

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 should look like after we do this:

```
<table>
    <tr>
        <th></th>
        <th>Title</th>
        <th>Release Date</th>
        <th>Genre</th>
        <th>Price</th>
        <th>Rating</th>
    </tr>
    @foreach (var item in Model) {
    <tr>
        <td>
            @Html.ActionLink("Edit", "Edit", new { id=item.ID }) |
            @Html.ActionLink("Details", "Details", new { id=item.ID }) |
            @Html.ActionLink("Delete", "Delete", new { id=item.ID })
        </td>
        <td>
            @item.Title
        </td>
        <td>
            @String.Format("{0:d}", item.ReleaseDate)
        </td>
        <td>
            @item.Genre
        </td>
```

```
        <td>
            @String.Format("{0:c}", item.Price)
        </td>
        <td>
            @item.Rating
        </td>
    </tr>
}
</table>
```

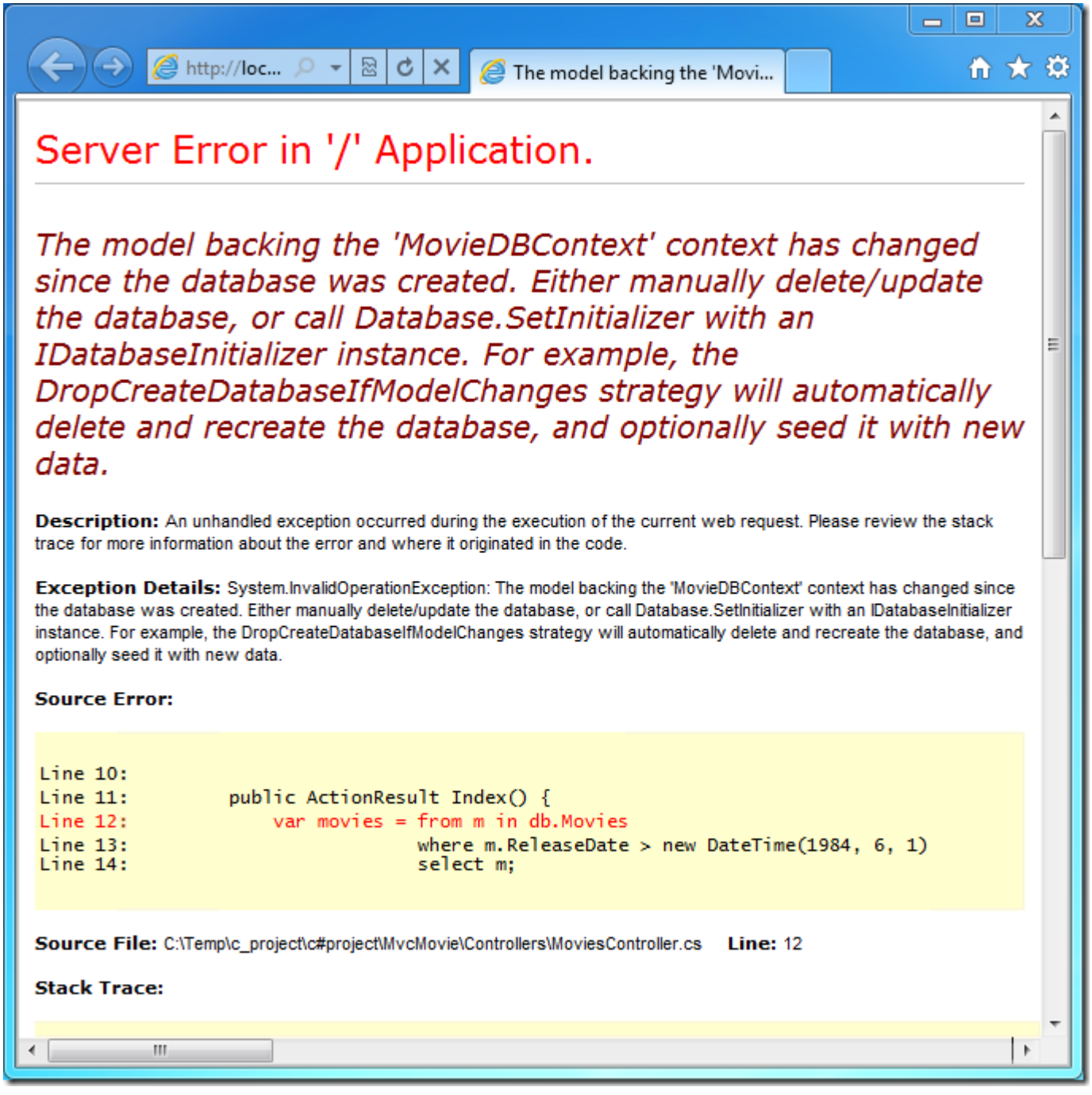
Next open the *Views**Movies**Create.cshtml* file and add the below markup near the end of the form. It will render a textbox so that a Rating can be specified when a new Movie is created:

```
<div class="editor-label">
    @Html.LabelFor(model => model.Rating)
</div>
<div class="editor-field">
    @Html.EditorFor(model => model.Rating)
    @Html.ValidationMessageFor(model => model.Rating)
</div>
```

Managing Model / Database Schema Differences

We've updated our application code to support the new Rating property.

Let's now re-run the application and navigate to the **/Movies** URL. When we do this, though, we'll find that the following error occurs:



We are seeing this error because the updated Movie model class within our application is now different than the schema of the Movie table of our existing database (there is no corresponding "Rating" column in the database table).

By default, when you use EF code-first to automatically create a database (like we did earlier in this tutorial), EF 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 it's not in sync, EF will throw 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. The sync checking feature is what causes the above error message to be displayed.

There are two approaches to resolving the above 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 when doing active development on a test database, as it allows you to quickly evolve your Model and database schema together. The downside, though, is that you lose existing data in the database (and so you ***don't*** want to use it on a production database!).

2. 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.

For this tutorial, we'll use the first approach – and have EF code-first automatically re-create the database anytime the model changes.

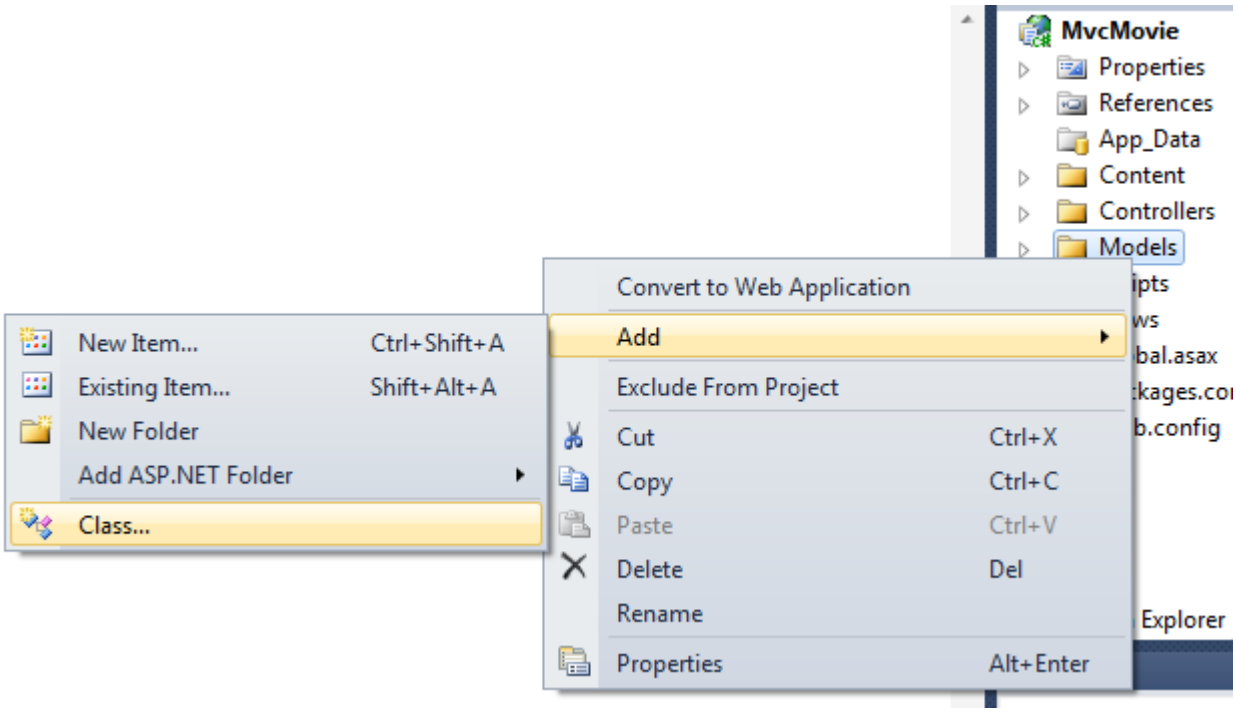
Automatically Recreate the Database on Model Changes

Let's update our application so that EF code-first automatically drops and re-creates our database anytime we evolve the model of our application.

Warning

: You should only enable this approach of automatically dropping and re-creating the database using a development/test database, and never on a production database with real data. Using it on a production server can lead to data loss.

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



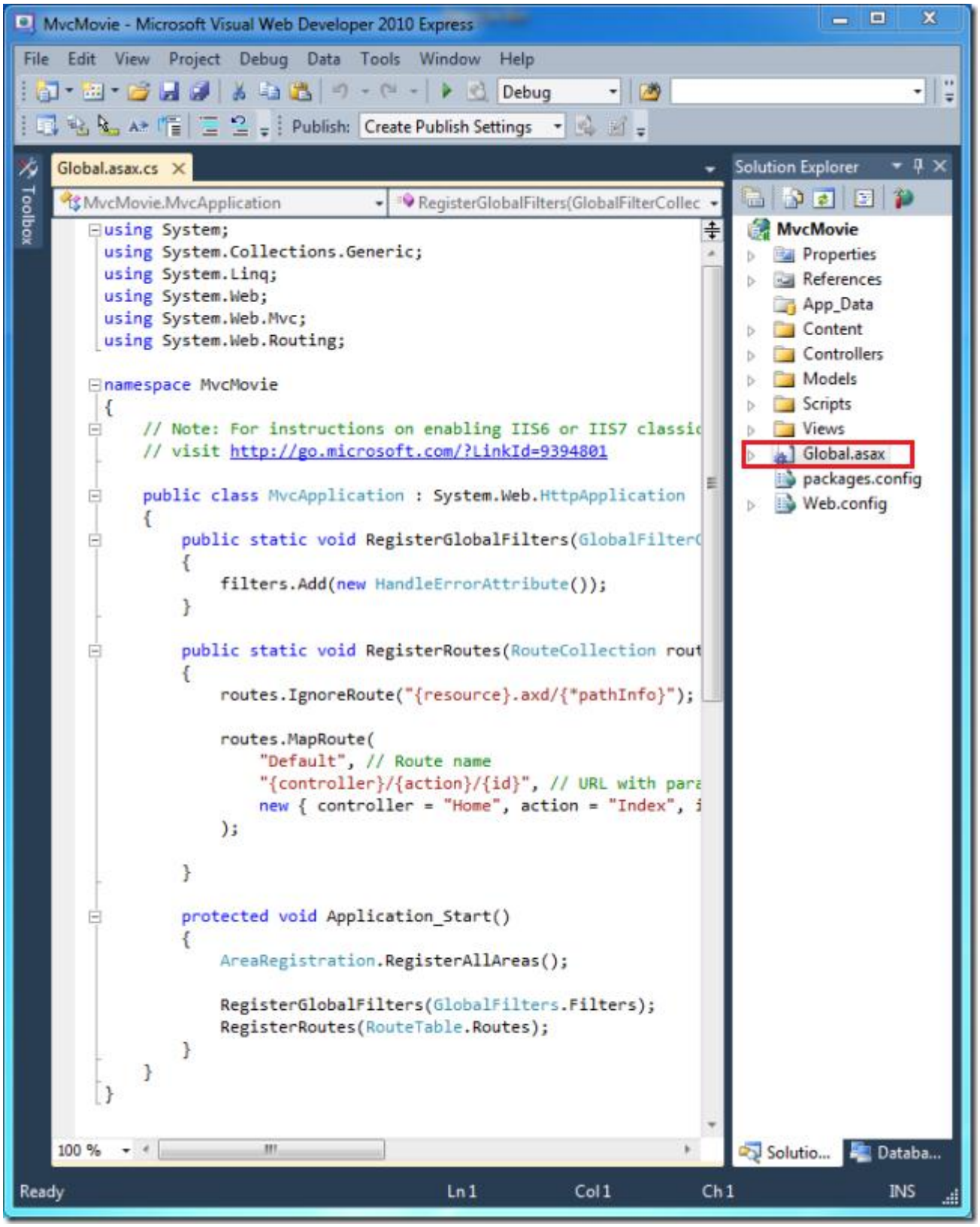
Name the class "MovieIntializer". Update the `MovieIntializer` class to contain the following code:

```
using System;
using System.Collections.Generic;
using System.Data.Entity.Database;

namespace MvcMovie.Models
{
    public class MovieInitializer : DropCreateDatabaseIfModelChanges<MovieDBContext>
    {
        protected override void Seed(MovieDBContext context)
        {
            var movies = new List<Movie> {

                 new Movie { Title = "When Harry Met Sally",
                             ReleaseDate=DateTime.Parse("1989-1-11"),
                             Genre="Romantic Comedy",
                             Rating="R",
                             Price=7.00M},

                 new Movie { Title = "Ghostbusters 2",
                             ReleaseDate=DateTime.Parse("1986-2-23"),
                             Genre="Comedy",
                             Rating="R",
                             Price=9.00M},
              };

            movies.ForEach(d => context.Movies.Add(d));
        }
    }
}
```

The `MovieInitializer` class above indicates that the database used by our Model should be dropped and automatically recreated if our Model classes ever change. We are using its “Seed” method to specify some default data that we want to automatically add to the database any time it is created (or re-created). This provides a useful way to populate some sample data into our database, without requiring us to manually populate it each time we make a database change.

Now that we’ve defined our `MovieInitializer` class, we’ll want to wire it up so that each time our application runs it checks to see whether our Model classes are different than the schema in our the database, and if so re-creates it to match (and then populates it with the sample seed data).

Open the Global.asax file located at the root of the MvcMovies project:



The Global.asax file contains the "Application Class" of our project, and contains an `Application_Start()` event handler that will run when our application first starts up.

Let's add two using statements to the top of the file. The first references the Entity Framework namespace, and the second references the namespace where our `MovieInitializer` class lives:

```
using System.Data.Entity.Database;  // DbDatabase.SetInitialize
using MvcMovie.Models;              // MovieInitializer
```

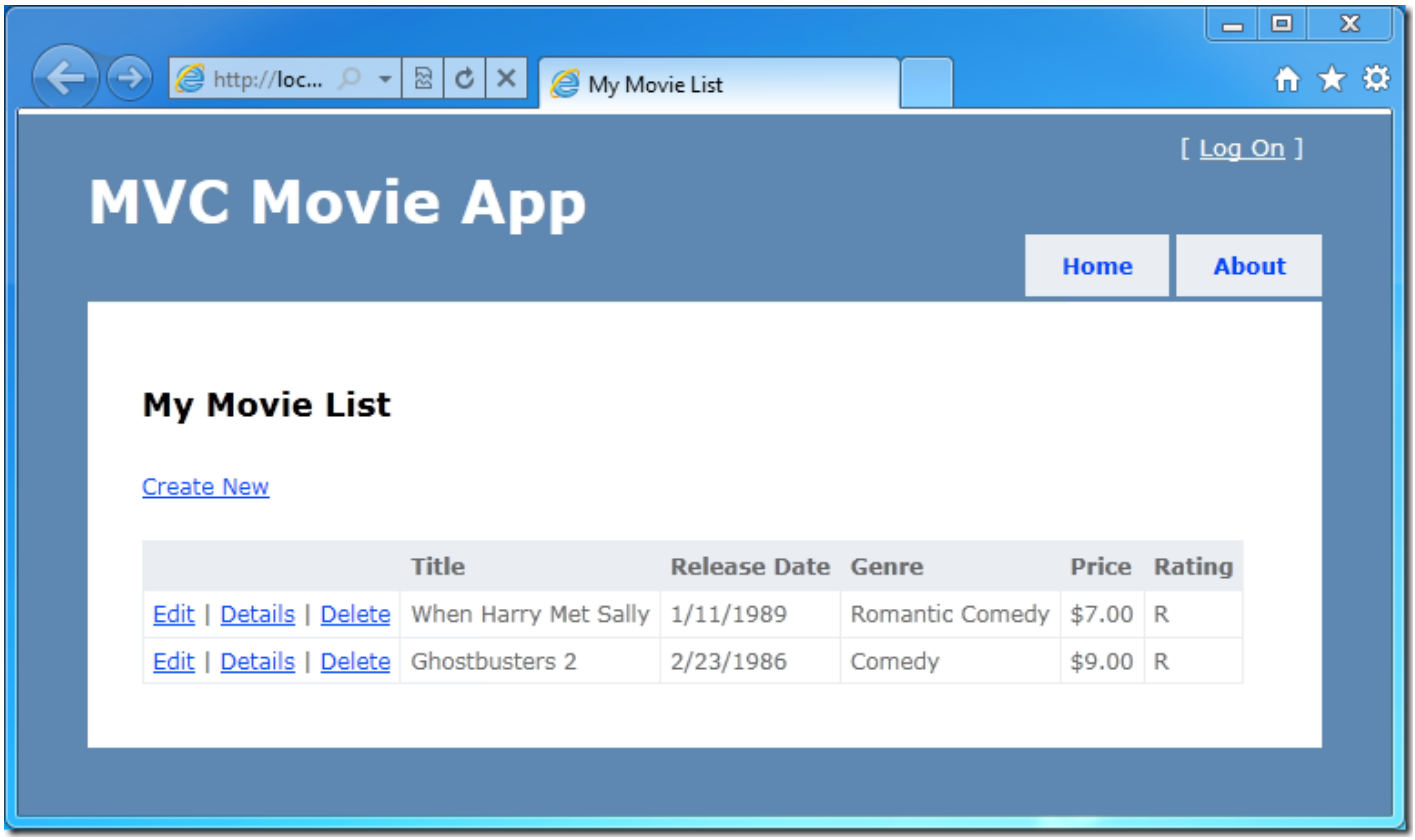
Then find the `Application_Start` method and add a call to `DbDatabase.SetInitializer()` at the beginning of the method as shown below:

```
protected void Application_Start()
{
    DbDatabase.SetInitializer<MovieDBContext>(new MovieInitializer());

    AreaRegistration.RegisterAllAreas();
    RegisterGlobalFilters(GlobalFilters.Filters);
    RegisterRoutes(RouteTable.Routes);
}
```

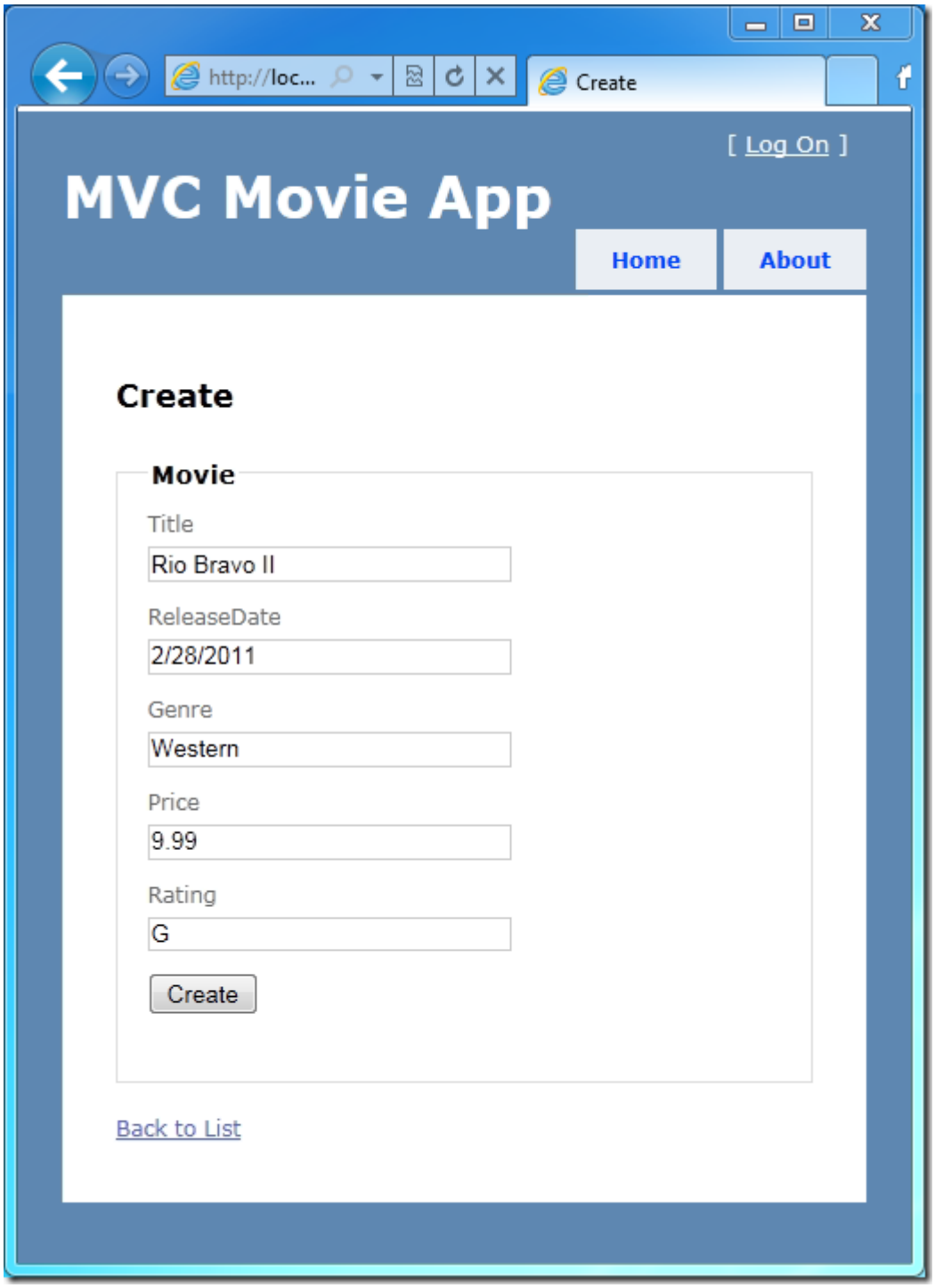
The `DbDatabase.SetInitializer` statement we just added indicates that the database used by our `MovieDBContext` should be automatically deleted and recreated if the schema in the database does not match the current state of our Movie model objects. It will then populate the database with the "seed" sample data specified within our `MovieInitializer` class.

Close the Global.asax file.

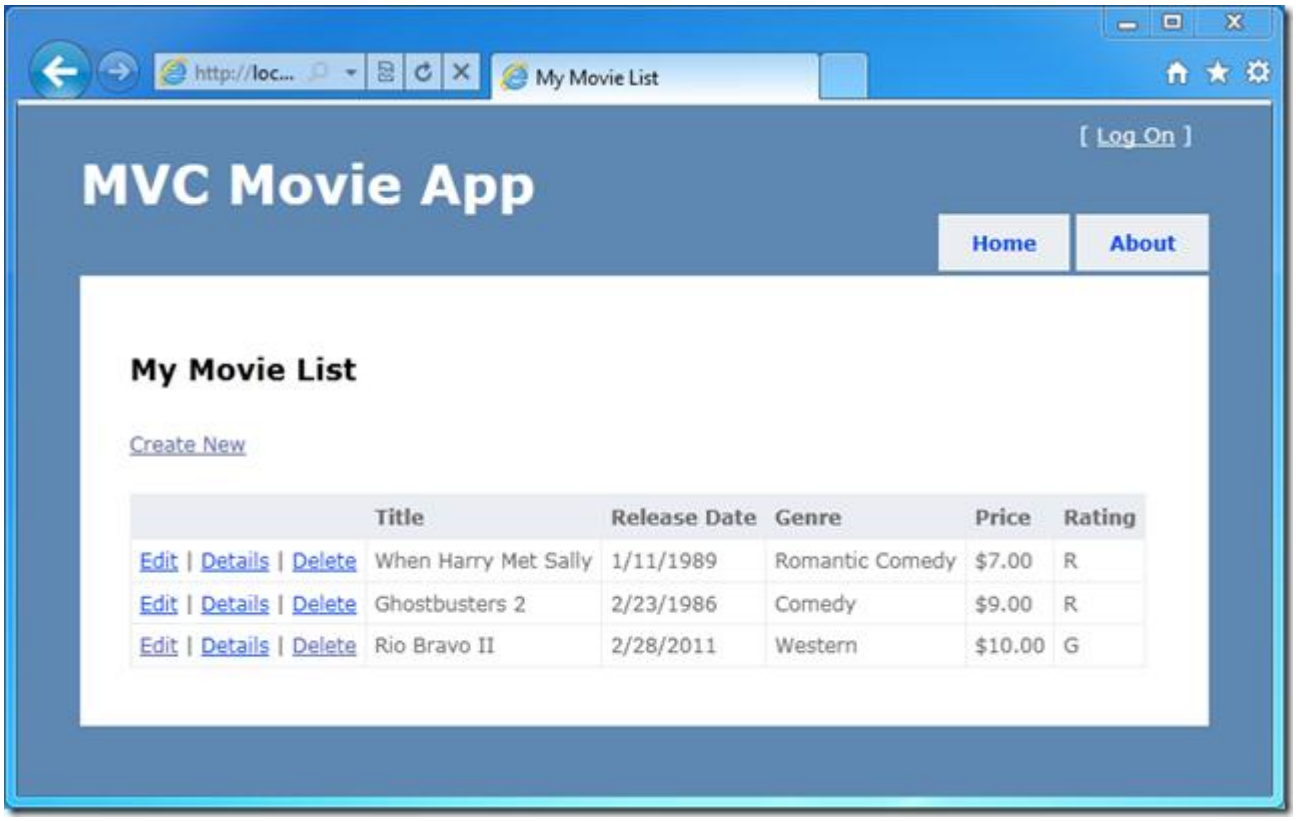
Let's now re-run our application again and navigate to the **/Movies** URL. When the application starts up, it will detect that our Model structure no longer matches the schema of our database, and automatically recreates the database to match the new Model structure. It will then populate the database with the two sample Movies we specified as "seed" data:



Click the “Create” link to add a new Movie:



Our new Movie, complete with Rating, will now show up in the Movies listing:



Fixing the Precision of our Price

In the screen-shots above you might have noticed that we have an issue with our Price column. We entered $9.99 as the price in our Create form – and yet it is showing up as $10.00 in our movie listing page. Why is that?

This is happening because when EF code-first created our database, it used a default precision setting of (18:0) when creating columns for Decimal data-types. This causes a value of $9.99 to be rounded up to $10. We'll want to change this so that we instead store at least two decimal places (18:2). The good news is that EF code-first allows you to easily override the mapping rules for how Models are persisted and loaded from a database. You can use this mechanism to override the default typing conventions and table inheritance rules used by EF code-first, and save data however you want within a database.

To change the precision of how our Price column is persisted in a database, open up the *Movie.cs* file within the *\Models* folder of the project. Add a using statement for `System.Data.Entity.ModelConfiguration`.

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

Add the following `OnModelCreating` override method to our existing `MovieDBContext` class:

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

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Movie>().Property(p => p.Price).HasPrecision(18, 2);
        }
    }
```

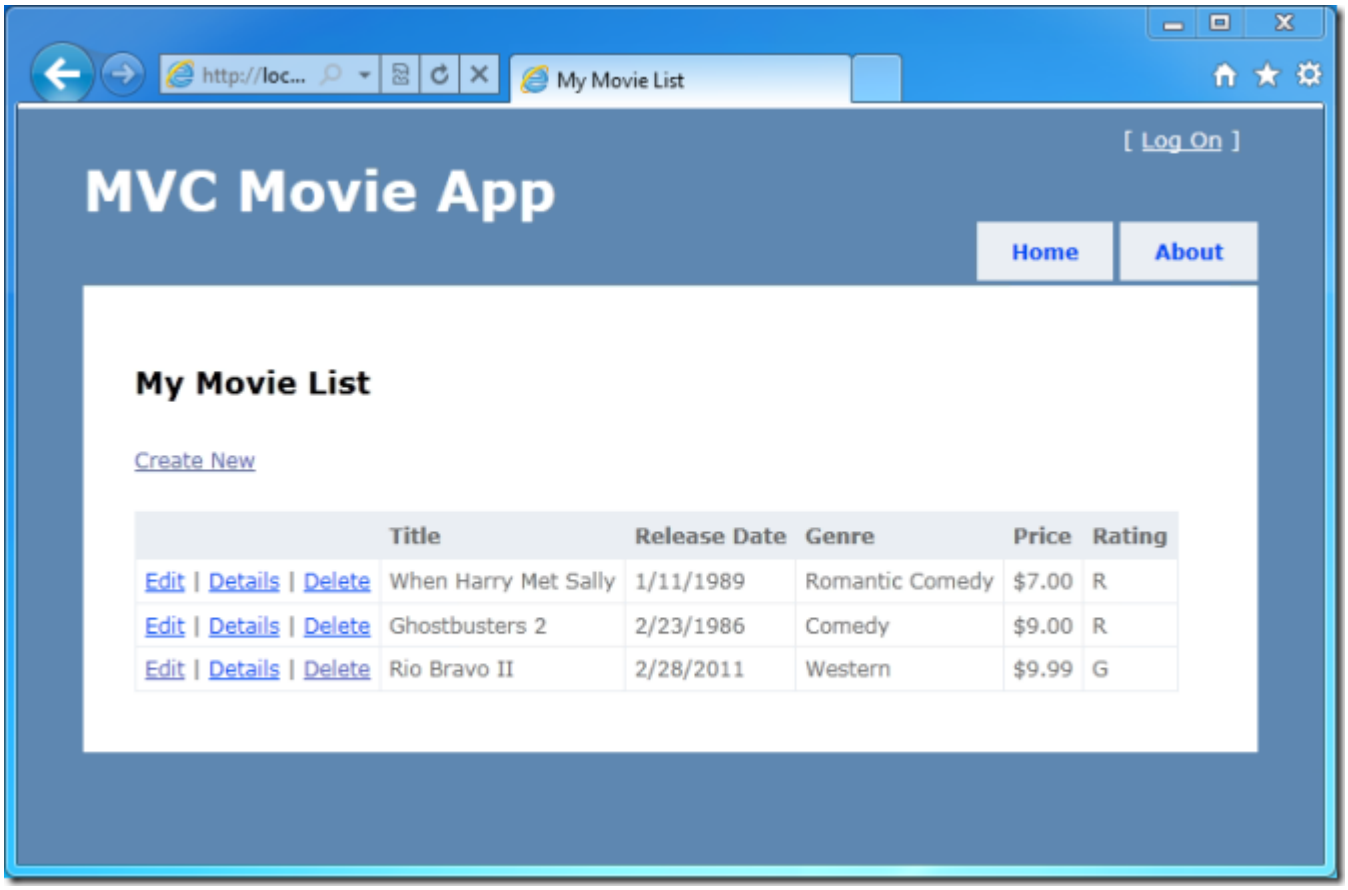
The `OnModelCreating`() method can be used to override/customize the mapping rules for how our Model classes are mapped to/from our database. The code above uses EF's `ModelBuilder` API to indicate that we want the`Price` property of our Movie objects to have a precision of two decimal places when persisted in the database.

The full code listing for the *Movie.cs* file is shown below:

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

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

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

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Movie>().Property(p => p.Price).HasPrecision(18, 2);
        }
    }
}
```

Let's now re-run our application and navigate again to the **/Movies** URL. When the application starts up, it will detect once again that our Model structure no longer matches the schema of our database, and automatically recreate the database to match the new Model structure (and have the new Price precision).

Create a new Movie and enter a price of 9.99. Notice how the decimal price is now correctly persisted in the database and shows up in our movie listing:



In this section we showed how you can quickly modify your Model objects, and keep your database in sync with the changes. We also showed how you can pre-populate your newly created databases with sample data to allow you to quickly try out and test scenarios. Next, let's look at how we can add richer validation logic to our Model classes, and enable some business rules to be enforced.

Adding Validation to the Model

In this section we'll add validation logic to our `Movie` model, and we'll ensure that the validation rules are enforced any time a user attempts to create or edit a movie using our application.

Keeping Things DRY

One of the core design tenets of ASP.NET MVC is DRY ("'Don't Repeat Yourself"). ASP.NET MVC encourages you to specify functionality or behavior only once, and then have it be reflected everywhere in an application. This reduces the amount of code you need to write and makes the code you do write much easier to maintain.

The validation support provided by ASP.NET MVC and EF code-first is a great example of the DRY principle in action. You can declaratively specify validation rules in one place (the model class) and then those rules are enforced *everywhere within the application*.

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

Adding Validation Rules to the Movie Model

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

Open the *Movie.cs* file. Add a `using` statement at the top of the file that references the System.ComponentModel.DataAnnotations namespace:

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

The namespace is part of the .NET Framework. It provides a built-in set of validation attributes that you can apply declaratively to any class or property.

Let's update the `Movie` class to take advantage of the built-in Required, StringLength, and Range validation attributes using the code below:

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

    [Required(ErrorMessage = "Title is required")]
    public string Title { get; set; }

    [Required(ErrorMessage = "Date is required")]
    public DateTime ReleaseDate { get; set; }

    [Required(ErrorMessage = "Genre must be specified")]
    public string Genre { get; set; }

    [Required(ErrorMessage = "Price Required")]
    [Range(1, 100, ErrorMessage = "Price must be between $1 and $100")]
    public decimal Price { get; set; }

    [StringLength(5)]
    public string Rating { get; set; }
}
```

The validation attributes above specify behavior we want enforced on the model properties they are applied to. The `Required` attribute indicates that a property must have a value; in this sample, a movie has to have a Title, ReleaseDate, Genre, and Price in order to be valid. The `Range` attribute constrains a value to within a specified range. The `StringLength` attribute lets you set the maximum length of a stringproperty, and optionally its minimum length.

EF code-first ensures that the validation rules you specify on a model class are enforced before allowing the application to save changes in the database. For example, the code below will throw an exception when the `SaveChanges` method is called, because several required `Movie` property values are missing and the price is zero (which is out of the valid range).

```
MovieDBContext db = new MovieDBContext();

Movie movie = new Movie();
movie.Title = "Gone with the Wind";
movie.Price = 0.0M;

db.Movies.Add(movie);
db.SaveChanges();        // <= Will throw validation exception
```

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

Here's a complete code listing for the updated *Movie.cs* file:

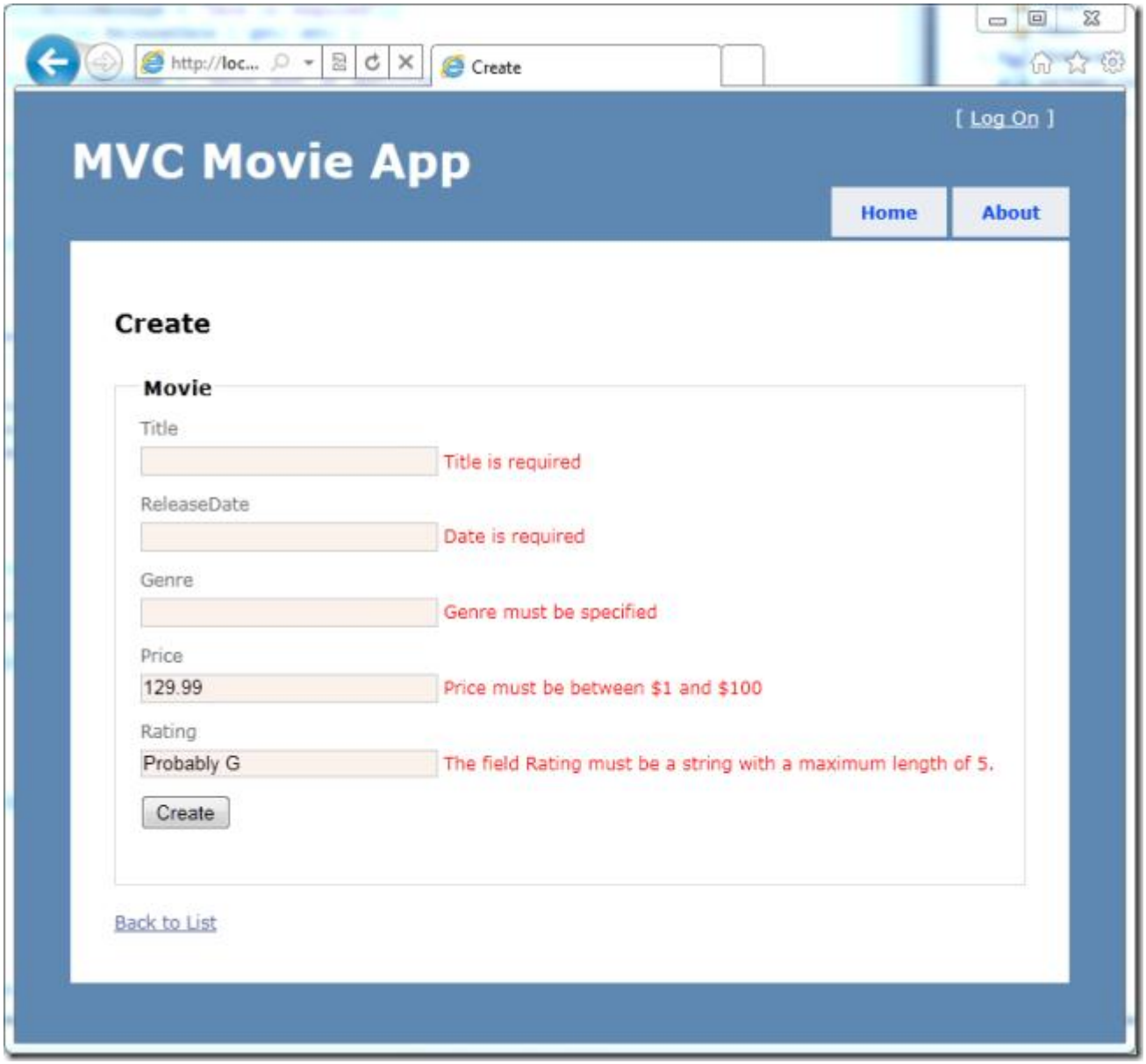
```
using System;
using System.ComponentModel.DataAnnotations;
using System.Data.Entity;
using System.Data.Entity.ModelConfiguration;

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

       [Required(ErrorMessage = "Title is required")]
       public string Title { get; set; }

       [Required(ErrorMessage = "Date is required")]
       public DateTime ReleaseDate { get; set; }

       [Required(ErrorMessage = "Genre must be specified")]
       public string Genre { get; set; }

       [Required(ErrorMessage = "Price Required")]
       [Range(1, 100, ErrorMessage = "Price must be between $1 and $100")]
       public decimal Price { get; set; }

       [StringLength(5)]
       public string Rating { get; set; }
    }

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


        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
```

```
            modelBuilder.Entity<Movie>().Property(p => p.Price).HasPrecision(18, 2);
        }
    }
}
```

Validation Error UI within ASP.NET MVC

Let's now re-run our application and navigate to the */Movies* URL.

Click the **Create Movie** link to add a new movie. Fill out the form with some invalid values and then click the **Create** button:



Notice how the form has automatically used a background color to highlight the text boxes that contain invalid data and has emitted an appropriate validation error message next to each one. The error messages match the error message strings we specified when we annotated the `Movie` class earlier in this tutorial. The errors are enforced both client-side (using JavaScript) and server-side (in case a user has JavaScript disabled).

What's really nice is that we 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 we created earlier in this tutorial automatically picked up the validation rules that we specified on the `Movie` model class.

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 our controller or views. The next listing shows what the `Create` methods in the `MovieController` class look like. They're unchanged from how we created them earlier in this tutorial.

```
//
// GET: /Movies/Create

public ActionResult Create()
{
    return View();
}

//
// POST: /Movies/Create

[HttpPost]
public ActionResult Create(Movie newMovie)
{
    if (ModelState.IsValid)
    {
        db.Movies.Add(newMovie);
        db.SaveChanges();

        return RedirectToAction("Index");
    }
    else
    {
        return View(newMovie);
    }
}
```

The first action method displays the initial Create form. The second handles the form post. The second `Create` method 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 redisplays the form. If there are no errors, the method saves the new movie in the database.

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

```
@model MvcMovie.Models.Movie
@{
    ViewBag.Title = "Create";
}
<h2>
    Create</h2>
<script src="@Url.Content("~/Scripts/jquery.validate.min.js")"
type="text/javascript"></script>
<script src="@Url.Content("~/Scripts/jquery.validate.unobtrusive.min.js")"
type="text/javascript"></script>
@using (Html.BeginForm())
{
```

```
    @Html.ValidationSummary(true)
    <fieldset>
        <legend>Movie</legend>
        <div class="editor-label">
            @Html.LabelFor(model => model.Title)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Title)
            @Html.ValidationMessageFor(model => model.Title)
        </div>
        <div class="editor-label">
            @Html.LabelFor(model => model.ReleaseDate)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.ReleaseDate)
            @Html.ValidationMessageFor(model => model.ReleaseDate)
        </div>
        <div class="editor-label">
            @Html.LabelFor(model => model.Genre)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Genre)
            @Html.ValidationMessageFor(model => model.Genre)
        </div>
        <div class="editor-label">
            @Html.LabelFor(model => model.Price)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Price)
            @Html.ValidationMessageFor(model => model.Price)
        </div>
        <div class="editor-label">
            @Html.LabelFor(model => model.Rating)
        </div>
        <div class="editor-field">
            @Html.EditorFor(model => model.Rating)
            @Html.ValidationMessageFor(model => model.Rating)
        </div>
        <p>
            <input type="submit" value="Create" />
        </p>
    </fieldset>
}
<div>
    @Html.ActionLink("Back to List", "Index")
</div>
```

Notice how we're using an `Html.EditorFor` helper to output the `<input>` element for each `Movie` property. Next to this helper is a call to the `Html.ValidationMessageFor` helper method. These two helper methods work with the model object that is 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 know 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.

If we want to change the validation logic later, we can do so in exactly one place. We won't have to worry about different parts of our application being inconsistent with how the rules are enforced — all validation logic will be defined in one place

and used everywhere. This keeps our code very clean. It means the code is easy to maintain and evolve. And it means that we fully honor the DRY principle.

Next, let's look at how we can finish up the application by enabling the ability to edit and delete existing movies, as well as display details for individual ones.

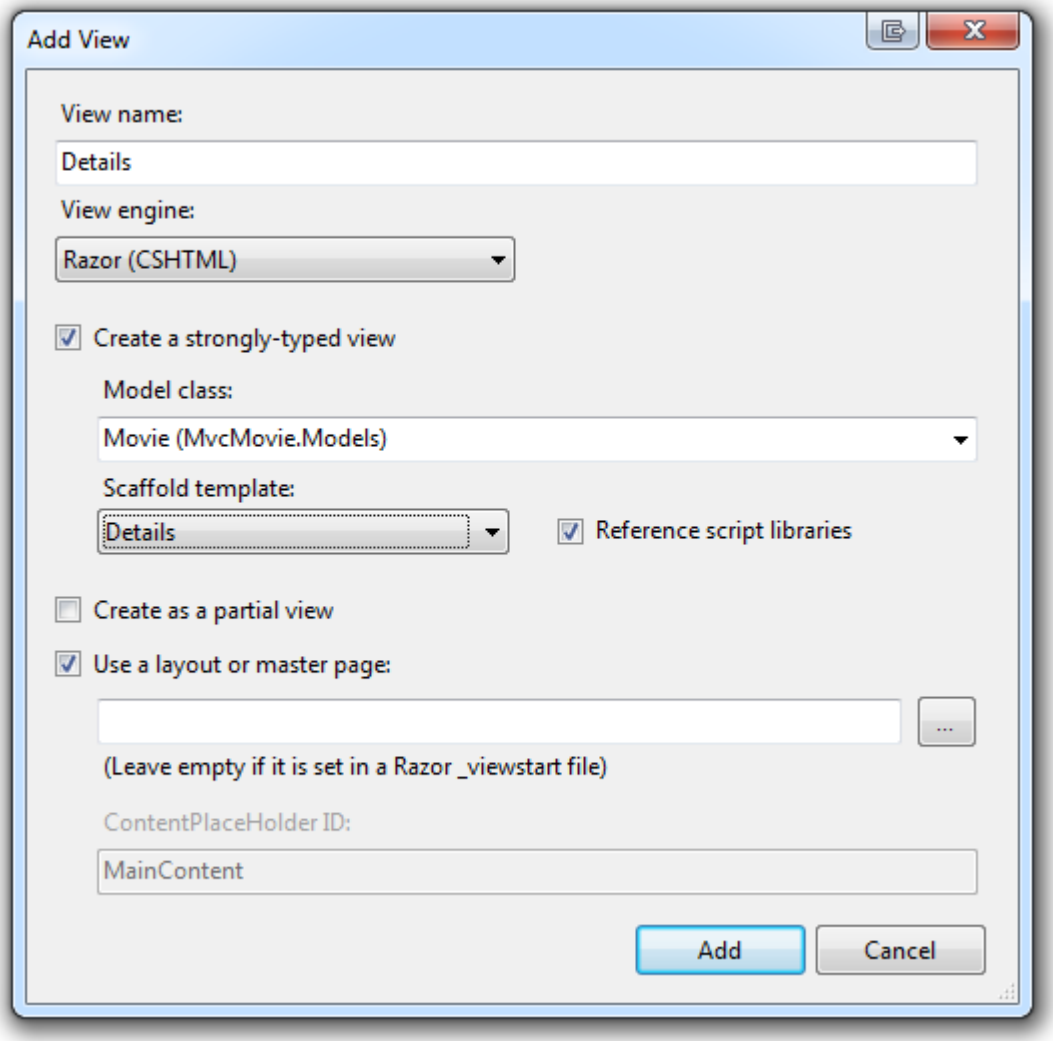
Implementing Edit, Details, and Delete Views

Open the `Movie` controller and add the following `Details` method:

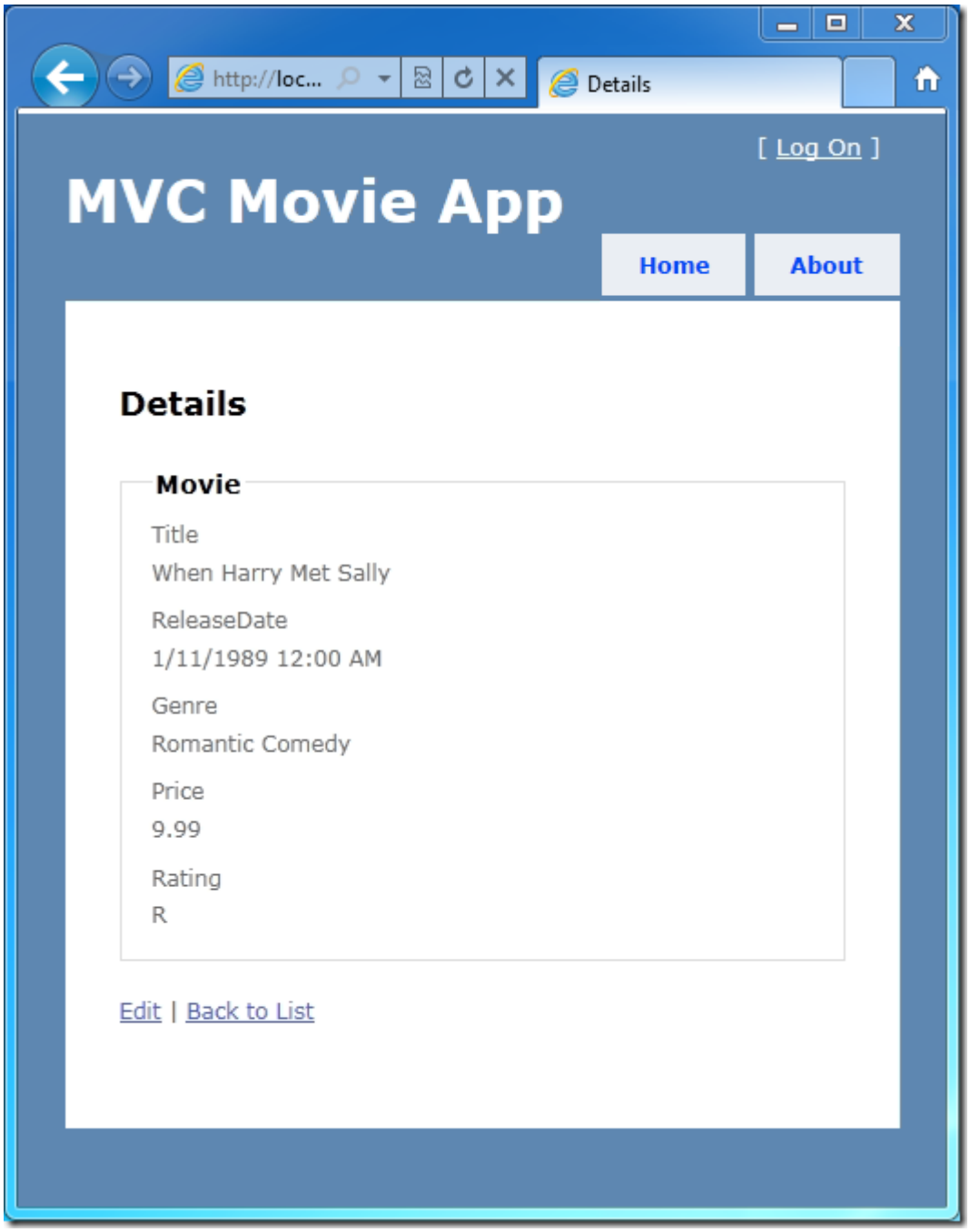
```
//
// GET: /Movies/Details

public ActionResult Details(int id)
{
    Movie movie = db.Movies.Find(id);
    if (movie == null)
        return RedirectToAction("Index");
    return View("Details", movie);
}
```

The code-first approach makes it easy to search for data using the `Find` method. An important security feature of this method is that we actually verify that we found a movie. 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 *http://localhost:xxxx/Movies/Details/12345*. Without the check for a null movie, this could result in a database error.

Right-click inside the `Details` method and select **Add View**. For **Scaffold template**, choose **Details**.



Run the application and select a **Details** link.



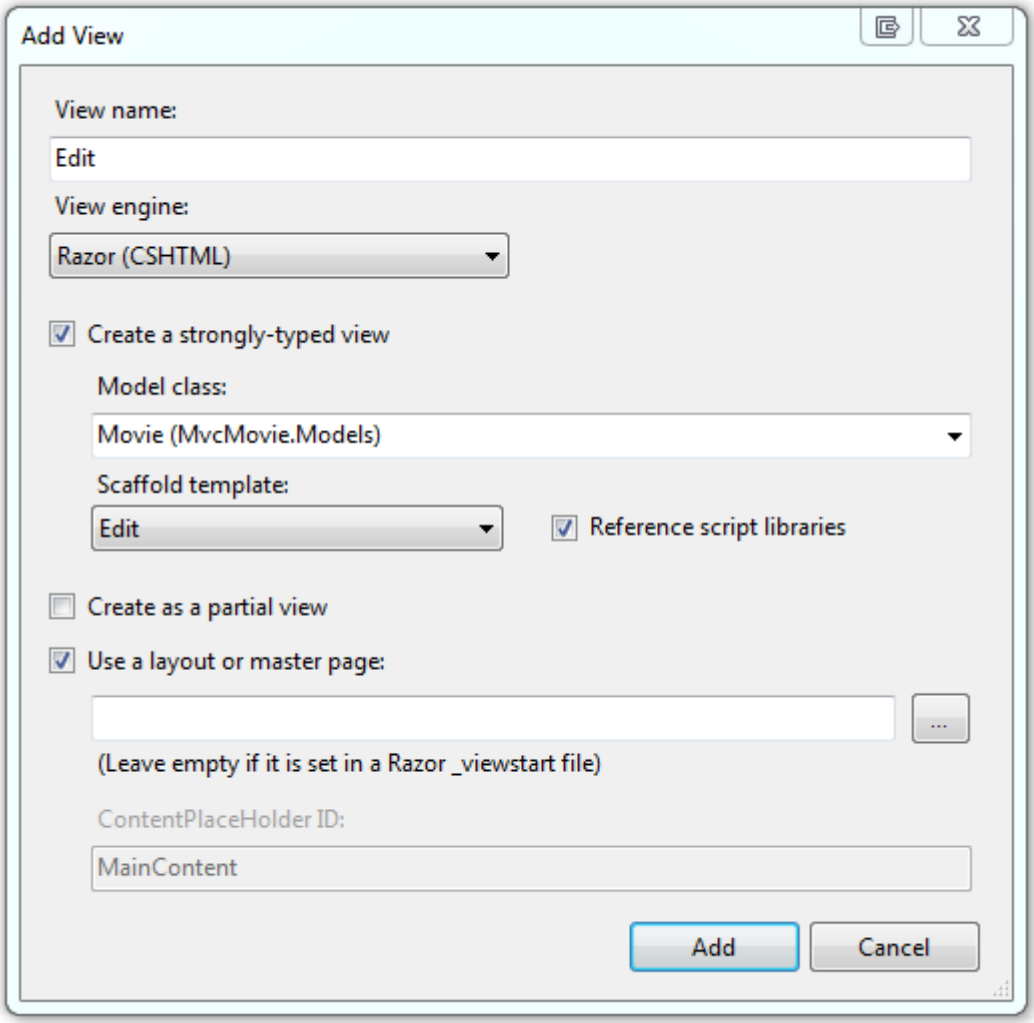
Implementing an Edit View

Back in the `Movie` controller, add the following `Edit` methods:

```
 //
// GET: /Movies/Edit

public ActionResult Edit(int id)
{
    Movie movie = db.Movies.Find(id);
    if (movie == null)
        return RedirectToAction("Index");

    return View(movie);
}

//
// POST: /Movies/Edit

[HttpPost]
public ActionResult Edit(Movie model)
{
    try
    {
        var movie = db.Movies.Find(model.ID);

        UpdateModel(movie);
        db.SaveChanges();
        return RedirectToAction("Details", new { id = model.ID });
    } catch (Exception)
    {
        ModelState.AddModelError("", "Edit Failure, see inner exception");
    }

    return View(model);
}
```

The first `Edit` method will be called when a user clicks one of the edit links. If the movie is found, the application will display the movie data in the Edit view. The `Edit` method marked with `[HttpPost]` takes a movie object created by the model binder from data posted in the Edit form(that is, when the user changes data on the Edit form and hits the **Save** button). The `UpdateModel(movie)` method invokes the model copier which copies the edited data (the `model` parameter) into the movie entry in the database. If any errors occur while the data is being saved to the database, the user is redirected to the Edit view with the data that was posted.

Right-click inside the `Edit` method and select **Add View**. For **Scaffold template**, choose **Edit**.



Run the application, select an **Edit** link, and try editing some data.

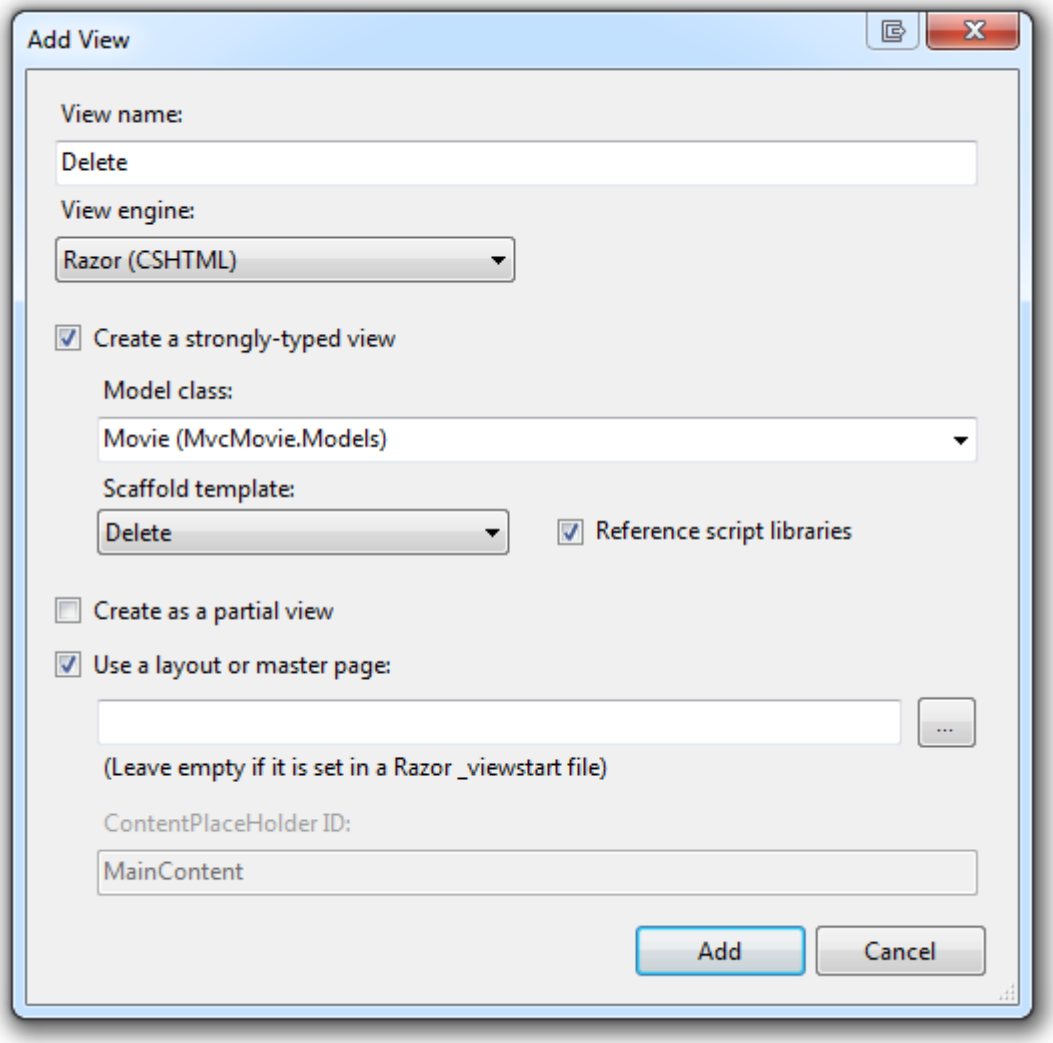
Implementing a Delete View

Add the following `Delete` methods to the `Movie` controller.

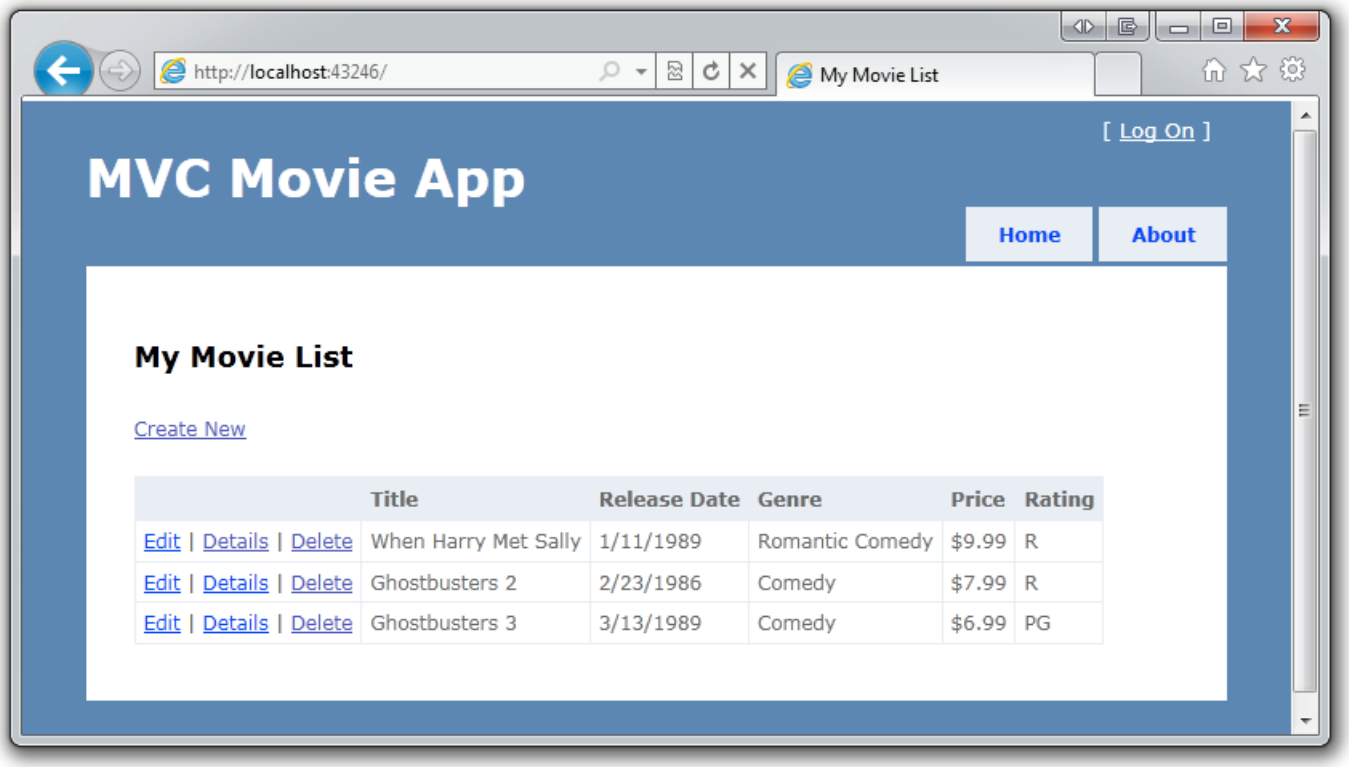
```
//
// GET: /Movies/Delete

public ActionResult Delete(int id)
{
    Movie movie = db.Movies.Find(id);
    if (movie == null)
        return RedirectToAction("Index");
    return View(movie);
}

//
// POST: /Movies/Delete

[HttpPost]
public RedirectToRouteResult Delete(int id, FormCollection collection)
{
    Movie movie = db.Movies.Find(id);
    db.Movies.Remove(movie);
    db.SaveChanges();

    return RedirectToAction("Index");
}
```

Note that the `Delete` method that *isn't* marked with `[HttpPost]` does not delete the data. 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 on this, see Stephen Walther's blog entry ASP.NET MVC Tip #46 — Don't use Delete Links because they create Security Holes.

Right-click inside the `Delete` method and select **Add View**. Select the **Delete** scaffold template.



We now have a complete MVC application that stores data in a SQL Server Express database. We can create, read, update, and delete movies.



This basic tutorial got you started making controllers, associating them with views, and passing around hard-coded data. Then we created and designed a data model. The code-first approach created a database from the data model on the fly. We retrieved the data from the database and displayed it in an HTML table. Then we added a Create form that let users add data to the database. We changed the database to include a new column of data, then updated two pages to create and display this new data. We added validation by marking the data model with attributes from the `DataAnnotations` namespace. The resulting validation runs on the client and on the server. Finally, we added code and view templates to support Edit, Details, and Delete actions.

I now encourage you to move on to our intermediate-level MVC Music Store tutorial, to explore the ASP.NET articles on MSDN, and to check out the many videos and resources at http://asp.net/mvc to learn even more about ASP.NET MVC! The MVC forums are a great place to ask questions.

Enjoy!

— Scott Hanselman (http://hanselman.com and@shanselman on Twitter)

Rev 1.0 - Edited by **Gustavo Azcona** (http://gustavoazcona.blogspot.com)
See the original tutorial at http://www.asp.net/mvc/tutorials#Getting-Started-With-MVC3
Download this document from http://www.avanic.com.ar/downloads/docs/getting_started_with_mvc3_cs.pdf