# Intro to ASP.NET MVC 4

This tutorial will teach you the basics of building an ASP.NET MVC 4 Web application using Microsoft [Visual Studio Express 2012](http://www.microsoft.com/visualstudio/11/en-us/products/express) or Visual Web Developer 2010 Express Service Pack 1. Visual Studio 2012 is recommended, you won't need to install anything to complete the tutorial. If you are using Visual Studio 2010 you must install the components below. You can install all of them by clicking the following links:

* [Visual Studio Web Developer Express SP1 prerequisites](http://www.microsoft.com/web/gallery/install.aspx?appid=VWD2010SP1Pack)
* [WPI installer for ASP.NET MVC 4](http://go.microsoft.com/fwlink/?LinkId=243392)
* [LocalDB](http://www.microsoft.com/web/gallery/install.aspx?appid=SQLLocalDBOnly_11_0)
* [SSDT](http://blogs.msdn.com/b/rickandy/archive/2012/08/02/installing-and-using-sql-server-data-tools-ssdt-on-visual-studio-2010-and-vwd.aspx)

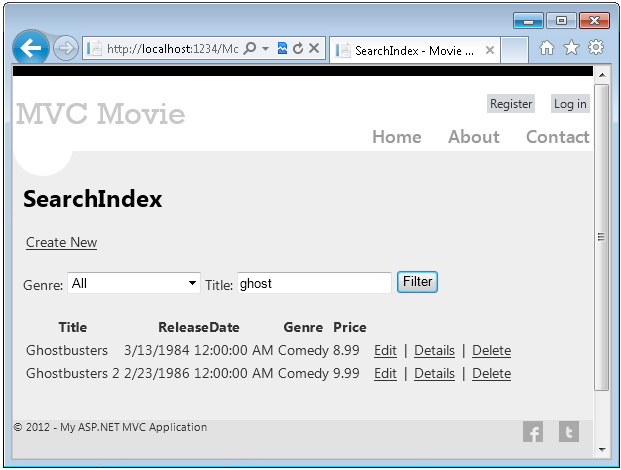
If you're using Visual Studio 2010 instead of Visual Web Developer 2010, install the [WPI installer for ASP.NET MVC 4](http://go.microsoft.com/fwlink/?LinkId=243392) and  the: [Visual Studio 2010 prerequisites](http://www.microsoft.com/web/gallery/install.aspx?appsxml=&appid=VS2010SP1Pack)

A Visual Web Developer project with C# source code is available to accompany this topic. [Download the C# version](http://archive.msdn.microsoft.com/aspnetmvcsamples/Release/ProjectReleases.aspx?ReleaseId=5892).

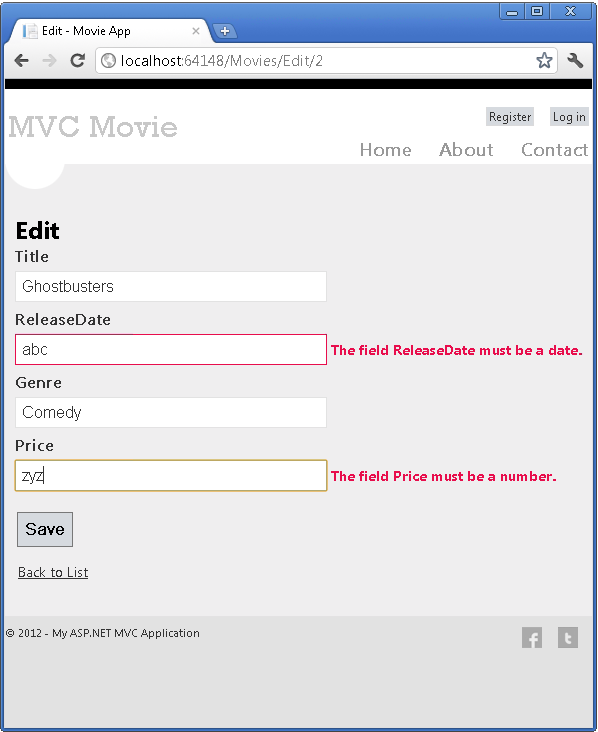
In the tutorial you run the application in Visual Studio. You can also make the application available over the Internet by deploying it to a hosting provider. Microsoft offers free web hosting for up to 10 web sites in a [free Windows Azure trial account](http://www.windowsazure.com/en-us/pricing/free-trial/?WT.mc_id=A443DD604). For information about how to deploy a Visual Studio web project to a Windows Azure Web Site, see [Create and deploy an ASP.NET web site and SQL Database with Visual Studio](http://www.windowsazure.com/en-us/develop/net/compute/). That tutorial also shows how to use Entity Framework Code First Migrations to deploy your SQL Server database to Windows Azure SQL Database (formerly SQL Azure).

## What You'll Build

You'll implement a simple movie-listing application that supports creating, editing, searching 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:



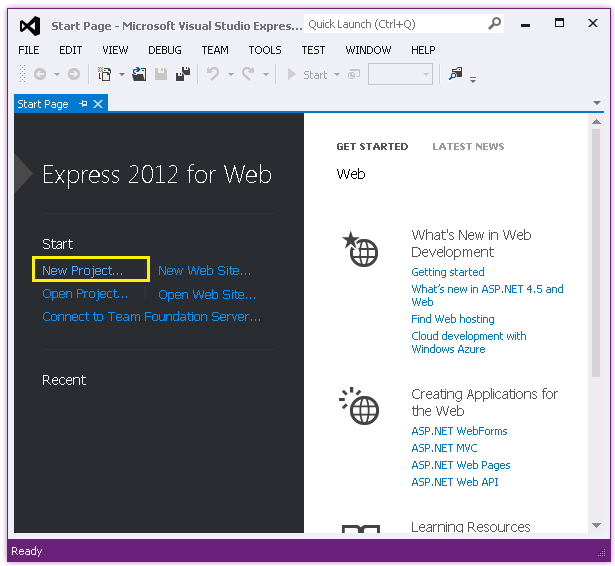
The application also lets you add, edit, and delete movies, as well as see details about individual ones. All data-entry scenarios include validation to ensure that the data stored in the database is correct.



## Getting Started

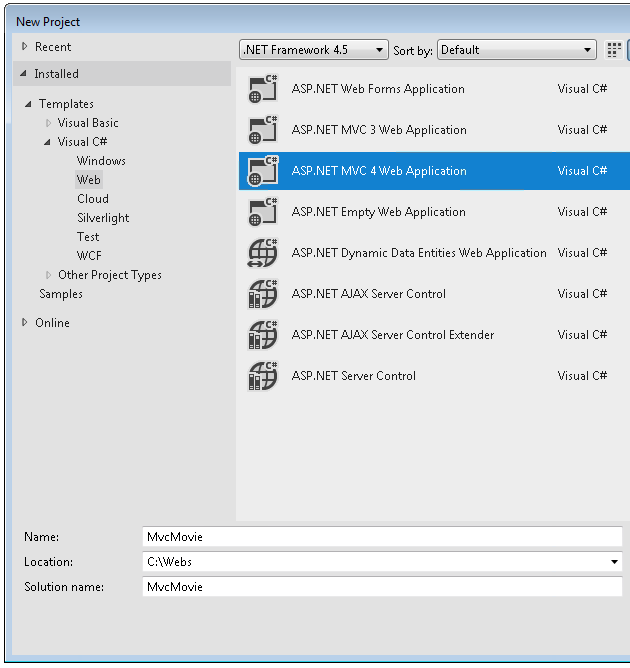
Start by running Visual Studio Express 2012  or Visual Web Developer 2010 Express. Most of the screen shots in this series use Visual Studio Express 2012, but you can complete this tutorial with Visual Studio 2010/SP1, Visual Studio 2012, Visual Studio Express 2012  or Visual Web Developer 2010 Express. Select **New Project** from the **Start** page.

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

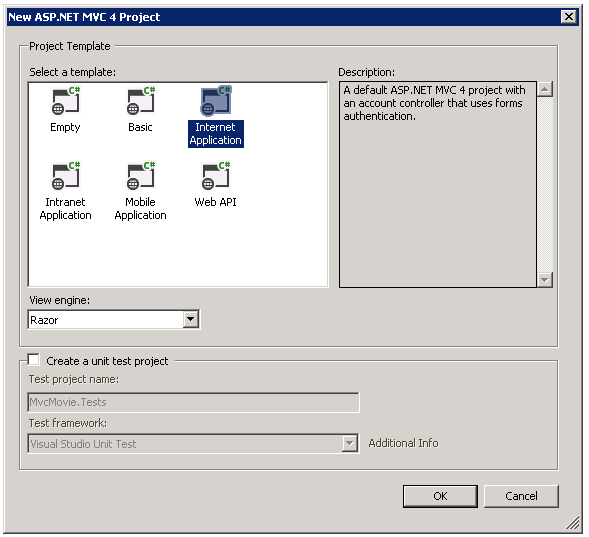


## Creating Your First Application

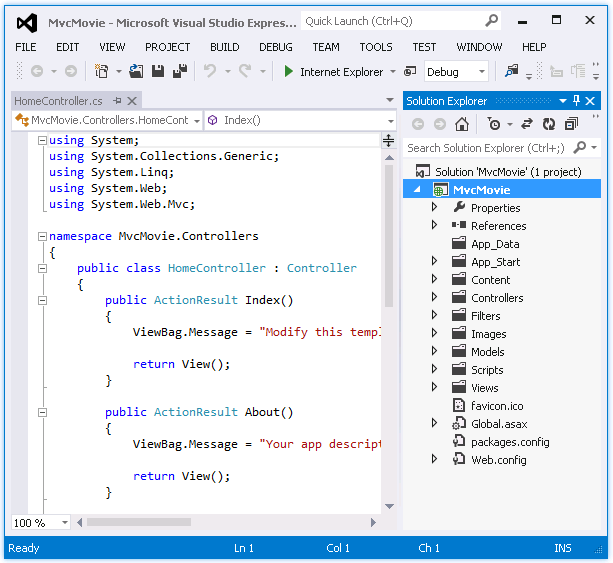
You can create applications using either Visual Basic or Visual C# as the programming language. Select Visual C# on the left and then select **ASP.NET MVC 4 Web Application**. Name your project "MvcMovie" and then click **OK**.



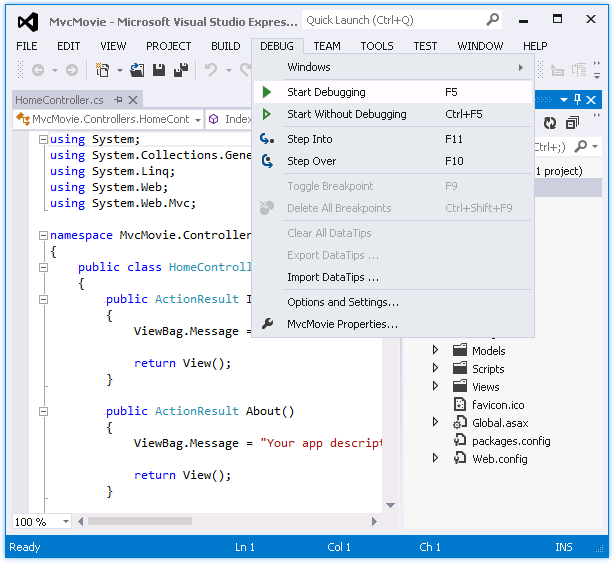
In the **New ASP.NET MVC 4 Project** dialog box, select **Internet Application**. Leave **Razor** as the default view engine.



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

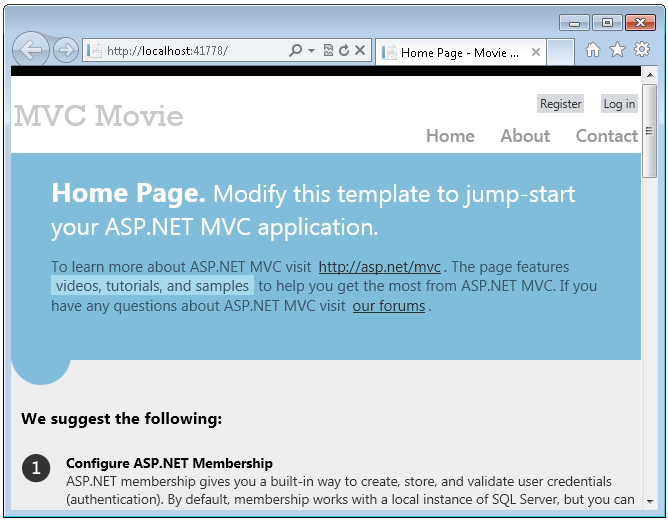


From the **Debug** menu, select **Start Debugging**.



Notice that the keyboard shortcut to start debugging is F5.

F5 causes Visual Studio to start IIS Express and run your web application. Visual Studio 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 Studio runs a web project, a random port is used for the web server. In the image below, the port number is 41788. When you run the application, you'll probably see a different port number.



Right out of the box this default template gives you  Home, Contact and About pages. It also provides support to register and log in, and links to Facebook and Twitter. The next step is to change how this application works and learn a little bit about ASP.NET MVC. Close your browser and let's change some code.

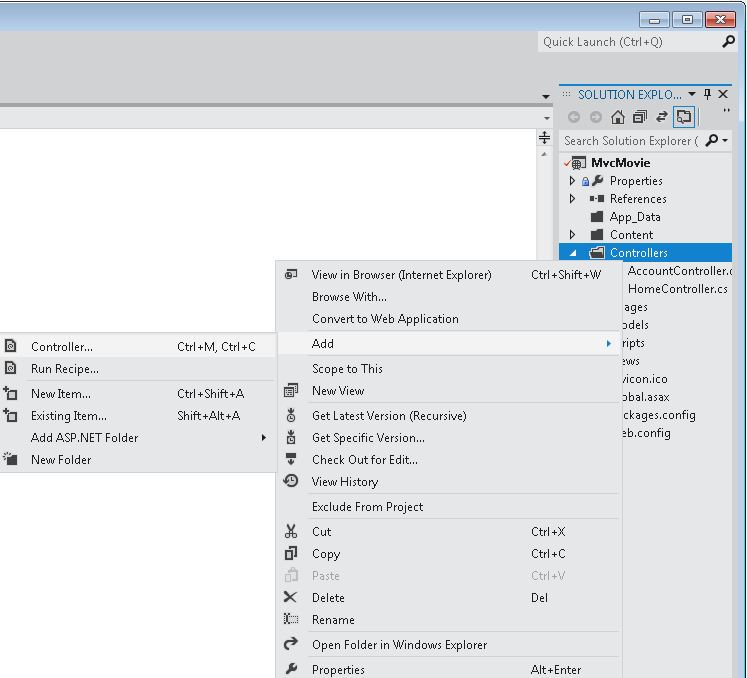
## Adding a Controller

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

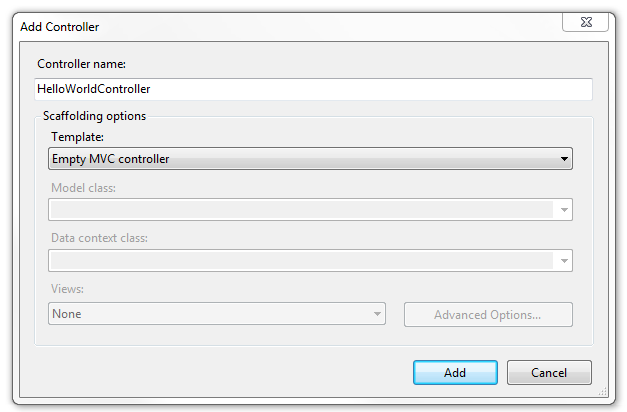
* **M**odels: Classes that represent the data of the application and that use validation logic to enforce business rules for that data.
* **V**iews: Template files that your application uses to dynamically generate HTML responses.
* **C**ontrollers: Classes that handle incoming browser requests, retrieve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

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

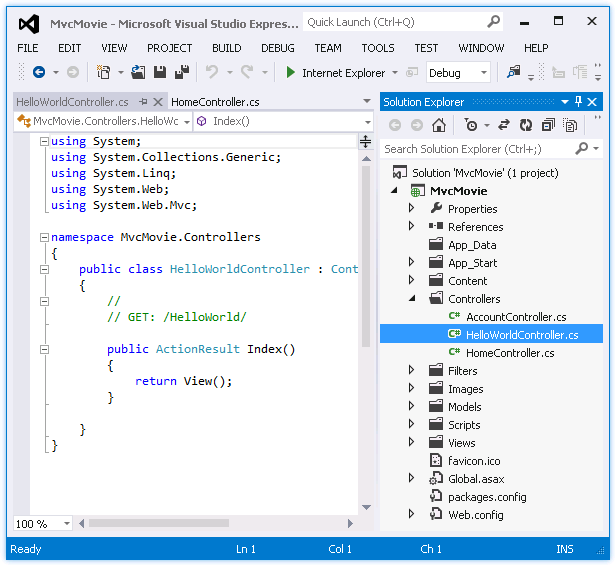
Let's begin by creating a controller class. In **Solution Explorer**, right-click the *Controllers* folder and then select **Add Controller**.



Name your new controller "HelloWorldController". Leave the default template as **Empty MVC controller** and click **Add**.



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



Replace the contents of the file with the following code.

using System.Web;

using System.Web.Mvc;

namespace MvcMovie.Controllers

{

public class HelloWorldController : Controller

{

//

// GET: /HelloWorld/

public string Index()

{

return "This is my <b>default</b> action...";

}

//

// GET: /HelloWorld/Welcome/

public string Welcome()

{

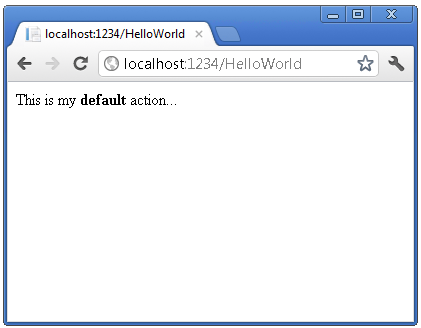
return "This is the Welcome action method...";

}

}

}

The controller methods will return a string of HTML as an example. The controller is named HelloWorldController and the first method 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, in the illustration below, it's *http://localhost:1234/HelloWorld.*) The page in the browser will look like the following screenshot. In the method above, the code returned a string directly. You told the system to just return some HTML, and it did!

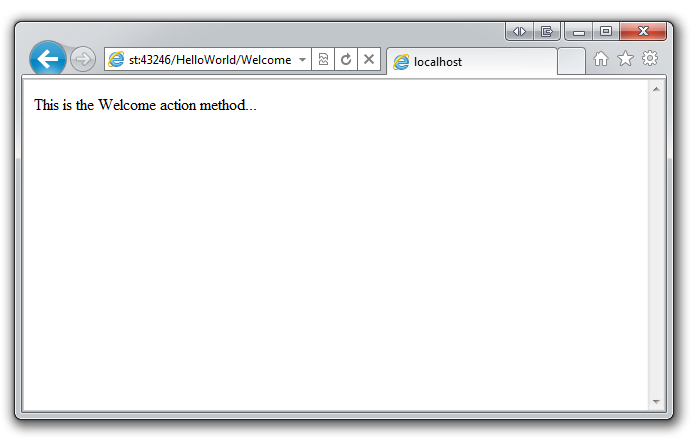


ASP.NET MVC invokes different controller classes (and different action methods within them) depending on the incoming URL. The default URL routing logic used by ASP.NET MVC uses a format like this to determine what code to invoke:

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

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.



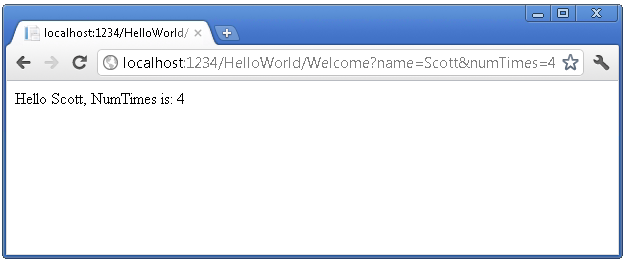
Let's modify the example slightly so that you can pass some parameter information from the URL to the controller (for example, */HelloWorld/Welcome?name=Scott&numtimes=4*). Change your Welcome method to include two parameters as shown below. Note that the code uses the C# optional-parameter feature to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

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

 return HttpUtility.HtmlEncode("Hello " + name + ", NumTimes is: " + numTimes);

}

Run your application and browse to the example URL (*http://localhost:xxxx/HelloWorld/Welcome?name=Scott&numtimes=4)*. You can try different values for name and numtimes in the URL. The [ASP.NET MVC model binding system](http://odetocode.com/Blogs/scott/archive/2009/04/27/6-tips-for-asp-net-mvc-model-binding.aspx) automatically maps the named parameters from the query string in the address bar to parameters in your method.



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

You'll create a view template file using the [Razor view engine](http://weblogs.asp.net/scottgu/archive/2010/07/02/introducing-razor.aspx) 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.

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

public ActionResult Index()

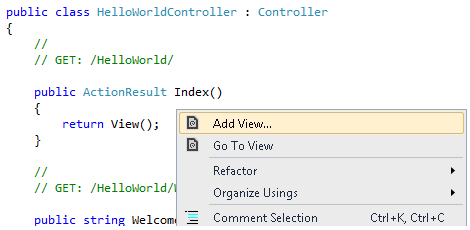
{

return View();

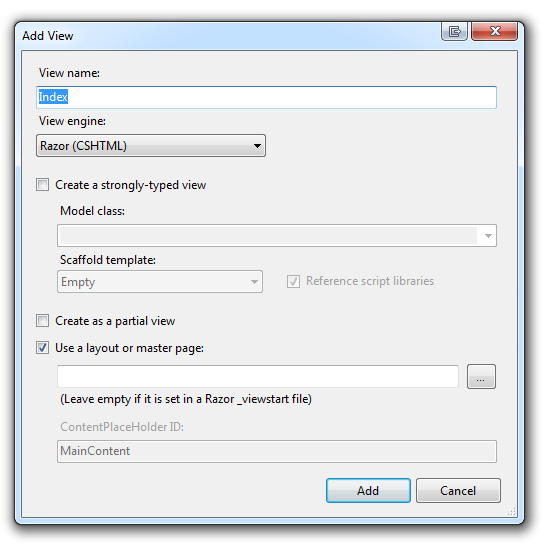
}

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

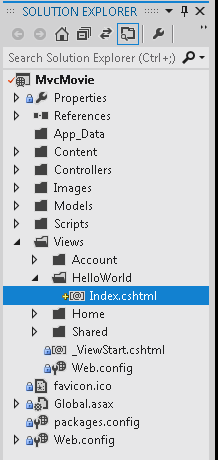
In the project, add a view template that you can use with the Index method. To do this, right-click inside the Index method and click **Add View**.



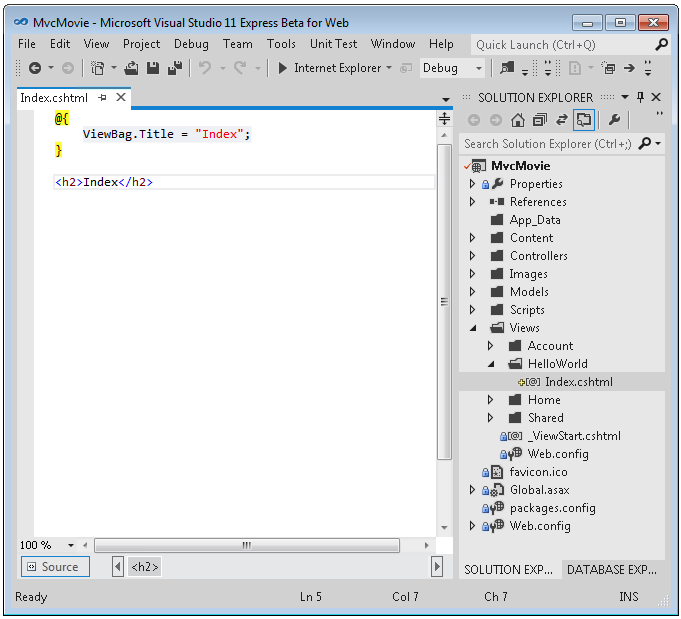
The **Add View** dialog box appears. Leave the defaults the way they are and click the **Add** button:



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



The following shows the *Index.cshtml* file that was created:



Add the following HTML under the <h2> tag.

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

 The complete *MvcMovie\Views\HelloWorld\Index.cshtml* file is shown below.

@{

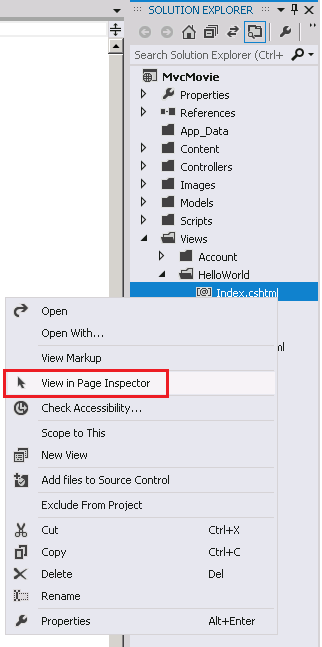
ViewBag.Title = "Index";

}

<h2>Index</h2>

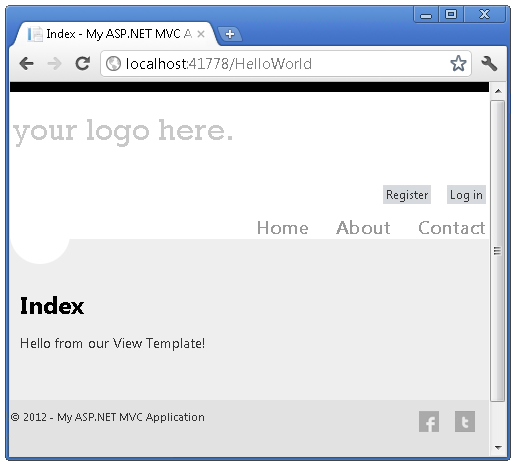
<p>Hello from our View Template!</p>

If you are using Visual Studio 2012, in solution explorer, right click the *Index.cshtml* file and select **View in Page Inspector**.



The [Page Inspector tutorial](http://www.asp.net/mvc/tutorials/mvc-4/using-page-inspector-in-aspnet-mvc) has more information about this new tool.

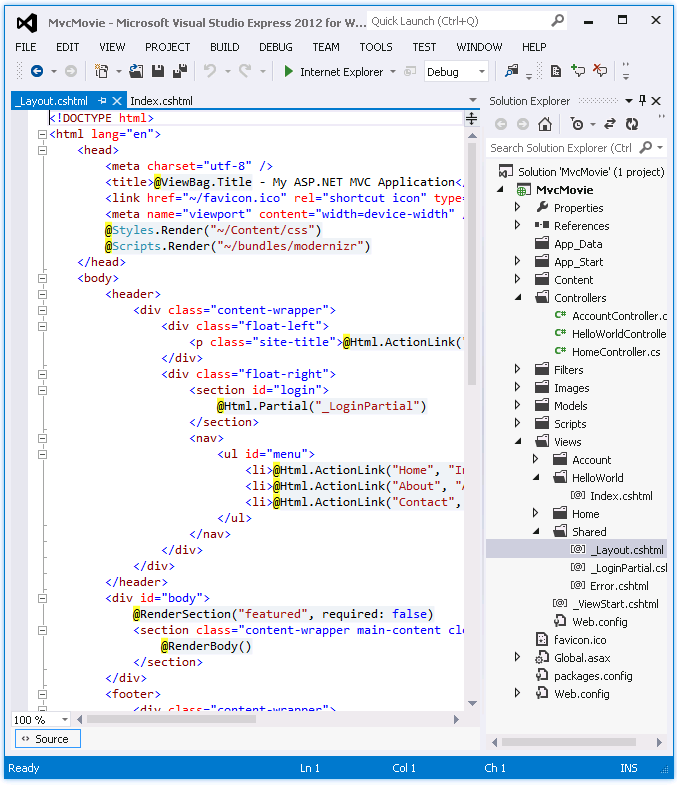
Alternatively, run the application and browse to the HelloWorld controller (*http://localhost:xxxx/HelloWorld*). The Index method in your controller didn't do much work; it simply ran the statement return View(), which specified that the method should use a view template file to render a response to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



Looks pretty good. However, notice that the browser's title bar shows  "Index My ASP.NET A" and the big link on the top of the page says "your logo here." Below the "your logo here." link are registration and log in links, and below that links to Home, About and Contact pages. Let's change some of these.

## Changing Views and Layout Pages

First, you want to change the "your logo here." title at the top of the page. That text is common to every page. It's actually implemented in only one place in the project, even though it appears on every page in the application. Go to the */Views/Shared* folder in **Solution Explorer** and open the *\_Layout.cshtml* file. This file is called a *layout page* and it's 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. Find the @RenderBody() line. [RenderBody](http://msdn.microsoft.com/en-us/gg618478) is a placeholder where all the view-specific pages you create show up, "wrapped" in the layout page. For example, if you select the About link, the *Views\Home\About.cshtml* view is rendered inside the RenderBody  method.

Change the site-title heading in the layout template from "your logo here" to "MVC Movie".

<div class="float-left">

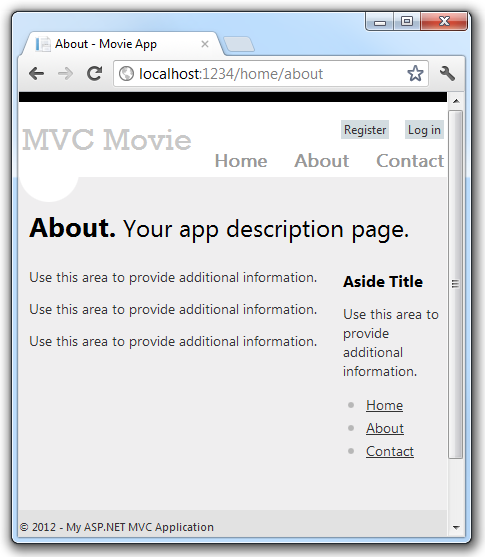
<p class="site-title">@Html.ActionLink("MVC Movie", "Index", "Home")</p>

</div>

Replace the contents of the title element with the following markup:

<title>@ViewBag.Title - Movie App</title>

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



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

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

@{

ViewBag.Title = "Movie List";

}

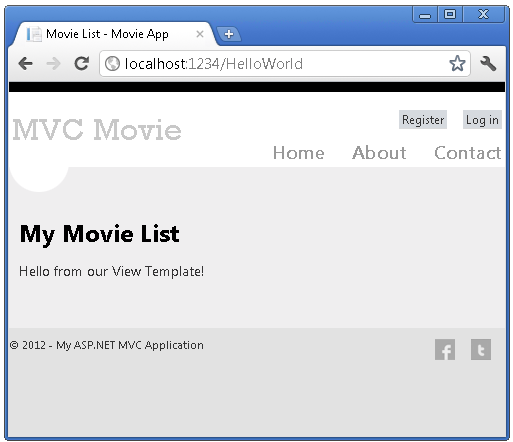
<h2>My Movie List</h2>

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

To indicate the HTML title to display, the code above sets a Title property of the ViewBag object (which is in the *Index.cshtml* view template). 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 that we modified previously. Using this ViewBag approach, you can easily pass other parameters between your view template and your layout file.

Run the application 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.) The browser title is created with the  ViewBag.Title we set in the *Index.cshtml* view template  and the additional  "- Movie App" added in the layout file.

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



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

## Passing Data from the Controller to the View

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

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

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

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

using System.Web;

using System.Web.Mvc;

namespace MvcMovie.Controllers

{

public class HelloWorldController : Controller

{

public ActionResult Index()

{

return View();

}

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

{

ViewBag.Message = "Hello " + name;

ViewBag.NumTimes = numTimes;

return View();

}

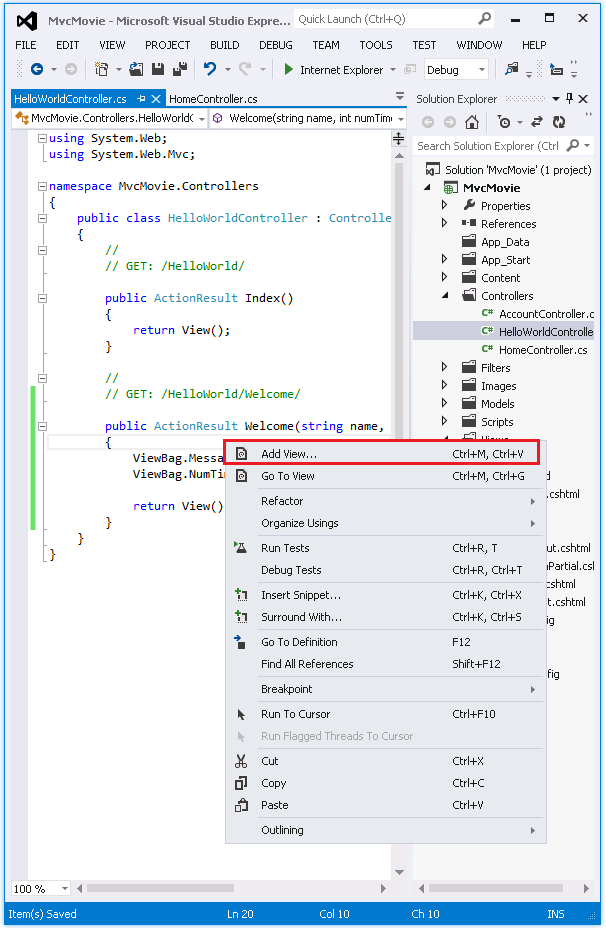
}

}

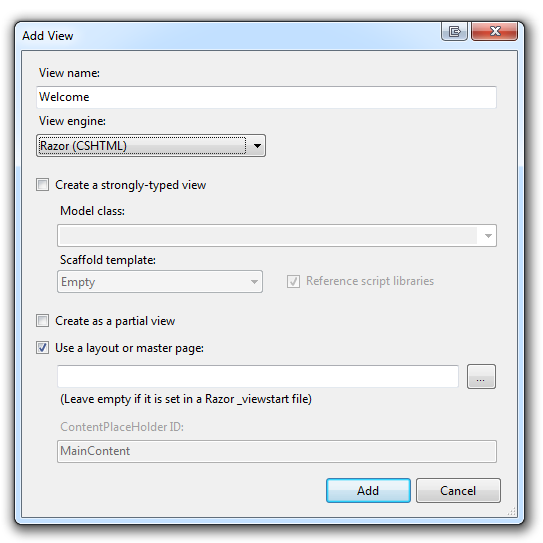
Now the ViewBag object contains data that will be passed to the view automatically.

Next, you need a Welcome view template! In the **Build** menu, select **Build 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. You'll create a loop that says "Hello" as many times as the user says it should. The complete *Welcome.cshtml* file is shown below.

@{

ViewBag.Title = "Welcome";

}

<h2>Welcome</h2>

<ul>

@for (int i=0; i < ViewBag.NumTimes; i++) {

<li>@ViewBag.Message</li>

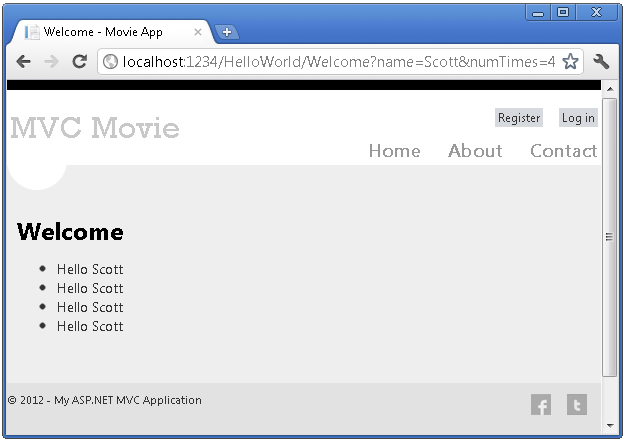
}

</ul>

Run the application and browse to the following URL:

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

Now data is taken from the URL and passed to the controller using the [model binder](http://odetocode.com/Blogs/scott/archive/2009/04/27/6-tips-for-asp-net-mvc-model-binding.aspx). The controller packages the data into a ViewBag object and passes that object to the view. The view then displays the data as HTML to the user.



 In the sample above, we used a ViewBag object to pass data from the controller to a view. Latter in the tutorial, we will use a view model to pass data from a controller to a view. The view model approach to passing data is generally much preferred over the view bag approach. See the blog entry [Dynamic V Strongly Typed Views](http://blogs.msdn.com/b/rickandy/archive/2011/01/28/dynamic-v-strongly-typed-views.aspx) for more information.

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

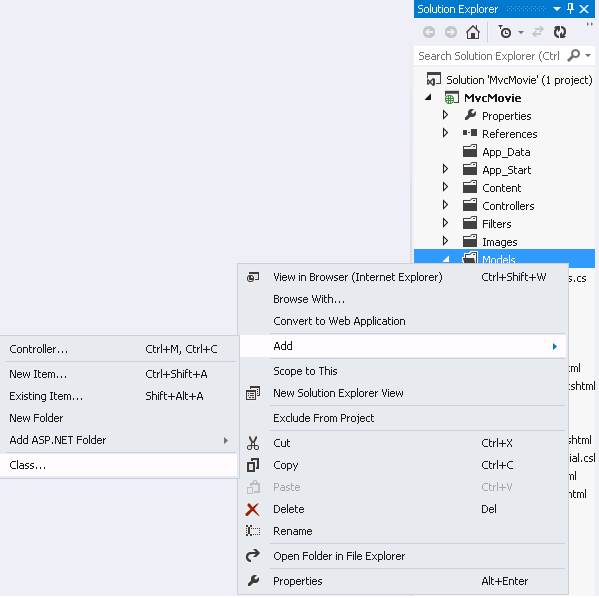
Adding a Model

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

You’ll use a .NET Framework data-access technology known as the [Entity Framework](http://msdn.microsoft.com/en-us/library/bb399572(VS.110).aspx) to define and work with these model classes. The Entity Framework (often referred to as EF) supports a development paradigm called *Code First*. Code First allows you to create model objects by writing simple classes. (These are also known as POCO classes, from "plain-old CLR objects.") You can then have the database created on the fly from your classes, which enables a very clean and rapid development workflow.

## Adding Model Classes

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



Enter the *class*  name "Movie".

Add the following five properties to the Movie class:

public class Movie

{

public int ID { get; set; }

public string Title { get; set; }

public DateTime ReleaseDate { get; set; }

public string Genre { get; set; }

public decimal Price { get; set; }

}

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

In the same file, add the following MovieDBContext class:

public class MovieDBContext : DbContext

{

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

}

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

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

using System.Data.Entity;

The complete *Movie.cs* file is shown below. (Several using statements that are not needed have been removed.)

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; }

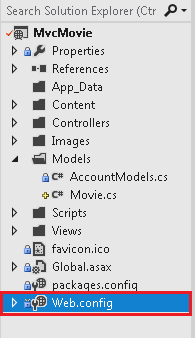
}

}

## Creating a Connection String and Working with SQL Server LocalDB

The MovieDBContext class you created handles the task of connecting to the database and mapping Movie objects to database records. One question you might ask, though, is how to specify which database it will connect to. You'll do that by adding connection information in the *Web.config* file of the application.

Open the application root *Web.config* file. (Not the *Web.config* file in the *Views* folder.) Open the *Web.config* file outlined in red.



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

<add name="MovieDBContext"

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

providerName="System.Data.SqlClient"

/>

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

<connectionStrings>

<add name="DefaultConnection"

connectionString="Data Source=(LocalDb)\v11.0;Initial Catalog=aspnet-MvcMovie-2012213181139;Integrated Security=true"

providerName="System.Data.SqlClient"

/>

<add name="MovieDBContext"

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

providerName="System.Data.SqlClient"

/>

</connectionStrings>

This small amount of code and XML is everything you need to write in order to represent and store the movie data in a database.

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

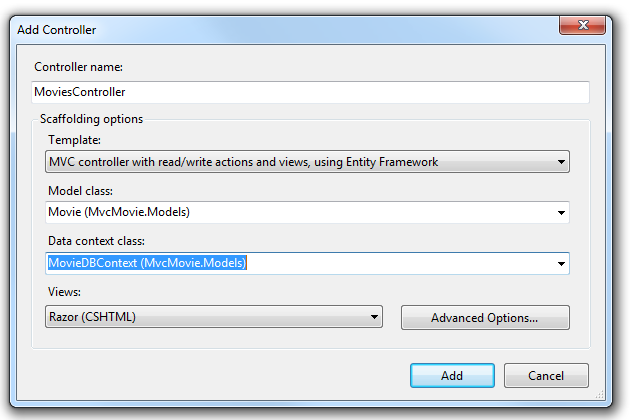
Accessing Your Model's Data from a Controller

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

**Build the application** before going on to the next step.

Right-click the Controllers folder and create a new MoviesController controller. The options below will not appear until you build your application. Select the following options:

* Controller name: **MoviesController**. (This is the default. )
* Template: **MVC Controller with read/write actions and views, using Entity Framework**.
* Model class: **Movie (MvcMovie.Models)**.
* Data context class: **MovieDBContext (MvcMovie.Models)**.
* Views: **Razor (CSHTML)**. (The default.)

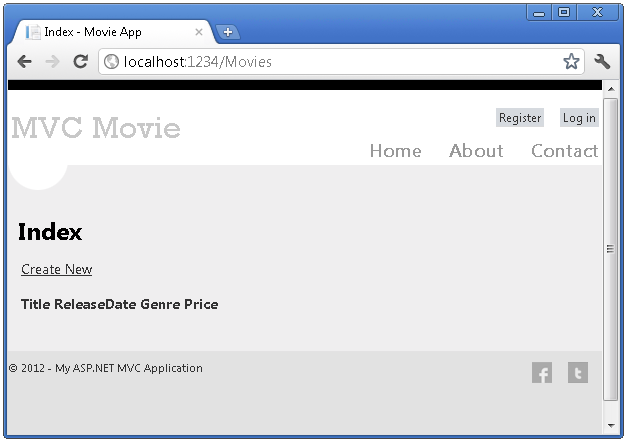


Click **Add**. Visual Studio Express creates the following files and folders:

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

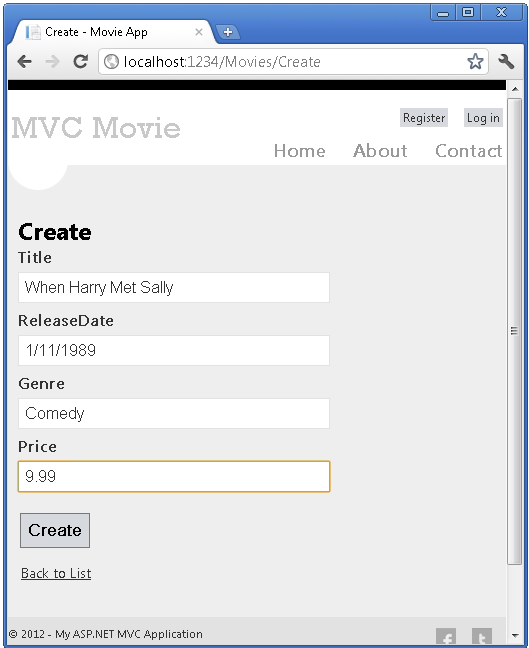
ASP.NET MVC 4  automatically created the CRUD (create, read, update, and delete) action methods and views for you (the automatic creation of CRUD action methods and views is known as scaffolding). You now have a fully functional web application that lets you create, list, edit, and delete movie entries.

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

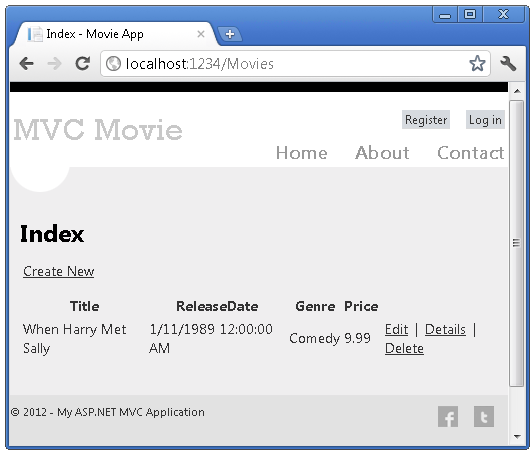


## Creating a Movie

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



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



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

## Examining the Generated Code

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

public class MoviesController : Controller

{

private MovieDBContext db = new MovieDBContext();

//

// GET: /Movies/

public ActionResult Index()

{

return View(db.Movies.ToList());

}

The following line from the MoviesController class instantiates a movie database context, as described previously. You can use the movie database context to query, edit, and delete movies.

private MovieDBContext db = new MovieDBContext();

A request to the Movies controller returns all the entries in the Movies table of the movie database and then passes the results to the Index view.

## Strongly Typed Models and the @model Keyword

Earlier in this tutorial, you saw how a controller can pass data or objects to a view template using the ViewBag object. The ViewBag is a dynamic object that provides a convenient late-bound way to pass information to a view.

ASP.NET MVC also provides the ability to pass strongly typed data or objects to a view template. This strongly typed approach enables better compile-time checking of your code and richer IntelliSense in the Visual Studio editor. The scaffolding mechanism in Visual Studio used  this approach with the MoviesController class and view templates when it created the methods and views.

In the Controllers\MoviesController.cs file examine the generated Details method. A portion of the movie controller with the Details method is shown below.

public ActionResult Details(int id = 0)

{

Movie movie = db.Movies.Find(id);

if (movie == null)

{

return HttpNotFound();

}

return View(movie);

}

If a  Movie is found, an instance of the Movie model is passed to the Details view. Examine the contents of the Views\Movies\Details.cshtml file.

By including a @model statement at the top of the view template file, you can specify the type of object that the view expects. When you created the movie controller, Visual Studio automatically included the following @model statement at the top of the Details.cshtml file:

@model MvcMovie.Models.Movie

This @model directive allows you to access the  movie that the controller passed to the view by using a Model object that's strongly typed. For example, in the Details.cshtml template, the code passes each movie field to the DisplayNameFor and  [DisplayFor](http://msdn.microsoft.com/en-us/library/system.web.mvc.html.displayextensions.displayfor(VS.98).aspx) HTML Helpers with  the strongly typed Model object.  The Create and Edit methods and view templates also pass a movie model object.

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

public ActionResult Index()

{

return View(db.Movies.ToList());

}

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

@model IEnumerable<MvcMovie.Models.Movie>

This @model directive allows you to access the list of movies that the controller passed to the view by using a Model object that's strongly typed. For example, in the Index.cshtml template, the code loops through the movies by doing a foreach statement over the strongly typed Model object:

@foreach (var item in Model) {

<tr>

<td>

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

</td>

<td>

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

</td>

<td>

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

</td>

<td>

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

</td>

<th>

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

</th>

<td>

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

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

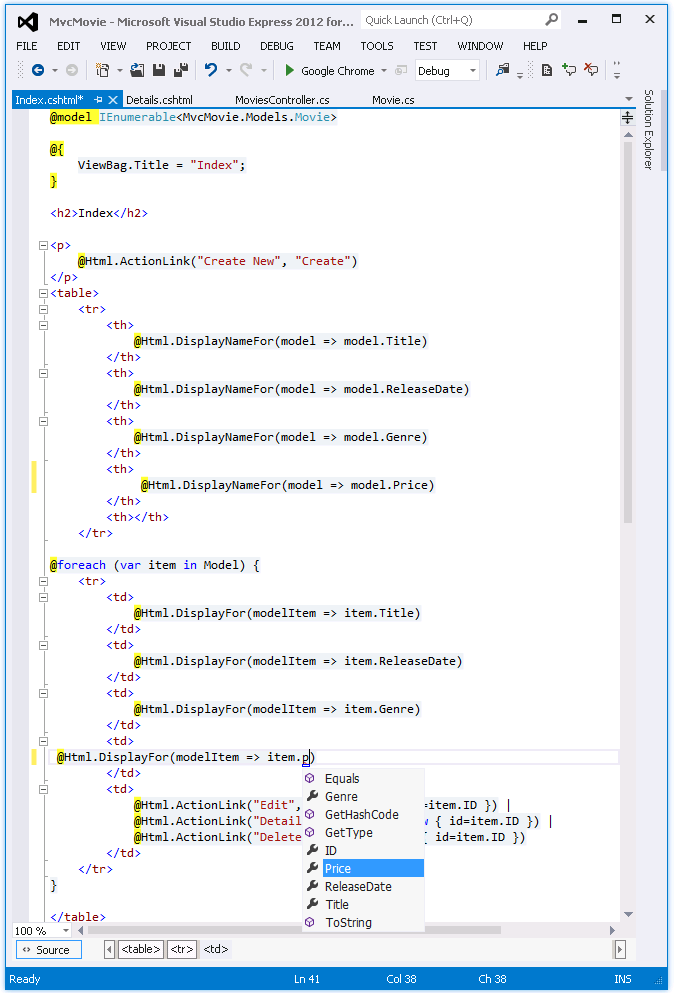
@Html.ActionLink("Delete", "Delete", { id=item.ID })

</td>

</tr>

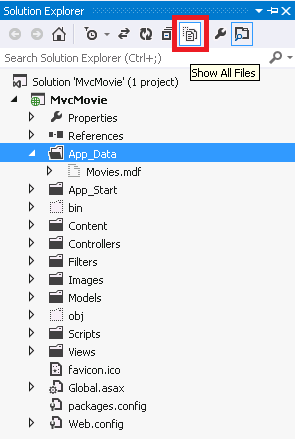
}

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

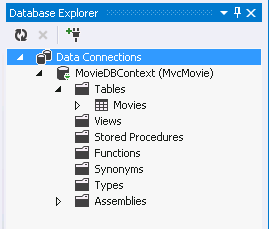


## Working with SQL Server LocalDB

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



Double-click Movies.mdf to open **DATABASE EXPLORER**, then expand the **Tables** folder to see the Movies table.



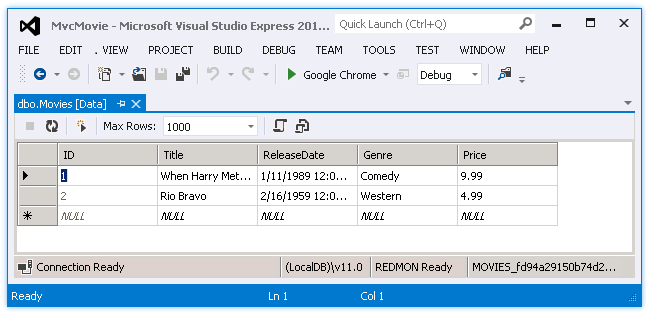
Note: If  the database explorer doesn't appear, from the **TOOLS** menu, select **Connect to Database**, then cancel the **Choose Data Source** dialog. This will force open the database explorer.

Note: If you are using VWD or Visual Studio 2010 and get an error similar to any of the following following:

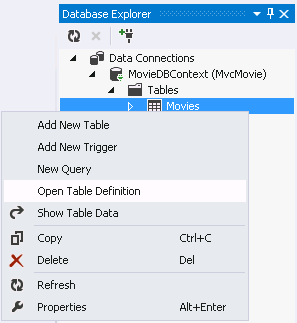
* The database 'C:\Webs\MVC4\MVCMOVIE\MVCMOVIE\APP\_DATA\MOVIES.MDF' cannot be opened because it is version 706. This server supports version 655 and earlier. A downgrade path is not supported.
* "InvalidOperation Exception was unhandled by user code" The supplied SqlConnection does not specify an initial catalog.

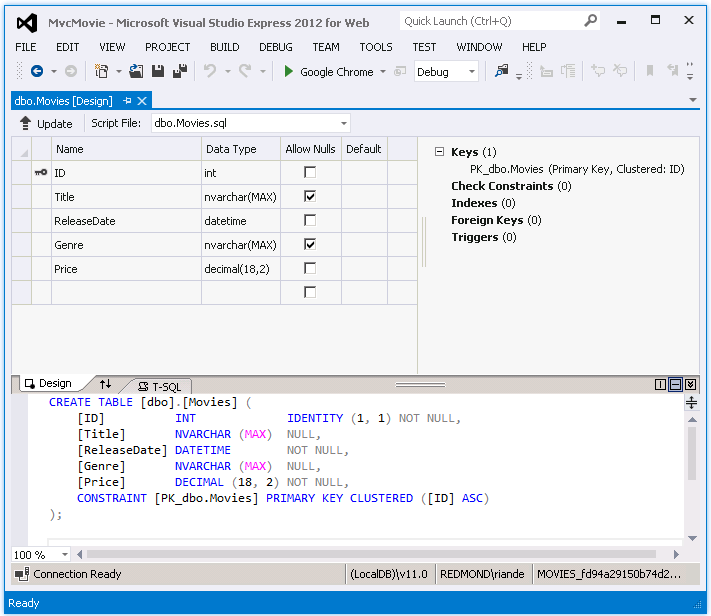
You need to install the [SQL Server Data Tools](http://blogs.msdn.com/b/rickandy/archive/2012/08/02/installing-and-using-sql-server-data-tools-ssdt-on-visual-studio-2010-and-vwd.aspx) and [LocalDB](http://www.microsoft.com/web/gallery/install.aspx?appid=SQLLocalDBOnly_11_0). Verify the MovieDBContext connection string specified on the previous page.

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



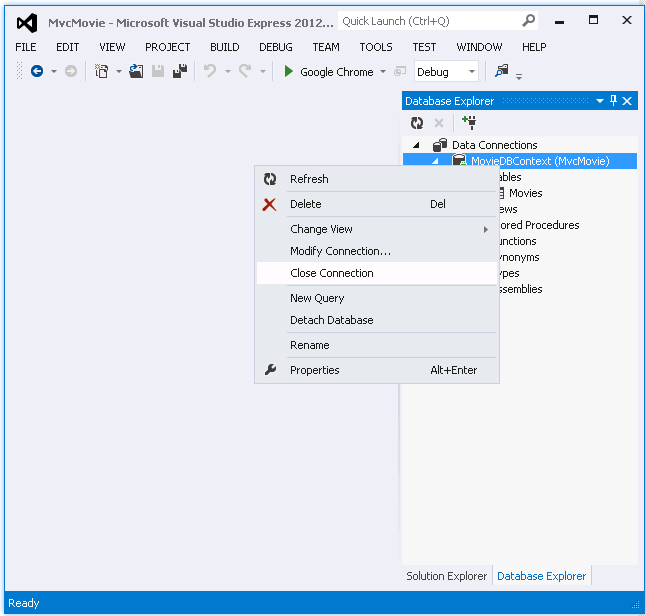
Right-click the Movies table and select **Open Table Definition** to see the table structure that Entity Framework Code First created for you.





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

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

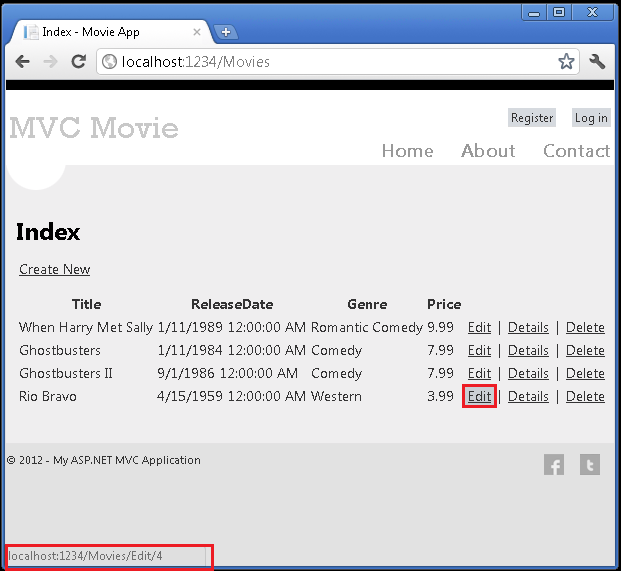


You now have the database and a simple listing page to display content from it. In the next tutorial, we'll examine the rest of the scaffolded code and add a SearchIndex method and a SearchIndex view that lets you search for movies in this database.

Examining the Edit Methods and Edit View

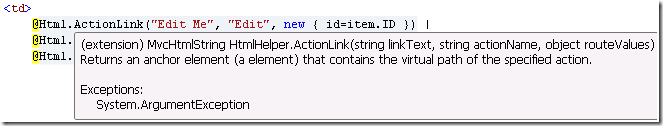
In this section, you'll examine the generated action methods and views for the movie controller. Then you'll add a custom search page.

Run the application and browse to the Movies controller by appending */Movies* to the URL in the address bar of your browser. Hold the mouse pointer over an **Edit** link to see the URL that it links to.



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

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



The Html object is a helper that's exposed using a property on the [System.Web.Mvc.WebViewPage](http://msdn.microsoft.com/en-us/library/gg402107(VS.98).aspx)  base class. The [ActionLink](http://msdn.microsoft.com/en-us/library/system.web.mvc.html.linkextensions.actionlink.aspx) method of the helper makes it easy to dynamically generate HTML hyperlinks that link to action methods on controllers. The first argument to the ActionLink method is the link text to render (for example, <a>Edit Me</a>). The second argument is the name of the action method to invoke. The final argument is an [anonymous object](http://weblogs.asp.net/scottgu/archive/2007/05/15/new-orcas-language-feature-anonymous-types.aspx) that generates the route data (in this case, the ID of 4).

The generated link shown in the previous image is *http://localhost:xxxxx/Movies/Edit/4*. The default route (established in *App\_Start\RouteConfig.cs*) takes the URL pattern {controller}/{action}/{id}. Therefore, ASP.NET translates *http://localhost:xxxxx/Movies/Edit/4* into a request to the Edit action method of the Movies controller with the parameter ID equal to 4.  Examine the following code from the *App\_Start\RouteConfig.cs* file.

public static void RegisterRoutes(RouteCollection routes)

{

routes.IgnoreRoute("{resource}.axd/{\*pathInfo}");

routes.MapRoute(

name: "Default",

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

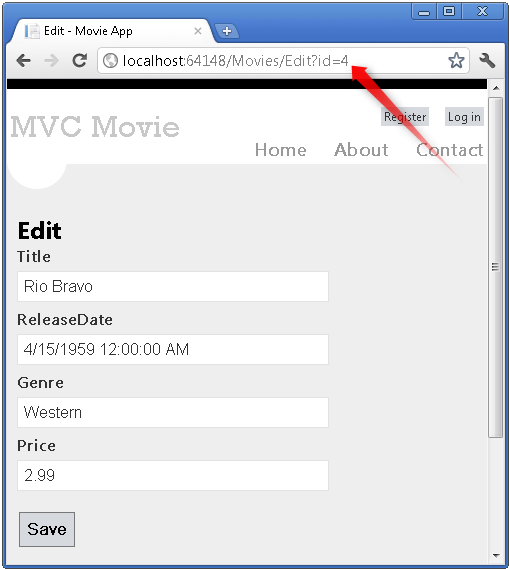
defaults: new { controller = "Home", action = "Index",

id = UrlParameter.Optional }

);

}

You can also pass action method parameters using a query string. For example, the URL *http://localhost:xxxxx/Movies/Edit?ID=4* also passes the parameter ID of 4 to the Edit action method of the Movies controller.



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

//

// GET: /Movies/Edit/5

public ActionResult Edit(int id = 0)

{

Movie movie = db.Movies.Find(id);

if (movie == null)

{

return HttpNotFound();

}

return View(movie);

}

//

// POST: /Movies/Edit/5

[HttpPost]

public ActionResult Edit(Movie movie)

{

if (ModelState.IsValid)

{

db.Entry(movie).State = EntityState.Modified;

db.SaveChanges();

return RedirectToAction("Index");

}

return View(movie);

}

Notice the second Edit action method is preceded by the [HttpPost](http://msdn.microsoft.com/en-us/library/system.web.mvc.httppostattribute.aspx) attribute. This attribute specifies that that overload of the Edit method can be invoked only for POST requests. You could apply the [HttpGet](http://msdn.microsoft.com/en-us/library/system.web.mvc.httpgetattribute.aspx) attribute to the first edit method, but that's not necessary because it's the default. (We'll refer to action methods that are implicitly assigned the HttpGet attribute as HttpGet methods.)

The HttpGet Edit method takes the movie ID parameter, looks up the movie using the Entity Framework Find method, and returns the selected movie to the Edit view. The ID parameter specifies a [default value](http://msdn.microsoft.com/en-us/library/dd264739.aspx) of zero if the Edit method is called without a parameter. If a movie cannot be found,  [HttpNotFound](http://msdn.microsoft.com/en-us/library/gg453938(VS.98).aspx) is returned. When the scaffolding system created the Edit view, it examined the Movie class and created code to render <label> and <input> elements for each property of the class. The following example shows the Edit view that was generated:

@model MvcMovie.Models.Movie

@{

ViewBag.Title = "Edit";

}

<h2>Edit</h2>

@using (Html.BeginForm()) {

@Html.ValidationSummary(true)

<fieldset>

<legend>Movie</legend>

@Html.HiddenFor(model => model.ID)

<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="Save" />

</p>

</fieldset>

}

<div>

@Html.ActionLink("Back to List", "Index")

</div>

@section Scripts {

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

}

Notice how the view template has a @model MvcMovie.Models.Movie statement at the top of the file — this specifies that the view expects the model for the view template to be of type Movie.

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

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

<form action="/Movies/Edit/4" method="post"> <fieldset>

<legend>Movie</legend>

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

<div class="editor-label">

<label for="Title">Title</label>

</div>

<div class="editor-field">

<input class="text-box single-line" id="Title" name="Title" type="text" value="Rio Bravo" />

<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-date="The field ReleaseDate must be a date." data-val-required="The ReleaseDate field is required." id="ReleaseDate" name="ReleaseDate" type="text" value="4/15/1959 12:00:00 AM" />

<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="Western" />

<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-required="The Price field is required." id="Price" name="Price" type="text" value="2.99" />

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

</div>

<p>

<input type="submit" value="Save" />

</p>

</fieldset>

</form>

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

## Processing the POST Request

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

[HttpPost]

public ActionResult Edit(Movie movie)

{

if (ModelState.IsValid)

{

db.Entry(movie).State = EntityState.Modified;

db.SaveChanges();

return RedirectToAction("Index");

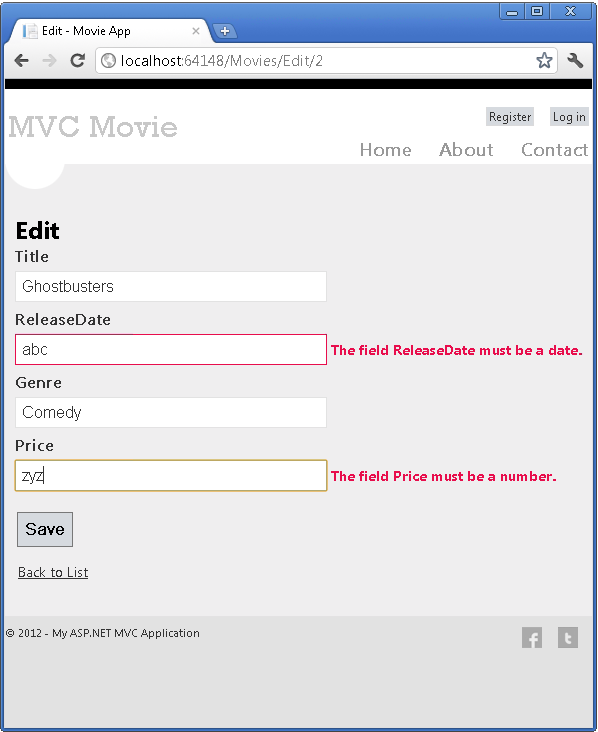
}

return View(movie);

}

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

If the posted values aren't valid, they are redisplayed in the form. The Html.ValidationMessageFor helpers in the *Edit.cshtml* view template take care of displaying appropriate error messages.



**Note** to support jQuery validation for non-English locales that use a comma (",") for a decimal point, you must include *globalize.js* and your specific *cultures/globalize.cultures.js* file(from <https://github.com/jquery/globalize> ) and JavaScript to use Globalize.parseFloat. The following code shows the modifications to the Views\Movies\Edit.cshtml file to work with the "fr-FR" culture:

@section Scripts {

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

<script src="~/Scripts/globalize.js"></script>

<script src="~/Scripts/globalize.culture.fr-FR.js"></script>

<script>

$.validator.methods.number = function (value, element) {

return this.optional(element) ||

!isNaN(Globalize.parseFloat(value));

}

$(document).ready(function () {

Globalize.culture('fr-FR');

});

</script>

<script>

jQuery.extend(jQuery.validator.methods, {

range: function (value, element, param) {

//Use the Globalization plugin to parse the value

var val = $.global.parseFloat(value);

return this.optional(element) || (

val >= param[0] && val <= param[1]);

}

});

</script>

}

The decimal field may require a comma, not a decimal point. As a temporary fix, you can add the globalization element to the projects root web.config file. The following code shows the globalization element with the culture set to United States English.

<system.web>

<globalization culture ="en-US" />

<!--elements removed for clarity-->

</system.web>

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

## Adding a Search Method and Search View

In this section you'll add a SearchIndex action method that lets you search movies by genre or name. This will be available using the */Movies/SearchIndex* URL. The request will display an HTML form that contains input elements that a user can enter in order to search for a movie. When a user submits the form, the action method will get the search values posted by the user and use the values to search the database.

## Displaying the SearchIndex Form

Start by adding a SearchIndex action method to the existing MoviesController class. The method will return a view that contains an HTML form. Here's the code:

public ActionResult SearchIndex(string searchString)

{

var movies = from m in db.Movies

select m;

if (!String.IsNullOrEmpty(searchString))

{

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

}

return View(movies);

}

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

var movies = from m in db.Movies

select m;

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

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

if (!String.IsNullOrEmpty(searchString))

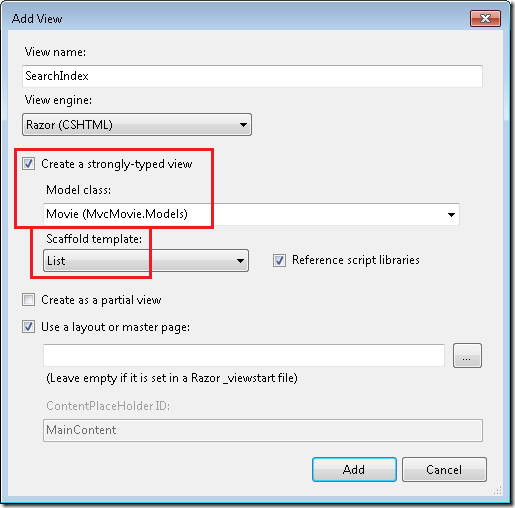
{

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

}

The s => s.Title code above is a [Lambda Expression](http://msdn.microsoft.com/en-us/library/bb397687.aspx). Lambdas are used in method-based [LINQ](http://msdn.microsoft.com/en-us/library/bb397926.aspx) queries as arguments to standard query operator methods such as the [Where](http://msdn.microsoft.com/en-us/library/system.linq.enumerable.where.aspx) method used in the above code. LINQ queries are not executed when they are defined or when they are modified by calling a method such as Where or OrderBy. Instead, query execution is deferred, which means that the evaluation of an expression is delayed until its realized value is actually iterated over or the [ToList](http://msdn.microsoft.com/en-us/library/bb342261.aspx) method is called. In the SearchIndex sample, the query is executed in the SearchIndex view. For more information about deferred query execution, see [Query Execution](http://msdn.microsoft.com/en-us/library/bb738633.aspx).

Now you can implement the SearchIndex view that will display the form to the user. Right-click inside the SearchIndex method and then click **Add View**. In the **Add View** dialog box, specify that you're going to pass a Movie object to the view template as its model class. In the **Scaffold template** list, choose **List**, then click **Add**.



When you click the **Add** button, the *Views\Movies\SearchIndex.cshtml* view template is created. Because you selected **List** in the **Scaffold template** list, Visual Studio automatically generated (scaffolded) some default markup in the view. The scaffolding created an HTML form. It examined the Movie class and created code to render <label> elements for each property of the class. The listing below shows the Create view that was generated:

@model IEnumerable<MvcMovie.Models.Movie>

@{

ViewBag.Title = "SearchIndex";

}

<h2>SearchIndex</h2>

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

</p>

<table>

<tr>

<th>

Title

</th>

<th>

ReleaseDate

</th>

<th>

Genre

</th>

<th>

Price

</th>

<th></th>

</tr>

@foreach (var item in Model) {

<tr>

<td>

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

</td>

<td>

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

</td>

<td>

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

</td>

<td>

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

</td>

<td>

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

@Html.ActionLink("Details", "Details", new { id=item.ID }) |

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

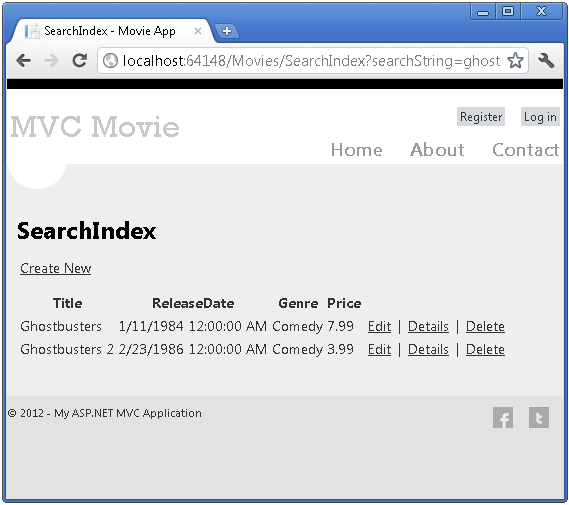
</td>

</tr>

}

</table>

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



If you change the signature of the SearchIndex method to have a parameter named id, the id parameter will match the {id} placeholder for the default routes set in the *Global.asax* file.

{controller}/{action}/{id}

The original SearchIndex method looks like this::

public ActionResult SearchIndex(string searchString)

{

var movies = from m in db.Movies

select m;

if (!String.IsNullOrEmpty(searchString))

{

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

}

return View(movies);

}

The modified SearchIndex method would look as follows:

public ActionResult SearchIndex(string id)

{

string searchString = id;

var movies = from m in db.Movies

select m;

if (!String.IsNullOrEmpty(searchString))

{

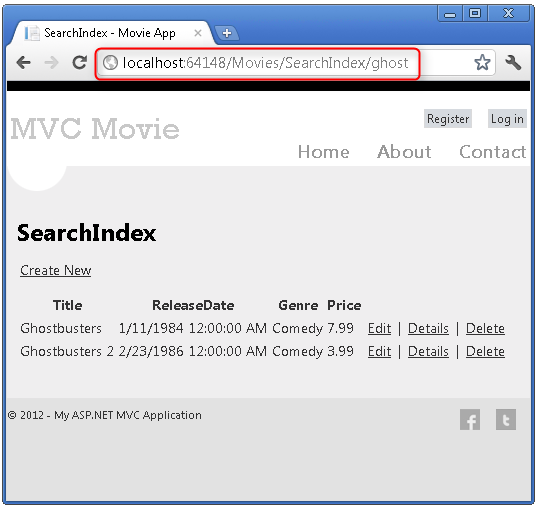
movies = movies.Where(s => s.Title.Contains(searchString));

}

return View(movies);

}

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



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

public ActionResult SearchIndex(string searchString)

{

var movies = from m in db.Movies

select m;

if (!String.IsNullOrEmpty(searchString))

{

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

}

return View(movies);

}

Open the *Views\Movies\SearchIndex.cshtml* file, and just after @Html.ActionLink("Create New", "Create"), add the following:

@using (Html.BeginForm()){

<p> Title: @Html.TextBox("SearchString")<br />

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

}

The following example shows a portion of the *Views\Movies\SearchIndex.cshtml* file with the added filtering markup.

@model IEnumerable<MvcMovie.Models.Movie>

@{

ViewBag.Title = "SearchIndex";

}

<h2>SearchIndex</h2>

<p>

@Html.ActionLink("Create New", "Create")

@using (Html.BeginForm()){

<p> Title: @Html.TextBox("SearchString") <br />

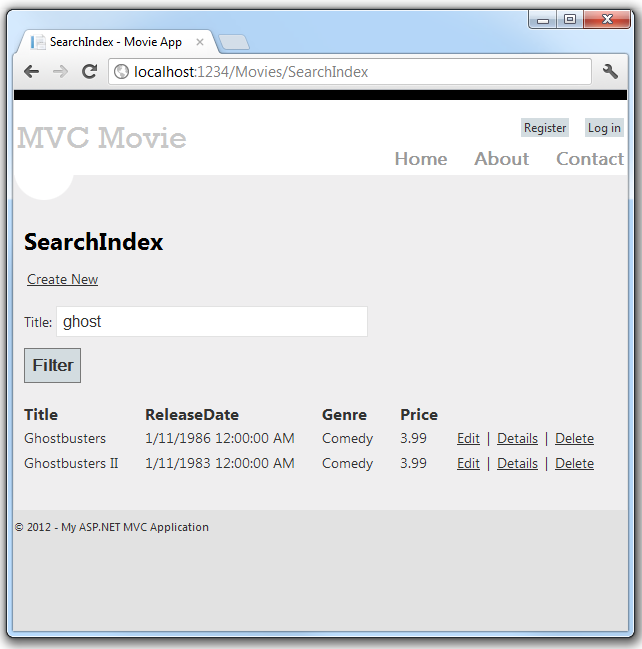
<input type="submit" value="Filter" /></p>

}

</p>

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

Run the application and try searching for a movie.



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

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

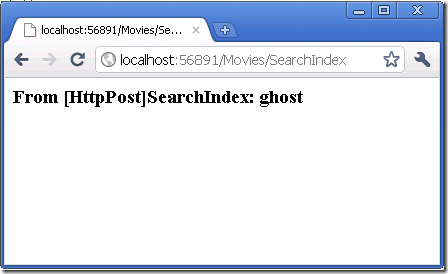
[HttpPost]

public string SearchIndex(FormCollection fc, string searchString)

{

return "<h3> From [HttpPost]SearchIndex: " + searchString + "</h3>";

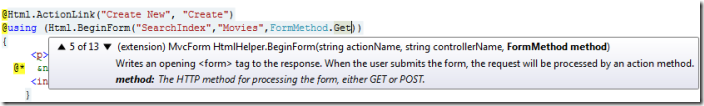
}



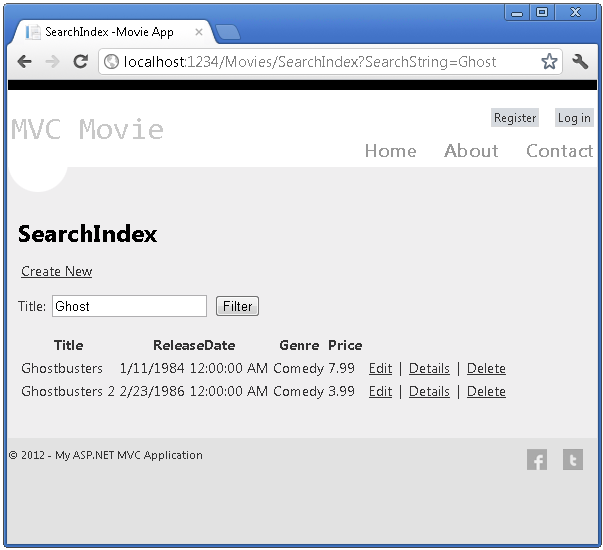
However, even if you add this HttpPost version of the SearchIndex method, there's a limitation in how this has all been implemented. Imagine that you want to bookmark a particular search or you want to send a link to friends that they can click in order to see the same filtered list of movies. Notice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

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

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



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



## Adding Search by Genre

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

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

public ActionResult SearchIndex(string movieGenre, string searchString)

{

var GenreLst = new List<string>();

var GenreQry = from d in db.Movies

orderby d.Genre

select d.Genre;

GenreLst.AddRange(GenreQry.Distinct());

ViewBag.movieGenre = new SelectList(GenreLst);

var movies = from m in db.Movies

select m;

if (!String.IsNullOrEmpty(searchString))

{

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

}

if (string.IsNullOrEmpty(movieGenre))

return View(movies);

else

{

return View(movies.Where(x => x.Genre == movieGenre));

}

}

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

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

var GenreQry = from d in db.Movies

orderby d.Genre

select d.Genre;

The code uses the [AddRange](http://msdn.microsoft.com/en-us/library/z883w3dc.aspx) method of the generic [List](http://msdn.microsoft.com/en-us/library/6sh2ey19.aspx) collection to add all the distinct genres to the list. (Without the Distinct modifier, duplicate genres would be added — for example, comedy would be added twice in our sample). The code then stores the list of genres in the ViewBag object.

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

if (string.IsNullOrEmpty(movieGenre))

return View(movies);

else

{

return View(movies.Where(x => x.Genre == movieGenre));

}

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

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

<p>

@Html.ActionLink("Create New", "Create")

@using (Html.BeginForm("SearchIndex","Movies",FormMethod.Get)){

<p>Genre: @Html.DropDownList("movieGenre", "All")

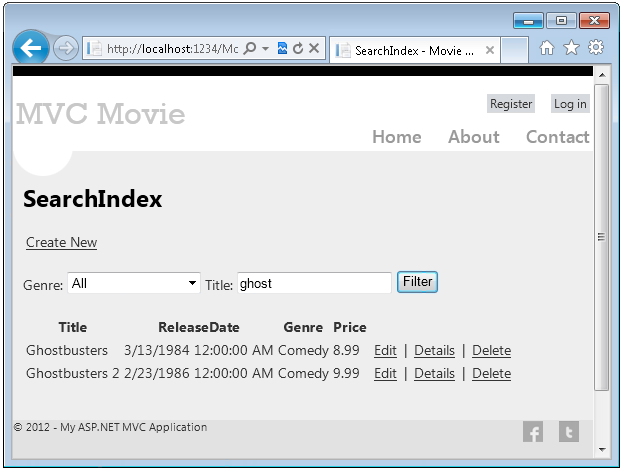
Title: @Html.TextBox("SearchString")

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

}

</p>

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



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

Adding a New Field to the Movie Model and Table

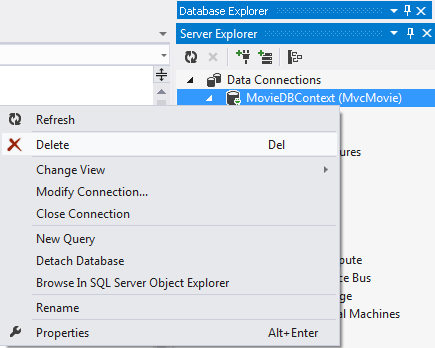
In this section you'll use Entity Framework Code First Migrations to migrate some changes to the model classes so the change is applied to the database.

By default, when you use Entity Framework Code First to automatically create a database, as you did earlier in this tutorial, Code First adds a table to the database to help track whether the schema of the database is in sync with the model classes it was generated from. If they aren't in sync, the Entity Framework throws an error. This makes it easier to track down issues at development time that you might otherwise only find (by obscure errors) at run time.

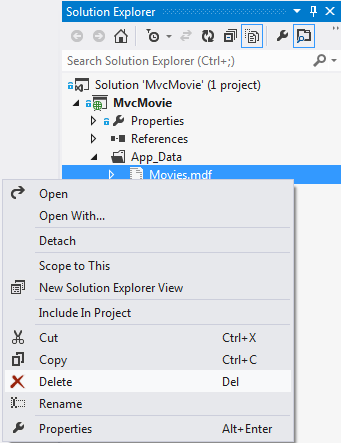
## Setting up Code First Migrations for Model Changes

If you are using Visual Studio 2012, double click the *Movies.mdf* file from Solution Explorer to open the database tool. Visual Studio Express for Web will show Database Explorer, Visual Studio 2012 will show Server Explorer. If you are using Visual Studio 2010, use SQL Server Object Explorer.

In the database tool ( Database Explorer, Server Explorer or SQL Server Object Explorer), right click on MovieDBContext and select **Delete** to drop the movies database.

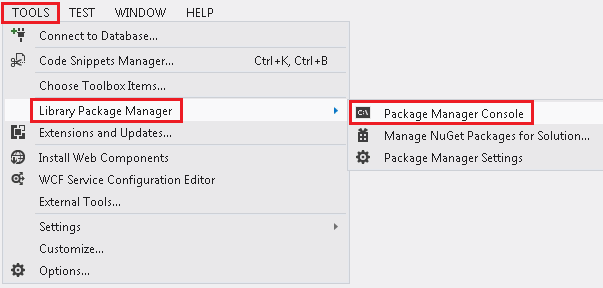


Navigate back to Solution Explorer. Right click on the *Movies.mdf* file and select **Delete** to remove the movies database.

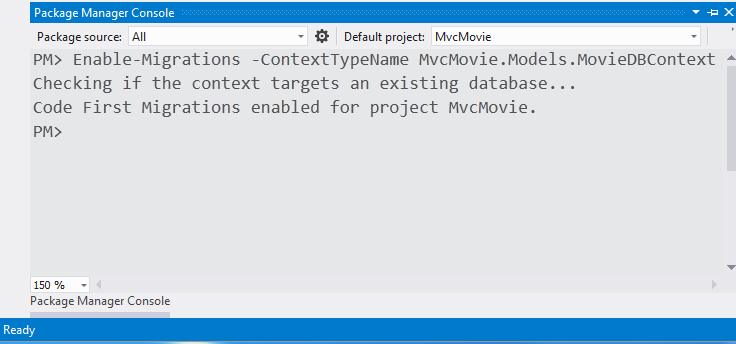


Build the application to make sure there are no errors.

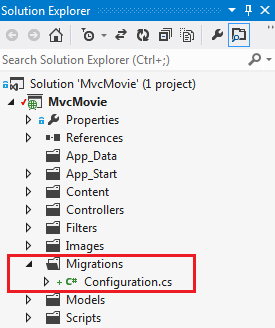
From the **Tools** menu, click **Library Package Manager** and then **Package Manager Console**.



In the **Package Manager Console** window at the PM> prompt enter "Enable-Migrations -ContextTypeName MvcMovie.Models.MovieDBContext".



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



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

protected override void Seed(MvcMovie.Models.MovieDBContext context)

{

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

new Movie

{

Title = "When Harry Met Sally",

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

Genre = "Romantic Comedy",

Price = 7.99M

},

new Movie

{

Title = "Ghostbusters ",

ReleaseDate = DateTime.Parse("1984-3-13"),

Genre = "Comedy",

Price = 8.99M

},

new Movie

{

Title = "Ghostbusters 2",

ReleaseDate = DateTime.Parse("1986-2-23"),

Genre = "Comedy",

Price = 9.99M

},

new Movie

{

Title = "Rio Bravo",

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

Genre = "Western",

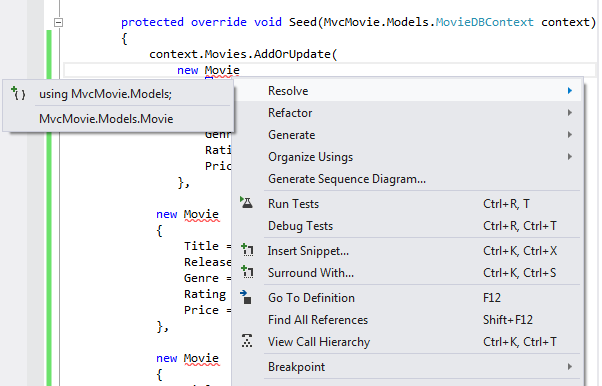
Price = 3.99M

}

);

}

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



Doing so adds the following using statement:

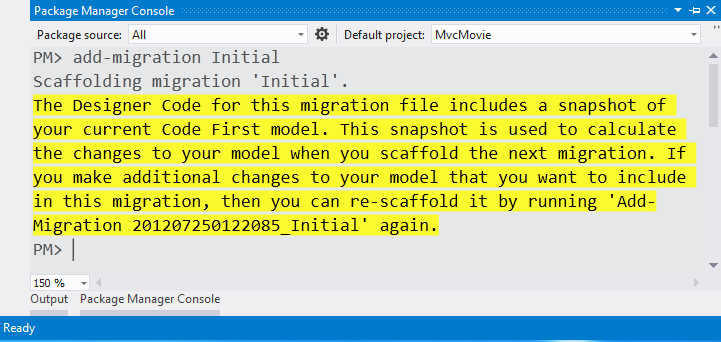
using MvcMovie.Models;

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

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

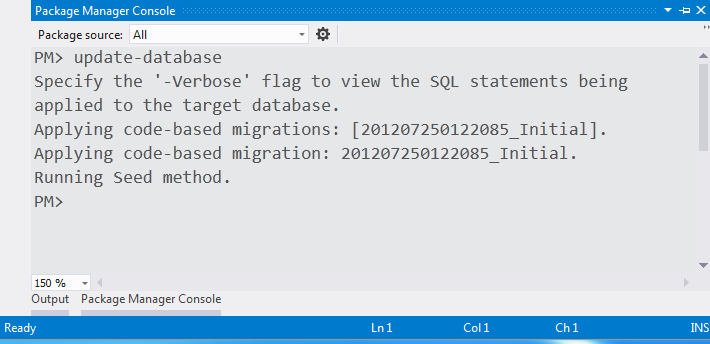
The next step is to create a DbMigration class for the initial migration. This migration to creates a new database, that's why you deleted the *movie.mdf* file in a previous step.

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



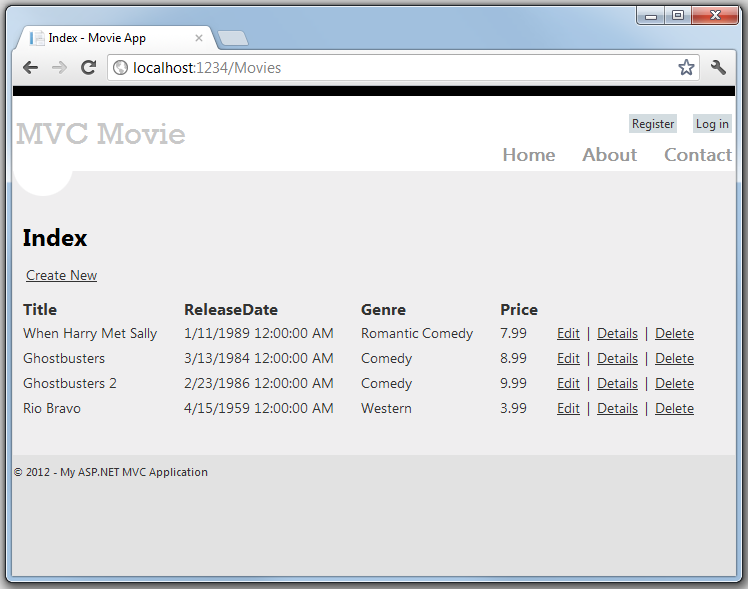
Code First Migrations creates another class file in the *Migrations* folder (with the name *{DateStamp}\_Initial.cs* ), and this class contains code that creates the database schema. The migration filename is pre-fixed with a timestamp to help with ordering.  Examine the *{DateStamp}\_Initial.cs*  file,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

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



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

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



## Adding a Rating Property to the Movie Model

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

public string Rating { get; set; }

The complete Movie class now looks like the following code:

public class Movie

{

public int ID { get; set; }

public string Title { get; set; }

public DateTime ReleaseDate { get; set; }

public string Genre { get; set; }

public decimal Price { get; set; }

public string Rating { get; set; }

}

Build the application using the **Build** > **Build Movie** menu command or by pressing CTRL-SHIFT-B.

Now that you've updated the Model class, you also need to update the *\Views\Movies\Index.cshtml* and *\Views\Movies\Create.cshtml* view templates in order to display the new Rating property in the browser view.

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

@model IEnumerable<MvcMovie.Models.Movie>

@{

ViewBag.Title = "Index";

}

<h2>Index</h2>

<p>

@Html.ActionLink("Create New", "Create")

</p>

<table>

<tr>

<th>

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

</th>

<th>

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

</th>

<th>

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

</th>

<th>

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

</th>

<th>

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

</th>

<th></th>

</tr>

@foreach (var item in Model) {

<tr>

<td>

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

</td>

<td>

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

</td>

<td>

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

</td>

<td>

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

</td>

<td>

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

</td>

<td>

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

@Html.ActionLink("Details", "Details", new { id=item.ID }) |

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

</td>

</tr>

}

</table>

Next, open the *\Views\Movies\Create.cshtml* file and add the following markup near the end of the form. This renders a text box so that you can specify a rating when a new movie is created.

<div class="editor-label">

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

</div>

<div class="editor-field">

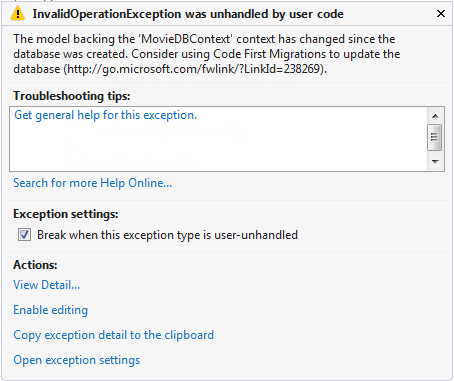
@Html.EditorFor(model => model.Rating)

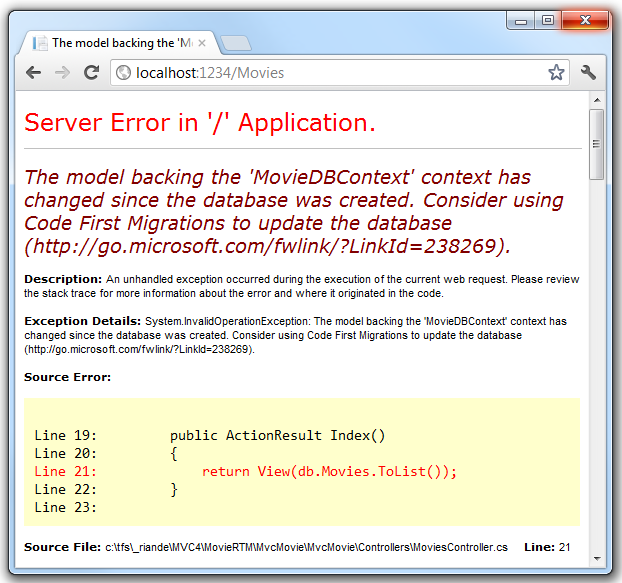
@Html.ValidationMessageFor(model => model.Rating)

</div>

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

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





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

There are a few approaches to resolving the error:

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

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

Update the Seed method so that it provides a value for the new column. Open Migrations\Configuration.cs file and add a Rating field to each Movie object.

new Movie

{

Title = "When Harry Met Sally",

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

Genre = "Romantic Comedy",

Rating = "G",

Price = 7.99M

},

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

add-migration AddRatingMig

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

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

public partial class AddRatingMig : DbMigration

{

public override void Up()

{

AddColumn("dbo.Movies", "Rating", c => c.String());

}

public override void Down()

{

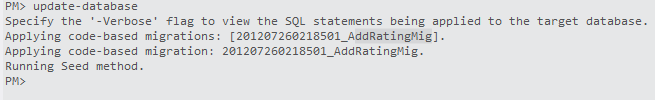
DropColumn("dbo.Movies", "Rating");

}

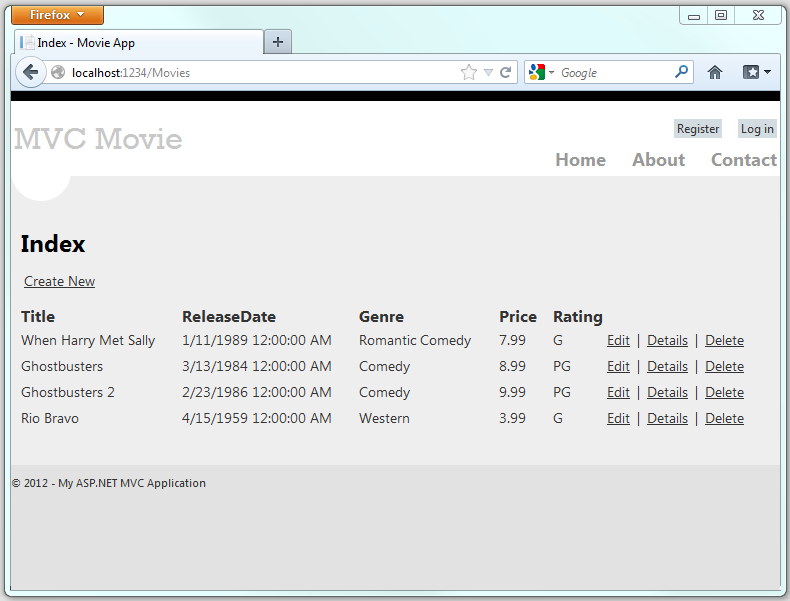
}

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

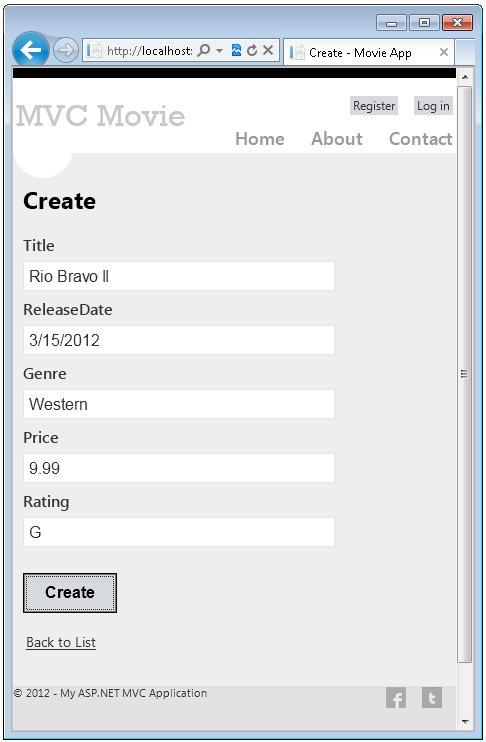
The following image shows the output in the **Package Manager Console** window (The date stamp prepending AddRatingMig will be different.)



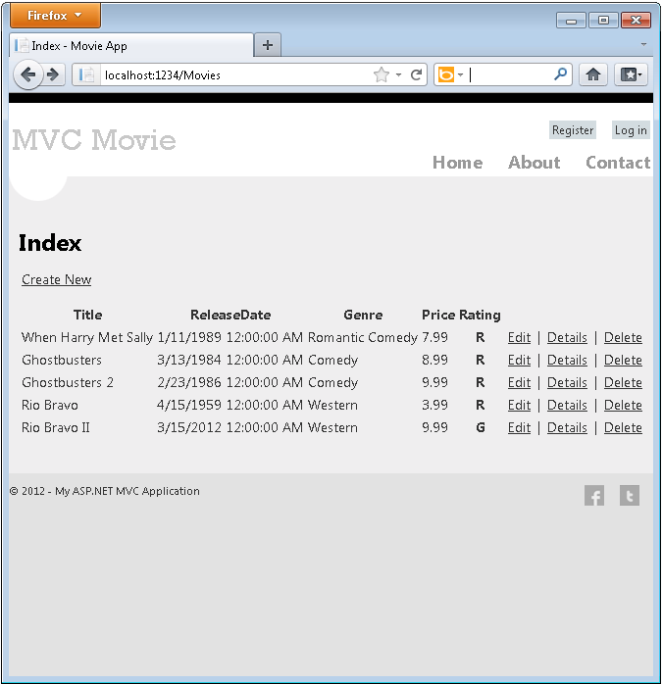
Re-run the application and navigate to the /Movies URL. You can see the new Rating field.



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



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



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

You could enter the "update-database" command in the **Package Manager Console** window again and no changes would be made, because the schema matches the model.

In this section you saw how you can modify model objects and keep the database in sync with the changes. You also learned a way to populate a newly created database with sample data so you can try out scenarios. Next, let's look at how you can add richer validation logic to the model classes and enable some business rules to be enforced.

Adding Validation to the Model

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

## Keeping Things DRY

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

The validation support provided by ASP.NET MVC and Entity Framework Code First is a great example of the DRY principle in action. You can declaratively specify validation rules in one place (in the model class) and the rules are enforced everywhere in the application.

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

## Adding Validation Rules to the Movie Model

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

Open the *Movie.cs* file. Add a using statement at the top of the file that references the [System.ComponentModel.DataAnnotations](http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.aspx) namespace:

using System.ComponentModel.DataAnnotations;

Notice the namespace does not contain System.Web. DataAnnotations provides a built-in set of validation attributes that you can apply declaratively to any class or property.

Now update the Movie class to take advantage of the built-in [Required](http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.requiredattribute.aspx), [StringLength](http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.stringlengthattribute.aspx), and [Range](http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.rangeattribute.aspx) validation attributes. Use the following code as an example of where to apply the attributes.

public class Movie {

public int ID { get; set; }

[Required]

public string Title { get; set; }

[DataType(DataType.Date)]

public DateTime ReleaseDate { get; set; }

[Required]

public string Genre { get; set; }

[Range(1, 100)]

[DataType(DataType.Currency)]

public decimal Price { get; set; }

[StringLength(5)]

public string Rating { get; set; }

}

Run the application and you will again get the following run time error:

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

We will us migrations to update the scheam. Build the solution, and then open the **Package Manager Console** window and enter the following commands:

add-migration AddDataAnnotationsMig  
update-database

When this command finishes, Visual Studio opens the class file that defines the new DbMIgration derived class with the name specified (*AddDataAnnotationsMig*), and in the Up method you can see the code that updates the schema constraints. The Title and Genre fields are no longer nullable (that is, you must enter a value) and the Rating field has a maximum length of 5.

The validation attributes specify behavior that you want to enforce on the model properties they are applied to. The [Required](http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.requiredattribute(VS.110).aspx) attribute indicates that a property must have a value; in this sample, a movie has to have values for the Title, ReleaseDate, Genre, and Price properties 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 string property, and optionally its minimum length. Intrinsic types (such as decimal, int, float, DateTime) are required by default and don't need the Required attribute.

Code First ensures that the validation rules you specify on a model class are enforced before the application saves changes in the database. For example, the code below will throw 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 server side validation exception

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

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

using System;

using System.Data.Entity;

using System.ComponentModel.DataAnnotations;

namespace MvcMovie.Models {

public class Movie {

public int ID { get; set; }

[Required]

public string Title { get; set; }

[DataType(DataType.Date)]

public DateTime ReleaseDate { get; set; }

[Required]

public string Genre { get; set; }

[Range(1, 100)]

[DataType(DataType.Currency)]

public decimal Price { get; set; }

[StringLength(5)]

public string Rating { get; set; }

}

public class MovieDBContext : DbContext {

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

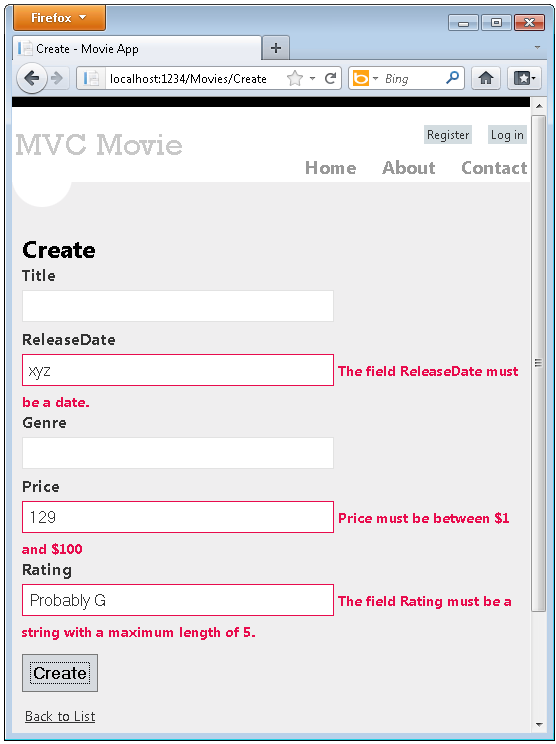
}

}

## Validation Error UI in ASP.NET MVC

Re-run the application and navigate to the */Movies* URL.

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



**Note** to support jQuery validation for non-English locales that use a comma (",") for a decimal point, you must include *globalize.js* and your specific *cultures/globalize.cultures.js* file(from <https://github.com/jquery/globalize> ) and JavaScript to use Globalize.parseFloat. The following code shows the modifications to the Views\Movies\Edit.cshtml file to work with the "fr-FR" culture:

@section Scripts {

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

<script src="~/Scripts/globalize.js"></script>

<script src="~/Scripts/globalize.culture.fr-FR.js"></script>

<script>

$.validator.methods.number = function (value, element) {

return this.optional(element) ||

!isNaN(Globalize.parseFloat(value));

}

$(document).ready(function () {

Globalize.culture('fr-FR');

});

</script>

<script>

jQuery.extend(jQuery.validator.methods, {

range: function (value, element, param) {

//Use the Globalization plugin to parse the value

var val = $.global.parseFloat(value);

return this.optional(element) || (

val >= param[0] && val <= param[1]);

}

});

</script>

}

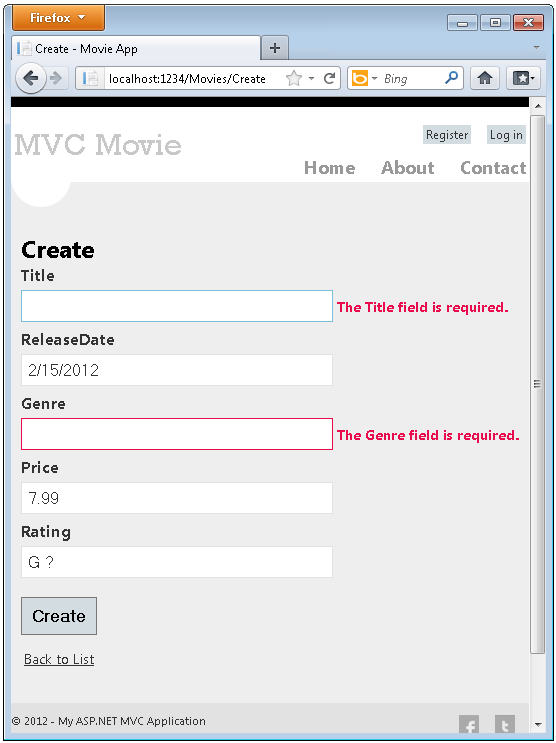
Notice how the form has automatically used a red border color to highlight the text boxes that contain invalid data and has emitted an appropriate validation error message next to each one. The errors are enforced both client-side (using JavaScript and jQuery) and server-side (in case a user has JavaScript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

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

You might have noticed for the properties Title and Genre, the required attribute is not enforced until you submit the form (hit the **Create** button), or enter text into the input field and removed it. For a field which is initially empty (such as the fields on the Create view) and which has only the required attribute and no other validation attributes, you can do the following to trigger validation:

1. Tab into the field.
2. Enter some text.
3. Tab out.
4. Tab back into the field.
5. Remove the text.
6. Tab out.

The above sequence will trigger the required validation without hitting the submit button. Simply hitting the submit button without entering any of the fields will trigger client side validation. The form data is not sent to the server until there are no client side validation errors. You can test this by putting a break point in the HTTP Post method or using the [fiddler tool](http://fiddler2.com/fiddler2/) or the IE 9 [F12 developer tools](http://msdn.microsoft.com/en-us/ie/aa740478).



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

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

//

// GET: /Movies/Create

public ActionResult Create()

{

return View();

}

//

// POST: /Movies/Create

[HttpPost]

public ActionResult Create(Movie movie)

{

if (ModelState.IsValid)

{

db.Movies.Add(movie);

db.SaveChanges();

return RedirectToAction("Index");

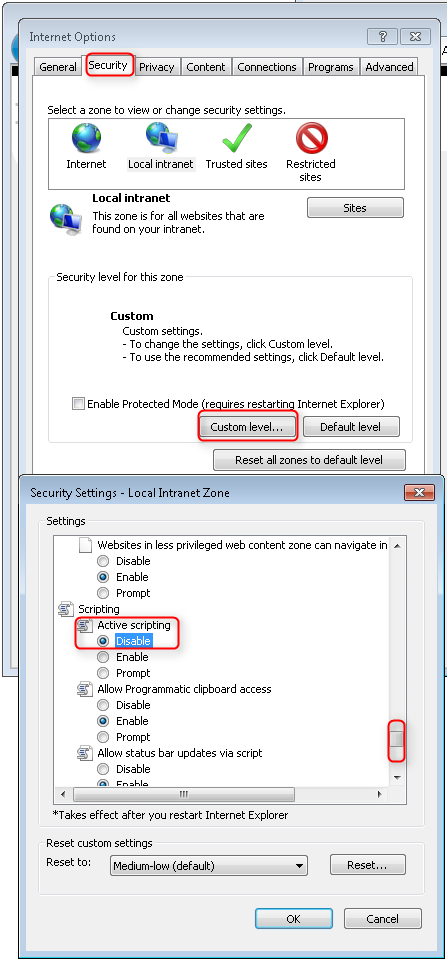
}

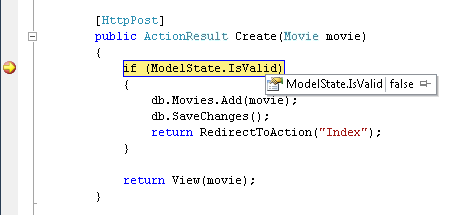
return View(movie);

}

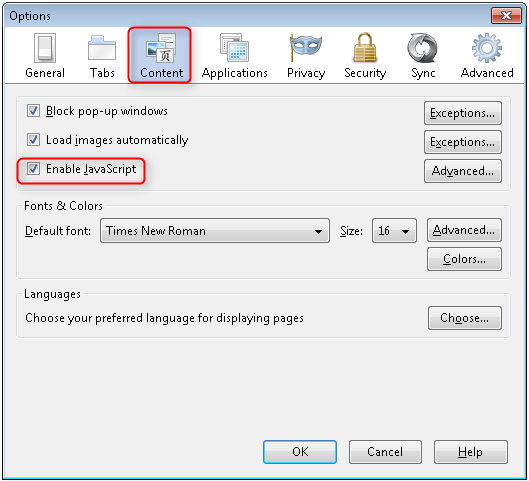
The first (HTTP GET) Create action method displays the initial Create form. The second ([HttpPost]) version handles the form post. The second Create method (The HttpPost version) calls ModelState.IsValid to check whether the movie has any validation errors. Calling this method evaluates any validation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

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

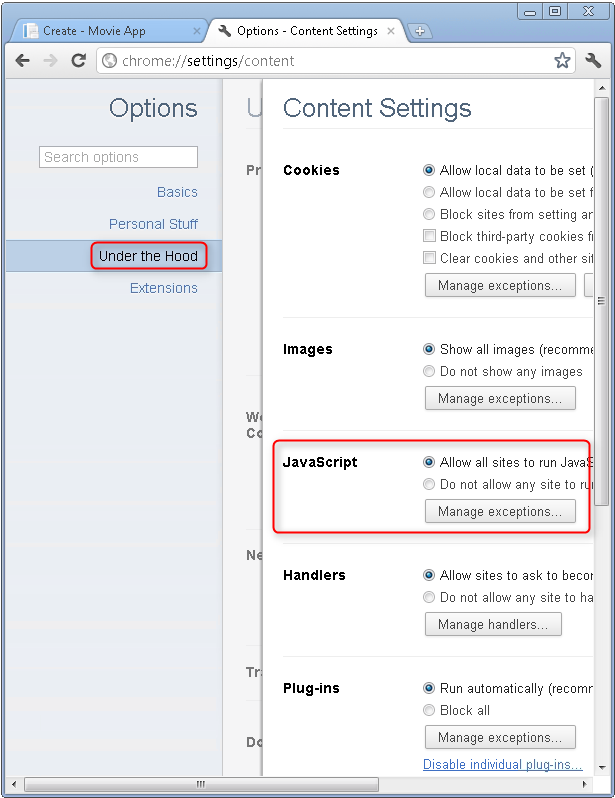




The following image shows how to disable JavaScript in the FireFox browser.



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



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

@model MvcMovie.Models.Movie

@{

ViewBag.Title = "Create";

}

<h2>Create</h2>

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

<script src="@Url.Content("~/Scripts/jquery.validate.unobtrusive.min.js")"></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 the code uses an Html.EditorFor helper to output the <input> element for each Movie property. Next to this helper is a call to the Html.ValidationMessageFor helper method. These two helper methods work with the model object that's passed by the controller to the view (in this case, a Movie object). They automatically look for validation attributes specified on the model and display error messages as appropriate.

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

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

## Adding Formatting to the Movie Model

Open the *Movie.cs* file and examine the Movie class. The [System.ComponentModel.DataAnnotations](http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.aspx) namespace provides formatting attributes in addition to the built-in set of validation attributes. We've already applied  a [DataType](http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.datatype.aspx) enumeration value to the release date and to the price fields. The following code shows the ReleaseDate and Price properties with the appropriate [DisplayFormat](http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.displayformatattribute.aspx) attribute.

[DataType(DataType.Date)]

public DateTime ReleaseDate { get; set; }

[DataType(DataType.Currency)]

public decimal Price { get; set; }

The [DataType](http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.datatype.aspx) attributes are not validation attributes, they are used to tell the view engine how to render the HTML. In the example above, the DataType.Date attribute displays the movie dates as dates only, with out time. For example, the following [DataType](http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.datatype.aspx) attributes don't validate the format of the data:

[DataType(DataType.EmailAddress)]

[DataType(DataType.PhoneNumber)]

[DataType(DataType.Url)]

The attributes listed above only provide hints for the view engine to format the data (and supply attributes such as <a> for URL's and <a href="mailto:EmailAddress.com"> for email. You can use the [RegularExpression](http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.regularexpressionattribute.aspx) attribute to validate the format of the data.

An alternative approach to using the DataType attributes, you could explicitly set a [DataFormatString](http://msdn.microsoft.com/en-us/library/system.string.format.aspx) value. The following code shows the release date property with a date format string (namely, "d"). You'd use this to specify that you don't want to time as part of the release date.

[DisplayFormat(DataFormatString = "{0:d}")]

public DateTime ReleaseDate { get; set; }

The following code formats the Price property as currency.

[DisplayFormat(DataFormatString = "{0:c}")]

public decimal Price { get; set; }

The complete Movie class is shown below.

public class Movie {

public int ID { get; set; }

[Required]

public string Title { get; set; }

[DataType(DataType.Date)]

public DateTime ReleaseDate { get; set; }

[Required]

public string Genre { get; set; }

[Range(1, 100)]

[DataType(DataType.Currency)]

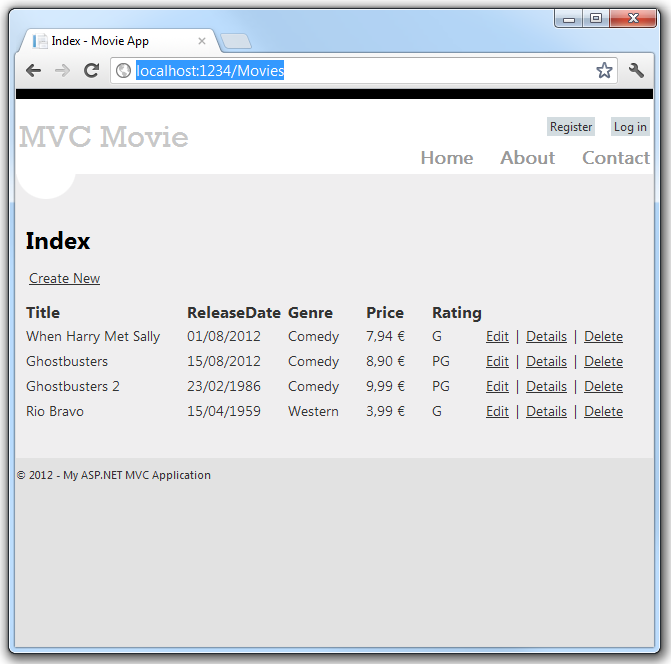
public decimal Price { get; set; }

[StringLength(5)]

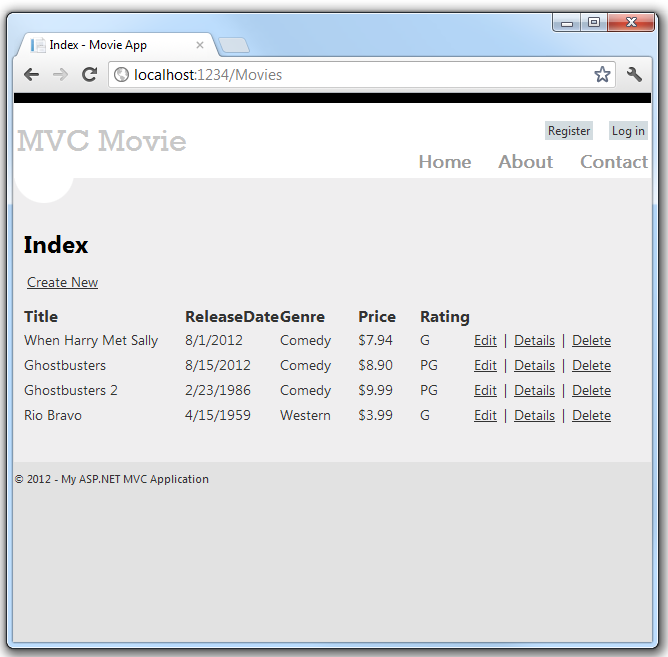
public string Rating { get; set; }

}

 Run the application and browse to the Movies controller. The release date and price are nicely formatted. The image below shows the release date and price using "fr-FR" as the culture.



The image below shows the same data displayed with the default culture (English US).



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

Examining the Details and Delete Methods

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

## Examining the Details and Delete Methods

Open the Movie controller and examine the Details method.

public ActionResult Details(int id = 0)

{

Movie movie = db.Movies.Find(id);

if (movie == null)

{

return HttpNotFound();

}

return View(movie);

}

Code First makes it easy to search for data using the Find method. An important security feature built into the method is that the code verifies that the Find method has found a movie before the code tries to do anything with it. For example, a hacker could introduce errors into the site by changing the URL created by the links from *http://localhost:xxxx/Movies/Details/1* to something like *http://localhost:xxxx/Movies/Details/12345* (or some other value that doesn't represent an actual movie). If you did not check for a null movie, a null movie would result in a database error.

Examine the Delete and DeleteConfirmed methods.

// GET: /Movies/Delete/5

public ActionResult Delete(int id = 0)

{

Movie movie = db.Movies.Find(id);

if (movie == null)

{

return HttpNotFound();

}

return View(movie);

}

//

// POST: /Movies/Delete/5

[HttpPost, ActionName("Delete")]

public ActionResult DeleteConfirmed(int id = 0)

{

Movie movie = db.Movies.Find(id);

if (movie == null)

{

return HttpNotFound();

}

db.Movies.Remove(movie);

db.SaveChanges();

return RedirectToAction("Index");

}

Note that the HTTP Get Delete method doesn't delete the specified movie, it returns a view of the movie where you can submit (HttpPost) the deletion.. Performing a delete operation in response to a GET request (or for that matter, performing an edit operation, create operation, or any other operation that changes data) opens up a security hole. For more information about this, see Stephen Walther's blog entry [ASP.NET MVC Tip #46 — Don't use Delete Links because they create Security Holes](http://stephenwalther.com/blog/archive/2009/01/21/asp.net-mvc-tip-46-ndash-donrsquot-use-delete-links-because.aspx).

The HttpPost method that deletes the data is named DeleteConfirmed to give the HTTP POST method a unique signature or name. The two method signatures are shown below:

// GET: /Movies/Delete/5

public ActionResult Delete(int id = 0)

//

// POST: /Movies/Delete/5

[HttpPost, ActionName("Delete")]

public ActionResult DeleteConfirmed(int id = 0)

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

To sort this out, you can do a couple of things. One is to give the methods different names. That's what the scaffolding mechanism did in he preceding example. However, this introduces a small problem: ASP.NET maps segments of a URL to action methods by name, and if you rename a method, routing normally wouldn't be able to find that method. The solution is what you see in the example, which is to add the ActionName("Delete") attribute to the DeleteConfirmed method. This effectively performs mapping for the routing system so that a URL that includes */Delete/* for a POST request will find the DeleteConfirmed method.

Another common way to avoid a problem with methods that have identical names and signatures is to artificially change the signature of the POST method to include an unused parameter. For example, some developers add a parameter type [FormCollection](http://msdn.microsoft.com/en-us/library/system.web.mvc.formcollection.aspx) that is passed to the POST method, and then simply don't use the parameter:

public ActionResult Delete(FormCollection fcNotUsed, int id = 0)

{

Movie movie = db.Movies.Find(id);

if (movie == null)

{

return HttpNotFound();

}

db.Movies.Remove(movie);

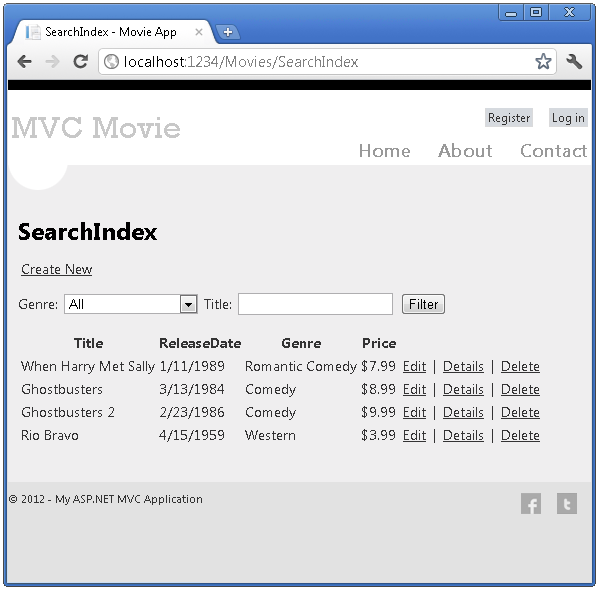
db.SaveChanges();

return RedirectToAction("Index");

}

## Summary

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



## Next Steps

After you have built and tested a web application, the next step is to make it available to other other people to use over the Internet. To do that, you have to deploy it to a web hosting provider. Microsoft offers free web hosting for up to 10 web sites in a [free Windows Azure trial account](http://www.windowsazure.com/en-us/pricing/free-trial/?WT.mc_id=A443DD604). For information about how to deploy a Visual Studio web project to a Windows Azure Web Site, see [Create and deploy an ASP.NET web site and SQL Database with Visual Studio](http://www.windowsazure.com/en-us/develop/net/compute/). That tutorial also shows how to use Entity Framework Code First Migrations to deploy your SQL Server database to Windows Azure SQL Database (formerly SQL Azure). For more information about deployment, see [ASP.NET Web Deployment Content Map](http://go.microsoft.com/fwlink/p/?LinkId=282413).

I now encourage you to move on to our intermediate-level [Creating an Entity Framework Data Model for an ASP.NET MVC Application](http://www.asp.net/entity-framework/tutorials/creating-an-entity-framework-data-model-for-an-asp-net-mvc-application) and [MVC Music Store](http://www.asp.net/mvc/tutorials/mvc-music-store-part-1) tutorials, to explore the [ASP.NET articles on MSDN](http://msdn.microsoft.com/en-us/library/gg416514(VS.98).aspx), and to check out the many videos and resources at <http://asp.net/mvc> to learn even more about ASP.NET MVC! The [ASP.NET MVC forums](http://forums.asp.net/1146.aspx) are a great place to ask questions.