ASP.NET MVC Music Store Tutorial

Version 2.0

Jon Galloway - Microsoft

1/12/2010

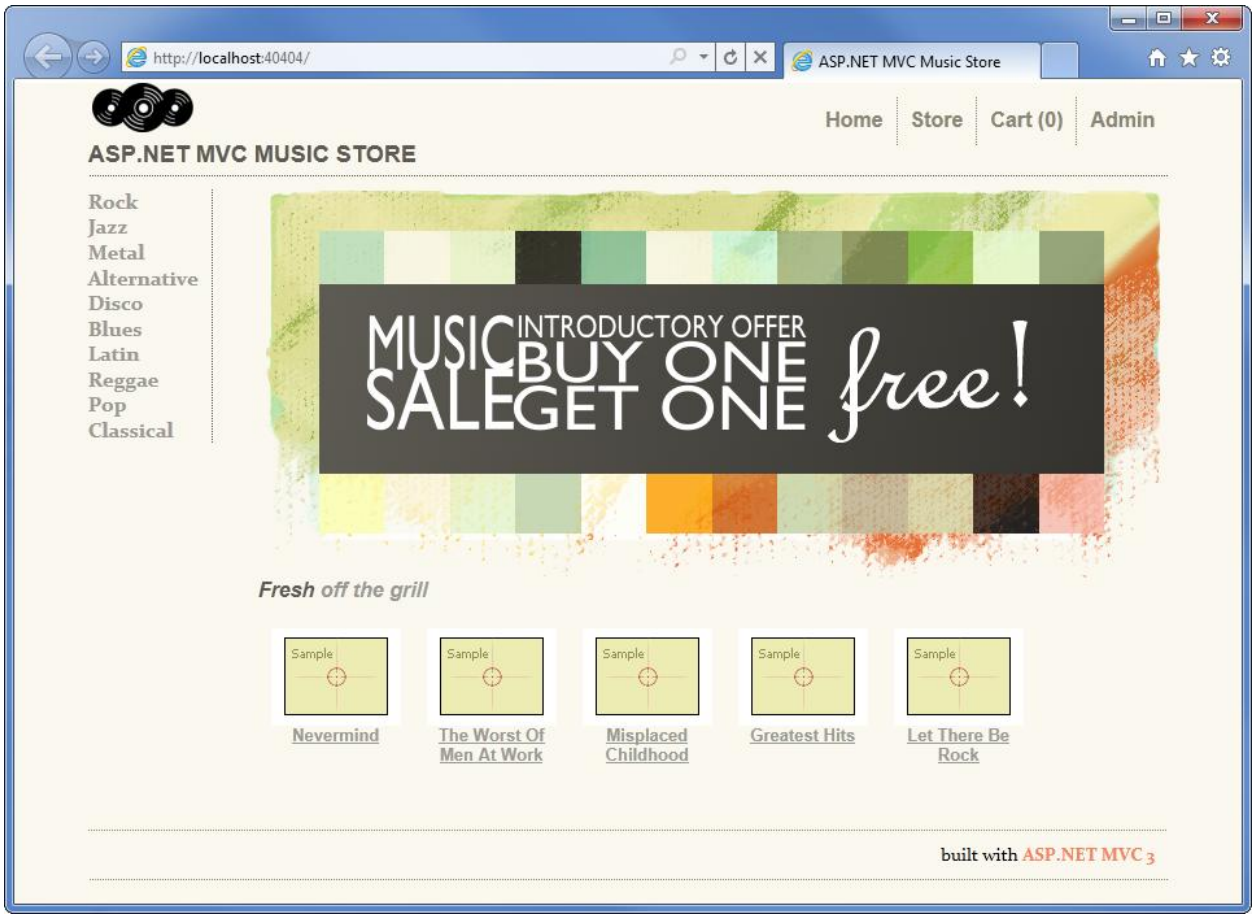
ASP.NET MVC Music Store Tutorial

Contents

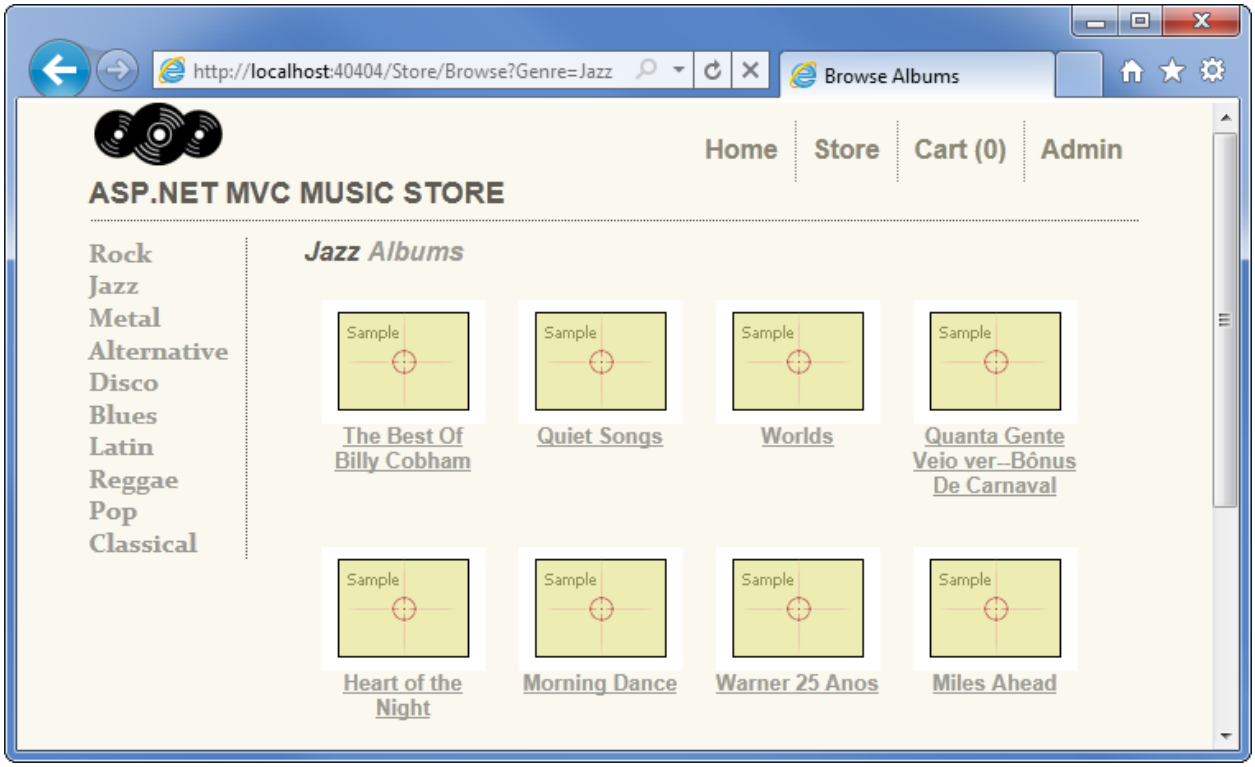
Overview

The MVC Music Store is a tutorial application that introduces and explains step-by-step how to use ASP.NET MVC and Visual Studio for web development. We'll be starting slowly, so beginner level web development experience is okay.

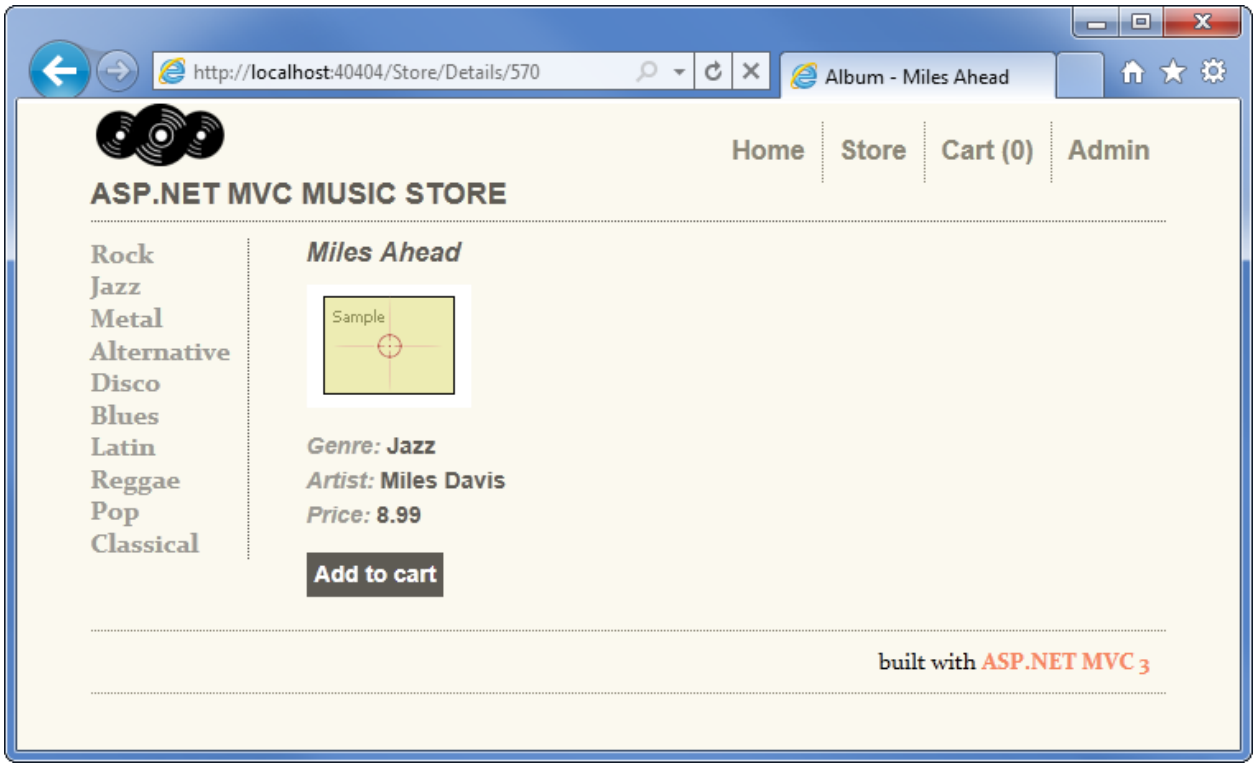
The application we'll be building is a simple music store. There are three main parts to the application: shopping, checkout, and administration.



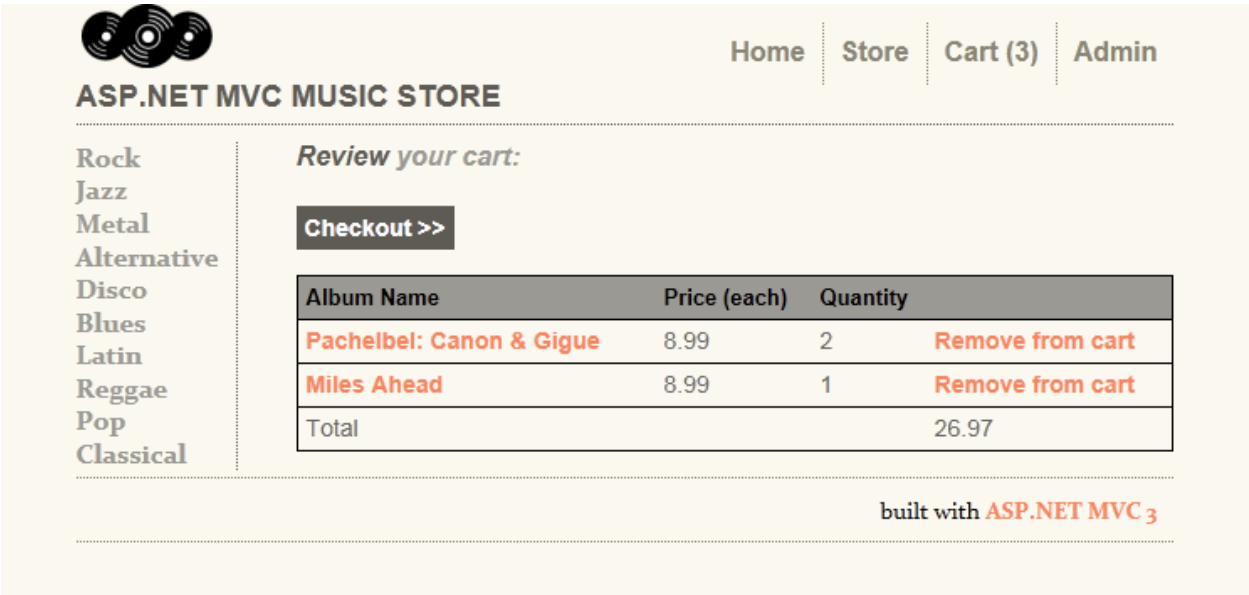
Visitors can browse Albums by Genre:



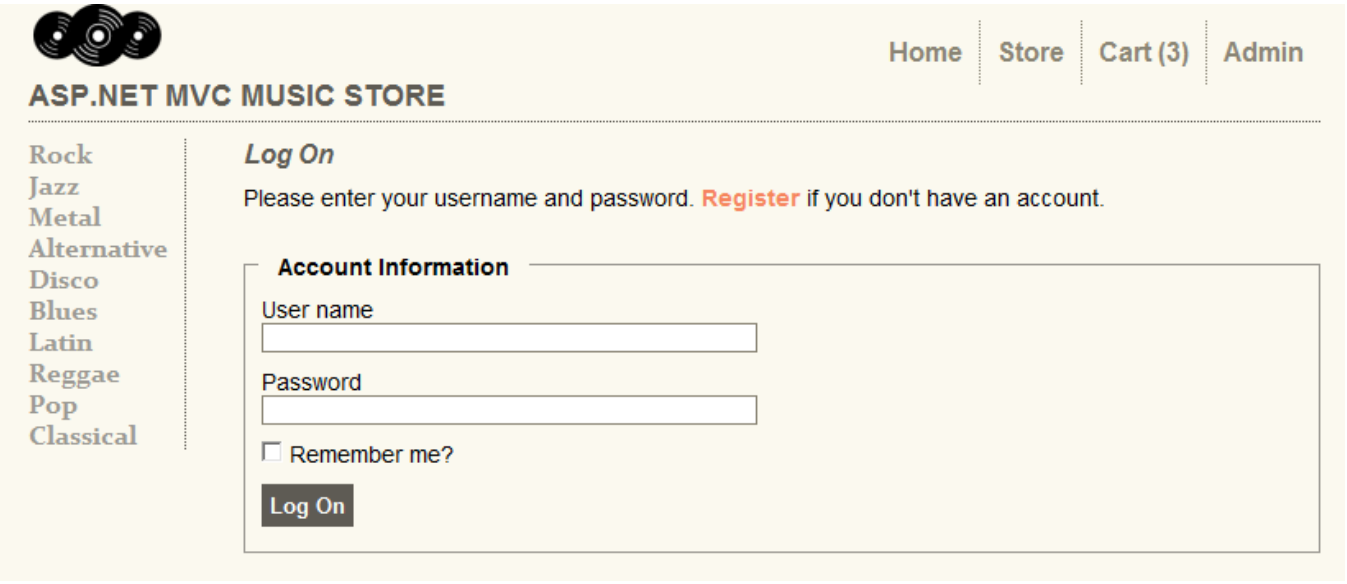
They can view a single album and add it to their cart:

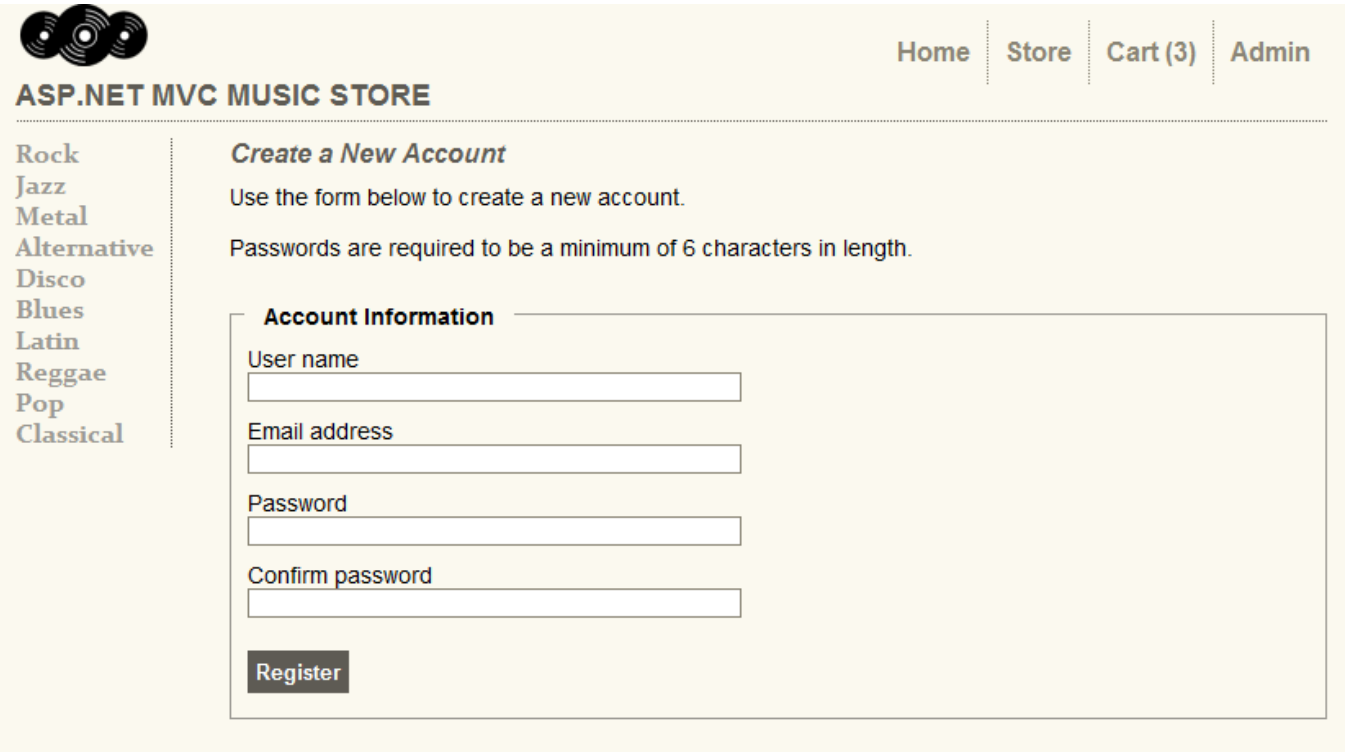


They can review their cart, removing any items they no longer want:

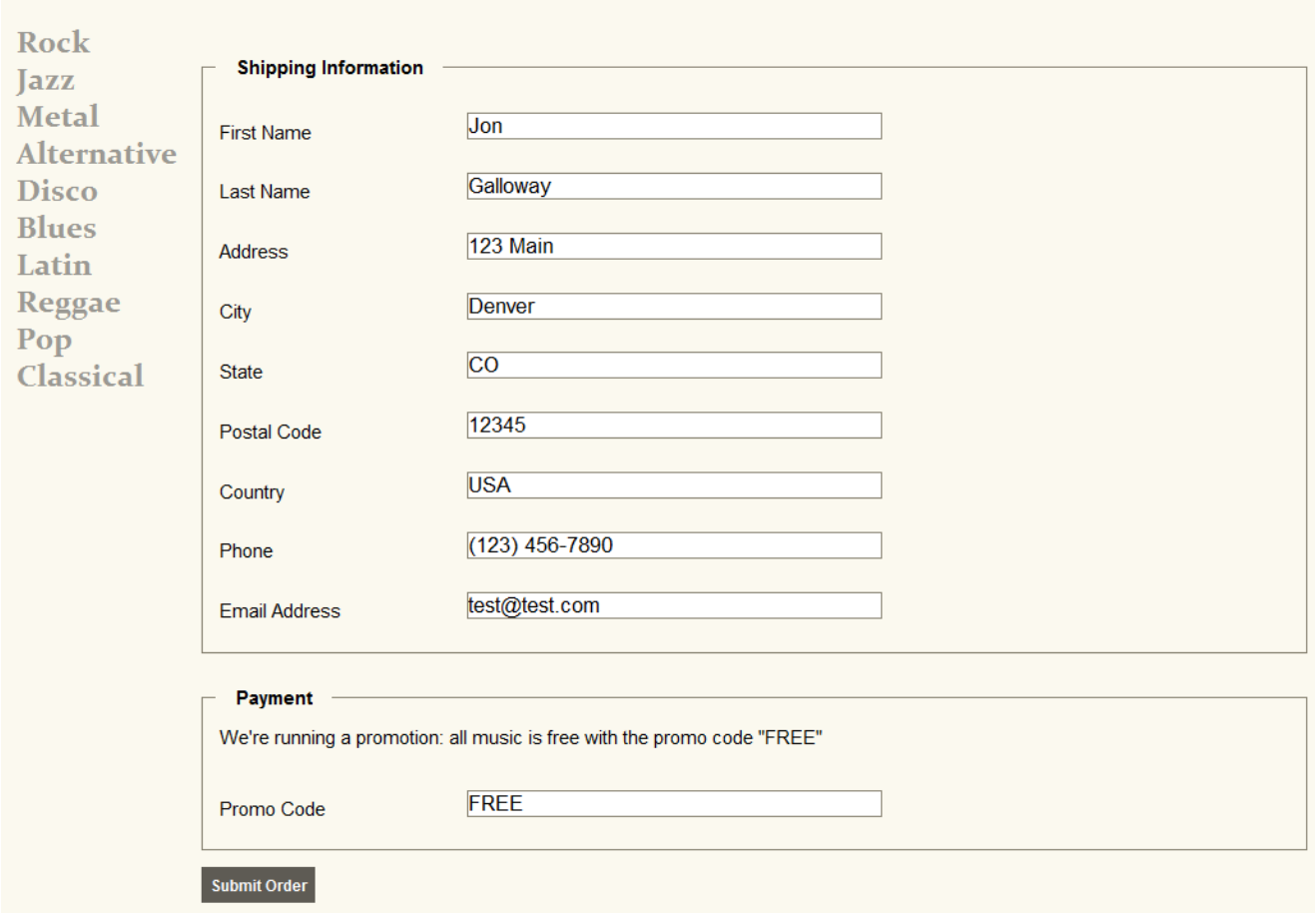


Proceeding to Checkout will prompt them to login or register for a user account.

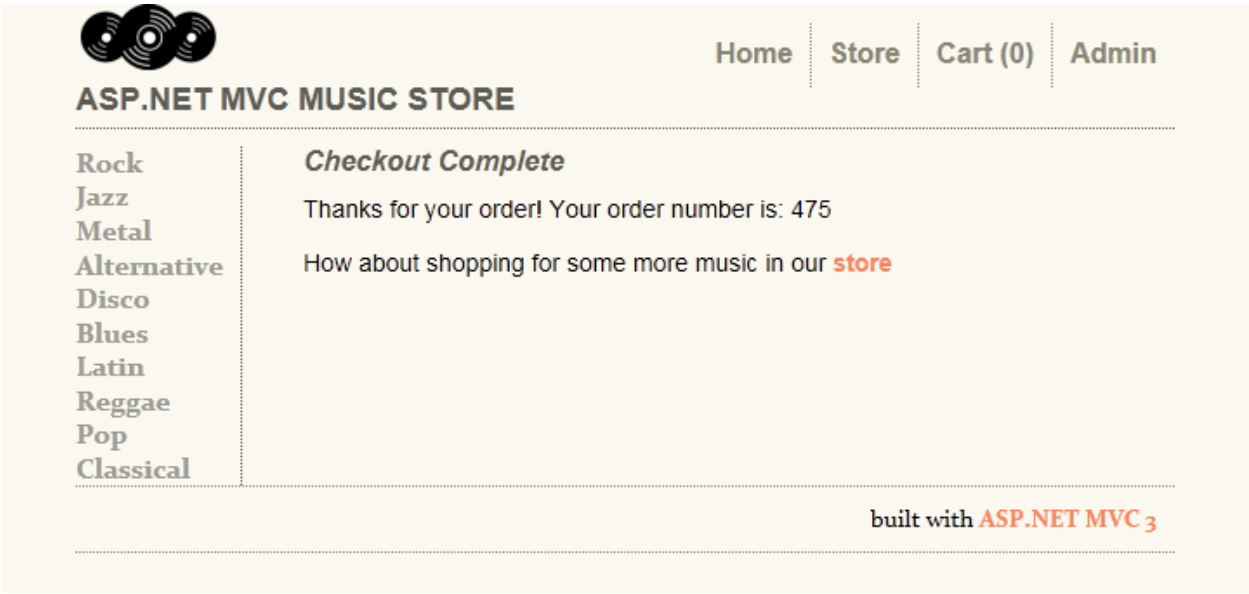




After creating an account, they can complete the order by filling out shipping and payment information. To keep things simple, we're running an amazing promotion: everything's free if they enter promotion code "FREE"!



After ordering, they see a simple confirmation screen:



In addition to customer-faceing pages, we'll also build an administrator section that shows a list of albums from which Administrators can Create, Edit, and Delete albums:

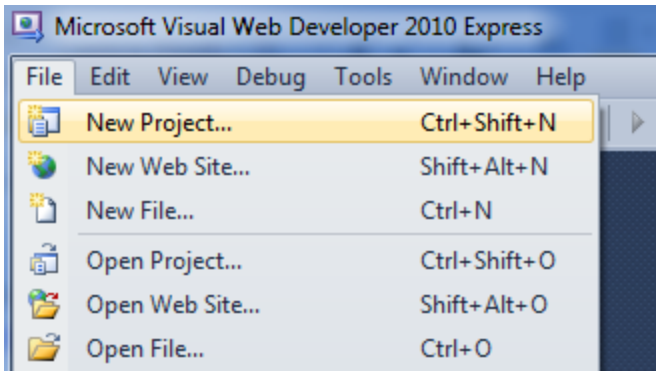
This tutorial will begin by creating a new ASP.NET MVC 2 project using the free Visual Web Developer 2010 Express (which is free), and then we'll incrementally add features to create a complete functioning application. Along the way, we'll cover database access, form posting scenarios,, data validation, using master pages for consistent page layout, using AJAX for page updates and validation, user login, and more.

You can follow along step by step, or you can download the completed application from http://mvcmusicstore.codeplex.com.

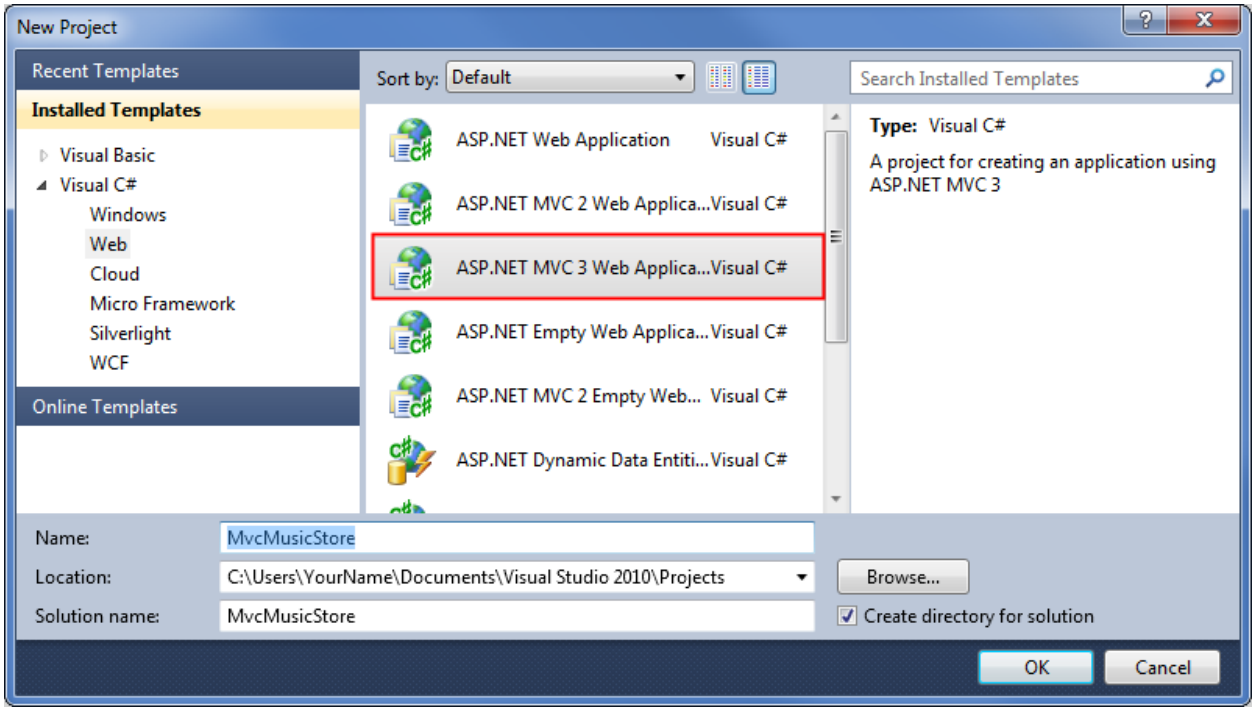
You can use either Visual Studio 2010 or the free Visual Web Developer 2010 Express to build the application. We'll be using the free SQL Server Express to host the database. You can install ASP.NET MVC, Visual Web Developer Express and SQL Server Express using a simple installer here: http://www.asp.net/downloads

1. File -> New Project

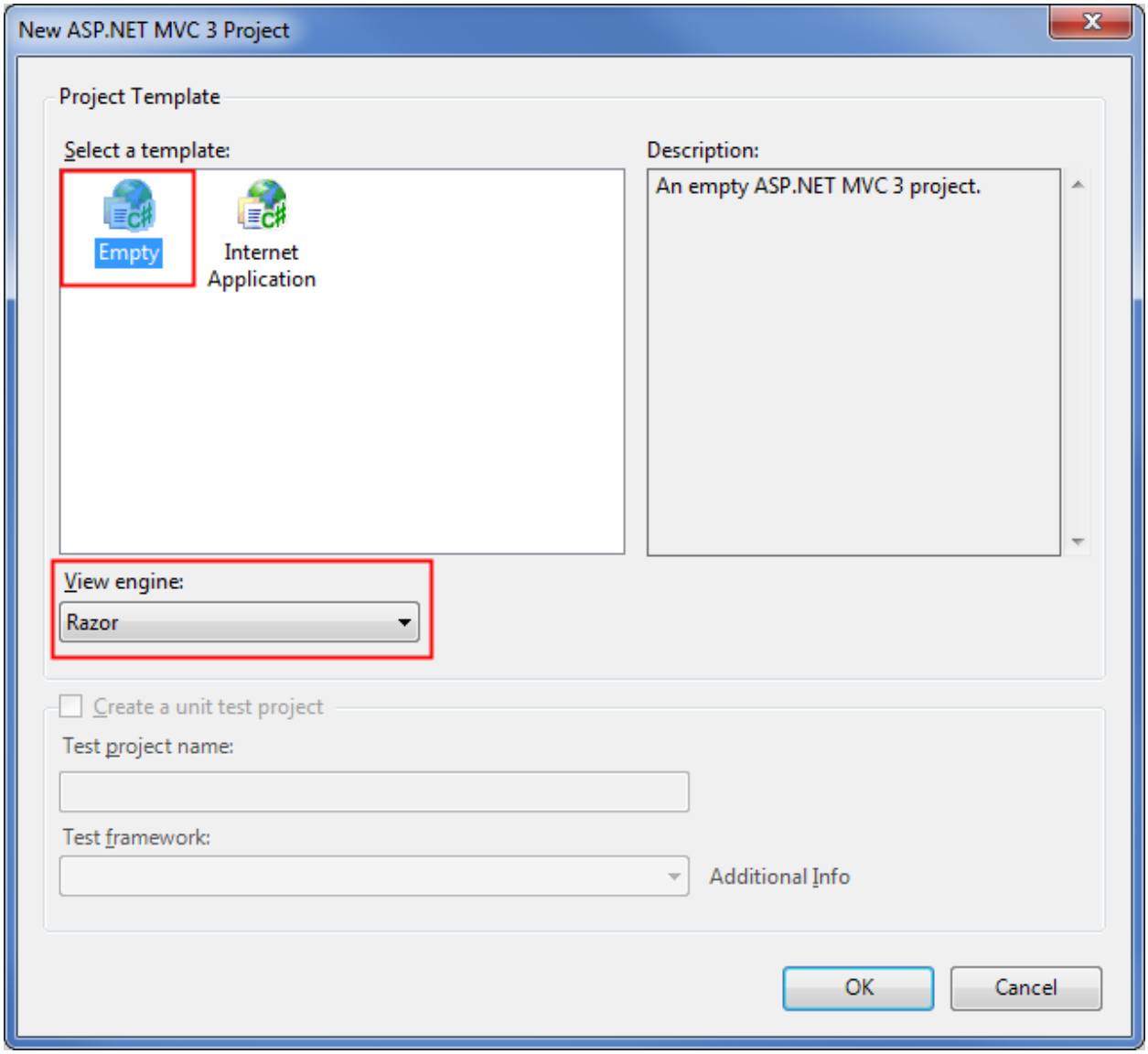
We'll start by selecting "New Project" from the File menu in Visual Web Developer. This brings up the New Project dialog.



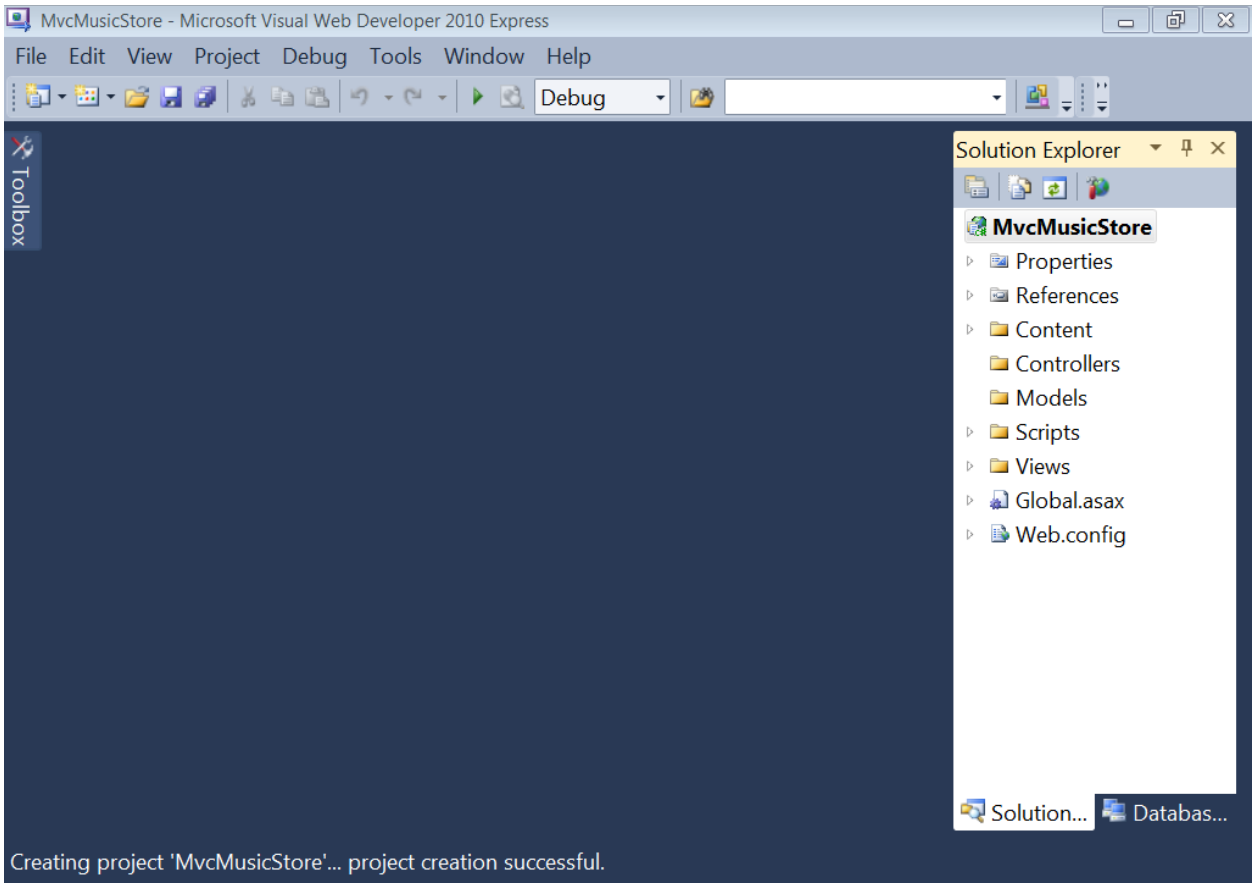
We'll select the Visual C# -> Web Templates group on the left, then choose the "ASP.NET MVC 3 Web Application" template in the center column. Name your project MvcMusicStore and press the OK button.



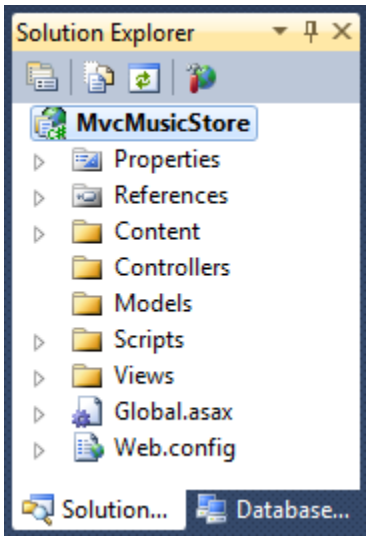
This will display a secondary dialog which allows us to make some MVC specific settings for our project. Ensure the "Empty" Project Template is selected and the View Engine is set to Razor as shown below, then press the OK button.



This will create our project. Let's take a look at the folders that have been added to our application in the Solution Explorer on the right side.



The Empty MVC 3 template isn't completely empty – it adds a basic folder structure:



ASP.NET MVC makes use of some basic naming conventions for folder names:

Folder	Purpose
/Controllers	Controllers respond to input from the browser, decide what to do with it, and return response to the user.
/Views	Views hold our UI templates

/Models	Models hold and manipulate data
/Content	This folder holds our images, CSS, and any other static content
/Scripts	This folder holds our JavaScript files
/App_Data	This folder holds our database files

These folders are included even in an Empty ASP.NET MVC application because the ASP.NET MVC framework by default uses a "convention over configuration" approach and makes some default assumptions based on folder naming conventions. For instance, controllers look for views in the Views folder by default without you having to explicitly specify this in your code. Sticking with the default conventions reduces the amount of code you need to write, and can also make it easier for other developers to understand your project. We'll explain these conventions more as we build our application.

2. Controllers

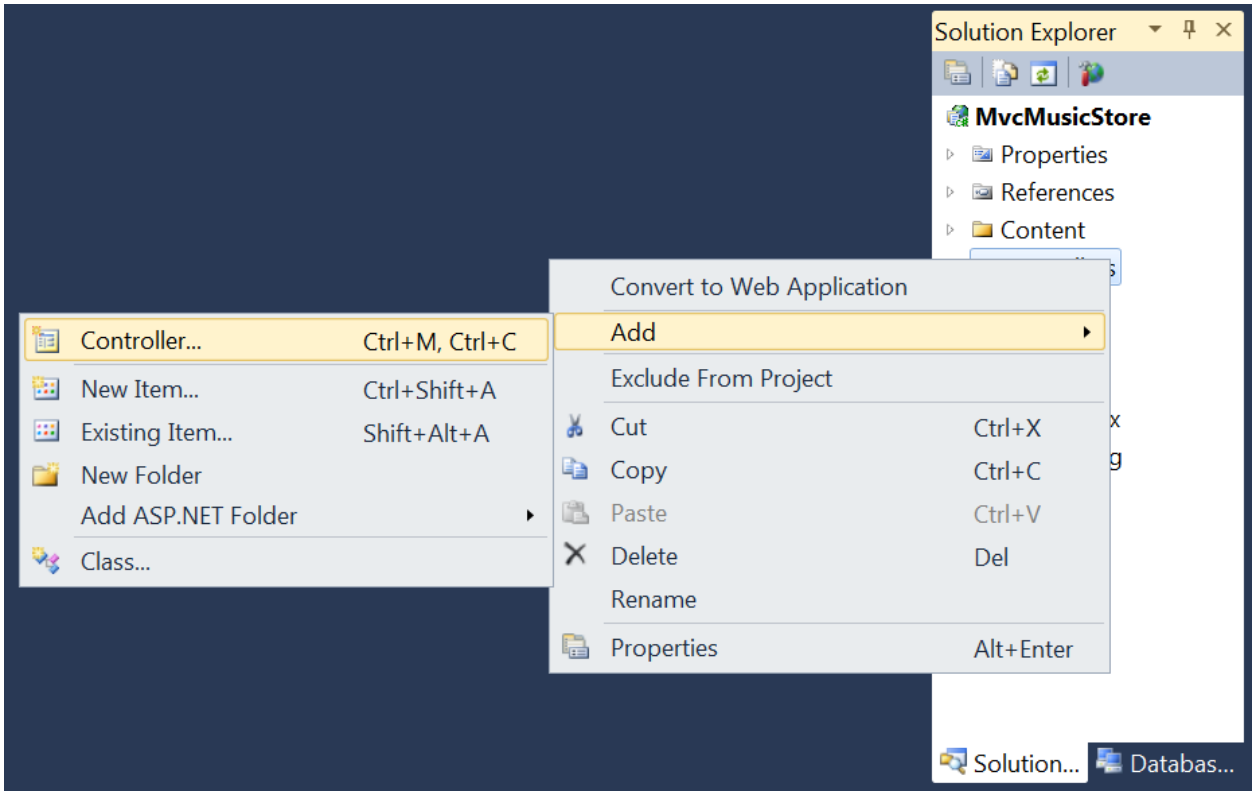
With traditional web frameworks, incoming URLs are typically mapped to files on disk. For example: a request for a URL like "/Products.aspx" or "/Products.php" might be processed by a "Products.aspx" or "Products.php" file.

Web-based MVC frameworks map URLs to server code in a slightly different way. Instead of mapping incoming URLs to files, they instead map URLs to methods on classes. These classes are called "Controllers" and they are responsible for processing incoming HTTP requests, handling user input, retrieving and saving data, and determining the response to send back to the client (display HTML, download a file, redirect to a different URL, etc.).

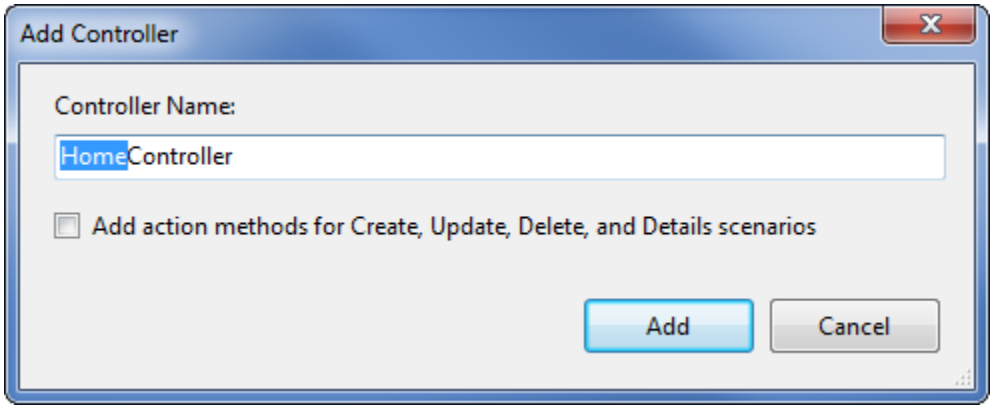
Adding a HomeController

We'll begin our MVC Music Store application by adding a Controller class that will handle URLs to the Home page of our site. We'll follow the default naming conventions of ASP.NET MVC and call it HomeController.

Right-click the "Controllers" folder within the Solution Explorer and select "Add", and then the "Controller…" command:



This will bring up the "Add Controller" dialog. Name the controller "HomeController" and press the Add button.



This will create a new file, HomeController.cs, with the following code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace MvcMusicStore.Controllers
{
    public class HomeController : Controller
```

```
    {
        //
        // GET: /Home/

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

    }
}
```

To start as simply as possible, let's replace the Index method with a simple method that just returns a string. We'll make two simple changes:

- Change the method to return a string instead of an ActionResult
- Change the return statement to return "Hello from Home"

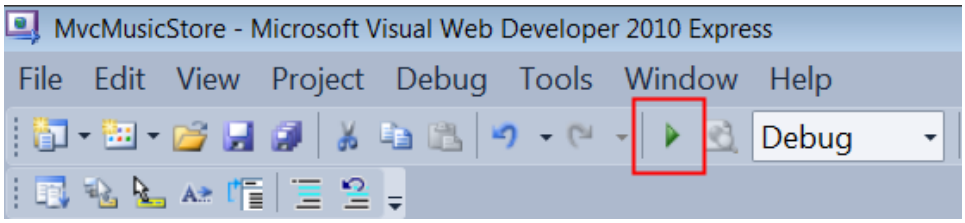
The method should now look like this:

```
public string Index()
{
    return "Hello from Home";
}
```

Running the Application

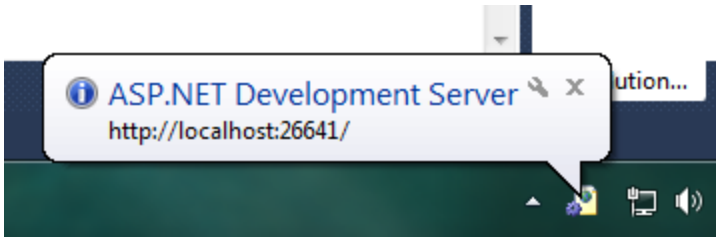
Now let's run the site. We can start our web-server and try out the site using any of the following::

- Choose the Debug ⇨ Start Debugging menu item
- Click the Green arrow button in the toolbar

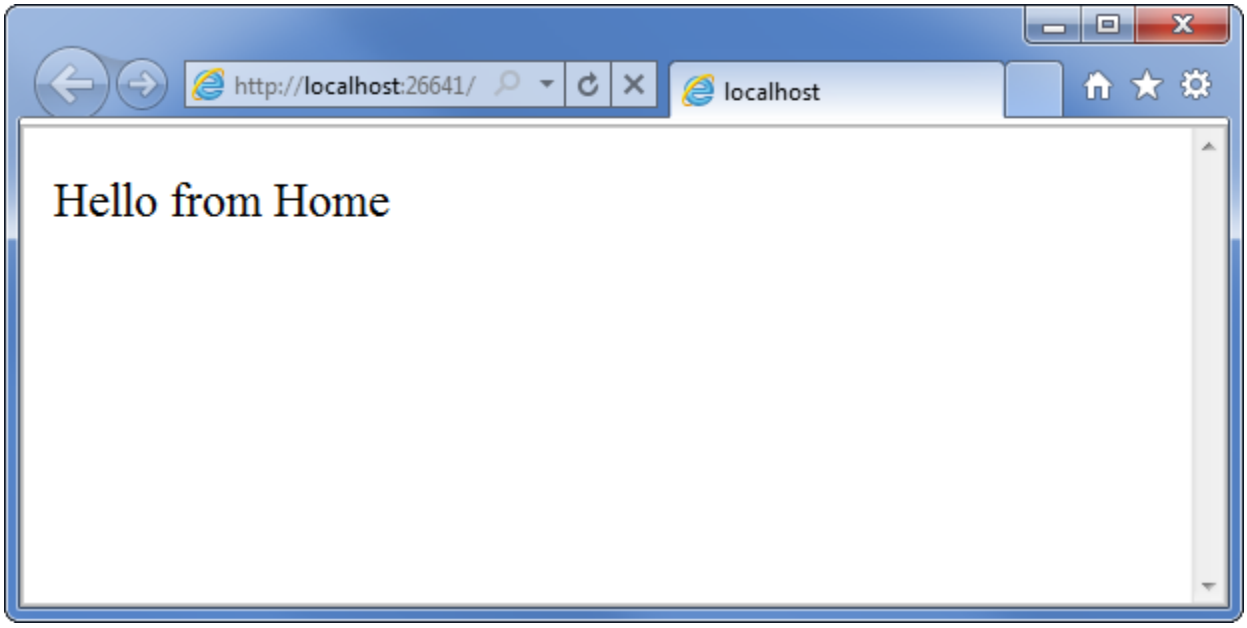


- Use the keyboard shortcut, F5.

Using any of the above steps will compile our project, and then cause the ASP.NET Development Server that is built-into Visual Web Developer to start. A notification will appear in the bottom corner of the screen to indicate that the ASP.NET Development Server has started up, and will show the port number that it is running under.



Visual Web Developer will then automatically open a browser window whose URL points to our web-server. This will allow us to quickly try out our web application:



Okay, that was pretty quick – we created a new website, added a three line function, and we've got text in a browser. Not rocket science, but it's a start.

Note: Visual Studio includes the ASP.NET Development Server, which will run your website on a random free "port" number. In the screenshot above, the site is running at http://localhost:26641/, so it's using port 26641. Your port number will be different. When we talk about URL's like /Store/Browse in this tutorial, that will go after the port number. Assuming a port number of 26641, browsing to /Store/Browse will mean browsing to http://localhost:26641/Store/Browse.

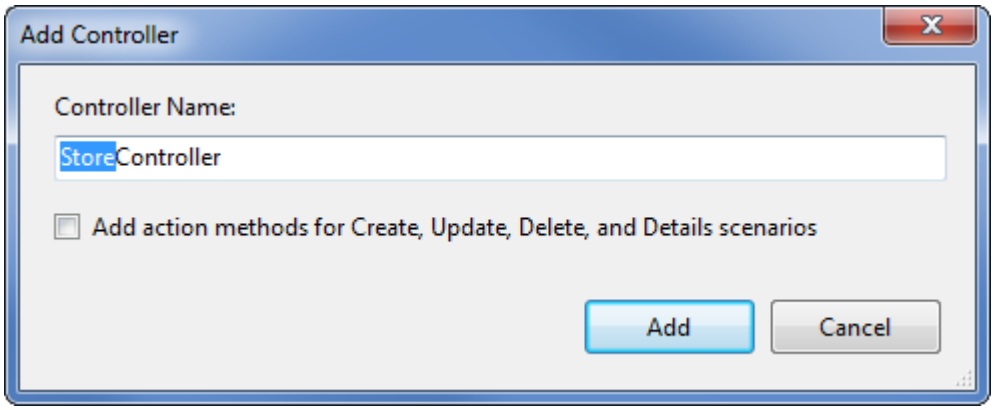
Adding a StoreController

We added a simple HomeController that implements the Home Page of our site. Let's now add another controller that we'll use to implement the browsing functionality of our music store. Our store controller will support three scenarios:

- A listing page of the music genres in our music store
- A browse page that lists all of the music albums in a particular genre
- A details page that shows information about a specific music album

We'll start by adding a new StoreController class.. If you haven't already, stop running the application either by closing the browser or selecting the Debug ⇨ Stop Debugging menu item.

Now add a new StoreController. Just like we did with HomeController, we'll do this by right-clicking on the "Controllers" folder within the Solution Explorer and choosing the Add->Controller menu item



Our new StoreController already has an "Index" method. We'll use this "Index" method to implement our listing page that lists all genres in our music store. We'll also add two additional methods to implement the two other scenarios we want our StoreController to handle: Browse and Details.

These methods (Index, Browse and Details) within our Controller are called "Controller Actions", and as you've already seen with the HomeController.Index()action method, their job is to respond to URL requests and (generally speaking) determine what content should be sent back to the browser or user that invoked the URL.

We'll start our StoreController implementation by changing theIndex() method to return the string "Hello from Store.Index()" and we'll add similar methods for Browse() and Details():

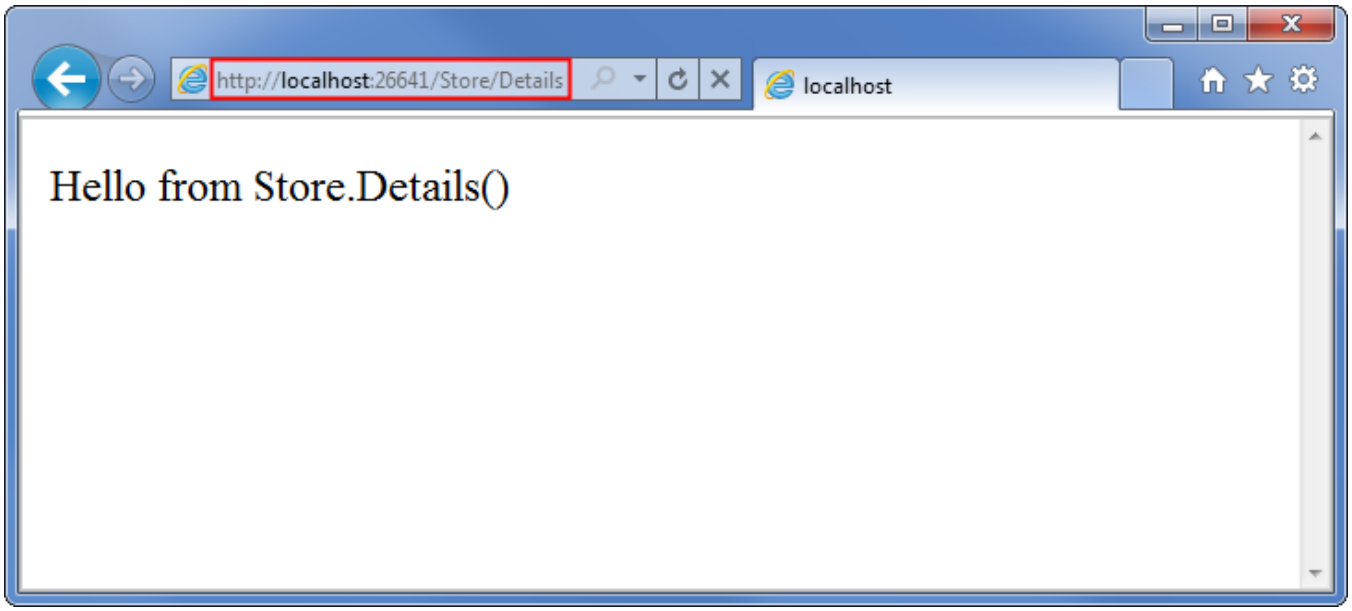
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace MvcMusicStore.Controllers
{
    public class StoreController : Controller
    {
        //
        // GET: /Store/

        public string Index()
        {
            return "Hello from Store.Index()";
        }
```

```
        //
        // GET: /Store/Browse

        public string Browse()
        {
            return "Hello from Store.Browse()";
        }

        //
        // GET: /Store/Details

        public string Details()
        {
            return "Hello from Store.Details()";
        }
    }
}
```

Run the project again and browse the following URLs:

- /Store
- /Store/Browse
- /Store/Details

Accessing these URLs will invoke the action methods within our Controller and return string responses:



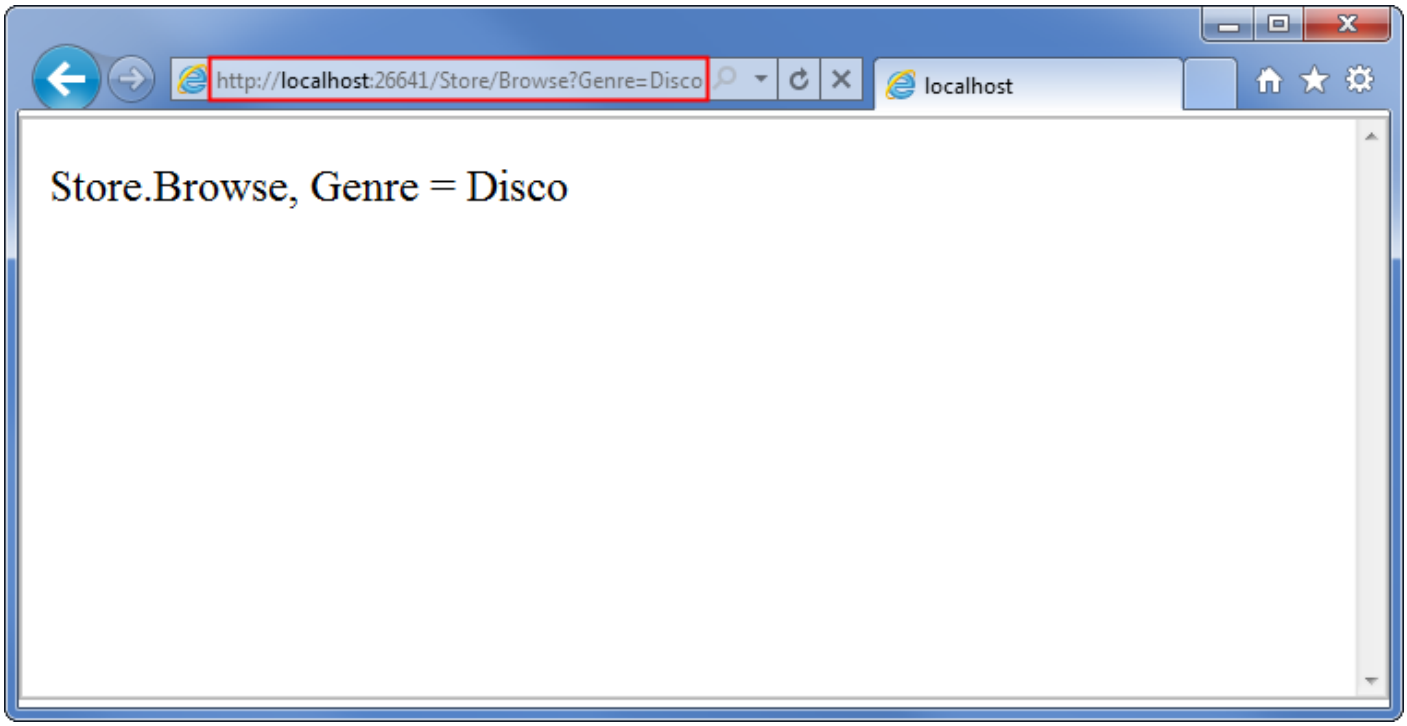
That's great, but these are just constant strings. Let's make them dynamic, so they take information from the URL and display it in the page output.

First we'll change the Browse action method to retrieve a querystring value from the URL. We can do this by adding a "genre" parameter to our action method. When we do this ASP.NET MVC will automatically pass any querystring or form post parameters named "genre" to our action method when it is invoked.

```
//
// GET: /Store/Browse?genre=?Disco

public string Browse(string genre)
{
    string message = HttpUtility.HtmlEncode("Store.Browse, Genre = " + genre);

    return message;
}
```

Note: We're using the HttpUtility.HtmlEncode utility method to sanitize the user input. This prevents users from injecting Javascript into our View with a link like /Store/Browse?Genre=<script>window.location='http://hackersite.com'</script>.

Now let's browse to /Store/Browse?Genre=Disco



Note: We're using the Server.HtmlEncode utility method to sanitize the user input. This prevents users from injecting Javascript into our View with a link like /Store/Browse?Genre=<script>window.location='http://hackersite.com'</script>.

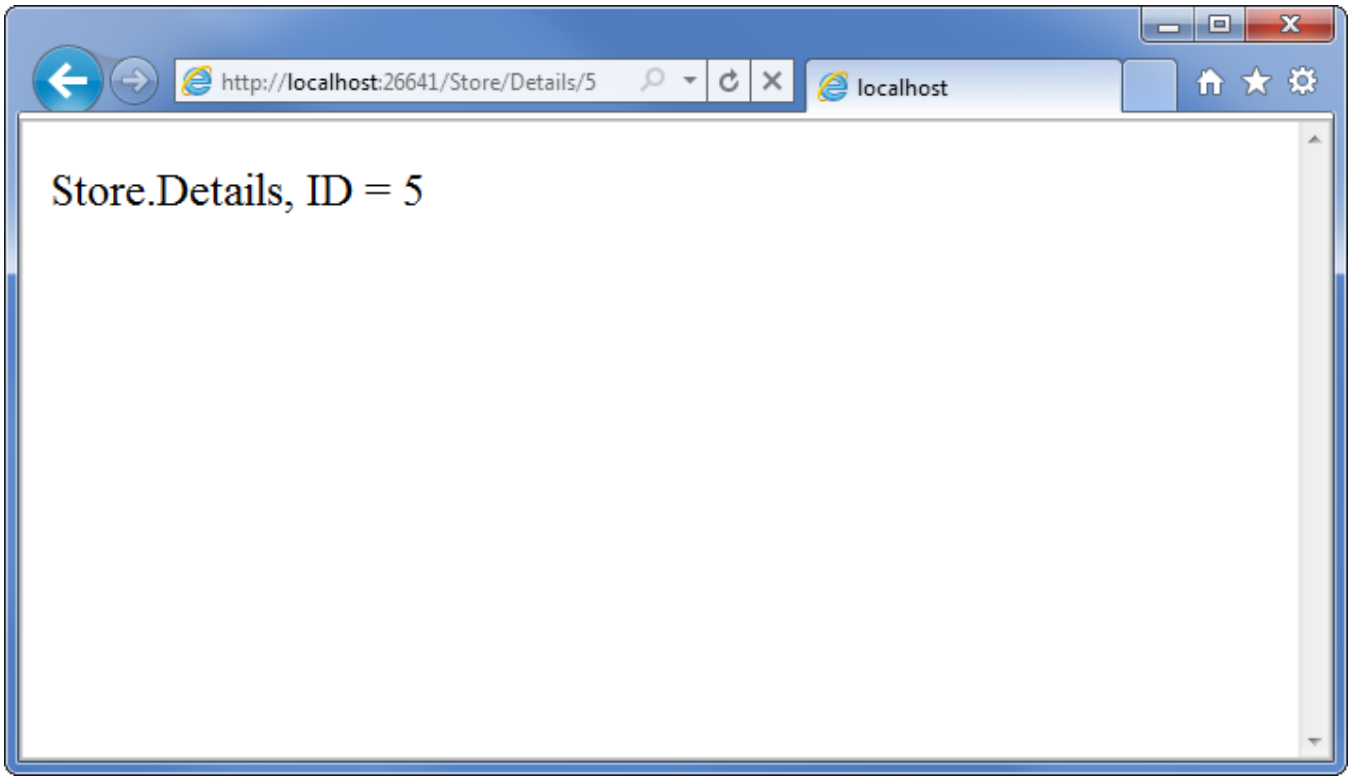
Let's next change the Details action to read and display an input parameter named ID. Unlike our previous method, we won't be embedding the ID value as a querystring parameter. Instead we'll embed it directly within the URL itself. For example: /Store/Details/5.

ASP.NET MVC lets us easily do this without having to configure anything. ASP.NET MVC's default routing convention is to treat the segment of a URL after the action method name as a parameter named "ID". If

your action method has a parameter named ID then ASP.NET MVC will automatically pass the URL segment to you as a parameter.

```
//
// GET: /Store/Details/5

public string Details(int id)
{
    string message = "Store.Details, ID = " + id;

    return message;
}
```

Run the application and browse to /Store/Details/5:



Let's recap what we've done so far:

- We've created a new ASP.NET MVC project in Visual Studio
- We've discussed the basic folder structure of an ASP.NET MVC application
- We've learned how to run our website using the ASP.NET Development Server
- We've created two Controller classes: a HomeController and a StoreController
- We've added Action Methods to our controllers which respond to URL requests and return text to the browser

3. Views and Models

So far we've just been returning strings from controller actions. That's a nice way to get an idea of how controllers work, but it's not how you'd want to build a real web application. We are going to want a better way to generate HTML back to browsers visiting our site – one where we can use template files to more easily customize the HTML content send back. That's exactly what Views do.

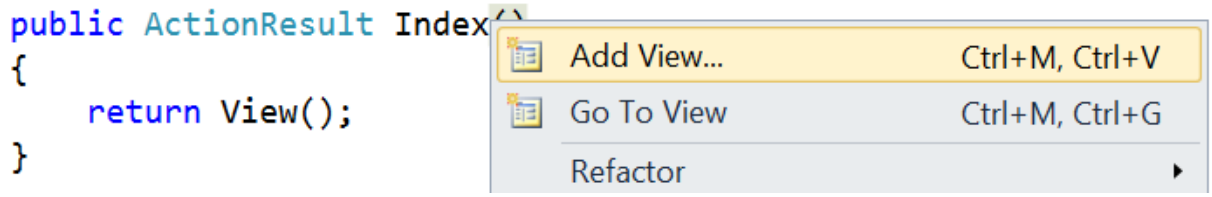
Adding a View template

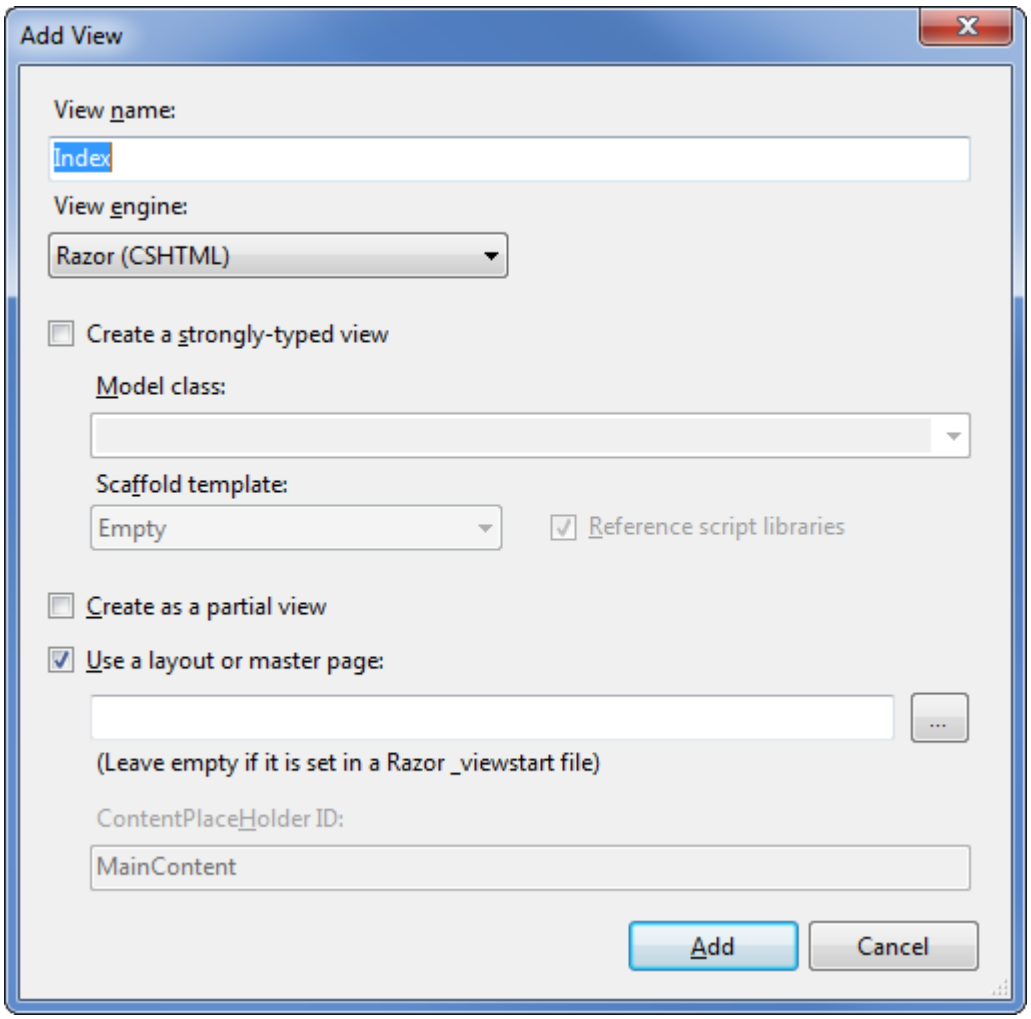
To use a view-template, we'll change the HomeController Index method to return an ActionResult, and have it return View(), like below:

```
public class HomeController : Controller
{
    //
    // GET: /Home/

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

The above change indicates that instead of returned a string, we instead want to use a "View" to generate a result back.

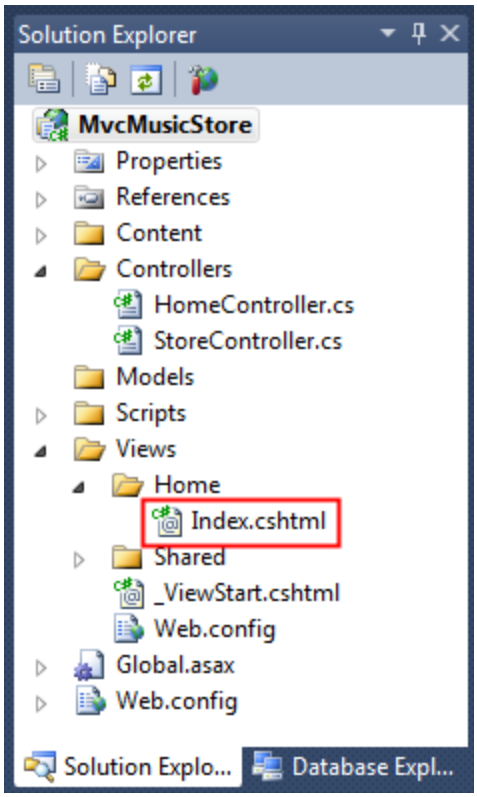
We'll now add an appropriate View template to our project. To do this we'll position the text cursor within the Index action method, then right-click and select "Add View". This will bring up the Add View dialog:





The "Add View" dialog allows us to quickly and easily generate View template files. By default the "Add View" dialog pre-populates the name of the View template to create so that it matches the action method that will use it. Because we used the "Add View" context menu within the Index() action method of our HomeController, the "Add View" dialog above has "Index" as the view name pre-populated by default. We don't need to change any of the options on this dialog, so click the Add button.

When we click the Add button, Visual Studio will create a new Index.cshtml view template for us in the \Views\Home directory, creating the folder if doesn't already exist.



The name and folder location of the "Index.cshtml" file is important, and follows the default ASP.NET MVC naming conventions. The directory name, \Views\Home, matches the controller - which is named HomeController. The view template name, Index, matches the controller action method which will be displaying the view.

ASP.NET MVC allows us to avoid having to explicitly specify the name or location of a view template when we use this naming convention to return a view. It will by default render the \Views\Home\Index.cshtml view template when we write code like below within our HomeController:

```
public class HomeController : Controller
{
    //
    // GET: /Home/

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

Visual Studio created and opened the "Index.cshtml" view template after we clicked the "Add" button within the "Add View" dialog. The contents of Index.cshtml are shown below.

```
@{
```

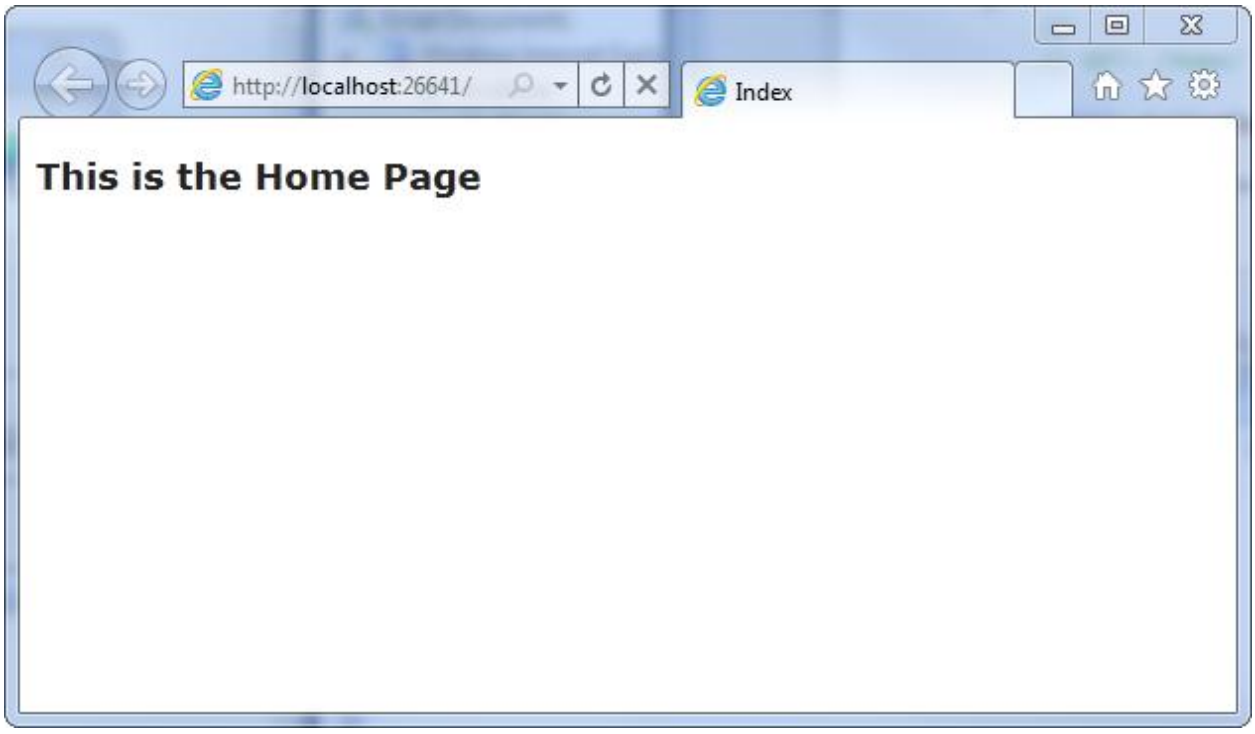
```
    ViewBag.Title = "Index";
}

<h2>Index</h2>
```

This view is using the Razor syntax, which is more concise than the Web Forms view engine used in ASP.NET Web Forms and previous versions of ASP.NET MVC. The Web Forms view engine is still available in ASP.NET MVC 3, but many developers find that the Razor view engine fits ASP.NET MVC development really well.

The first three lines set the page title using ViewBag.Title. We'll look at how this works in more detail soon, but first let's update the text heading text and view the page. Update the <h2> tag to say "This is the Home Page" as shown below.

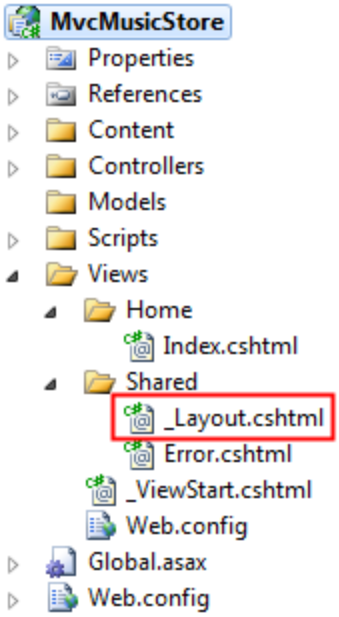
```
@{
    ViewBag.Title = "Index";
}

<h2>This is the Home Page</h2>
```

Running the application shows that our new text is visible on the home page.



Using a Layout for common site elements

Most websites have content which is shared between many pages: navigation, footers, logo images, stylesheet references, etc. The Razor view engine makes this easy to manage using a page called _Layout.cshtml has automatically been created for us inside the /Views/Shared folder.



Double-click on this folder to view the contents, which are shown below.

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

<body>
    @RenderBody()
</body>
</html>
```

The content from our individual views will be displayed by the @RenderBody() command, and any common content that we want to appear outside of that can be added to the _Layout.cshtml markup. We'll want our MVC Music Store to have a common header with links to our Home page and Store area on all pages in the site, so we'll add that to the template directly above that @RenderBody() statement.

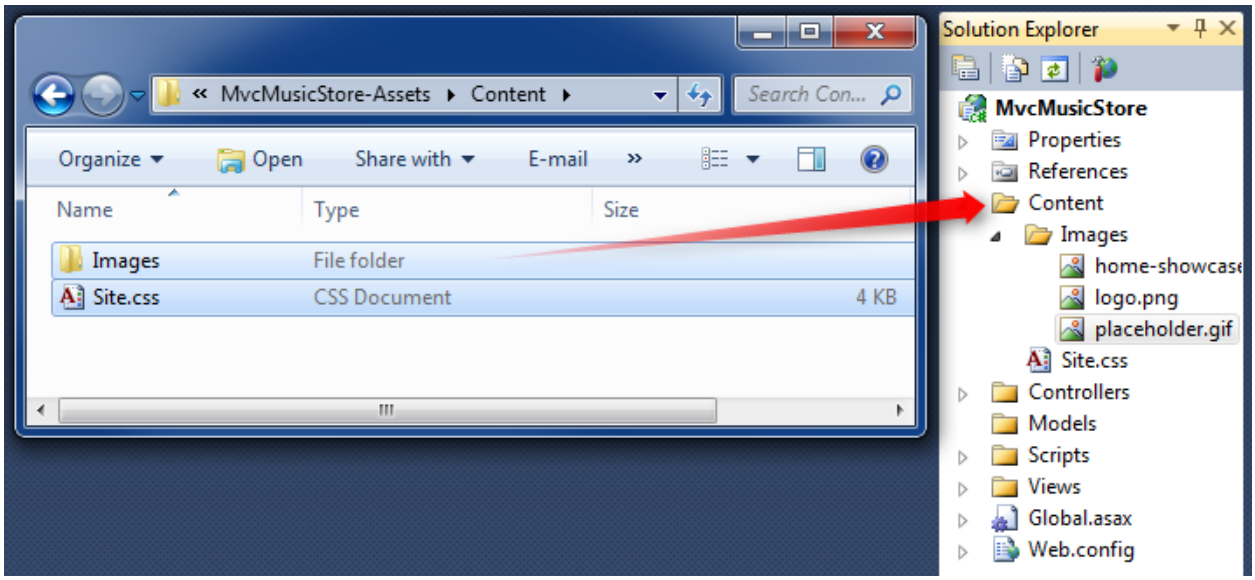
```
<!DOCTYPE html>
<html>
<head>
    <title>@ViewBag.Title</title>
    <link href="@Url.Content("~/Content/Site.css")" rel="stylesheet" type="text/css" />
    <script src="@Url.Content("~/Scripts/jquery-1.4.4.min.js")"
type="text/javascript"></script>
</head>
<body>
    <div id="header">
        <h1>
```

```
            ASP.NET MVC MUSIC STORE</h1>
        <ul id="navlist">
            <li class="first"><a href="/" id="current">Home</a></li>
            <li><a href="/Store/">Store</a></li>
        </ul>
    </div>

    @RenderBody()
</body>
</html>
```

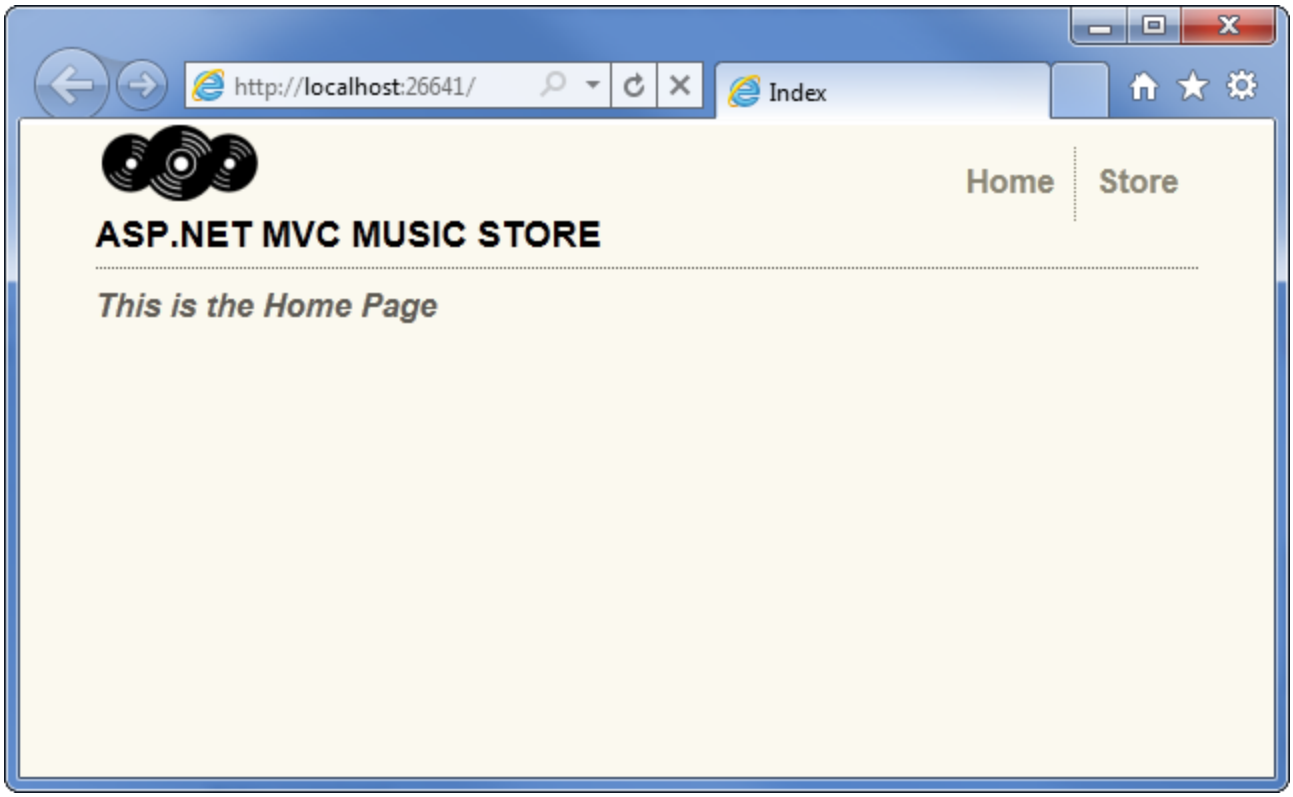
Updating the StyleSheet

The empty project template includes a very streamlined CSS file which just includes styles used to display validation messages. Our designer has provided some additional CSS and images to define the look and feel for our site, so we'll add those in now.

The updated CSS file and Images are included in the Content directory of MvcMusicStore-Assets.zip which is available at http://mvcmusicstore.codeplex.com. We'll select both of them in Windows Explorer and drop them into our Solution's Content folder in Visual Web Developer, as shown below:



Now let's run the application and see how our changes look on the Home page.



- Let's review what's changed: The HomeController's Index action method found and displayed the \Views\Home\Index.cshtmlView template, even though our code called "return View()", because our View template followed the standard naming convention.
- The Home Page is displaying a simple welcome message that is defined within the \Views\Home\Index.cshtml view template.
- The Home Page is using our _Layout.cshtml template, and so the welcome message is contained within the standard site HTML layout.

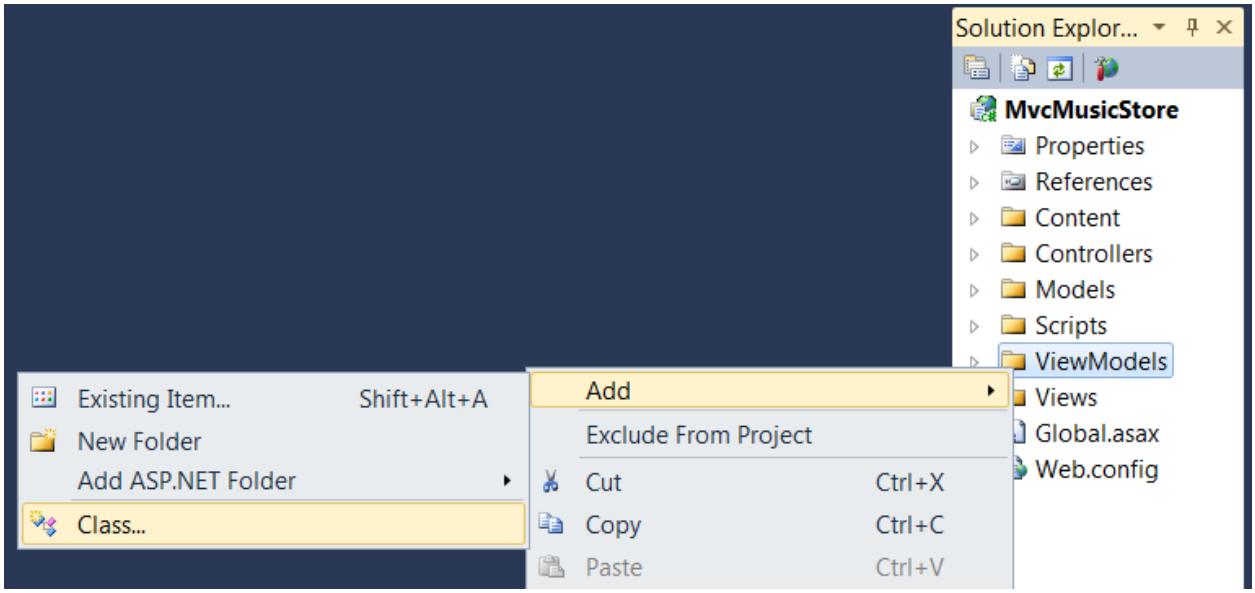
Using a Model to pass information to our View

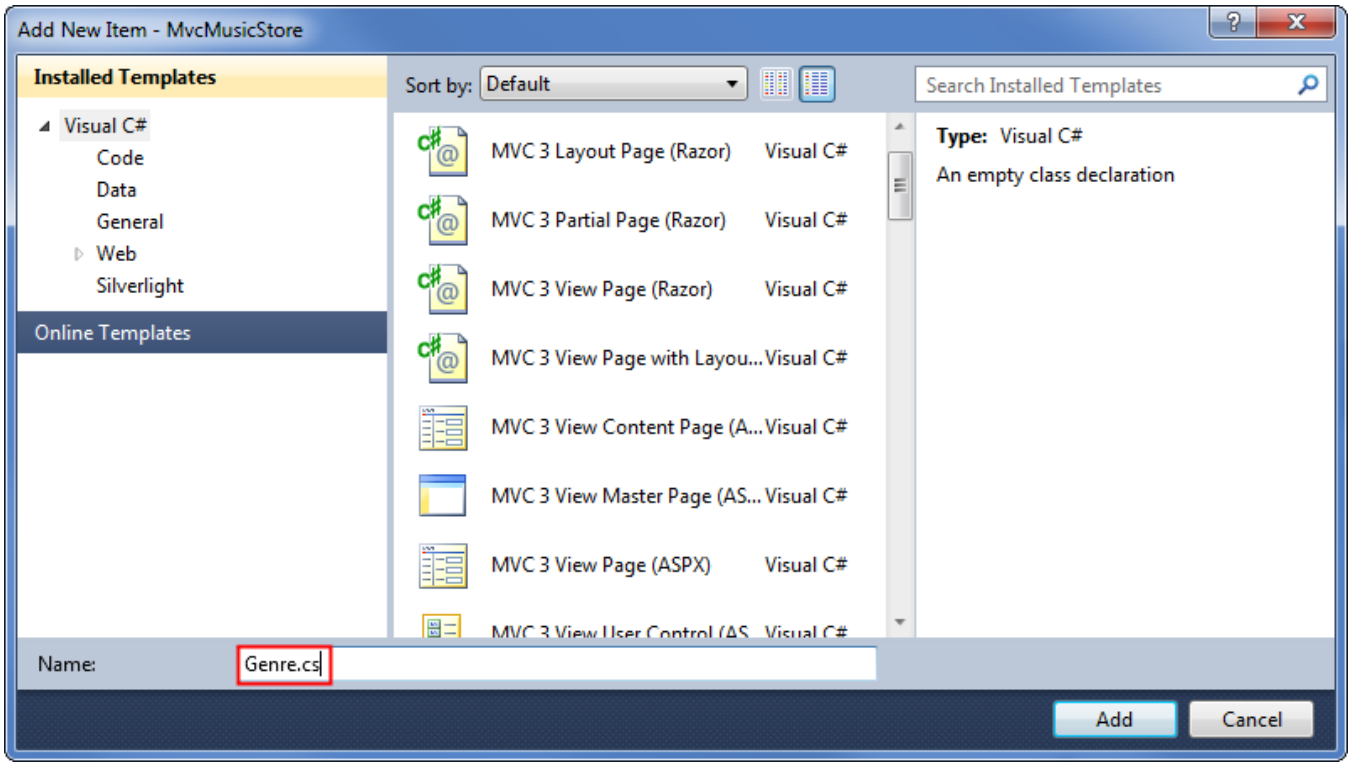
A View template that just displays hardcoded HTML isn't going to make a very interesting web site. To create a dynamic web site, we'll instead want to pass information from our controller actions to our view templates.

In the Model-View-Controller pattern, the term Model refers to objects which represent the data in the application. Often, model objects correspond to tables in your database, but they don't have to.

Controller action methods which return an ActionResult can pass a model object to the view. This allows a Controller to cleanly package up all the information needed to generate a response, and then pass this information off to a View template to use to generate the appropriate HTML response. This is easiest to understand by seeing it in action, so let's get started.

First we'll create some Model classes to represent Genres and Albums within our store. Let's start by creating a Genre class. Right-click the "Models" folder within your project, choose the "Add Class" option, and name the file "Genre.cs".





Then add a public string Name property to the class that was created:

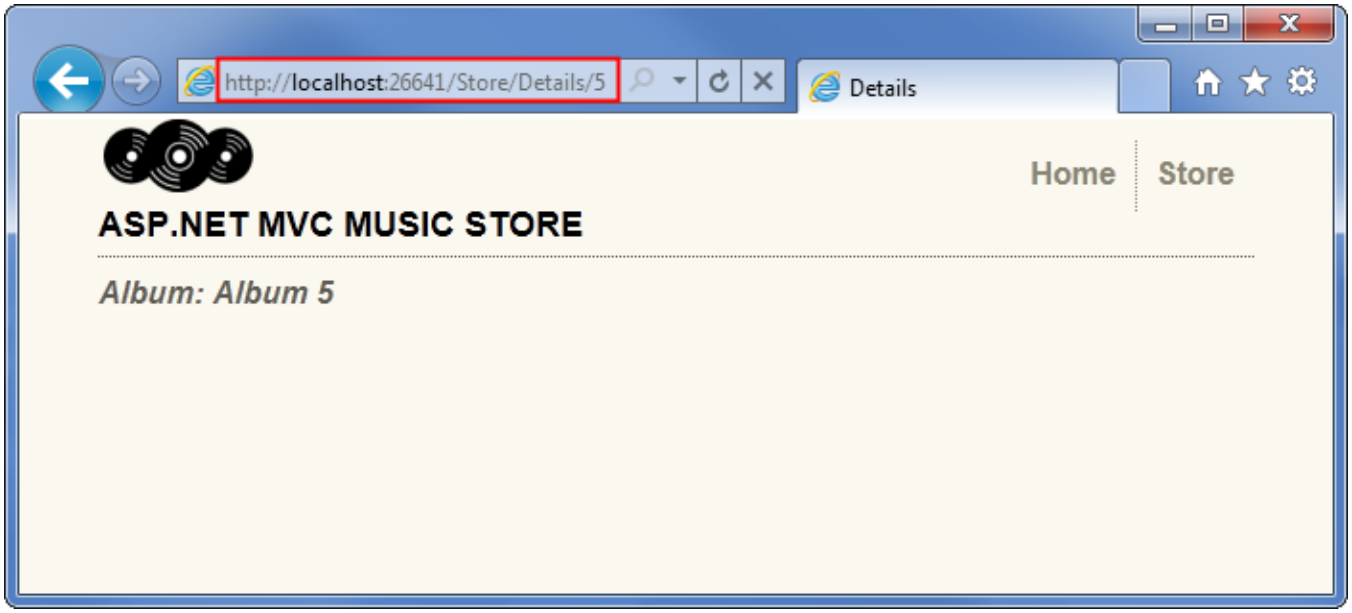
```
public class Genre
{
    public string Name { get; set; }
}
```

Note: In case you're wondering, the { get; set; } notation is making use of C#'s auto-implemented properties feature. This gives us the benefits of a property without requiring us to declare a backing field.

Next, follow the same steps to create an Album class (named Album.cs) that has a Title and a Genre property:

```
public class Album
{
    public string Title { get; set; }
    public Genre Genre { get; set; }
}
```

Now we can modify the StoreController to use Views which display Model information as shown below.



We'll start by changing the Store Details action so it shows the information for a single album. Add a "using" statement to the top of the StoreControllers class to include the MvcMusicStore.Models namespace, so we don't need to type MvcMusicStore.Models.Album every time we want to use the album class. The "usings" section of that class should now appear as below.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using MvcMusicStore.Models;
```

Next, we'll update the Details controller action so that it returns an ActionResult rather than a string, as we did with the HomeController's Index method.

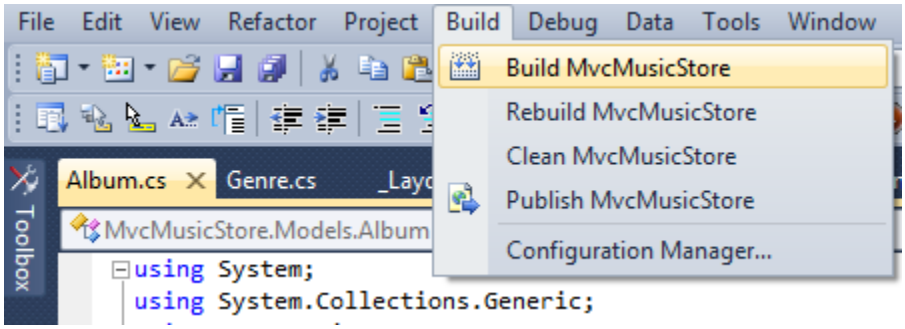
```
public ActionResult Details(int id)
```

Now we can modify the logic to return an Album object to the view. Later in this tutorial we will be retrieving the data from a database – but for right now we will use "dummy data" to get started.

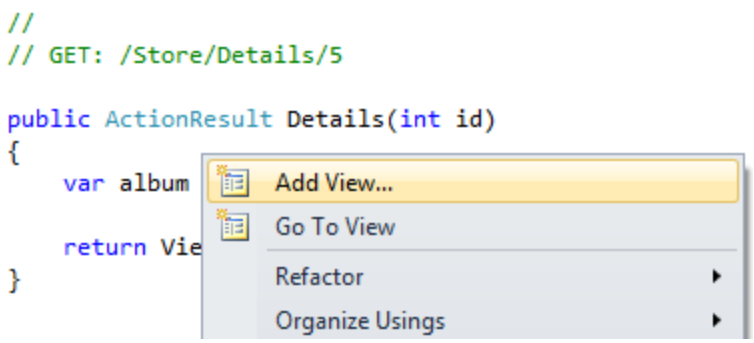
```
public ActionResult Details(int id)
{
    var album = new Album { Title = "Album " + id };

    return View(album);
}
```

Note: If you're unfamiliar with C#, you may assume that using var means that our album variable is late-bound. That's not correct – the C# compiler is using type-inference based on what we're assigning to the variable to determine that album is of type Album and compiling the local album variable as an Album type, so we get compile-time checking and Visual Studio code-editor support.

Let's now create a View template that uses our Album to generate an HTML response. Before we do that we need to build the project so that the Add View dialog knows about our newly created Album class. You can build the project by selecting the Debug⇨Build MvcMusicStore menu item.

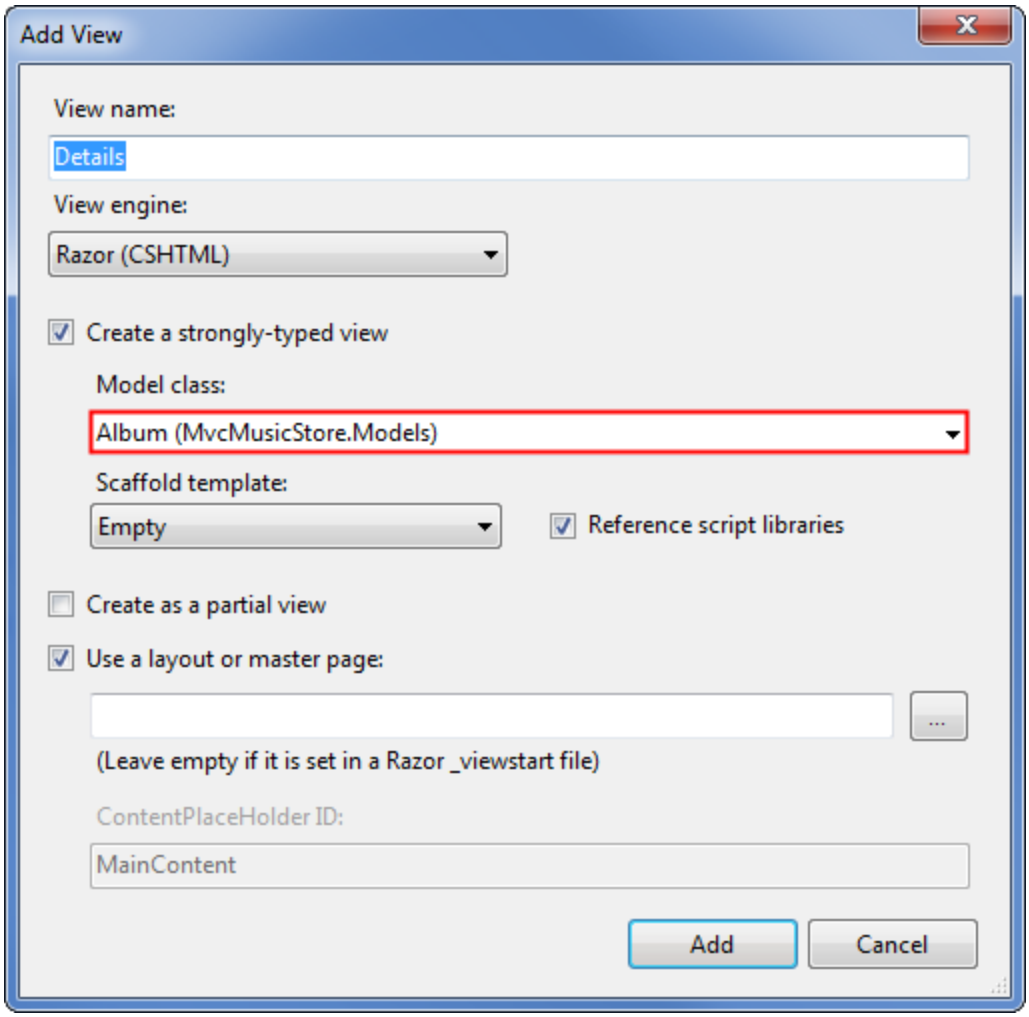


Now that we've set up our supporting classes, we're ready to build our View template. Right-click within the Details method in the Browse and select "Add View…" from the context menu.



We are going to create a new View template like we did before with the HomeController. Because we are creating it from the StoreController it will by default be generated in a \Views\Store\Index.cshtml file.

Unlike before, we are going to check the "Create a strongly-typed" view checkbox. We are then going to select our "Album" class within the "View data-class" drop-downlist. This will cause the "Add View" dialog to create a View template that expects that an Album object will be passed to it to use.



When we click the "Add" button our \Views\Store\Details.cshtml View template will be created, containing the following code.

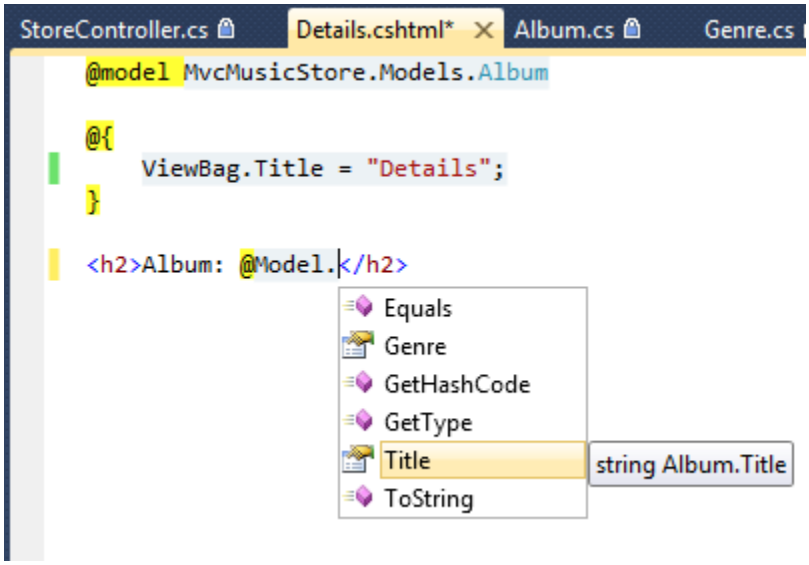
```
@model MvcMusicStore.Models.Album

@{
    ViewBag.Title = "Details";
}

<h2>Details</h2>
```

Notice the first line, which indicates that this view is strongly-typed to our Album class. The Razor view engine understands that it has been passed an Album object, so we can easily access model properties and even have the benefit of IntelliSense in the Visual Studio editor.

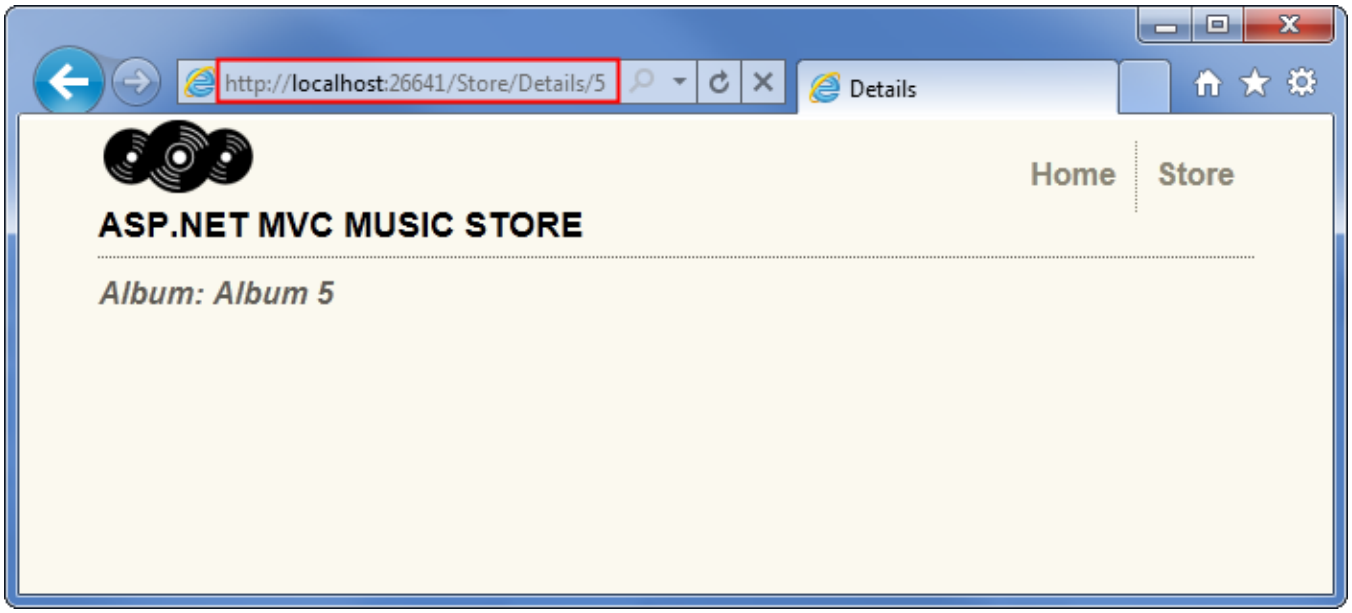
Update the <h2> tag so it displays the Album's Title property by modifying that line to appear as follows.

```
<h2>Album: @Model.Title</h2>
```

Notice that IntelliSense is triggered when you enter the period after the @Model keyword, showing the properties and methods that the Album class supports.



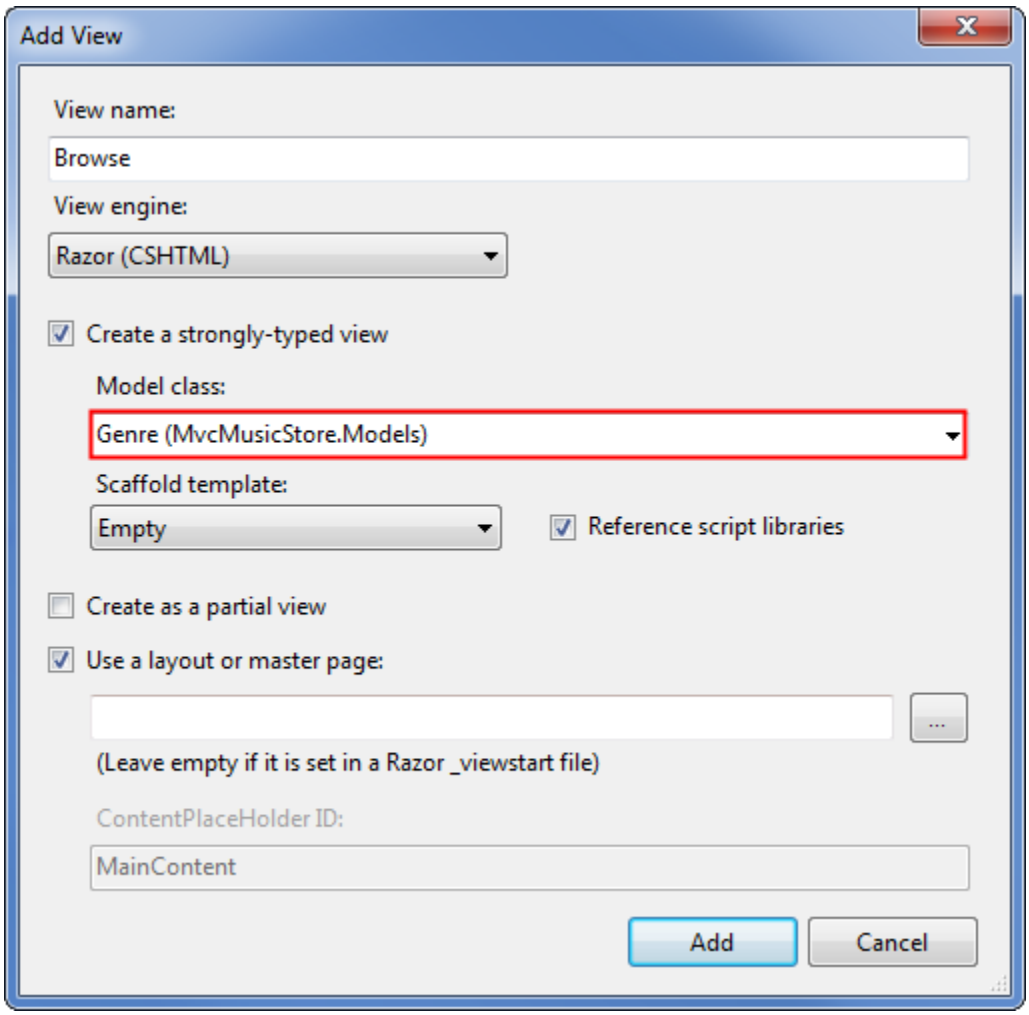
Let's now re-run our project and visit the /Store/Details/1 URL. We'll see details of an Album like below.



Now we'll make a similar update for the Store Browse action method. Update the method so it returns an ActionResult, and modify the method logic so it creates a new Genre object and returns it to the View.

```
public ActionResult Browse(string genre)
{
    var genreModel = new Genre { Name = genre };

    return View(genreModel);
}
```

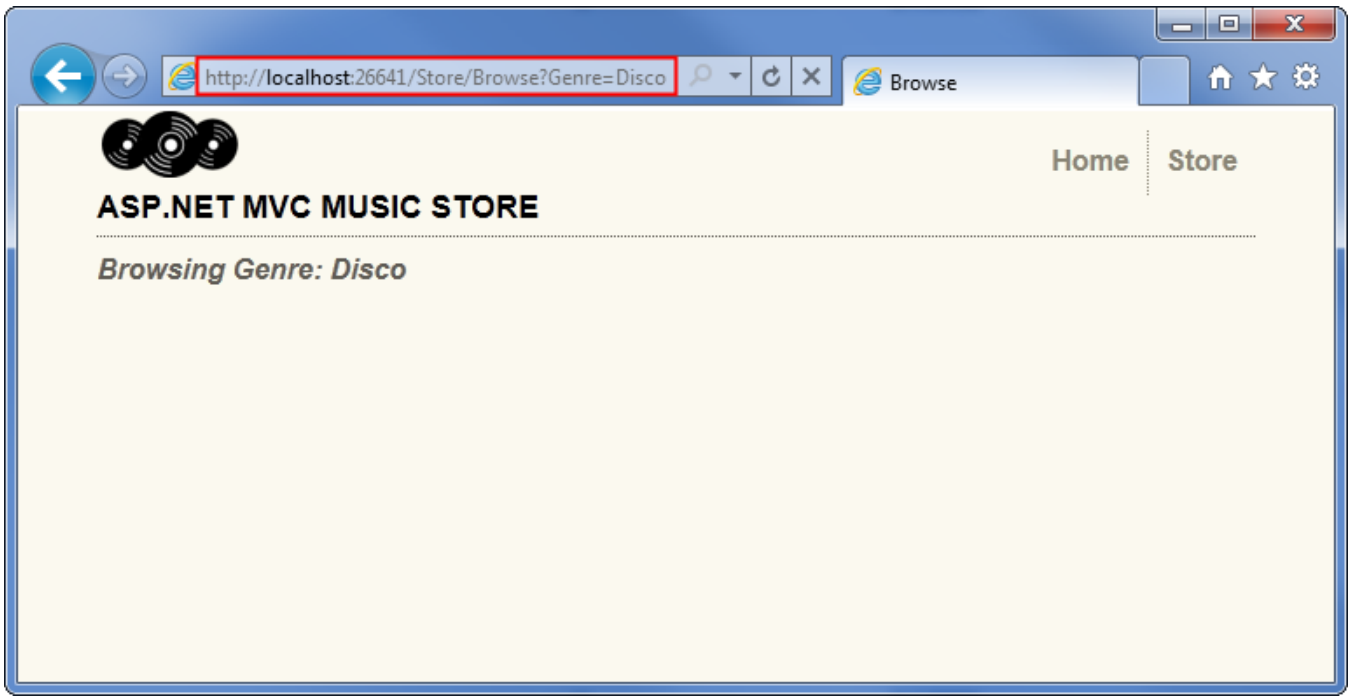
Right-click in the Browse method and select "Add View…" from the context menu, then add a View that is strongly-typed add a strongly typed to the Genre class.



Update the view code (in /Views/Store/Browse.cshtml) to display the Genre information.

```
@model MvcMusicStore.Models.Genre

@{
    ViewBag.Title = "Browse";
}

<h2>Browsing Genre: @Model.Name</h2>
```

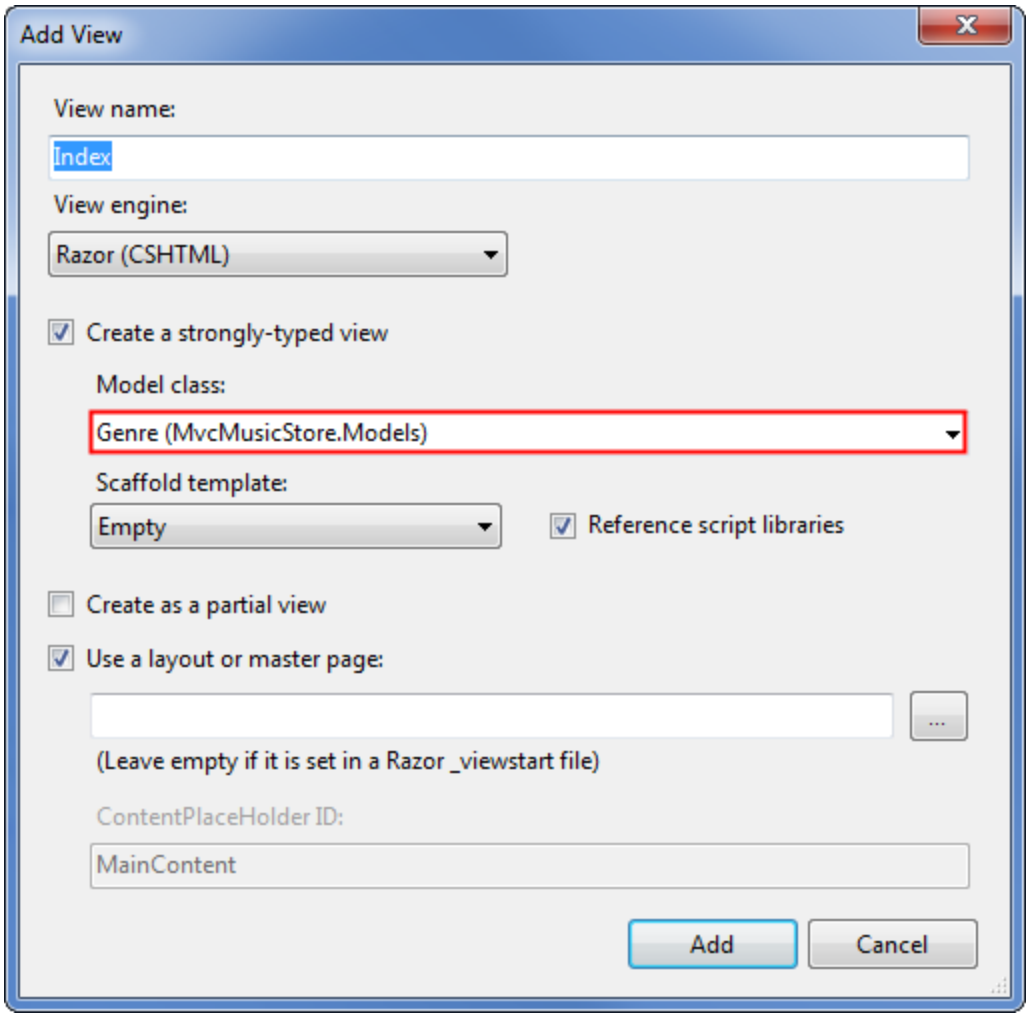
Now let's re-run our project and browse to the /Store/Browse?Genre=Disco URL. We'll see the Browse page displayed like below.



Finally, let's make a slightly more complex update to the Store Index action method and view to display a list of all the Genres in our store. We'll do that by using a List of Genres as our model object, rather than just a single Genre.

```
public ActionResult Index()
{
    var genres = new List<Genre>
    {
        new Genre { Name = "Disco"},
        new Genre { Name = "Jazz"},
        new Genre { Name = "Rock"}
    };

    return View(genres);
}
```

Right-click in the Store Index action method and select Add View as before, select Genre as the Model class, and press the Add button.



First we'll change the @model declaration to indicate that the view will be expecting several Genre objects rather than just one. Change the first line of /Store/Index.cshtml to read as follows:

```
@model IEnumerable<MvcMusicStore.Models.Genre>
```

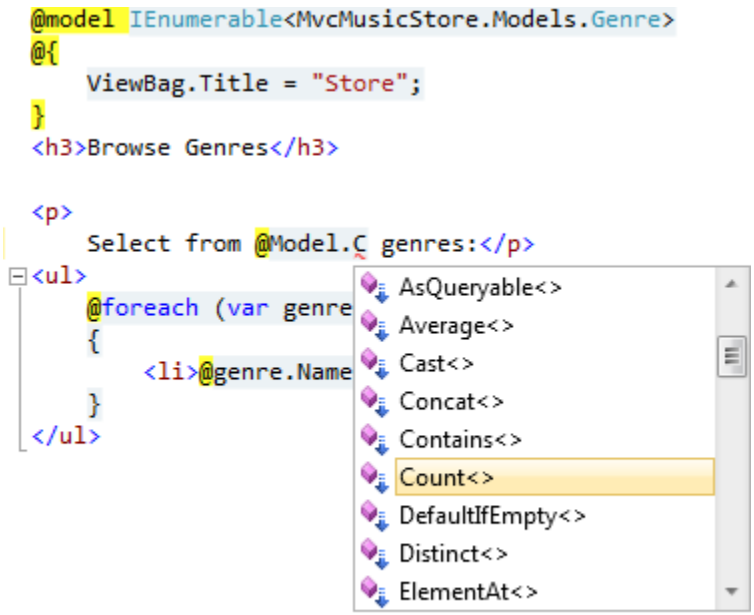
This tells the Razor view engine that it will be working with a model object that can hold several Genre objects. We're using an IEnumerable<Genre> rather than a List<Genre> since it's more generic, allowing us to change our model type later to any object type to any other container that supports the IEnumerable interface.

Next, we'll loop through the Genre objects in the model as shown in the completed view code below.

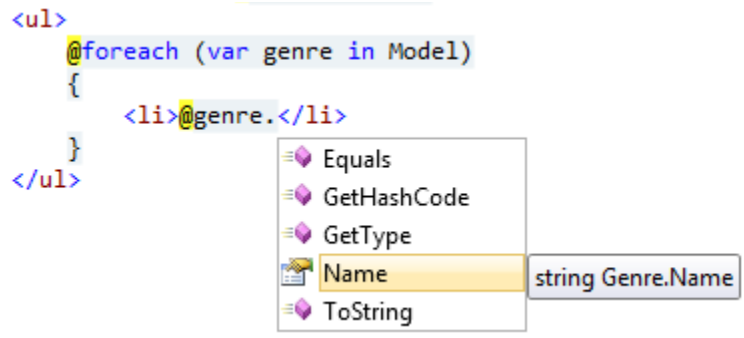
```
@model IEnumerable<MvcMusicStore.Models.Genre>
@{
    ViewBag.Title = "Store";
}
<h3>Browse Genres</h3>

<p>
    Select from @Model.Count() genres:</p>
```

```
<ul>
    @foreach (var genre in Model)
    {
        <li>@genre.Name</li>
    }
</ul>
```

Notice that we have full IntelliSense support as we enter this code, so that when we type "@Model." we see all methods and properties supported by an IEnumerable of type Genre.

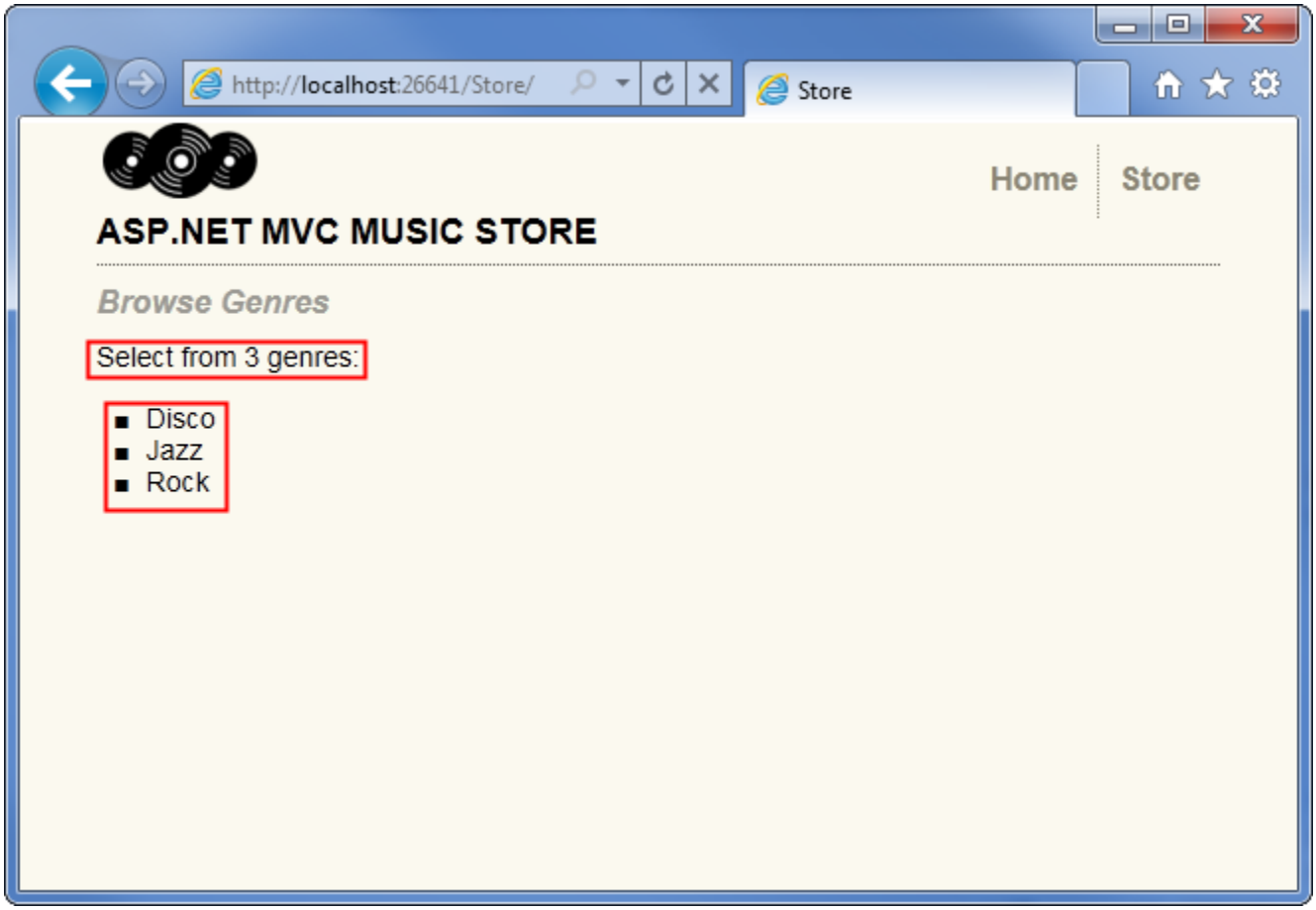


Within our "foreach" loop, Visual Studio knows that each item is of type Genre, so we see IntelliSence for each the Genre type.



Next, the scaffolding feature examined the Genre object and determined that each will have a Name property, so it loops through and writes them out. It also generates Edit, Details, and Delete links to each individual item. We'll take advantage of that later in our store manager, but for now we'd like to have a simple list instead.

When we run the application and browse to /Store, we see that both the count and list of Genres is displayed.



Adding Links between pages

Our /Store URL that lists Genres currently lists the Genre names simply as plain text. Let's change this so that instead of plain text we instead have the Genre names link to the appropriate /Store/Browse URL, so that clicking on a music genre like "Disco" will navigate to the /Store/Browse?genre=Disco URL. We could update our \Views\Store\Index.cshtml View template to output these links using code like below:

```
<ul>
    @foreach (var genre in Model)
    {
        <li><a href="/Store/Browse?genre=@genre.Name">@genre.Name</a></li>
    }
</ul>
```

That works, but it could lead to trouble later since it relies on a hardcoded string. For instance, if we wanted to rename the Controller, we'd need to search through our code looking for links that need to be updated.

An alternative approach we can use is to take advantage of an HTML Helper method. ASP.NET MVC includes HTML Helper methods which are available from our View template code to perform a variety of

common tasks just like this. The Html.ActionLink() helper method is a particularly useful one, and makes it easy to build HTML <a> links and takes care of annoying details like making sure URL paths are properly URL encoded.

Html.ActionLink() has several different overloads to allow specifying as much information as you need for your links. In the simplest case, you'll supply just the link text and the Action method to go to when the hyperlink is clicked on the client. For example, we can link to "/Store/" Index() method on the Store Details page with the link text "Go to the Store Index" using the following call:

```
@Html.ActionLink("Go to the Store Index", "Index")
```

Note: In this case, we didn't need to specify the controller name because we're just linking to another action within the same controller that's rendering the current view.

Our links to the Browse page will need to pass a parameter, though, so we'll use another overload of the Html.ActionLink method that takes three parameters:

1. Link text, which will display the Genre name
2. Controller action name (Browse)
3. Route parameter values, specifying both the name (Genre) and the value (Genre name)

Putting that all together, here's how we'll write those links to the Store Index view:

```
<ul>
    @foreach (var genre in Model)
    {
        <li>@Html.ActionLink(genre.Name, "Browse", new { genre = genre.Name })</li>
    }
</ul>
```

Now when we run our project again and access the /Store/ URL we will see a list of genres. Each genre is a hyperlink – when clicked it will take us to our /Store/Browse?genre=*[genre]* URL.

The HTML for the genre list looks like this:

```
<ul>
    <li><a href="/Store/Browse?genre=Rock">Rock</a> </li>
    <li><a href="/Store/Browse?genre=Jazz">Jazz</a> </li>
    <li><a href="/Store/Browse?genre=Country">Country</a> </li>
    <li><a href="/Store/Browse?genre=Pop">Pop</a> </li>
    <li><a href="/Store/Browse?genre=Disco">Disco</a> </li>
</ul>
```

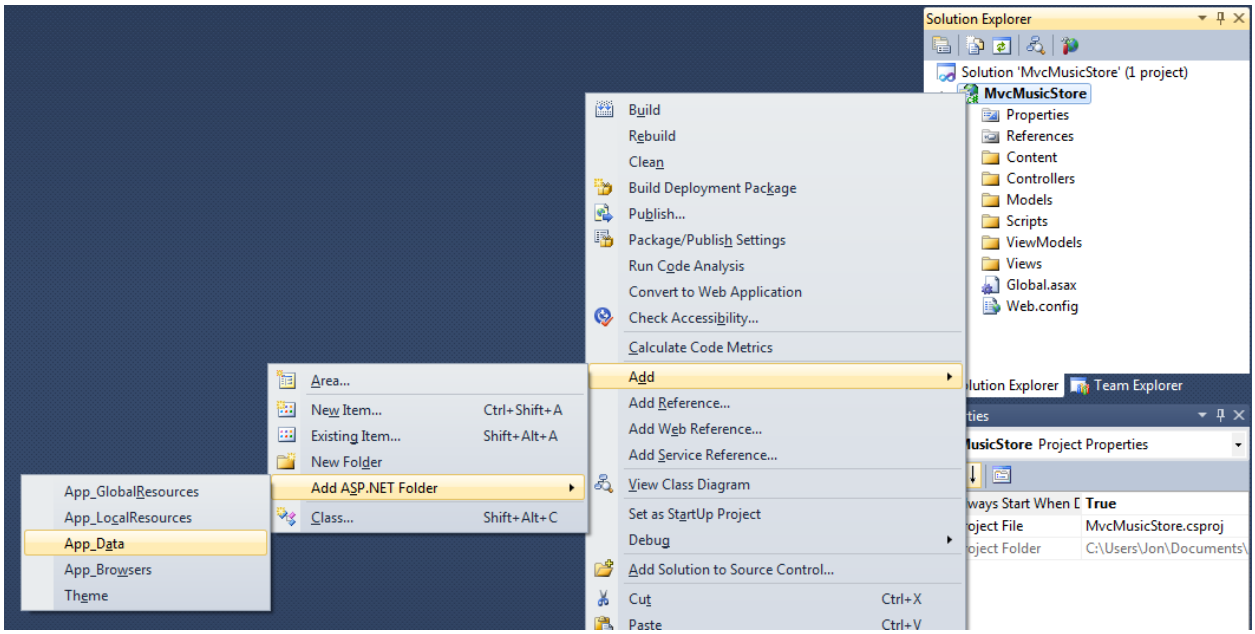
4. Data Access

So far, we've just been passing "dummy data" from our Controllers to our View templates. Now we're ready to hook up a real database. In this tutorial we'll be covering how to use the free SQL Server Express as our database engine. The code will also work with the full SQL Server.

We'll start by adding an App_Data directory to our project to hold our SQL Server Express database files. App_Data is a special directory in ASP.NET which already has the correct security access permissions for database access.

Adding a Database

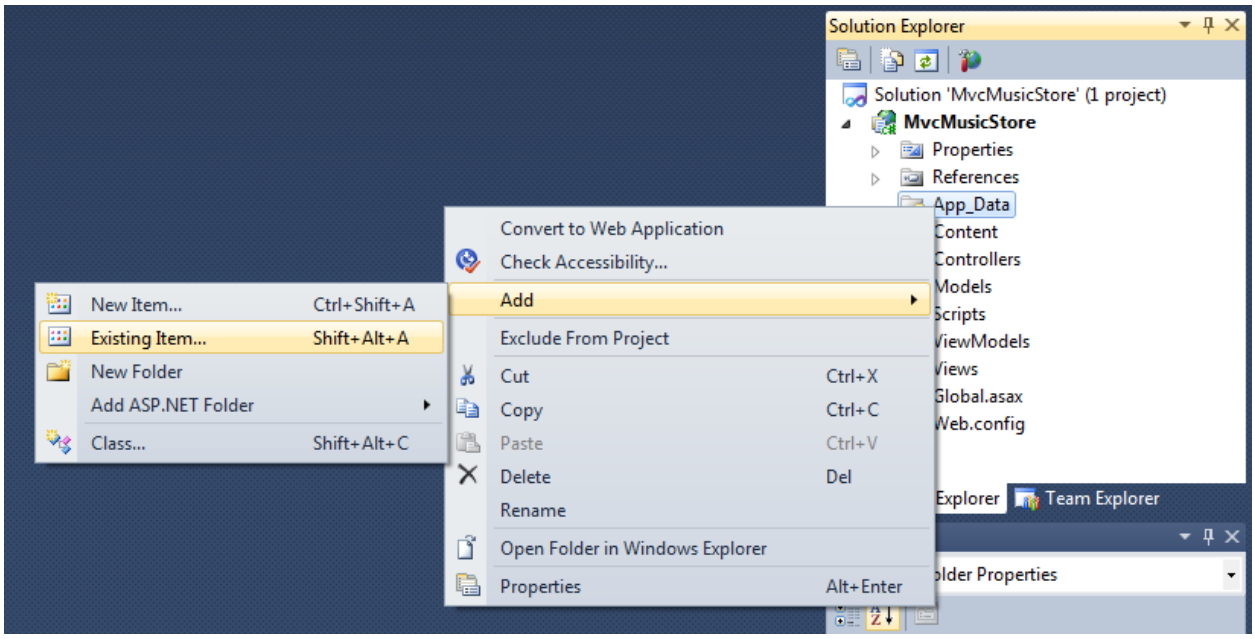
Right-click the project and select Add⇨Add ASP.NET Folder⇨App_Data.

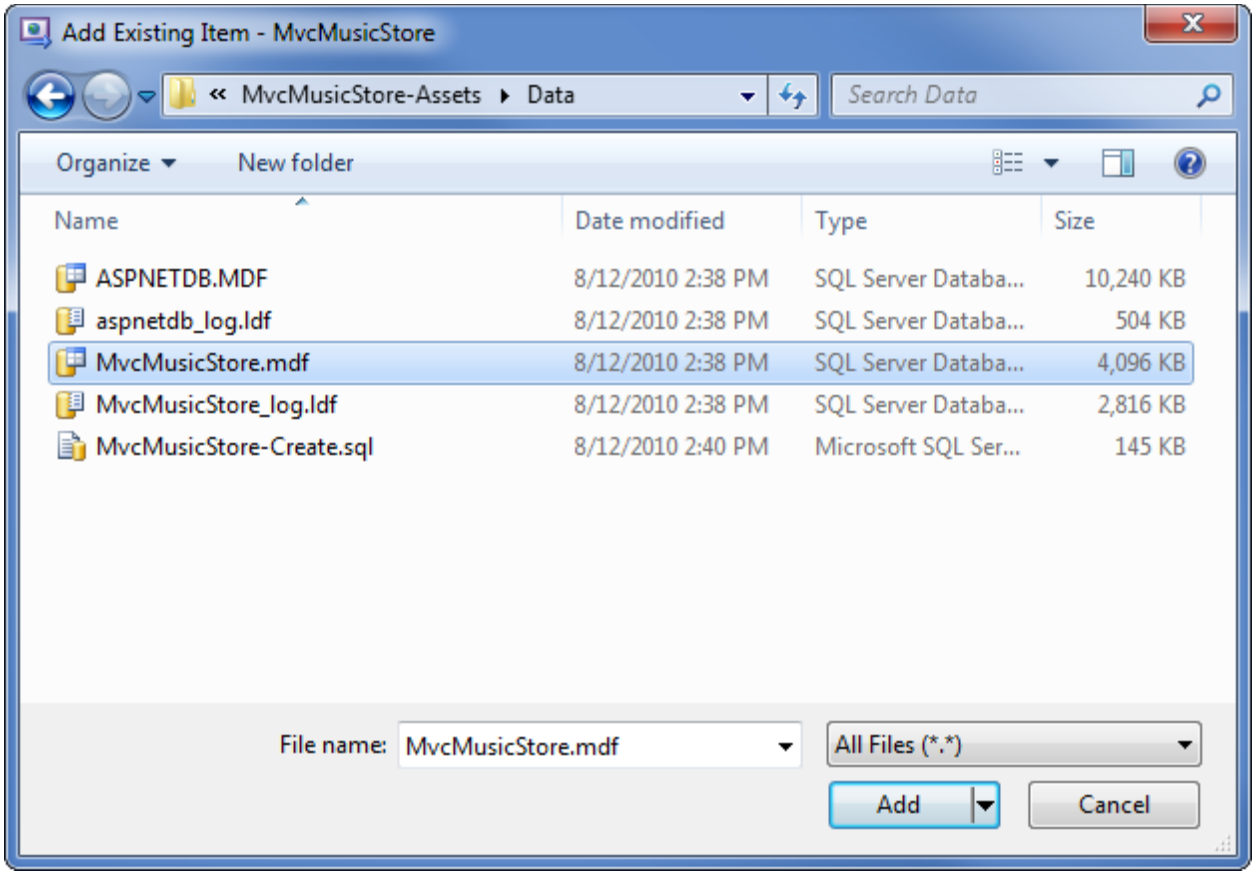


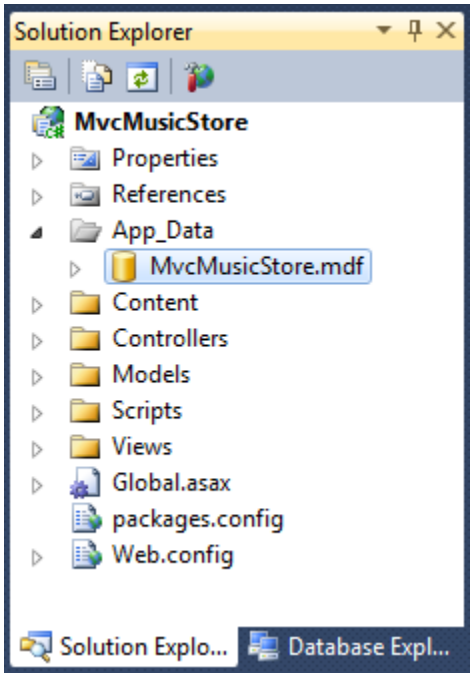
Now we'll add a database file. For this tutorial we'll be using a database that we've already created called MvcMusicStore.mdf – it is included in the MvcMusicStore-Assets/Data directory of the project downloads available at http://mvcmusicstore.codeplex.com.

Note: The MvcMusicStore-Assets/Data directory also includes a T-SQL script (MvcMusicStore-Create.sql) which can be used to create the MvcMusicStore database on a SQL Server instance if you're unable to use SQL Server Express.

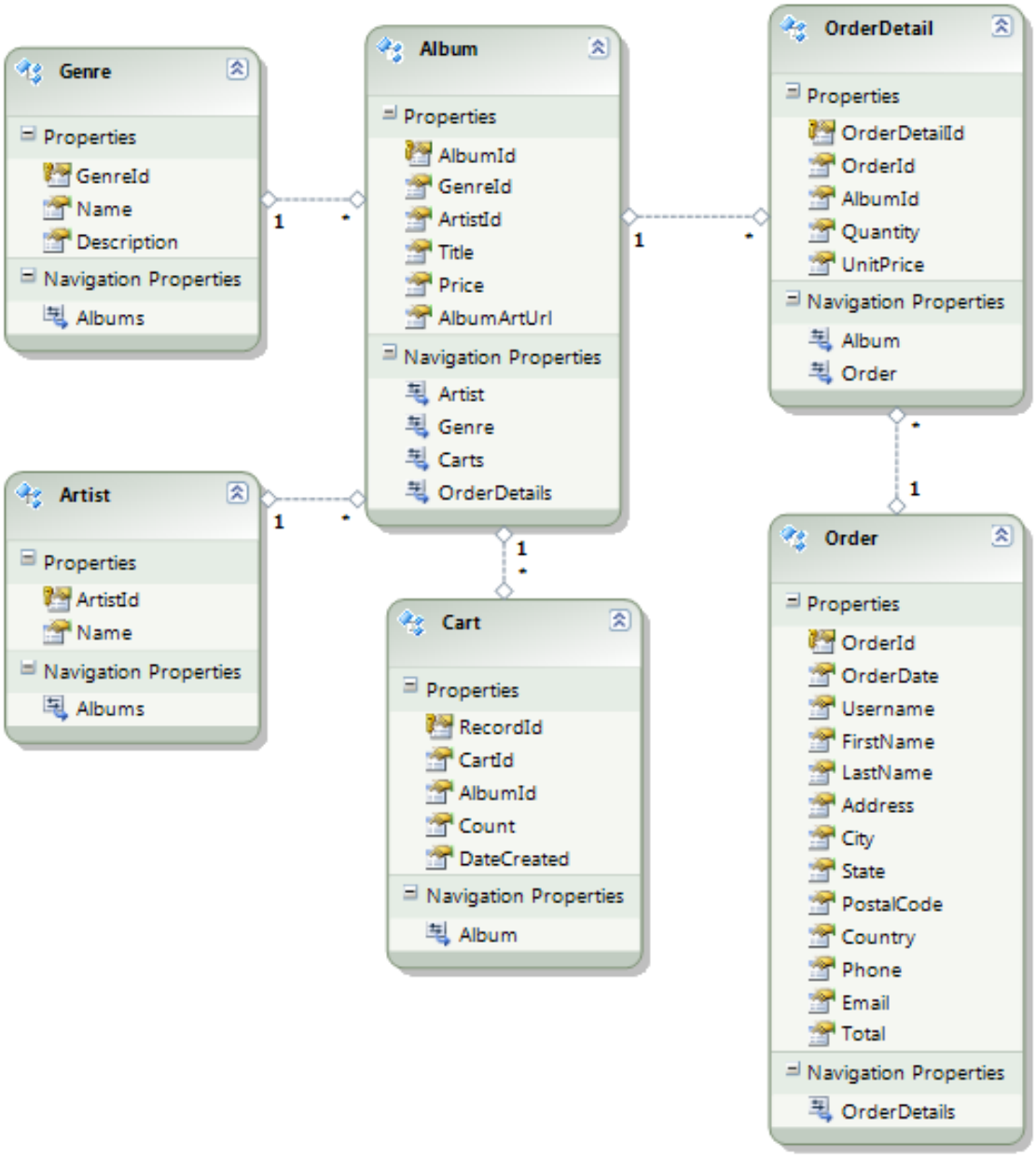
To add the database to our project, we can right-click the new App_Data folder, select Add⇨Existing Item… and browse to select the MvcMusicStore.mdf file we've downloaded to our local computer.


Solution Explorer
Solution 'MvcMusicStore' (1 project)
MvcMusicStore
Properties
References
App_Data
Content
Controllers
Models
Scripts
ViewModels
Views
Global.asax
Web.config
Explorer
Team Explorer
Convert to Web Application
Check Accessibility...
Add
Exclude From Project
Cut Ctrl+X
Copy Ctrl+C
Paste Ctrl+V
Delete Del
Rename
Open Folder in Windows Explorer
Properties Alt+Enter
New Item... Ctrl+Shift+A
Existing Item... Shift+Alt+A
New Folder
Add ASP.NET Folder
Class... Shift+Alt+C


Add Existing Item - MvcMusicStore
« MvcMusicStore-Assets ▸ Data
Search Data
Organize
New folder
Name
Date modified
Type
Size
ASPNETDB.MDF 8/12/2010 2:38 PM SQL Server Databa... 10,240 KB
aspnetdb_log.ldf 8/12/2010 2:38 PM SQL Server Databa... 504 KB
MvcMusicStore.mdf 8/12/2010 2:38 PM SQL Server Databa... 4,096 KB
MvcMusicStore_log.ldf 8/12/2010 2:38 PM SQL Server Databa... 2,816 KB
MvcMusicStore-Create.sql 8/12/2010 2:40 PM Microsoft SQL Ser... 145 KB
File name: MvcMusicStore.mdf
All Files (*.*)
Add
Cancel



Let's look at a diagram of the database we've just added to the project.



Above you can see that we have Album, Genre, and Artist classes to track our music. Since we're running a store, we also have tables for Cart, Order, and OrderDetails.

Connecting to the database using Entity Framework Code-First

Now that the database has been added to the project, we can write code to query and update the database. We'll use the Entity Framework (EF) that is built into .NET 4 to do this. EF is a flexible object relational mapping (ORM) data API that enables developers to query and update data stored in a database in an object-oriented way.

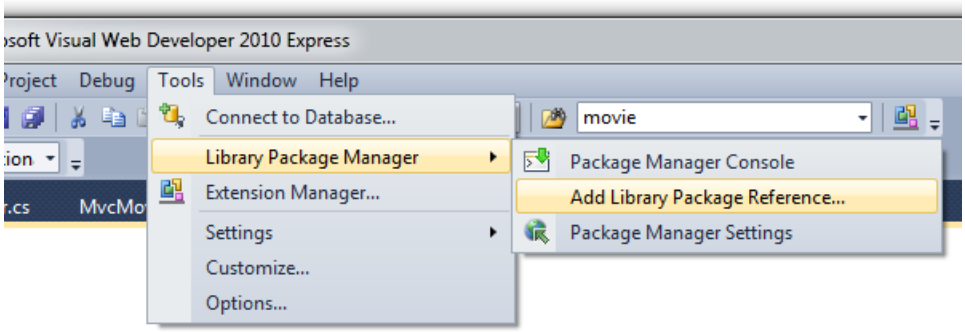
Entity Framework version 4 supports a development paradigm called code-first. Code-first allows you to create model object by writing simple classes (also known as POCO from "plain-old" CLR objects), and can even create the database on the fly from your classes. In order to use code-first, you must install the EFCodeFirst library.

Note: As mentioned, Entity Framework Code-First can be used to generate a database based on your model classes. In this tutorial, we'll be working with an existing database which is pre-loaded with a music catalog consisting of albums complete with their genre and artist information. For an example demonstrating the generation of a database from a model, see Scott Hanselman's Intro to ASP.NET MVC tutorial, available at http://www.asp.net/mvc/tutorials/getting-started-with-mvc-part1.

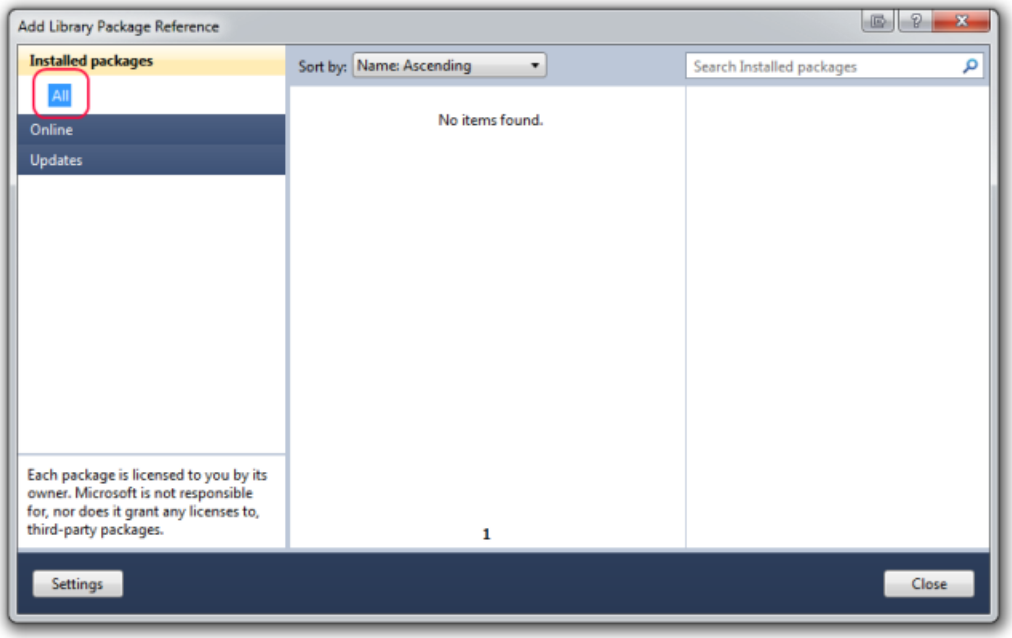
Use NuGet to install EFCodeFirst

In this section we are going to use the NuGet package manager (automatically installed by ASP.NET MVC 3) to add the EFCodeFirst library to the MvcMusicStore project. The NuGet package manager is installed with ASP.NET MVC 3.

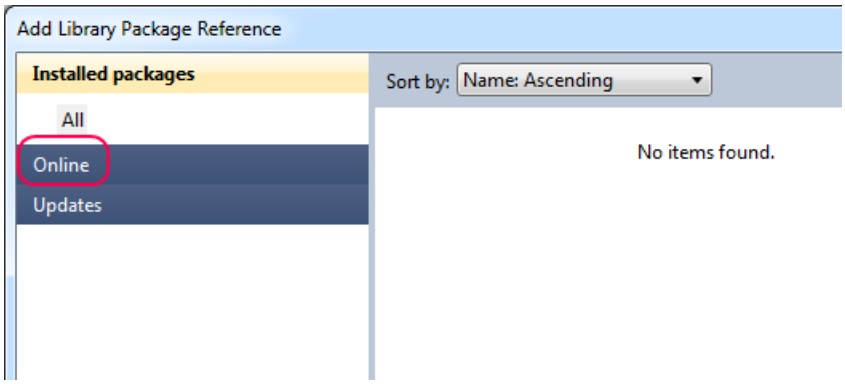
From the tools menu, select Library Package Manager\Add Library Package Reference



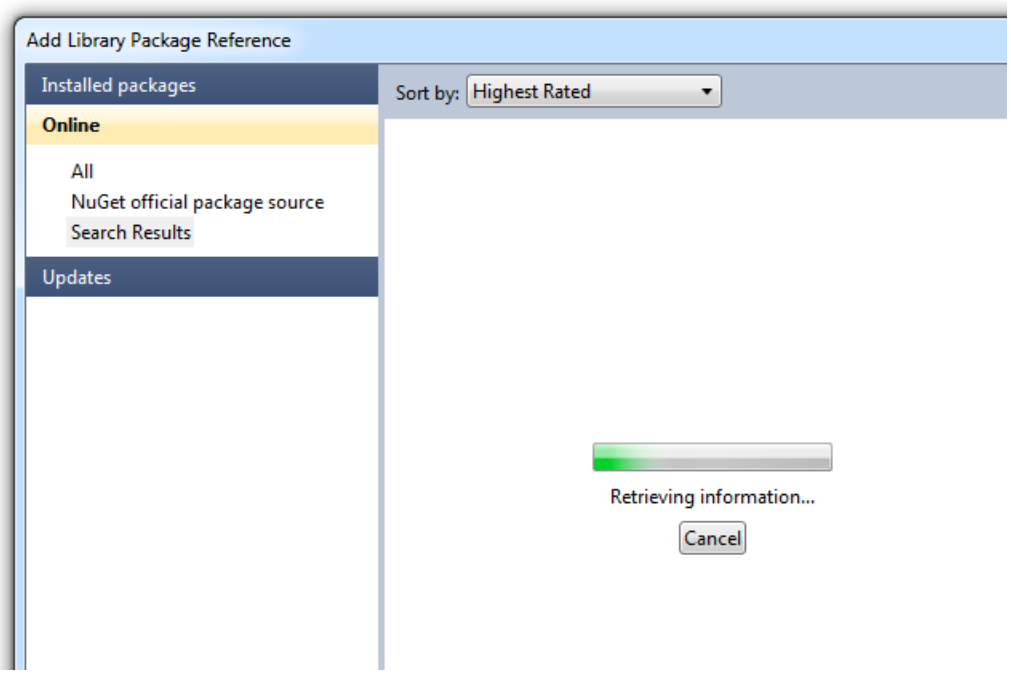
The Add Library Package Reference dialog box appears.



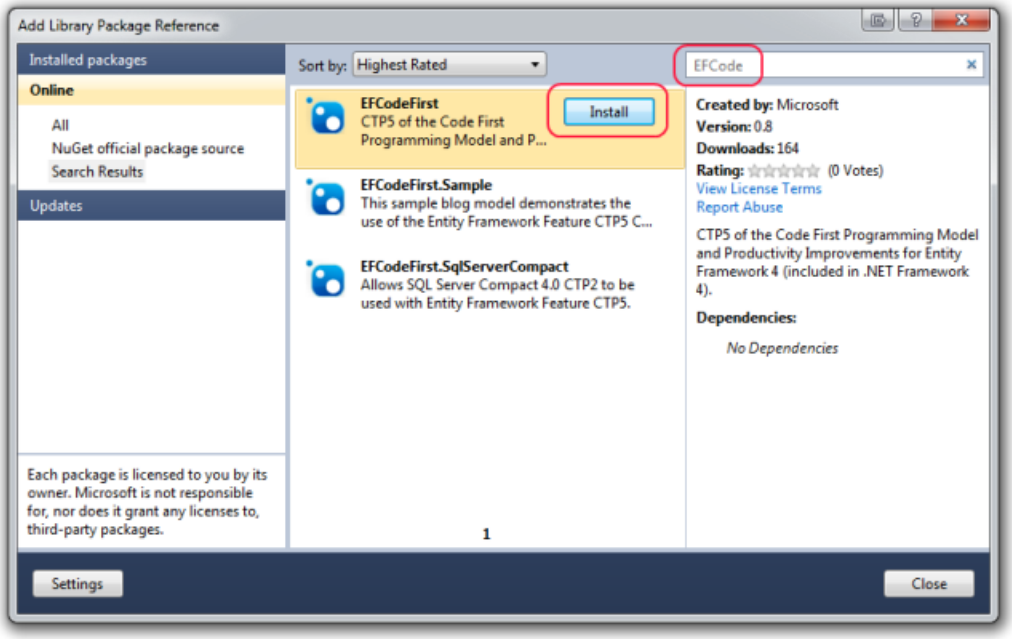
All is selected by default in the left pane. Because you have no packages installed, the center pane shows No items found. Select Online in the left pane.



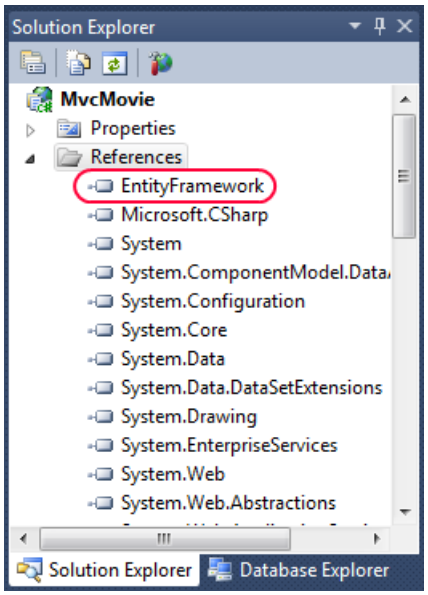
NuGET will query the server for all available packages.



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

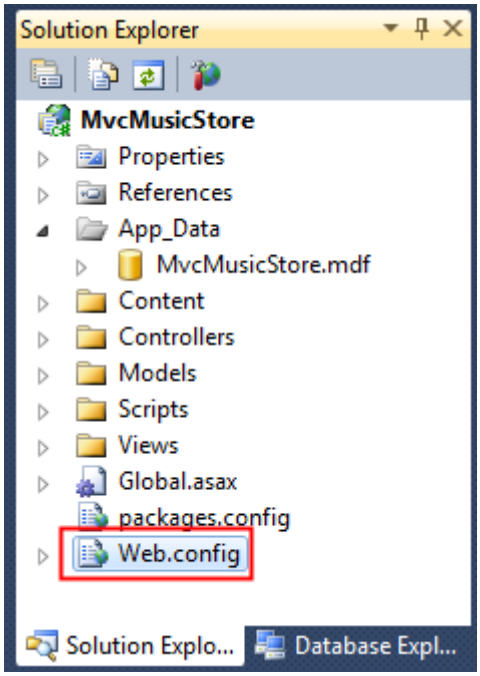


Once the package installs, select the Close button. The installation downloaded the EFCodeFirst library and added it to the MvcMusicStore project. The EFCodeFirst library is contained in the EntityFramework dll.



Creating a Connection String in the web.config file

We're going to add lines to the website's configuration file so that Entity Framework knows how to connect to our database. Double-click on the Web.config file located in the root of the project.



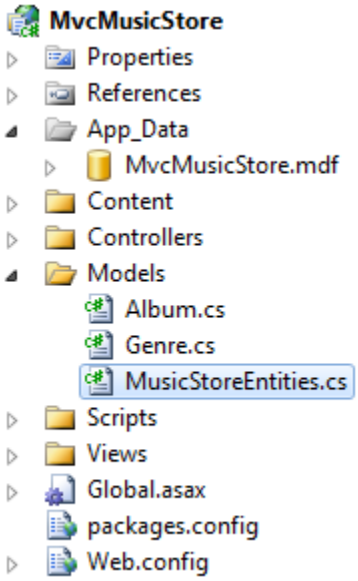
Scroll to the bottom of this file and add a <connectionStrings> section directly above the last line, as shown below.

```
<connectionStrings>
  <add name="MusicStoreEntities"
       connectionString="data source=.\SQLEXPRESS;
       Integrated Security=SSPI;
```

```
        AttachDBFilename=|DataDirectory|\MvcMusicStore.mdf;
        User Instance=true"
        providerName="System.Data.SqlClient" />
  </connectionStrings>
</configuration>
```

Adding a Context Class

Right-click the Models folder and add a new class named MusicStoreEntities.cs.



This class will represent the Entity Framework database context, and will handle our create, read, update, and delete operations for us. The code for this class is shown below.

```
using System.Data.Entity;

namespace MvcMusicStore.Models
{
    public class MusicStoreEntities : DbContext
    {
        public DbSet<Album> Albums { get; set; }
        public DbSet<Genre> Genres { get; set; }
    }
}
```

That's it - there's no other configuration, special interfaces, etc. By extending the DbContext base class, our MusicStoreEntities class is able to handle our database operations for us. Now that we've got that hooked up, let's add a few more properties to our model classes to take advantage of some of the additional information in our database.

Updating our Model Classes

Update the Album class as shown below.

```
namespace MvcMusicStore.Models
{
```

```
    public class Album
    {
        public int      AlbumId     { get; set; }
        public int      GenreId     { get; set; }
        public int      ArtistId    { get; set; }
        public string   Title       { get; set; }
        public decimal  Price       { get; set; }
        public string   AlbumArtUrl { get; set; }

        public Genre    Genre       { get; set; }
    }
}
```

Next, make the following updates to the Genre class.

```
using System.Collections.Generic;

namespace MvcMusicStore.Models
{
    public partial class Genre
    {
        public int      GenreId     { get; set; }
        public string   Name        { get; set; }
        public string   Description { get; set; }
        public List<Album> Albums   { get; set; }
    }
}
```

Querying the Database

Now let's update our StoreController so that instead of using "dummy data" it instead call into our database to query all of its information. We'll start by declaring a field on the StoreController to hold an instance of the MusicStoreEntities class, named storeDB:

```
public class StoreController : Controller
{
    MusicStoreEntities storeDB = new MusicStoreEntities();
```

Store Index using a LINQ Query Expression

The MusicStoreEntities class is maintained by the Entity Framework and exposes a collection property for each table in our database. We can use a cool capability of .NET called LINQ (language integrated query) to write strongly-typed query expressions against these collections – which will execute code against the database and return objects that we can easily program against.

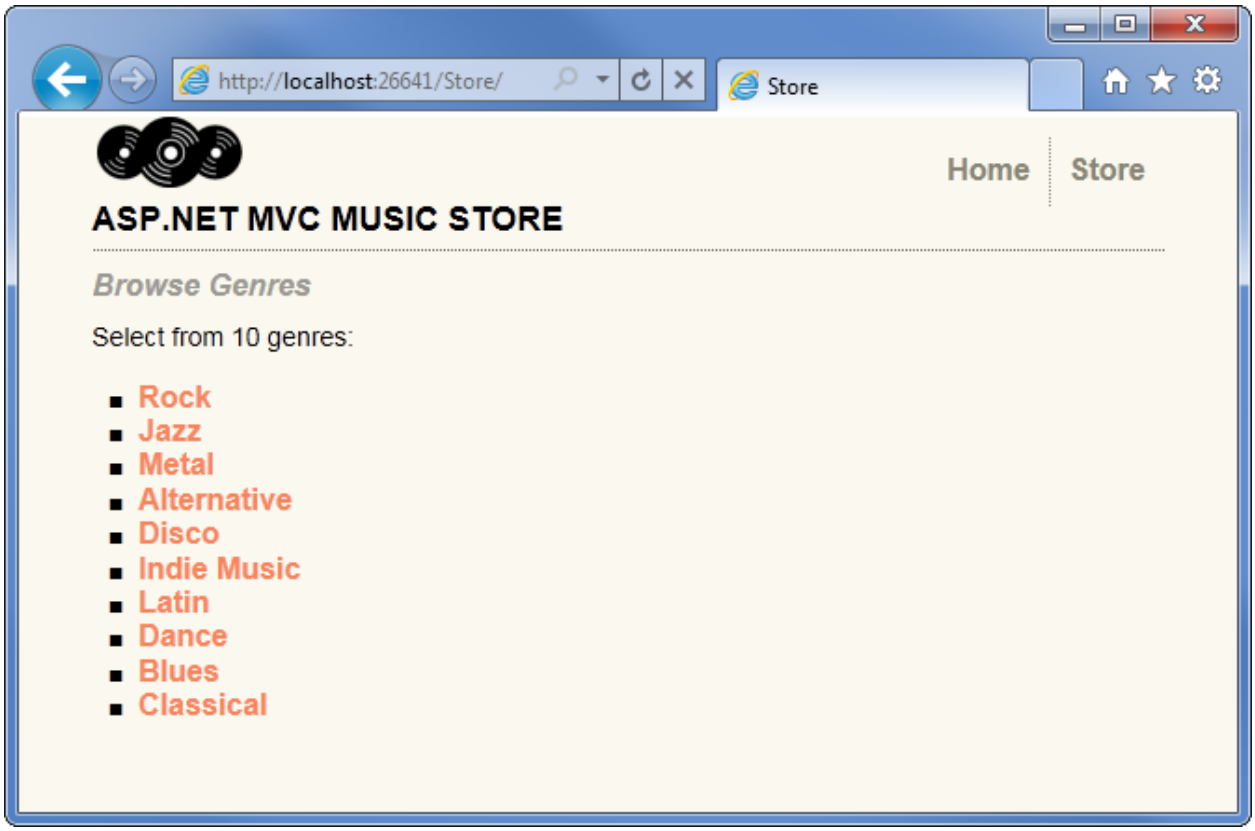
Let's update our StoreController's Index action to retrieve all Genre names in our database Previously we did this by hard-coding string data. Now we can instead write a LINQ query expression like below which retrieves the "Name" property of each Genre within our database:

```
public ActionResult Index()
{
    var genres = storeDb.Genres.ToList();
```

```
    return View(genres);
}
```

No changes need to happen to our View template since we're still returning the same StoreIndexViewModel we returned before - we're just returning live data from our database now.

When we run the project again and visit the "/Store" URL, we'll now see a list of all Genres in our database:



Store Browse, Details, and Index using a LINQ Extension Method

With the /Store/Browse?genre=*[some-genre]* action method, we're searching for a Genre by name. We only expect one result, since we shouldn't ever have two entries for the same Genre name, and so we can use the .Single() extension in LINQ to query for the appropriate Genre object like this:

```
var example = storeDB.Genres.Single(g => g.Name == "Disco");
```

The Single method takes a Lambda expression as a parameter, which specifies that we want a single Genre object such that its name matches the value we've defined. In the case above, we are loading a single Genre object with a Name value matching Disco.

We'll take advantage of an Entity Framework feature that allows us to indicate other related entities we want loaded as well when the Genre object is retrieved. This feature is called Query Result Shaping, and enables us to reduce the number of times we need to access the database to retrieve all of the information we need. We want to pre-fetch the Albums for Genre we retrieve, so we'll update our query to include from

Genres.Include("Albums") to indicate that we want related albums as well. This is more efficient, since it will retrieve both our Genre and Album data in a single database request.

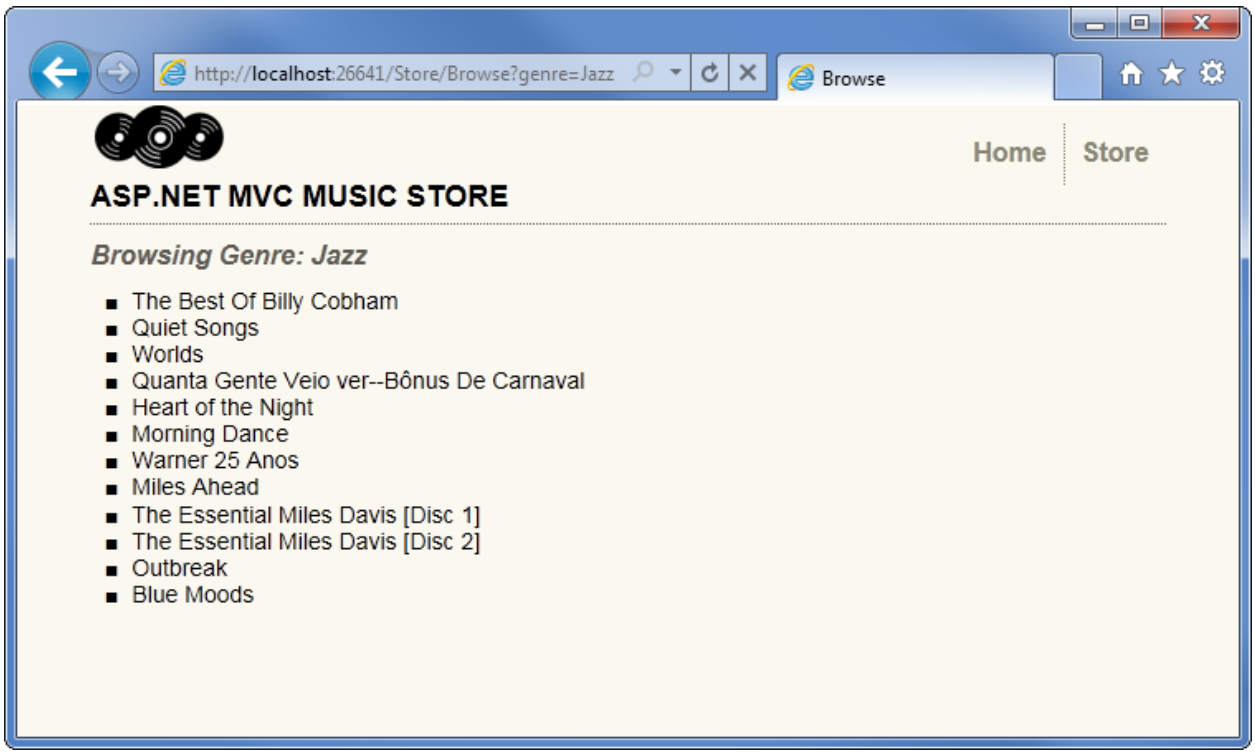
With the explanations out of the way, here's how our updated Browse controller action looks:

```
public ActionResult Browse(string genre)
{
    // Retrieve Genre and its Associated Albums from database
    var genreModel = storeDB.Genres.Include("Albums")
        .Single(g => g.Name == genre);

    return View(genreModel);
}
```

We can now update the Store Browse View to display the albums which are available in each Genre. Open the view template (found in /Views/Store/Browse.cshtml) and add a bulleted list of Albums as shown below.

```
@model MvcMusicStore.Models.Genre
@{
    ViewBag.Title = "Browse";
}
<h2>Browsing Genre: @Model.Name</h2>

<ul>
    @foreach (var album in Model.Albums)
    {
        <li>
            @album.Title
        </li>
    }
</ul>
```

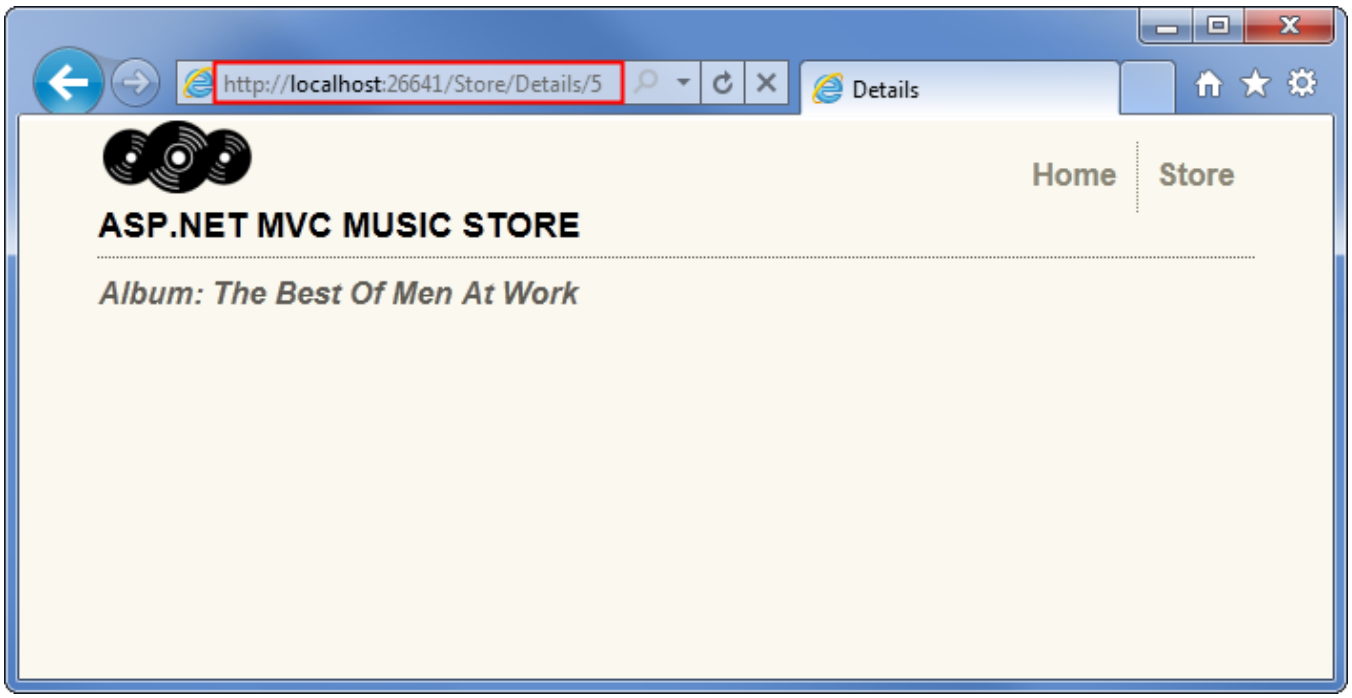
Running our application and browsing to /Store/Browse?genre=Jazz shows that our results are now being pulled from the database, displaying all albums in our selected Genre.



We'll make the same change to our /Store/Details/[id] URL, and replace our dummy data with a database query which loads an Album whose ID matches the parameter value.

```
public ActionResult Details(int id)
{
    var album = storeDB.Albums.Find(id);

    return View(album);
}
```

Running our application and browsing to /Store/Details/500 shows that our results are now being pulled from the database.



Now that our Store Details page is set up to display an album by the Album ID, let's update the Browse view to link to the Details view. We will use Html.ActionLink, exactly as we did to link from Store Index to Store Browse at the end of the previous section. The complete source for the Browse view appears below.

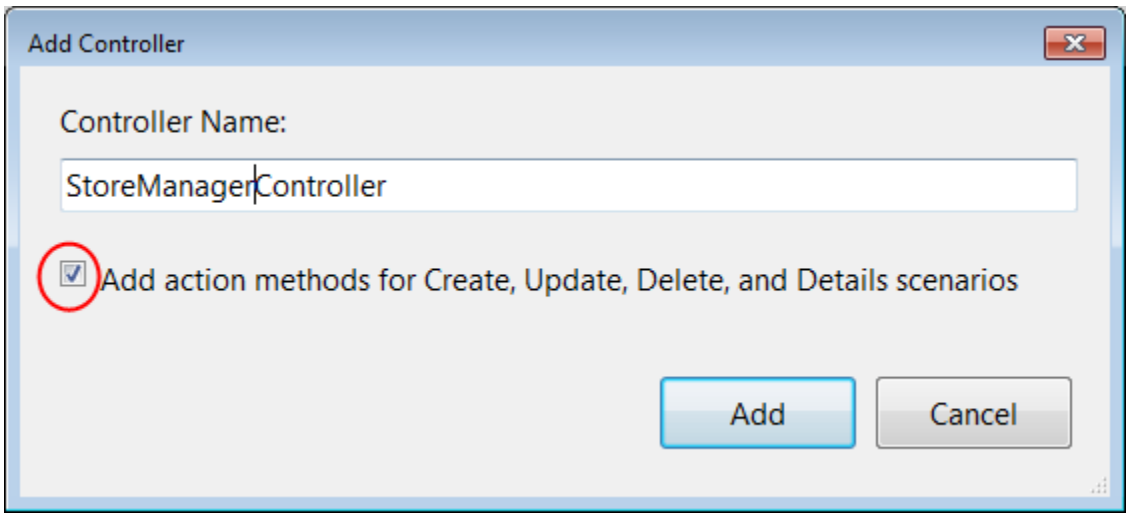
```
@model MvcMusicStore.Models.Genre
@{
    ViewBag.Title = "Browse";
}
<h2>Browsing Genre: @Model.Name</h2>

<ul>
    @foreach (var album in Model.Albums)
    {
        <li>
            @Html.ActionLink(album.Title, "Details", new { id = album.AlbumId })
        </li>
    }
</ul>
```

We're now able to browse from our Store page to a Genre page, which lists the available albums, and by clicking on an album we can view details for that album.

5. Edit Forms and Templating

In the past chapter, we were loading data from our database and displaying it. In this chapter, we'll also enable editing the data.

We'll begin by creating a new controller called StoreManagerController. This controller will support Create and Update actions, so we'll check the checkbox to "Add action methods for Create, Update, and Details scenarios." when we create the controller class:



This generates a new controller class that has stub methods for common CRUD controller actions, with TODO comments filled in to prompt us to put in our application specific logic.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace MvcMusicStore.Controllers
{
    public class StoreManagerController : Controller
    {
        //
        // GET: /StoreManager/

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

        //
        // GET: /StoreManager/Create

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

```
//
// POST: /StoreManager/Create

[HttpPost]
public ActionResult Create(FormCollection collection)
{
    try
    {
        // TODO: Add insert logic here

        return RedirectToAction("Index");
    }
    catch
    {
        return View();
    }
}

//
// GET: /StoreManager/Edit/5

public ActionResult Edit(int id)
{
    return View();
}

//
// POST: /StoreManager/Edit/5

[HttpPost]
public ActionResult Edit(int id, FormCollection collection)
{
    try
    {
        // TODO: Add update logic here

        return RedirectToAction("Index");
    }
    catch
    {
        return View();
    }
}

//
// GET: /StoreManager/Delete/5

public ActionResult Delete(int id)
{
    return View();
}

//
// POST: /StoreManager/Delete/5
```

```
        [HttpPost]
        public ActionResult Delete(int id, FormCollection collection)
        {
            try
            {
                // TODO: Add delete logic here

                return RedirectToAction("Index");
            }
            catch
            {
                return View();
            }
        }
    }
}
```

We don't need the Details controller action, so we can delete that method.

Adding the Artist class

Our Store Manager controller will be managing Artist information for our Albums, so before we write our controller code we'll add a simple Artist model class. Right-click the Models folder and add a class named Artist.cs, adding the following code to it.

```
namespace MvcMusicStore.Models
{
    public class Artist
    {
        public int ArtistId { get; set; }
        public string Name { get; set; }
    }
}
```

Next, we'll add an Artist property to our Album class. Entity Framework will understand how to load this information based on naming convention, and will be able to populate it without any additional work. The updated Album class appears below.

```
namespace MvcMusicStore.Models
{
    public class Album
    {
        public int      AlbumId     { get; set; }
        public int      GenreId     { get; set; }
        public int      ArtistId    { get; set; }
        public string   Title       { get; set; }
        public decimal  Price       { get; set; }
        public string   AlbumArtUrl { get; set; }

        public virtual Genre    Genre       { get; set; }
        public virtual Artist   Artist      { get; set; }
    }
}
```

Finally, we'll add a DbSet<Artist> to our MusicStoreEntities class so that we can work with Artist data later, without having to load them through the Album class.

```
using System.Data.Entity;

namespace MvcMusicStore.Models
{
    public class MusicStoreEntities : DbContext
    {
        public DbSet<Album>     Albums  { get; set; }
        public DbSet<Genre>     Genres  { get; set; }
        public DbSet<Artist>    Artists { get; set; }
    }
}
```

Customizing the Store Manager Index

As in our Store Controller, we'll begin by adding a field on our StoreManagerController to hold an instance of our MusicStoreEntities. First, add a using statement to reference the MvcMusicStore.Models namespace.

```
using MvcMusicStore.Models;
```

Now add a the storeDB field, as shown below.

```
public class StoreManagerController : Controller
{
    MusicStoreEntities storeDB = new MusicStoreEntities();
```

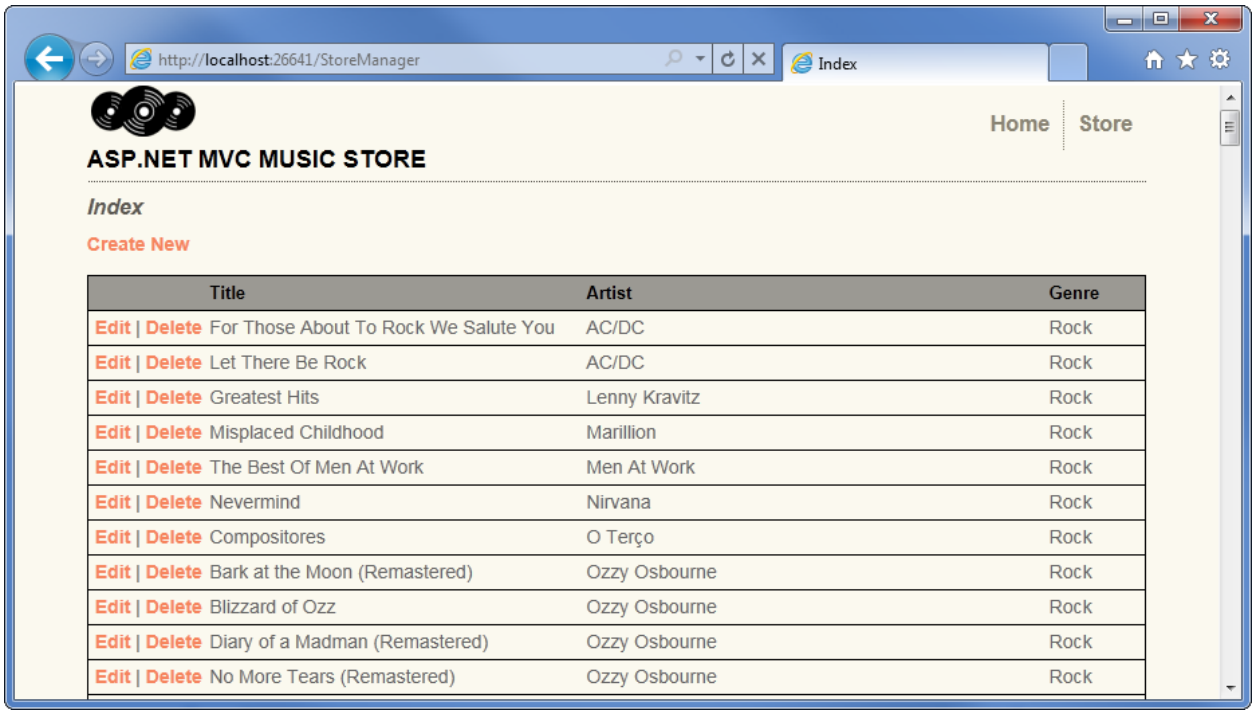
Now we will implement the Store Manager Index action. This action will display a list of albums, so the controller action logic will be pretty similar to the Store Controller's Index action we wrote earlier. We'll use LINQ to retrieve all albums, including Genre and Artist information for display.

```
//
// GET: /StoreManager/

public ActionResult Index()
{
    var albums = storeDB.Albums
        .Include("Genre").Include("Artist")
        .ToList();

    return View(albums);
}
```

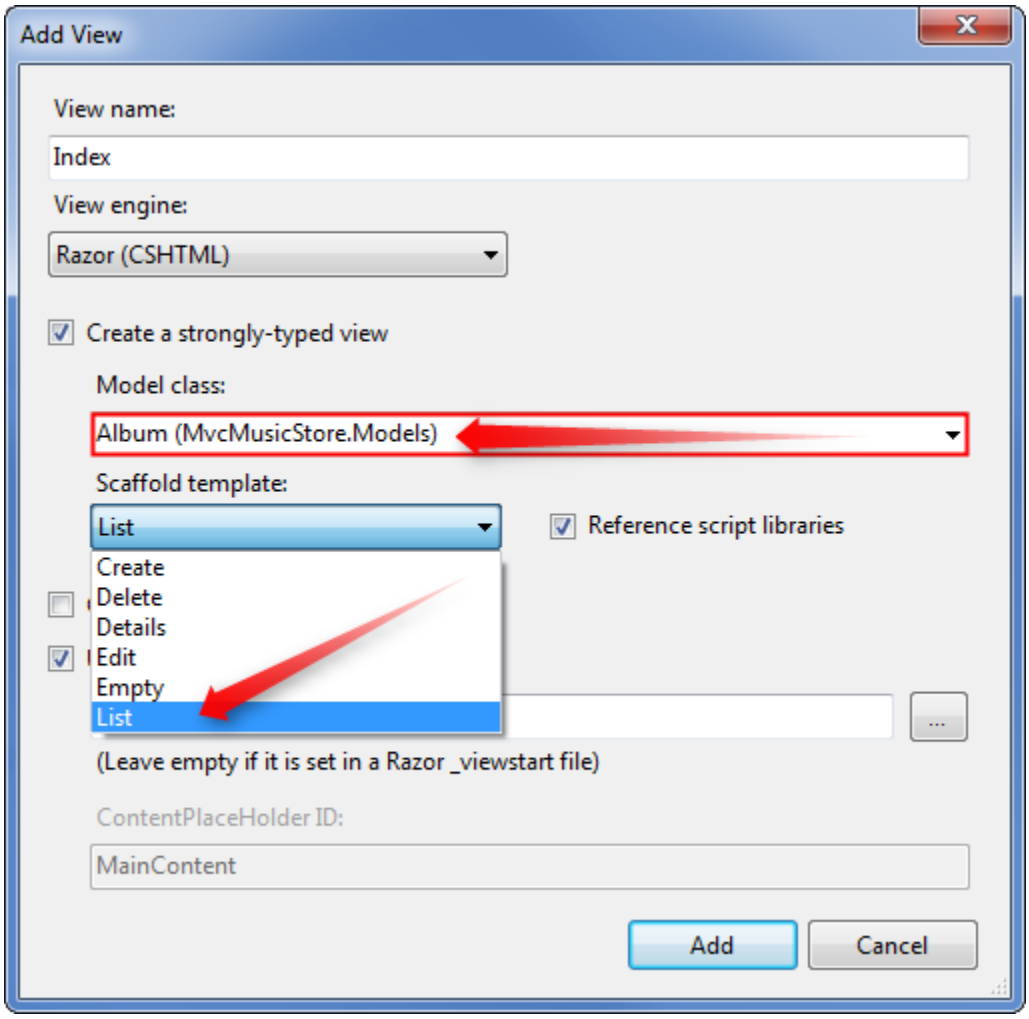
Scaffold View templates

Our Store Index view will make use of the *scaffolding* feature in ASP.NET MVC. Scaffolding allows us to automatically generate view templates that are based on the type of object passed to the view by the Controller action method. The scaffolding infrastructure uses reflection when creating view templates – so it can scaffold new templates based on any object passed to it.

Scaffolding provides a quick way to get started on a strongly typed view. Rather than having to write the view template manually, we can use scaffolding to quickly generate a default template and then just modify the generated code. In this case, by generating a view which is strongly typed to the Album class using the List template, we can generate a view which displays a table which lists information about all the albums, like the example shown below.



As before, we'll right-click within the Index() action method to bring up the Add View dialog. We'll adjust two settings on this dialog. First, check the "Create a strongly-typed view" and select the Album class from the dropdown. Next, we'll set the "Scaffold template" to List, which will generate a Scaffold View template. We'll discuss Scaffold View templates next.



Let's look at the generated template code (I've done some minor reformatting):

```
@model IEnumerable<MvcMusicStore.Models.Album>

@{
    ViewBag.Title = "Index";
}

<h2>Index</h2>

<p>
    @Html.ActionLink("Create New", "Create")
</p>
<table>
    <tr>
        <th></th>
        <th>GenreId</th>
        <th>ArtistId</th>
        <th>Title</th>
```

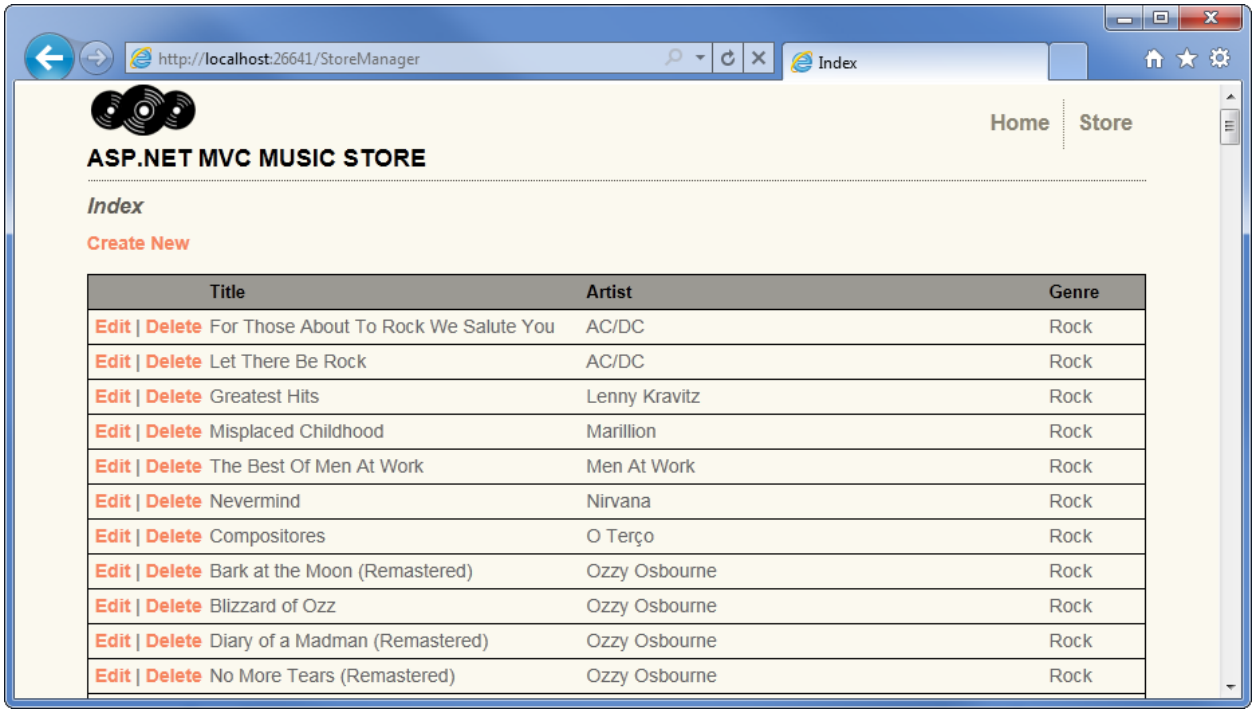
```
        <th>Price</th>
        <th>AlbumArtUrl</th>
    </tr>

@foreach (var item in Model) {
    <tr>
        <td>
            @Html.ActionLink("Edit", "Edit", new { id=item.AlbumId }) |
            @Html.ActionLink("Details", "Details", new { id=item.AlbumId }) |
            @Html.ActionLink("Delete", "Delete", new { id=item.AlbumId })
        </td>
        <td>@item.GenreId</td>
        <td>@item.ArtistId</td>
        <td>@item.Title</td>
        <td>@String.Format("{0:F}", item.Price)</td>
        <td>@item.AlbumArtUrl</td>
    </tr>
}

</table>
```

Note: That this template is following the same coding practices we've been learning so far in our hand-written templates. For example, it's using Html.ActionLink to generate links to other controller actions.

We just want to display Album Title, Artist, and Genre, so we can delete the AlbumId, Price, and Album Art URL columns. The GenreId and ArtistId aren't near as useful as the Artist and Genre Names, so we'll change them to display the linked class properties as follows:

```
<p>
    @Html.ActionLink("Create New", "Create")
</p>
<table>
    <tr>
        <th></th>
        <th>Title</th>
        <th>Artist</th>
        <th>Genre</th>
    </tr>

@foreach (var item in Model) {
    <tr>
        <td>
            @Html.ActionLink("Edit", "Edit", new { id=item.AlbumId }) |
            @Html.ActionLink("Delete", "Delete", new { id=item.AlbumId })
        </td>
        <td>@item.Title</td>
        <td>@item.Artist.Name</td>
        <td>@item.Genre.Name</td>
    </tr>
}

</table>
```

You can run the application and browse to /StoreManager to see the list.



	Title	Artist	Genre
Edit | Delete	For Those About To Rock We Salute You	AC/DC	Rock
Edit | Delete	Let There Be Rock	AC/DC	Rock
Edit | Delete	Greatest Hits	Lenny Kravitz	Rock
Edit | Delete	Misplaced Childhood	Marillion	Rock
Edit | Delete	The Best Of Men At Work	Men At Work	Rock
Edit | Delete	Nevermind	Nirvana	Rock
Edit | Delete	Compositores	O Terço	Rock
Edit | Delete	Bark at the Moon (Remastered)	Ozzy Osbourne	Rock
Edit | Delete	Blizzard of Ozz	Ozzy Osbourne	Rock
Edit | Delete	Diary of a Madman (Remastered)	Ozzy Osbourne	Rock
Edit | Delete	No More Tears (Remastered)	Ozzy Osbourne	Rock

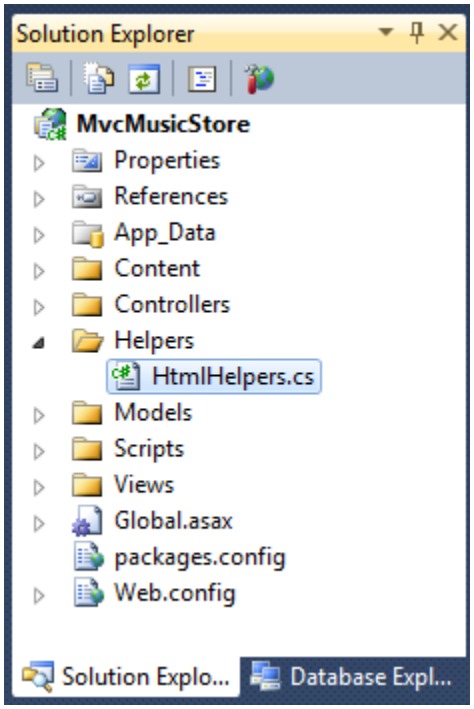
Using a custom HTML Helper to truncate text

We've got one potential issue with our Store Manager Index page. Our Album Title and Artist Name properties can both be long enough that they could throw off our table formatting. We'll create a custom HTML Helper to allow us to easily truncate these and other properties in our Views.

	Title	Artist	Genre
Edit | Delete	For Those About To Rock W...	AC/DC	Rock
Edit | Delete	Let There Be Rock	AC/DC	Rock
Edit | Delete	Greatest Hits	Lenny Kravitz	Rock
Edit | Delete	Misplaced Childhood	Marillion	Rock
Edit | Delete	The Best Of Men At Work	Men At Work	Rock
Edit | Delete	Nevermind	Nirvana	Rock
Edit | Delete	Compositores	O Terço	Rock
Edit | Delete	Bark at the Moon (Remaste...	Ozzy Osbourne	Rock
Edit | Delete	Blizzard of Ozz	Ozzy Osbourne	Rock
Edit | Delete	Diary of a Madman (Remast...	Ozzy Osbourne	Rock
Edit | Delete	No More Tears (Remastered...	Ozzy Osbourne	Rock

Note: This topic is a bit advanced, so if it doesn't make sense to you, don't worry about it. Learning to write your own HTML Helpers can simplify your code, but it's not a fundamental topic that you need to master to complete this tutorial.

Add a new directory named Helpers, and add a class to it named HtmlHelpers.cs.



Our HTML Helper will add a new "Truncate" method to the "Html" object exposed within ASP.NE T MVC Views. We'll do this by implementing an "extension method" to the built-in System.Web.Mvc.HtmlHelper class provided by ASP.NET MVC. Our helper class and method must both be static. Other than that, it's pretty simple.

```
using System.Web.Mvc;

namespace MvcMusicStore.Helpers
{
    public static class HtmlHelpers
    {
        public static string Truncate(this HtmlHelper helper, string input, int length)
        {
            if (input.Length <= length)
            {
                return input;
            }
            else
            {
                return input.Substring(0, length) + "...";
            }
        }
    }
}
```

This helper method takes a string and a maximum length to allow. If the text supplied is shorter than the length specified, the helper outputs it as-is. If it is longer, then it truncates the text and renders "…" for the remainder.

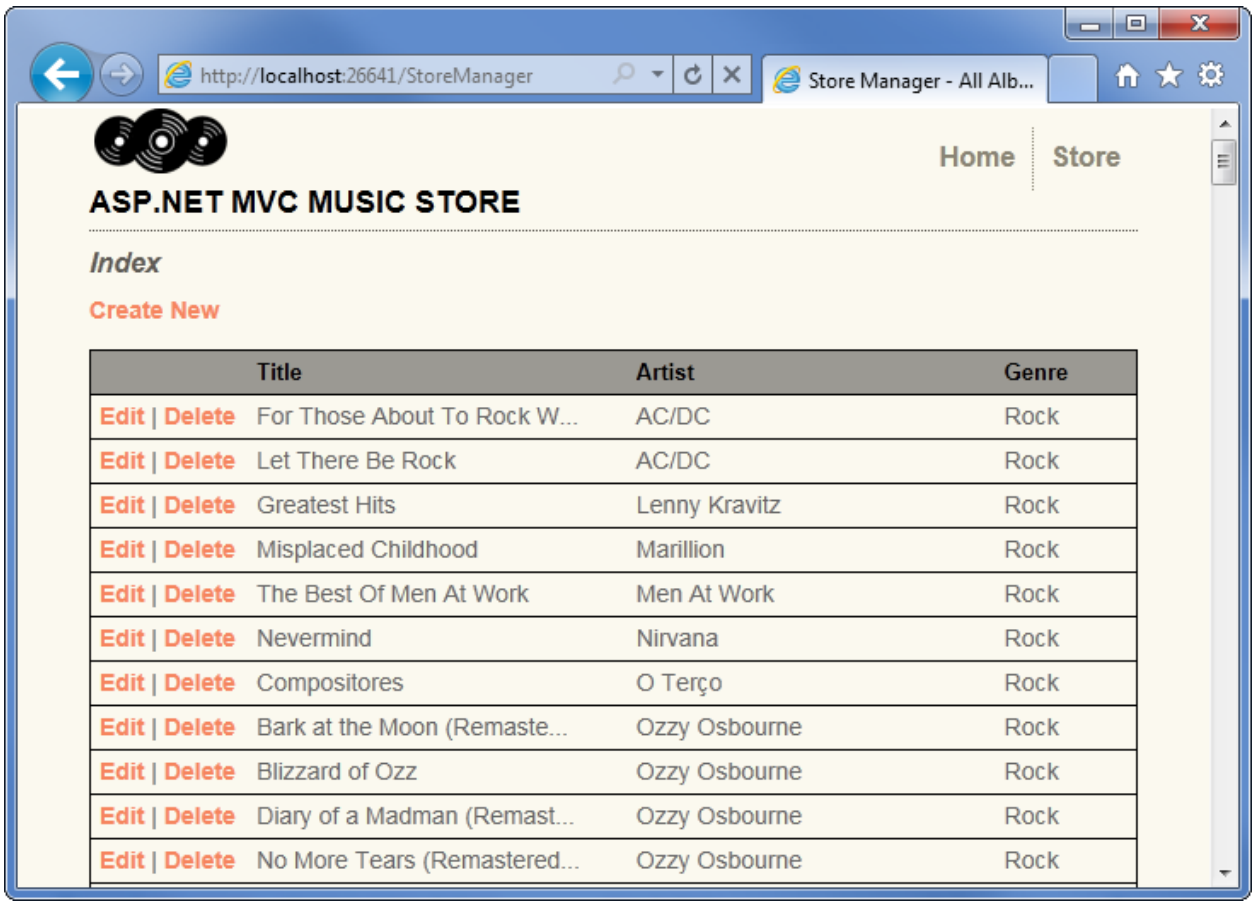
To use our custom HTML helper we need to add the namespace reference to our View, just as we add using statements in our Controller code. Add using statement just below the @model line at the top of our StoreManager Index page, like this:

```
@model IEnumerable<MvcMusicStore.Models.Album>
@using MvcMusicStore.Helpers
```

Now we can use our Html.Truncate helper to ensure that both the Album Title and Artist Name properties are less than 25 characters. The complete view code using our new Truncate helper appears below.

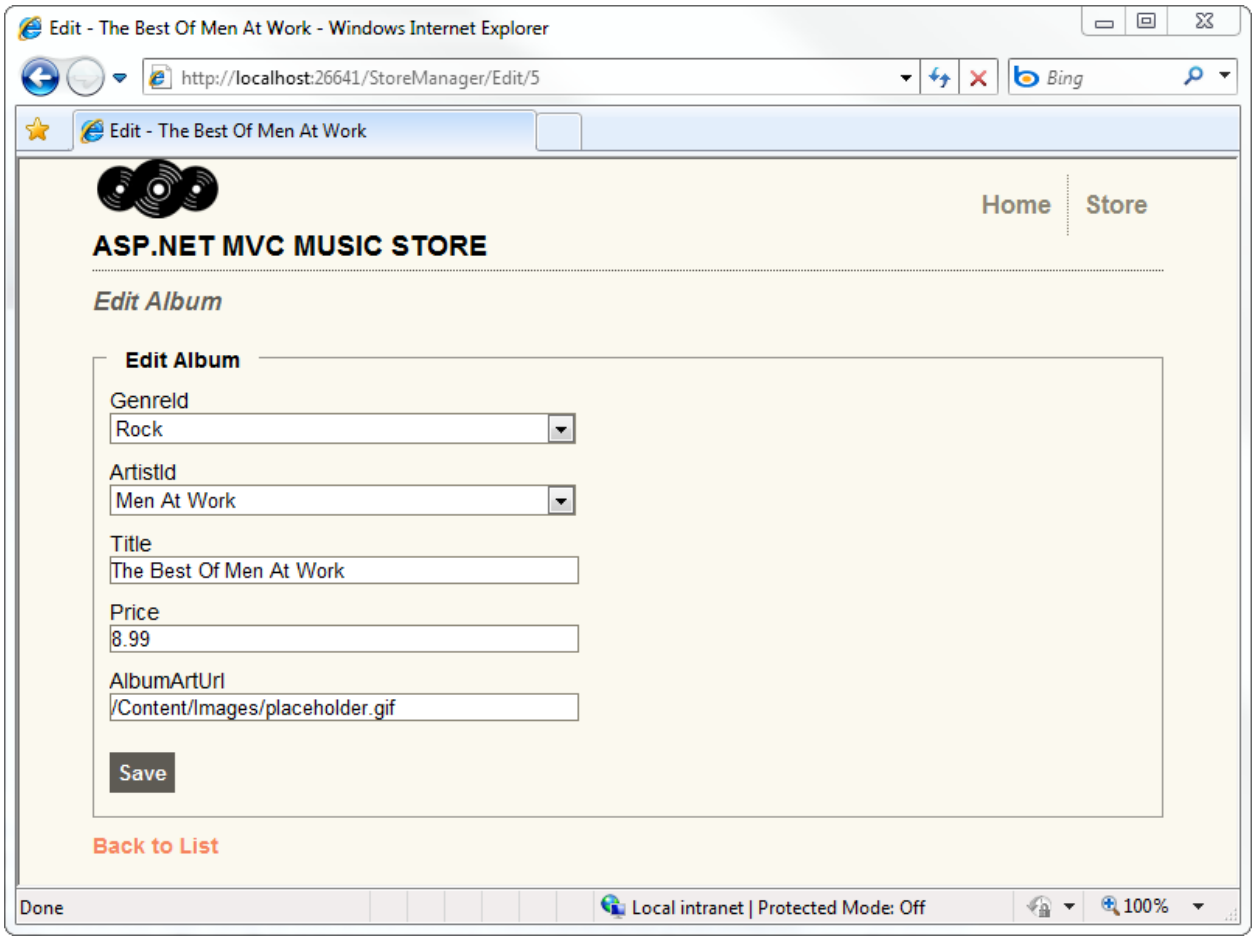
```
@model IEnumerable<MvcMusicStore.Models.Album>
@using MvcMusicStore.Helpers

@{
    ViewBag.Title = "Store Manager - All Albums";
}

<h2>Index</h2>

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

@foreach (var item in Model) {
    <tr>
        <td>
            @Html.ActionLink("Edit", "Edit", new { id=item.AlbumId }) |
            @Html.ActionLink("Delete", "Delete", new { id=item.AlbumId })
        </td>
        <td>@Html.Truncate(item.Title, 25) </td>
        <td>@Html.Truncate(item.Artist.Name, 25)</td>
        <td>@item.Genre.Name</td>
    </tr>
}

</table>
```

Now when we browse the /StoreManager/ URL, the albums and titles are kept below our maximum lengths.



Creating the Edit View

Let's next create a form to allow mangers to edit an album. It will include text fields for Album Title, Price, and Album Art URL. Later, we will add drop-downs to enable selecting the Artist and Genre from a list.



Implementing the Edit Action Methods

Store Managers will be able to edit an Album by visiting the /StoreManager/Edit/[id] URL – where the [id] value in the URL indicates the unique ID of the album to edit.

When a manager first visits this URL we will run code within our application to retrieve the appropriate Album from the database, create an Album object to encapsulate it and a list of Artists and Genres, and then pass our Album object off to a view template to render a HTML page back to the user. This HTML page will contain a <form> element that contains textboxes with our Album's details.

The user can make changes to the Album form values, and then press the "Save" button to submit these changes back to our application to save within the database. When the user presses the "save" button the <form> will perform an HTTP-POST back to the /StoreManager/Edit/[id] URL and submit the <form> values as part of the HTTP-POST.

ASP.NET MVC allows us to easily split up the logic of these two URL invocation scenarios by enabling us to implement two separate "Edit" action methods within our StoreManagerController class – one to handle the initial HTTP-GET browse to the /StoreManager/Edit/[id] URL, and the other to handle the HTTP-POST of the submitted changes.

We can do this by defining two "Edit" action methods like so:

```
//
// GET: /StoreManager/Edit/5

public ActionResult Edit(int id)
{
    //Display Edit form
}

//
// POST: /StoreManager/Edit/5

[HttpPost]
public ActionResult Edit(int id, FormCollection formValues)
{
    //Save Album
}
```

ASP.NET MVC will automatically determine which method to call depending on whether the incoming /StoreManager/Edit/[id] URL is an HTTP-GET or HTTP-POST. If it is a HTTP-POST, then the second method will be called. If it is anything else (including an HTTP-GET), the first method will be called.

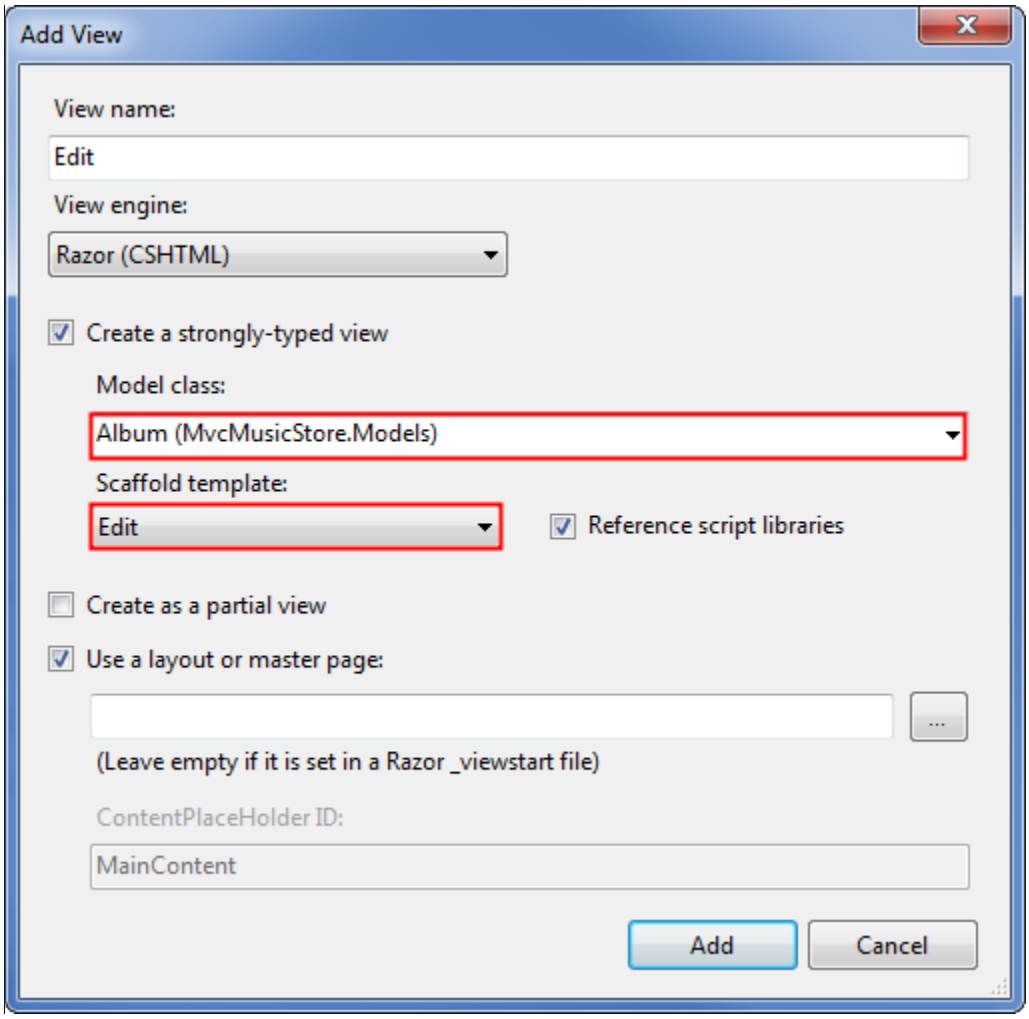
Writing the HTTP-GET Edit Controller Action

We will implement the HTTP-GET version of our "Edit" action method using the code below. It retrieves the appropriate Album from the database, then passes it to a View template to render the response.

```
//
// GET: /StoreManager/Edit/5

public ActionResult Edit(int id)
{
    Album album = storeDB.Albums.Find(id);

    return View(album);
}
```

Creating the Edit View

We'll create an Edit view template by right-clicking within the Edit action method and selecting the Add⇨View menu command. The Edit view should be strongly typed to the Album, and we'll select the "Edit" scaffold template from the "view content" dropdown:

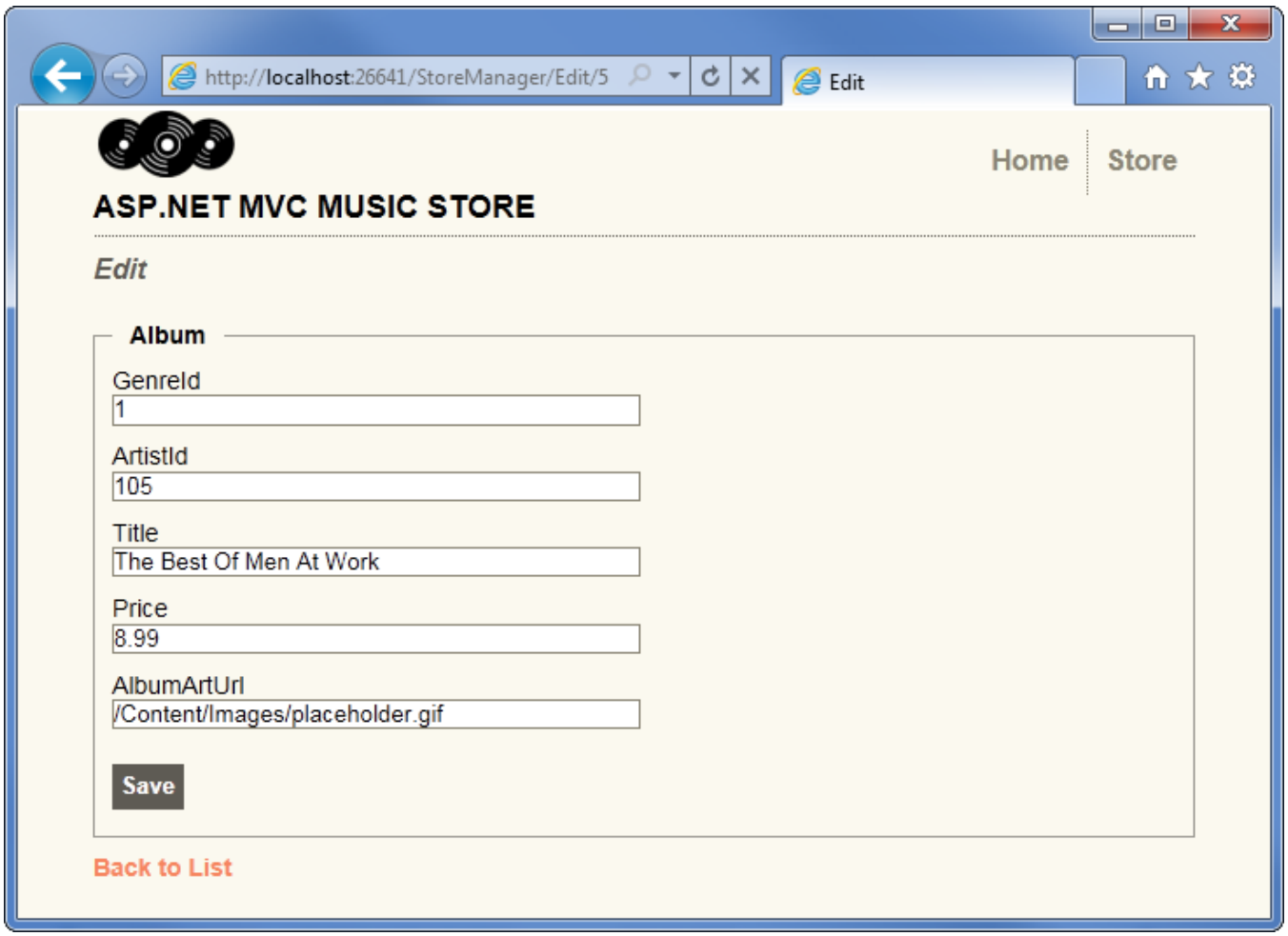


Here's the markup that's generated by the default Edit view template for the Album.

```
@model MvcMusicStore.Models.Album

@{
    ViewBag.Title = "Edit";
}

<h2>Edit</h2>

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

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

        @Html.HiddenFor(model => model.AlbumId)
```

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

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

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

        <p>
            <input type="submit" value="Save" />
        </p>
    </fieldset>
}

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

Let's now run our application and visit the /StoreManager/Edit/5 URL. We'll see that the above Edit view displayed an Album editing UI like below:



Using an Editor Template

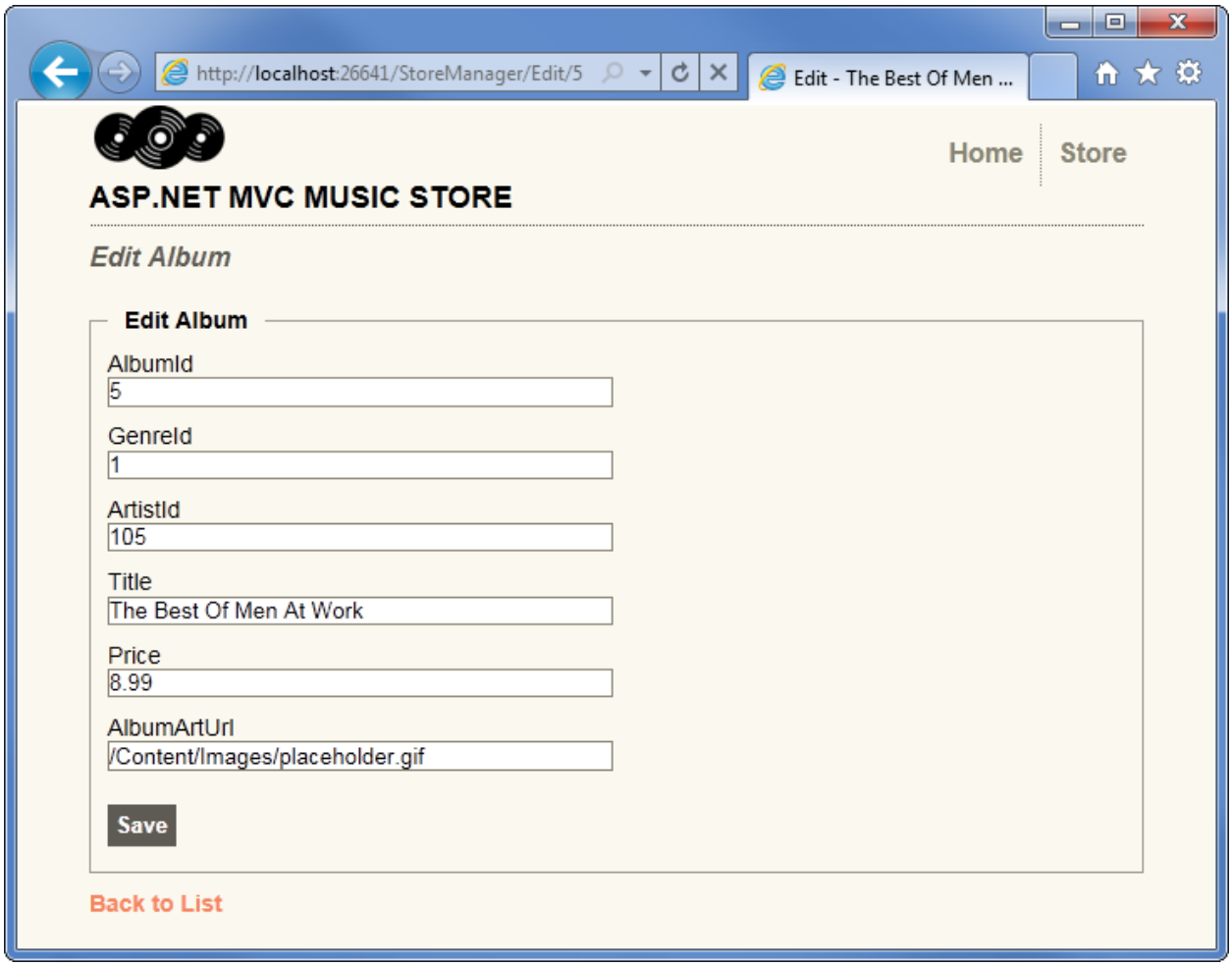
We just saw the view code in a generated Editor template, as well as the form that's shown when the view is displayed. Next, we'll look at another way to create an Edit form by convert this page to use the built-in Html.EditorFor() helper method. This has some advantages, including the ability to re-use the form in other views within our application.

Let's update our view template to contain the following code:

```
@model MvcMusicStore.Models.Album

@{
    ViewBag.Title = "Edit - " + Model.Title;
}

<h2>Edit Album</h2>

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

```
@using (Html.BeginForm()) {
    @Html.ValidationSummary(true)
    <fieldset>
        <legend>Edit Album</legend>
        @Html.EditorForModel()
        <p>
            <input type="submit" value="Save" />
        </p>
    </fieldset>
}

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

Above we are indicating that we want to create an Editing view for our model. Run the application and visit the /StoreManager/Edit/5 URL again. An almost identical form is displayed - the only difference is that the EditorForModel form shows the AlbumID, whereas it as rendered as a hidden field before.



We've seen two ways to generate an Edit form:

1) We can use the Editor View template to generate edit fields for all the model properties at design time, meaning that the code which displays the individual form fields is visual in the Edit.cshtml view.
2) Alternatively, we can use the templating feature in ASP.NET MVC to generate the edit fields for the form when the form is displayed.

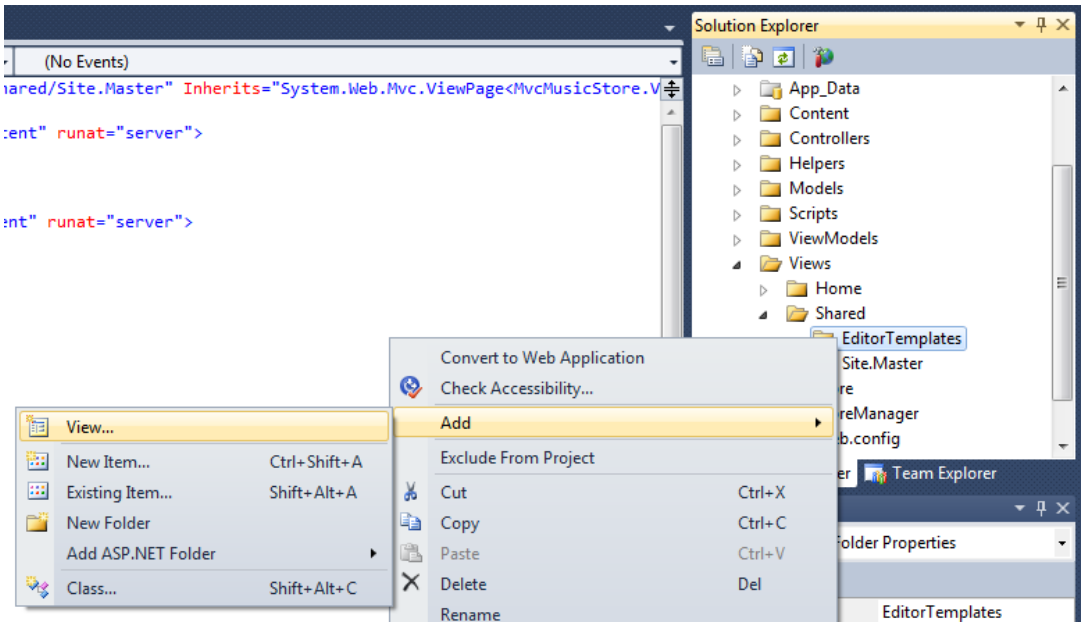
Both of these options are useful in different scenarios. In this case, we'll continue to use the Html.EditorForModel() templating feature to allow for reuse other Views within the site.

Creating a Shared Album Editor Template

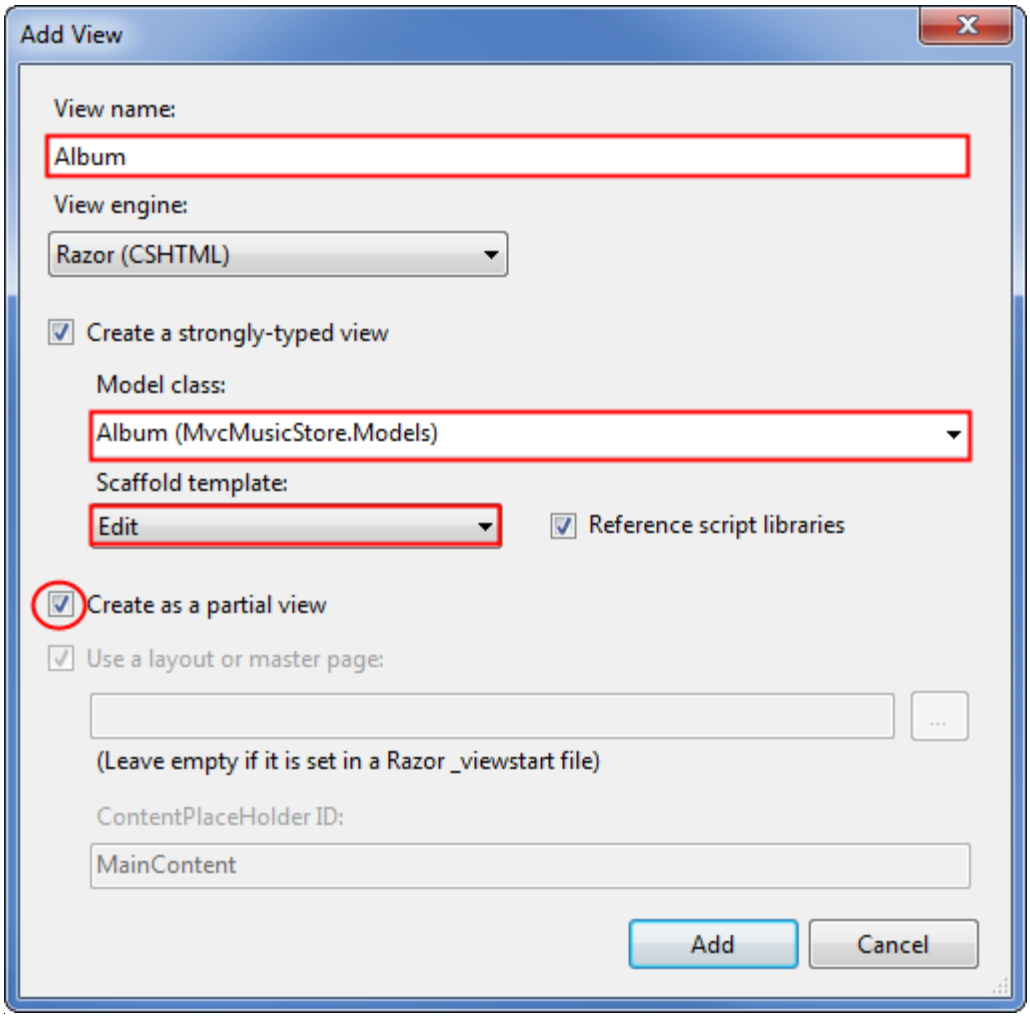
Our Album Edit view and Album Create view will have the same form fields, so we can make use of a shared Album Editor Template. Reusing an Edito rTemplate throughout our site gives our users a consistent user interface. It also makes it easier to manage and maintain our code by eliminating duplicated code between views.

ASP.NET MVC follows a convention based approach for templates, so both the folder and filenames are important. When we use Html.EditorForModel() in a view, the MVC runtime checks for a template with the same name as our Model in /Views/EditorTemplates and if it finds one, it will use that rather than use the default template. This allows us to customize the display for any model type.

We need to create an EditorTemplates folder inside the /Views/Shared folder. First create a new folder and name it EditorTemplates, then add a new View template as shown below.



This view will be a little different than the views we've been creating so far. It will be a Partial View, meaning that it's intended to be displayed inside another view. Name the view Album, select Album for the View data class, set the View content template to Edit, and press the Add button.

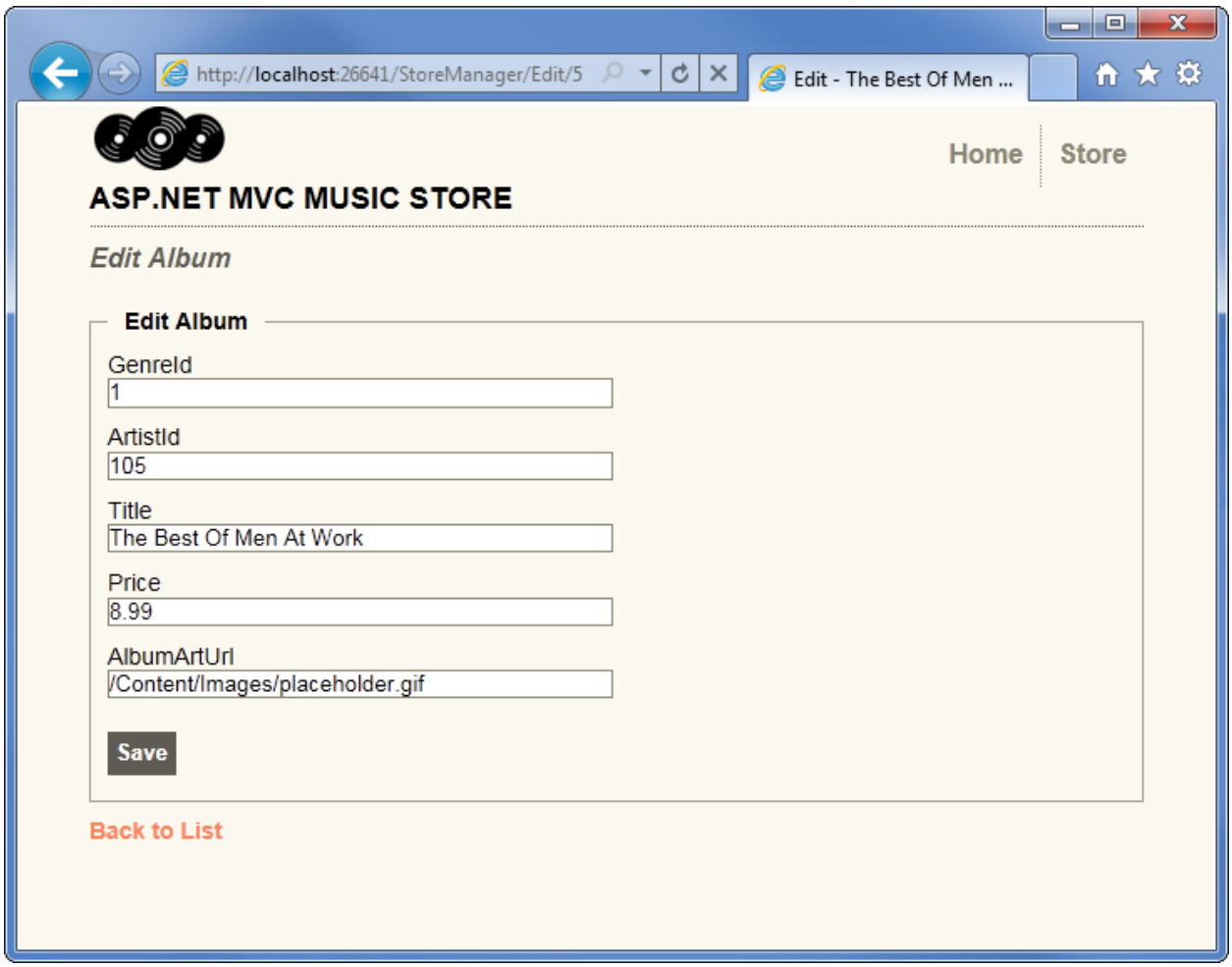


The generated view code for Album.cshtml contains a form tag, but since we will be using it in views which already have a surrounding form tag, we can remove the surrounding tags, as shown below.

```
@model MvcMusicStore.Models.Album

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

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

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

```
</div>

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

Running the application and browsing to /StoreManager/Edit/5 will show that we are using the Album edit form, as the AlbumId is being rendered as a hidden form field.



Now it's time to customize the Album editor template.

Passing additional information to Views using ViewBag

The next step is to change our Edit form to display dropdowns for Album and Genre, rather than just showing textboxes for AlbumId and GenreID. In order to do that, our View is going to need several bits of data. Specifically, it will need:

1) An Album object that represents the current Album being edited
2) A List of all Genres – we'll use this to populate the Genre dropdownlist
3) A List of all Artists – we'll use this to populate the Artist dropdownlist

Controllers can pass additional information - such as our Genre and Artist lists - to the view using the ViewBag class.

To do that, we will change the StoreManagerController's Edit action as follows:

```
public ActionResult Edit(int id)
{
    ViewBag.Genres = storeDB.Genres.OrderBy(g => g.Name).ToList();
    ViewBag.Artists = storeDB.Artists.OrderBy(a => a.Name).ToList();
```

```
    var album = storeDB.Albums.Single(a => a.AlbumId == id);

    return View(album);
}
```

At this point, our controller is passing an Album to the Edit view as the Model, and the Edit view is displaying the Album editor template. The Album editor template has been provided with all the information it needs to show dropdown for Album and Genre, since it has access to information contained in the ViewBag. We are now ready to set up the Album and Genre dropdowns in the Album editor template.

Implementing Dropdowns on the Album Editor Template

We will make use of another HTML Helper to create our dropdowns, Html.DropDownList. Let's look at the information we need to pass for the Artist dropdown:

- The name of the form field (ArtistId)
- The list of values for the dropdown, passed as a SelectList
- The Data Value field which should be posted back with the form
- The Data Text field which should be displayed in the dropdown list
- The Selected Value which is used to set the dropdown list value when the form is displayed

Here's what our call looks like:

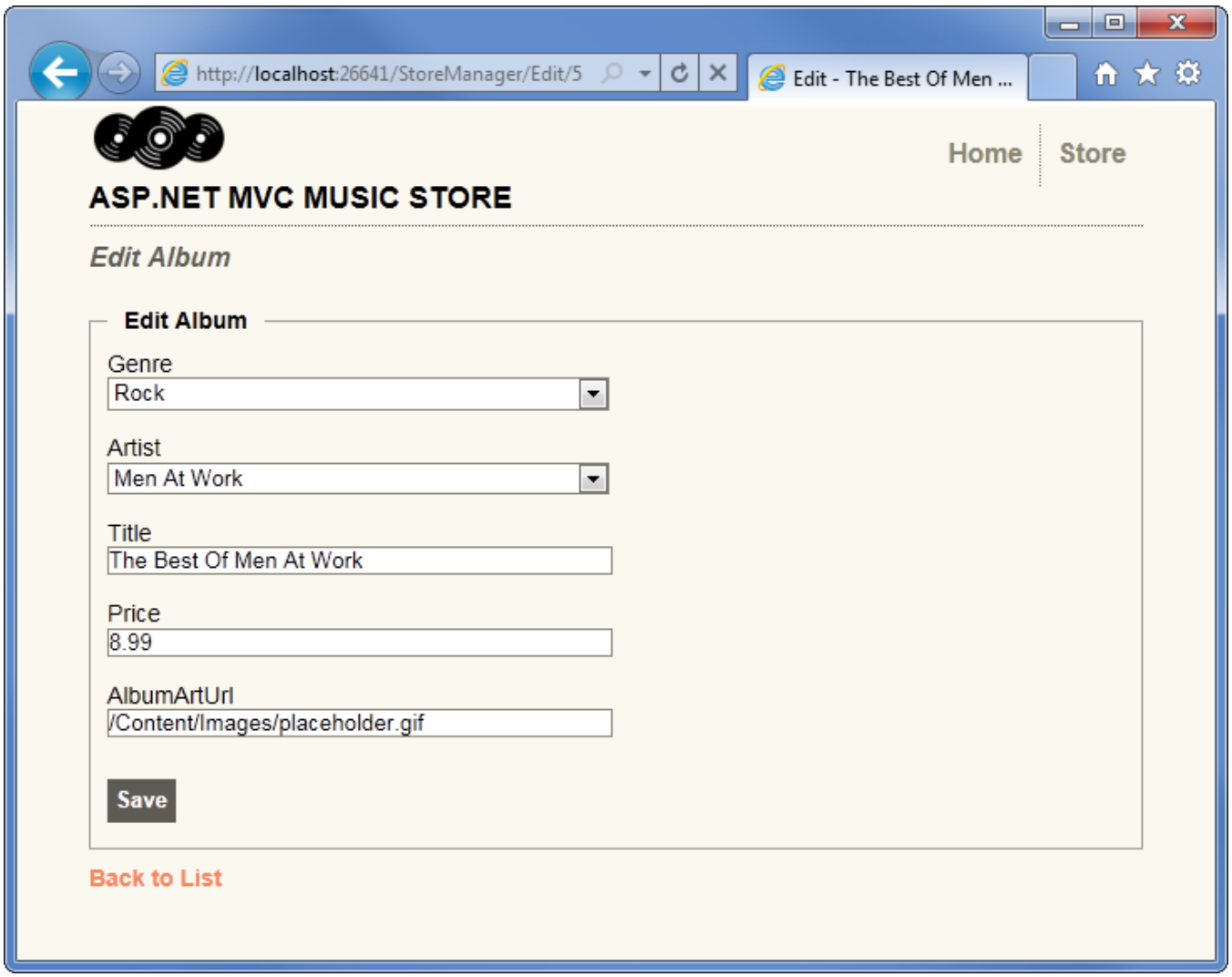
```
@Html.DropDownList("ArtistId",
    new SelectList(ViewBag.Artists as System.Collections.IEnumerable,
    "ArtistId", "Name", Model.ArtistId))
```

Our completed Album.ascx editor template code appears as follows:

```
@model MvcMusicStore.Models.Album
<p>
    @Html.LabelFor(model => model.Genre)
    @Html.DropDownList("GenreId",
        new SelectList(ViewBag.Genres as System.Collections.IEnumerable,
        "GenreId", "Name", Model.GenreId))
</p>
<p>
    @Html.LabelFor(model => model.Artist)
    @Html.DropDownList("ArtistId",
        new SelectList(ViewBag.Artists as System.Collections.IEnumerable,
        "ArtistId", "Name", Model.ArtistId))
</p>
<p>
    @Html.LabelFor(model => model.Title)
    @Html.TextBoxFor(model => model.Title)
    @Html.ValidationMessageFor(model => model.Title)
</p>
<p>
    @Html.LabelFor(model => model.Price)
    @Html.TextBoxFor(model => model.Price)
    @Html.ValidationMessageFor(model => model.Price)
</p>
```

```
<p>
    @Html.LabelFor(model => model.AlbumArtUrl)
    @Html.TextBoxFor(model => model.AlbumArtUrl)
    @Html.ValidationMessageFor(model => model.AlbumArtUrl)
</p>
```

Now when we edit an album from within our Store Manager, we see dropdowns instead of Artist and Genre ID text fields.



Implementing the HTTP-POST Edit Action Method

Our next step is to handle the scenario where the store manager presses the "Save" button and performs a HTTP-POST of the form values back to the server to update and save them. We'll implement this logic using an Edit action method which takes an ID (read from the route parameter values) and a FormCollection (read from the HTML Form). This method will be decorated with an [HttpPost] attribute to indicate that it should be used for the HTTP-POST scenarios with the /StoreManager/Edit/[id] URL.

This controller action method will perform three steps:

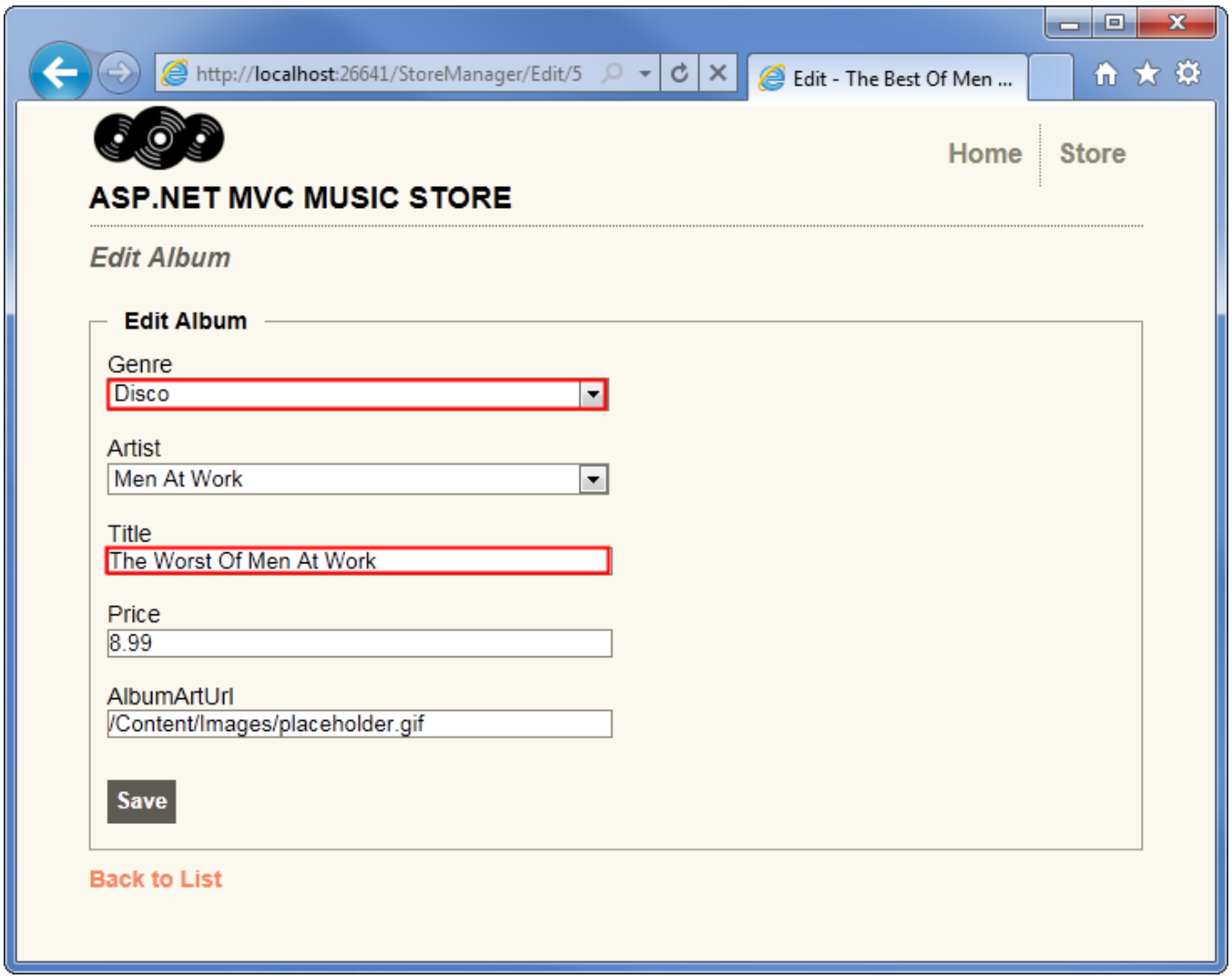
1. It will load the existing album object from the database by the ID passed in the URL
2. It will try to update the album using the values posted from the client, using the Controller's built-in UpdateModel method.
3. It will display results back to the user – either by redisplaying the form in case of an error, or by redirecting back to the list of albums in case of a successful update

Below is the code we'll use to implement the above three steps:

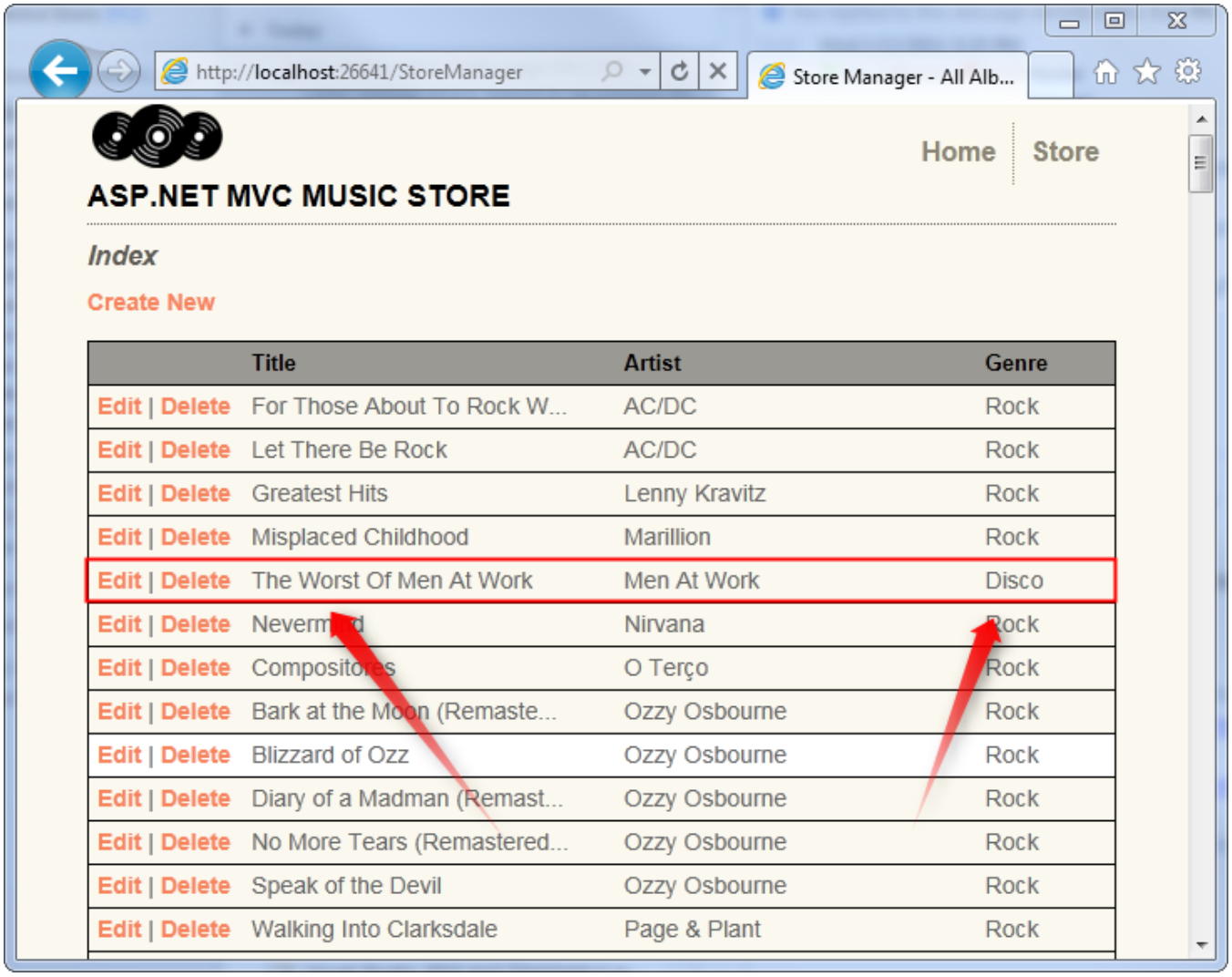
```
//
// POST: /StoreManager/Edit/5

[HttpPost]
public ActionResult Edit(int id, FormCollection collection)
{
    var album = storeDB.Albums.Find(id);

    if(TryUpdateModel(album))
    {
        storeDB.SaveChanges();
        return RedirectToAction("Index");
    }
    else
    {
        ViewBag.Genres = storeDB.Genres.OrderBy(g => g.Name).ToList();
        ViewBag.Artists = storeDB.Artists.OrderBy(a => a.Name).ToList();

        return View(album);
    }
}
```

We use the [HttpPost] attribute to indicate that this form will only respond to the HTTP-POST method, so it is only displayed when a user submits the form.

Now we are ready to Edit an Album. Run the application and browse to /Edit/5. Let's update both the Genre and Title for this album as shown.



When we click the Save button, our updates are saved and we are returned to the Store Manager Index when we can see our updated information.



Implementing the Create Action

Now that we've enabled the ability to "Edit" albums with our Store Manager Controller, let's next provide the ability to "Create" them as well.

Like we did with our Edit scenario, we'll implement the Create scenario using two separate methods within our Controller class. One action method will display an empty form when store managers first visit the /StoreManager/Create URL. A second action method will handle the scenario when the store manager presses the Save button within the form and submits the values back to the /StoreManager/Create URL as an HTTP-POST.

Implementing the HTTP-GET Create Action Method

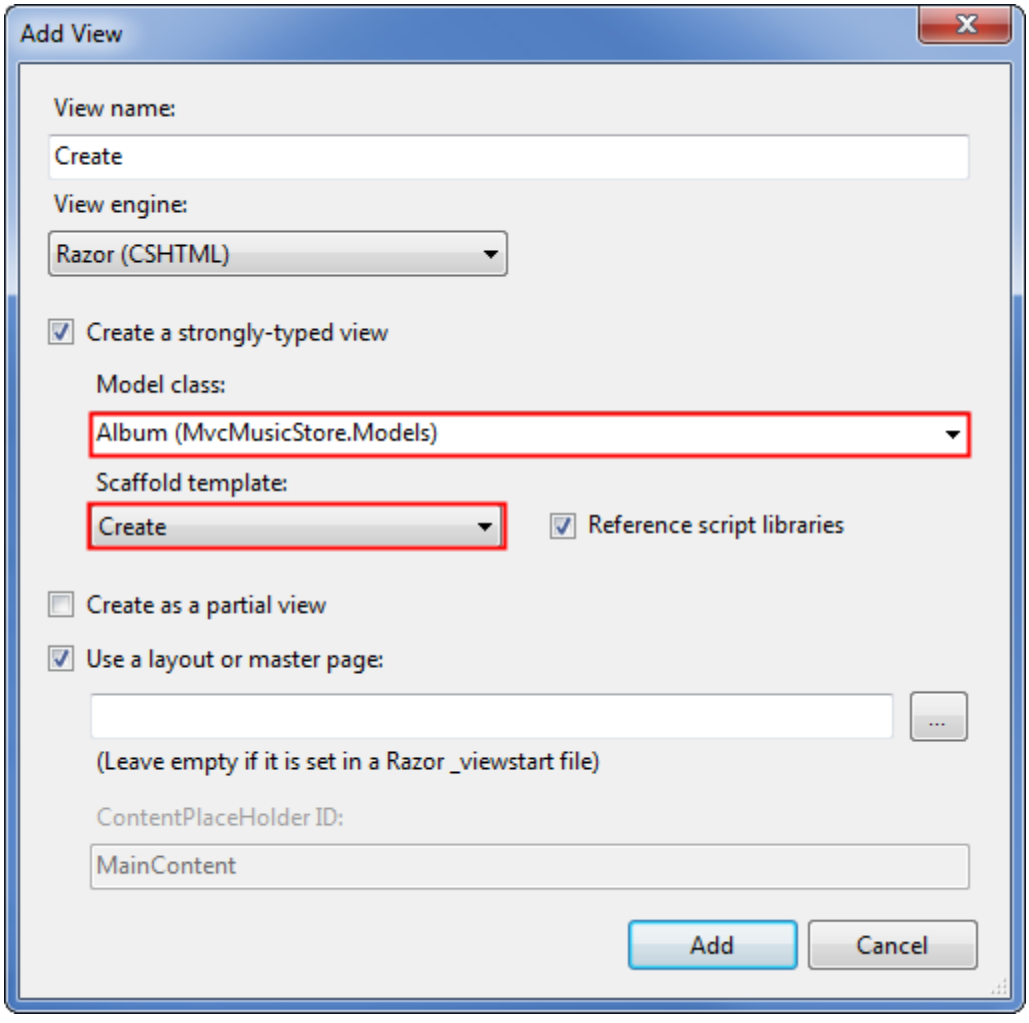
We'll begin by defining a "Create" controller action method that displays a new (empty) Album form to users using the code below:

```
//
// GET: /StoreManager/Create
```

```
public ActionResult Create()
{
    ViewBag.Genres = storeDB.Genres.OrderBy(g => g.Name).ToList();
    ViewBag.Artists = storeDB.Artists.OrderBy(a => a.Name).ToList();

    var album = new Album();

    return View(album);
}
```

Note: You have probably noticed that we are duplicating code which loads the Genres and Artists information from the Edit to the Create actions. This code could be refactored in a number of ways to remove this duplication, but in the interest of simplicity we will include this code in both methods.

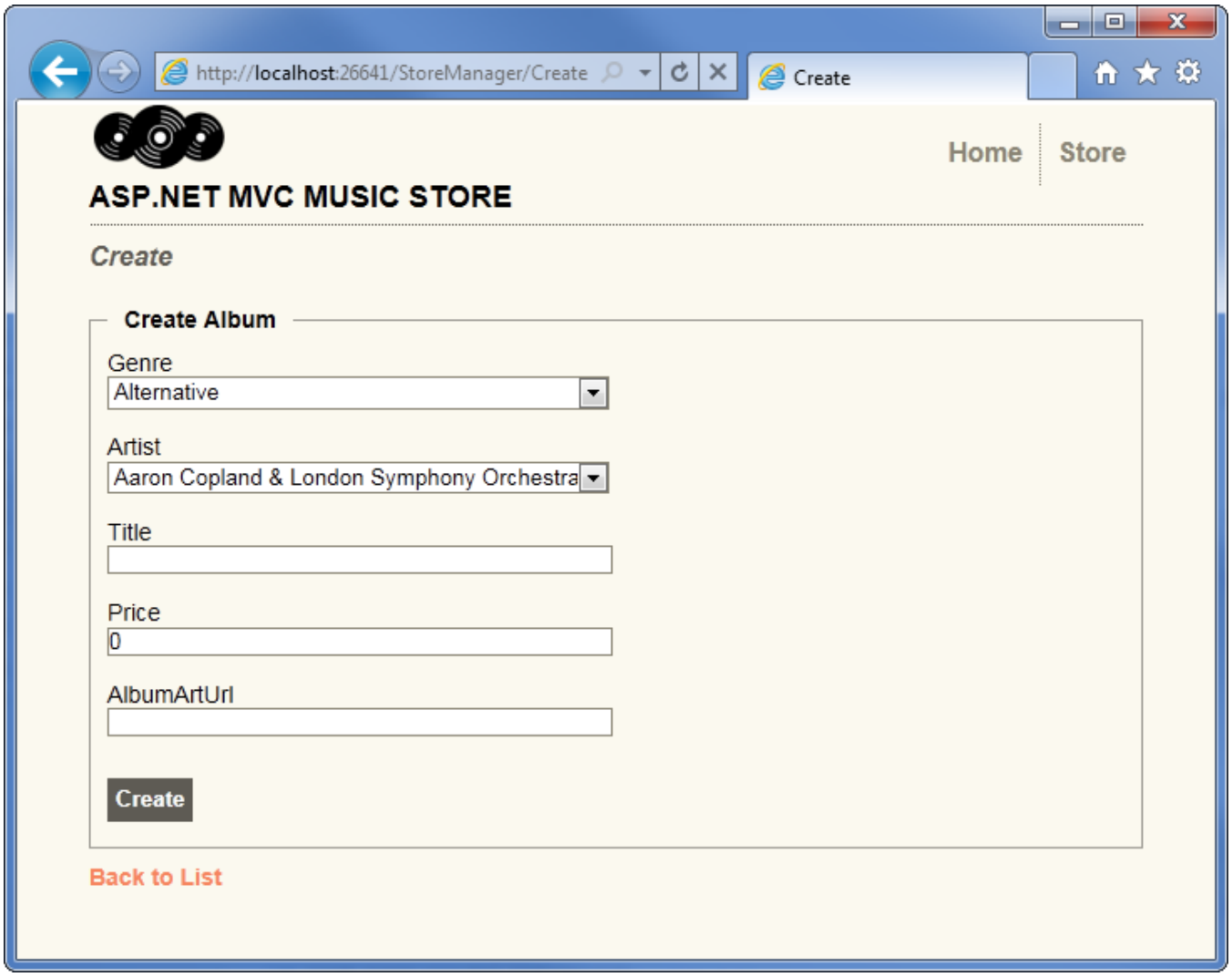
We'll then right-click within the method and use the "Add->View" menu command to create a new "Create" view that takes a StoreManagerViewModel:



We will update the generated Create.cshtml view template to invoke the Html.EditorFor() helper method, just as we did in the Edit form before. As planned, this View will be able to leverage our Album editor template.

```
@model MvcMusicStore.Models.Album

@{
    ViewBag.Title = "Create";
}

<h2>Create</h2>

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

@using (Html.BeginForm()) {
    @Html.ValidationSummary(true)
    <fieldset>
        <legend>Create Album</legend>
        @Html.EditorForModel()
        <p>
            <input type="submit" value="Create" />
        </p>
    </fieldset>
}

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

Let's then run the application again and visit the /StoreManager/Create URL. We'll now get back an empty Create form:



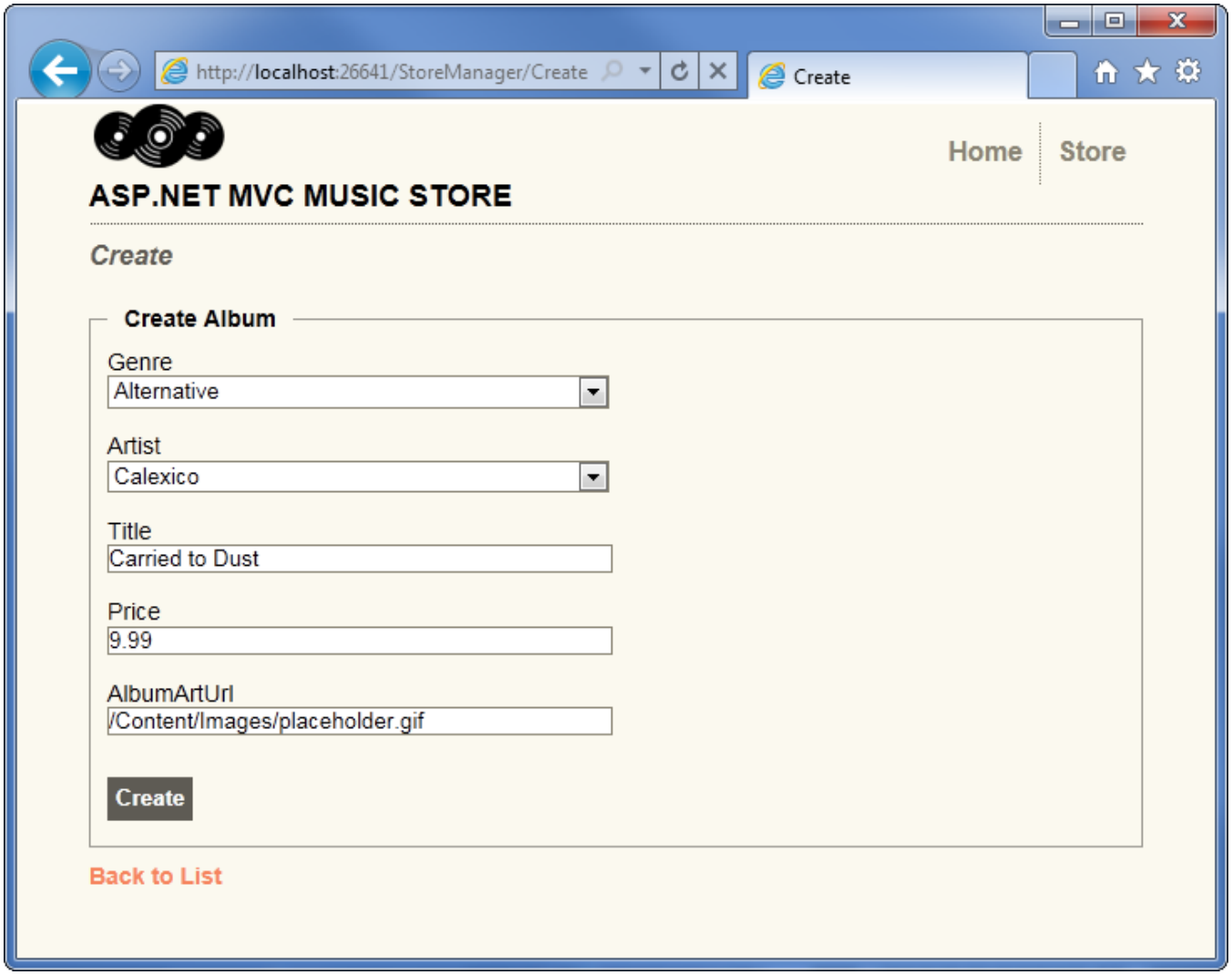
Lastly, we'll implement our HTTP-POST Create controller action method, which will be invoked when a store manager posts a new album back to the server. We'll implement it using the code below:

```
//
// POST: /StoreManager/Create

[HttpPost]
public ActionResult Create(Album album)
{
    if (ModelState.IsValid)
    {

        //Save Album
        storeDB.Albums.Add(album);
        storeDB.SaveChanges();

        return RedirectToAction("Index");

    }
```

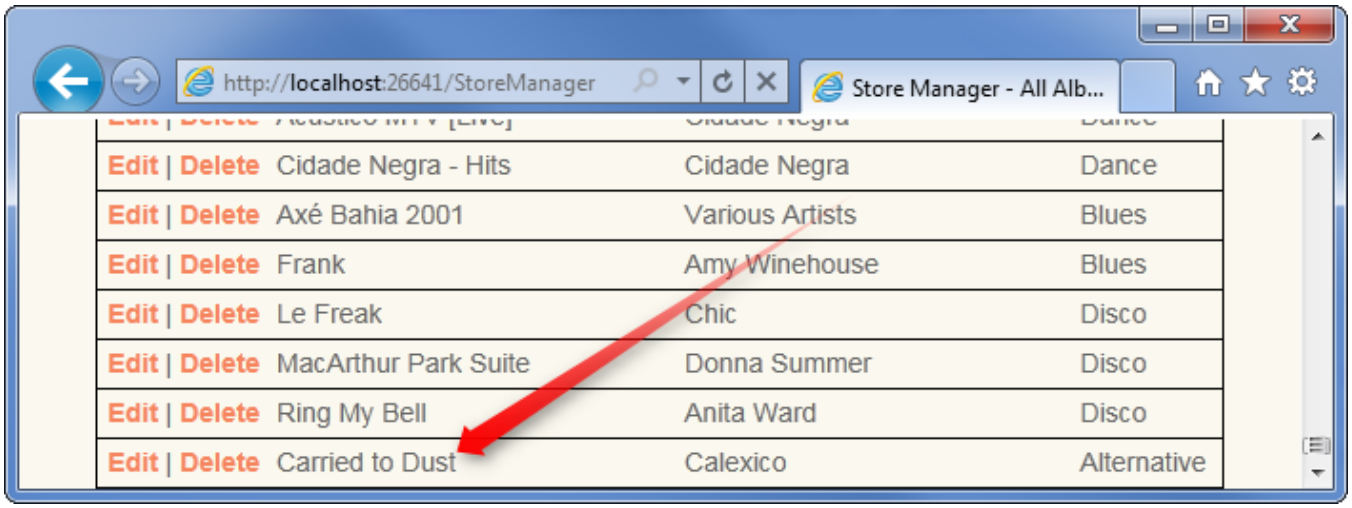
```
    // Invalid - redisplay with errors
    ViewBag.Genres = storeDB.Genres.OrderBy(g => g.Name).ToList();
    ViewBag.Artists = storeDB.Artists.OrderBy(a => a.Name).ToList();

    return View(album);
}
```

One difference from our previous Edit action method is that instead of loading an existing Album and calling UpdateModel, we are instead accepting an Album as the Action Method parameter. ASP.NET MVC will handle automatically creating this Album object for us from the posted <form> values. Our code then checks that the submitted values are valid, and if they are saves the new Album in the database.

Browsing to /StoreManager/Create, we're shown an empty edit screen where we can enter details for a new album.



Pressing the Save button adds this album to the list.



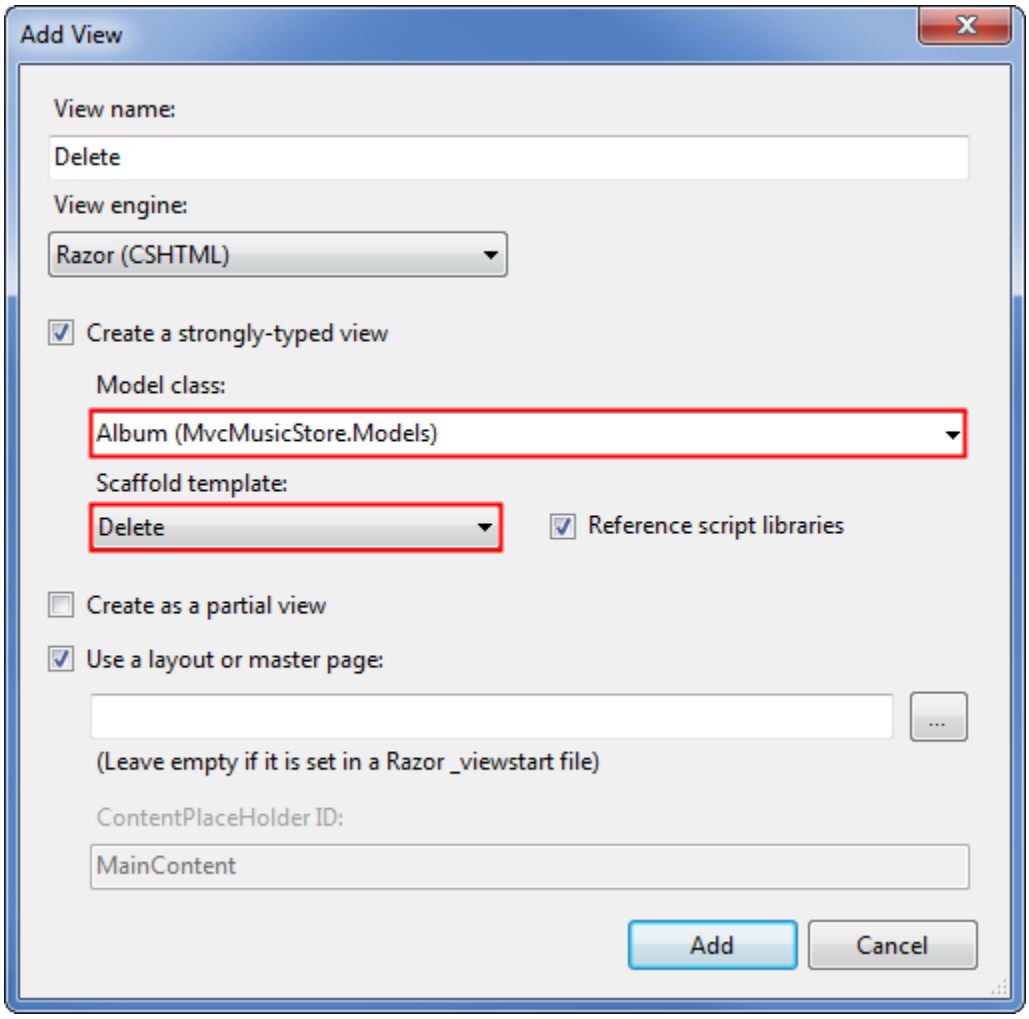
Handling Deletion

Deletion follows the same pattern as Edit and Create, using one controller action to display the confirmation form, and another controller action to handle the form submission.

The HTTP-GET Delete controller action is exactly the same as our previous Store Details controller action.

```
//
// GET: /StoreManager/Delete/5

public ActionResult Delete(int id)
{
    var album = storeDB.Albums.Find(id);

    return View(album);
}
```

We display a form that's strongly typed to an Album type, using the Delete view content template.



The Delete template shows all the fields for the model, but we can simplify that down quite a bit.

```
@model MvcMusicStore.Models.Album

@{
    ViewBag.Title = "Delete";
}

<h2>Delete Confirmation</h2>

<p>Are you sure you want to delete the album titled
   <strong>@Model.Title</strong>?
</p>

@using (Html.BeginForm()) {
    <p>
        <input type="submit" value="Delete" />
    </p>
    <p>
        @Html.ActionLink("Back to List", "Index")
    </p>
```

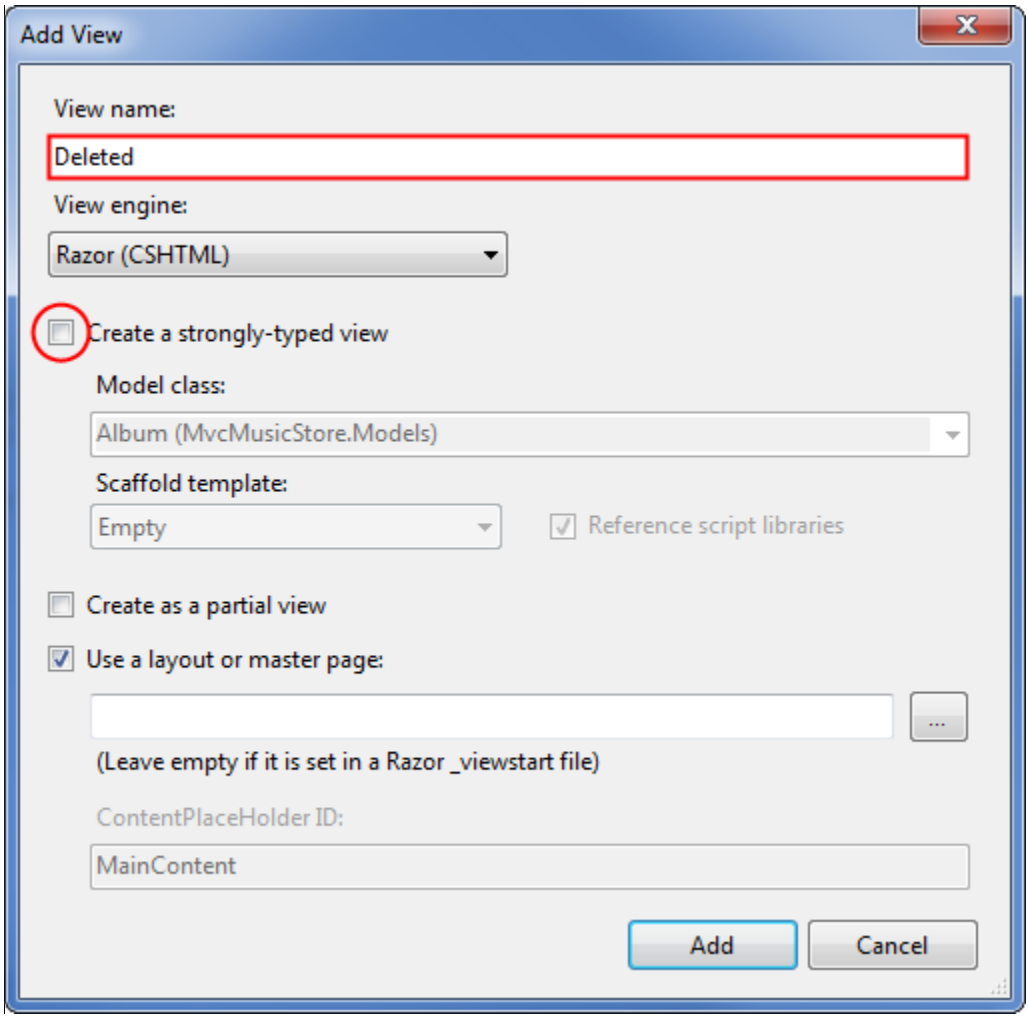
```
}
```

Now our HTTP-POST Delete Controller Action takes the following actions:

1. Loads the Album by ID
2. Deletes it the album and save changes
3. Redirect to a simple Deleted view template which indicates that the delete was successful

```
//
// POST: /StoreManager/Delete/5

[HttpPost]
public ActionResult Delete(int id, FormCollection collection)
{
    var album = storeDB.Albums.Find(id);

    storeDB.Albums.Remove(album);
    storeDB.SaveChanges();

    return View("Deleted");
}
```

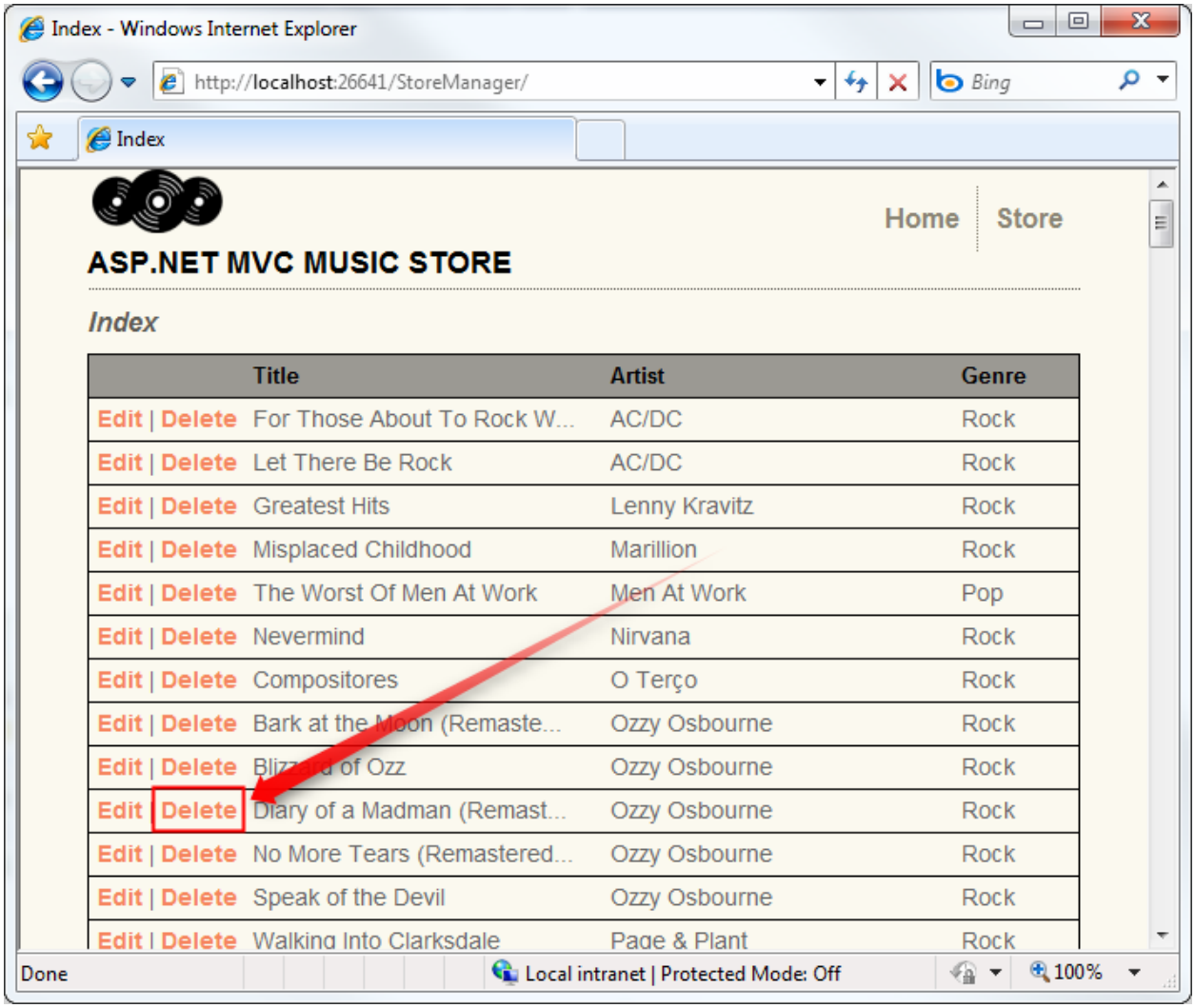
We will need to add one more View which is shown when the album is deleted. Right-click on the Delete controller action and add a view named Deleted which is not strongly typed.



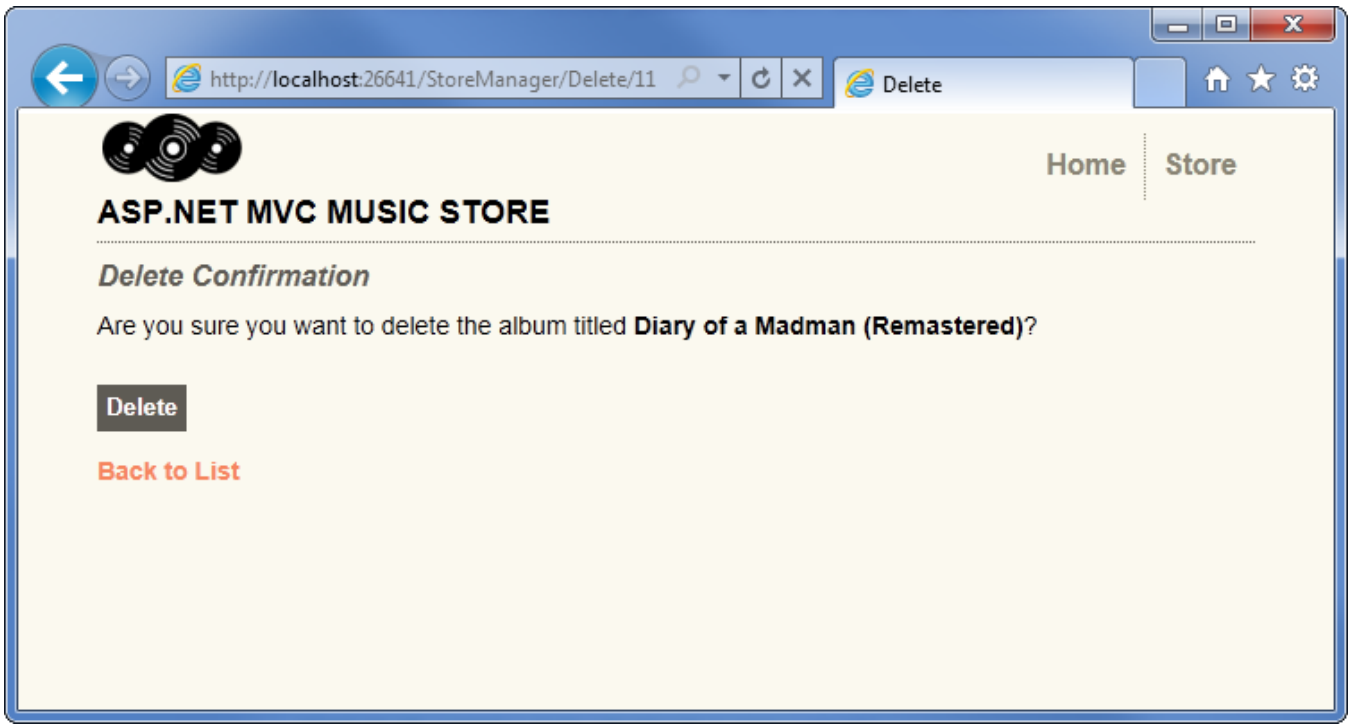
The Deleted view just shows a message that the album was deleted and shows a link back to the Store Manager Index.

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

<h2>Deleted</h2>

<p>Your album was successfully deleted.</p>

<p>
    @Html.ActionLink("Click here", "Index")
    to return to the album list.
</p>
```

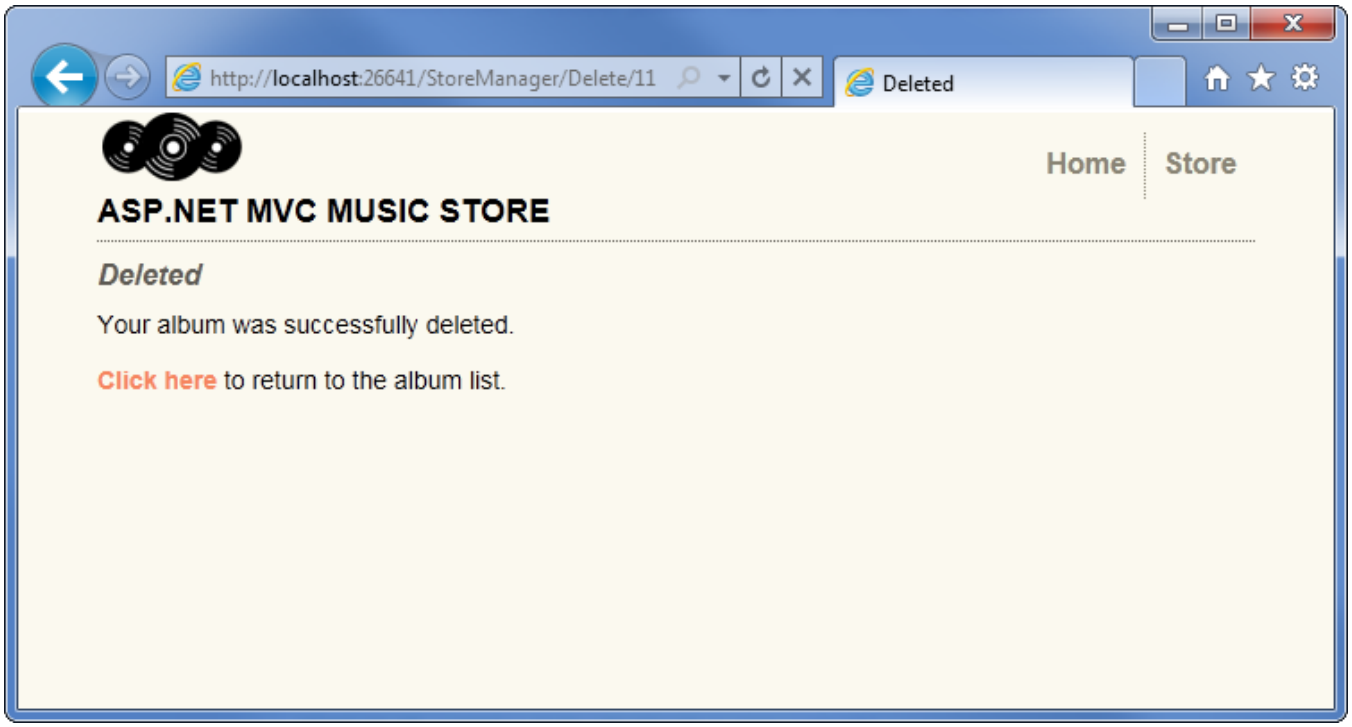
To test this, run the application and browse to /StoreManager. Select an album from the list and click the Delete link.



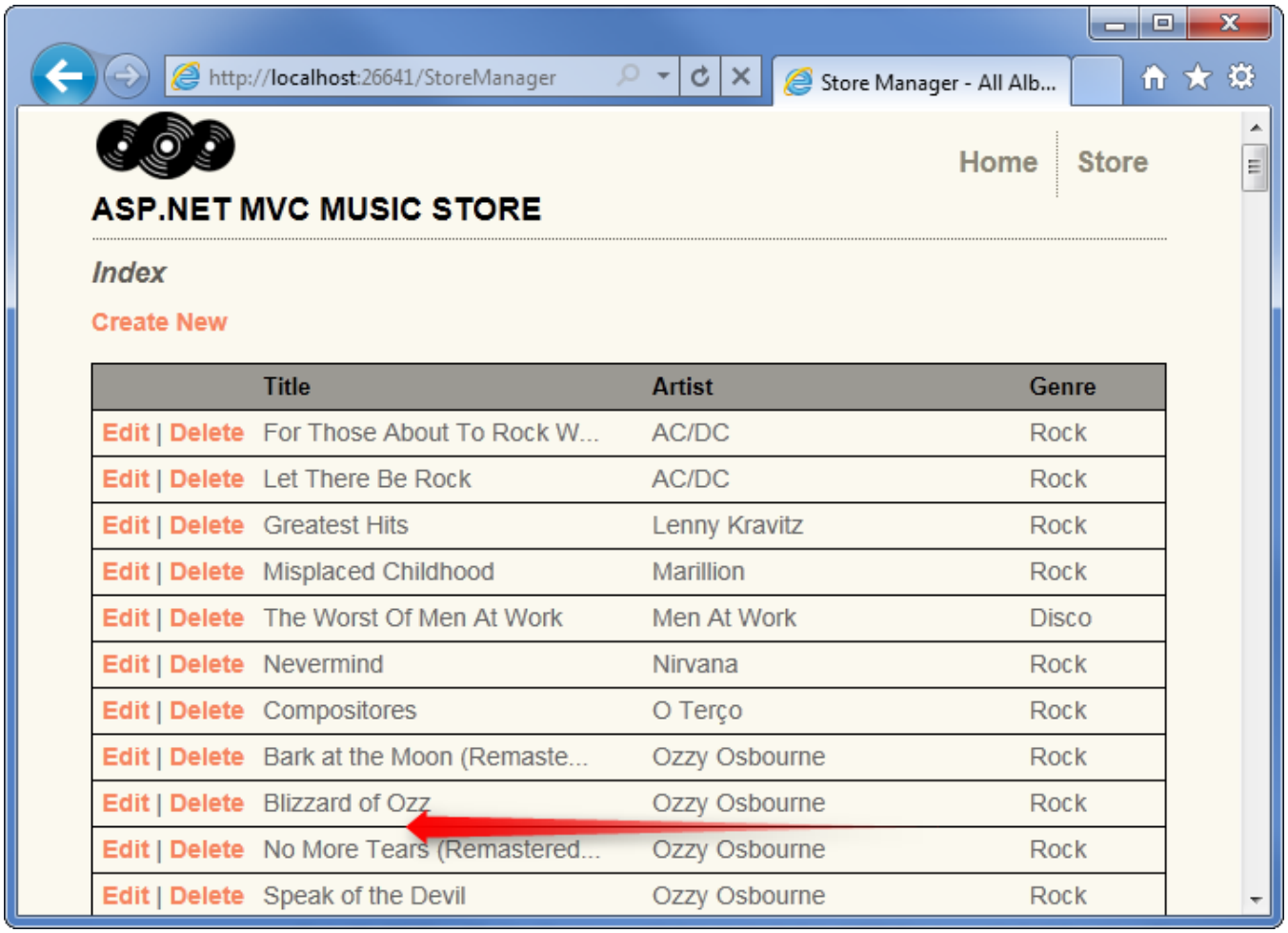
This displays our Delete confirmation screen.



Clicking the Delete button removes the album and shows the confirmation page.



Clicking on the link returns us to the Store Manager Index page, which shows that the album has been deleted.


http://localhost:26641/StoreManager
Store Manager - All Alb...
Home
Store
ASP.NET MVC MUSIC STORE
Index
Create New
Title
Artist
Genre
Edit | Delete For Those About To Rock W... AC/DC Rock
Edit | Delete Let There Be Rock AC/DC Rock
Edit | Delete Greatest Hits Lenny Kravitz Rock
Edit | Delete Misplaced Childhood Marillion Rock
Edit | Delete The Worst Of Men At Work Men At Work Disco
Edit | Delete Nevermind Nirvana Rock
Edit | Delete Compositores O Terço Rock
Edit | Delete Bark at the Moon (Remaste... Ozzy Osbourne Rock
Edit | Delete Blizzard of Ozz Ozzy Osbourne Rock
Edit | Delete No More Tears (Remastered... Ozzy Osbourne Rock
Edit | Delete Speak of the Devil Ozzy Osbourne Rock

6. Using Data Annotations for Model Validation

We have a major issue with our Create and Edit forms: they're not doing any validation. We can do things like leave required fields blank or type letters in the Price field, and the first error we'll see is from the database.

We can easily add validation to our application by adding Data Annotations to our model classes. Data Annotations allow us to describe the rules we want applied to our model properties, and ASP.NET MVC will take care of enforcing them and displaying appropriate messages to our users.

Adding Validation to our Album Forms

We'll use the following Data Annotation attributes:

- Required – Indicates that the property is a required field
- DisplayName – Defines the text we want used on form fields and validation messages
- StringLength – Defines a maximum length for a string field
- Range – Gives a maximum and minimum value for a numeric field
- Bind – Lists fields to exclude or include when binding parameter or form values to model properties
- ScaffoldColumn – Allows hiding fields from editor forms

Note: For more information on Model Validation using Data Annotation attributes, see the MSDN documentation at http://go.microsoft.com/fwlink/?LinkId=159063

Open the Album class and add the following *using* statements to the top.

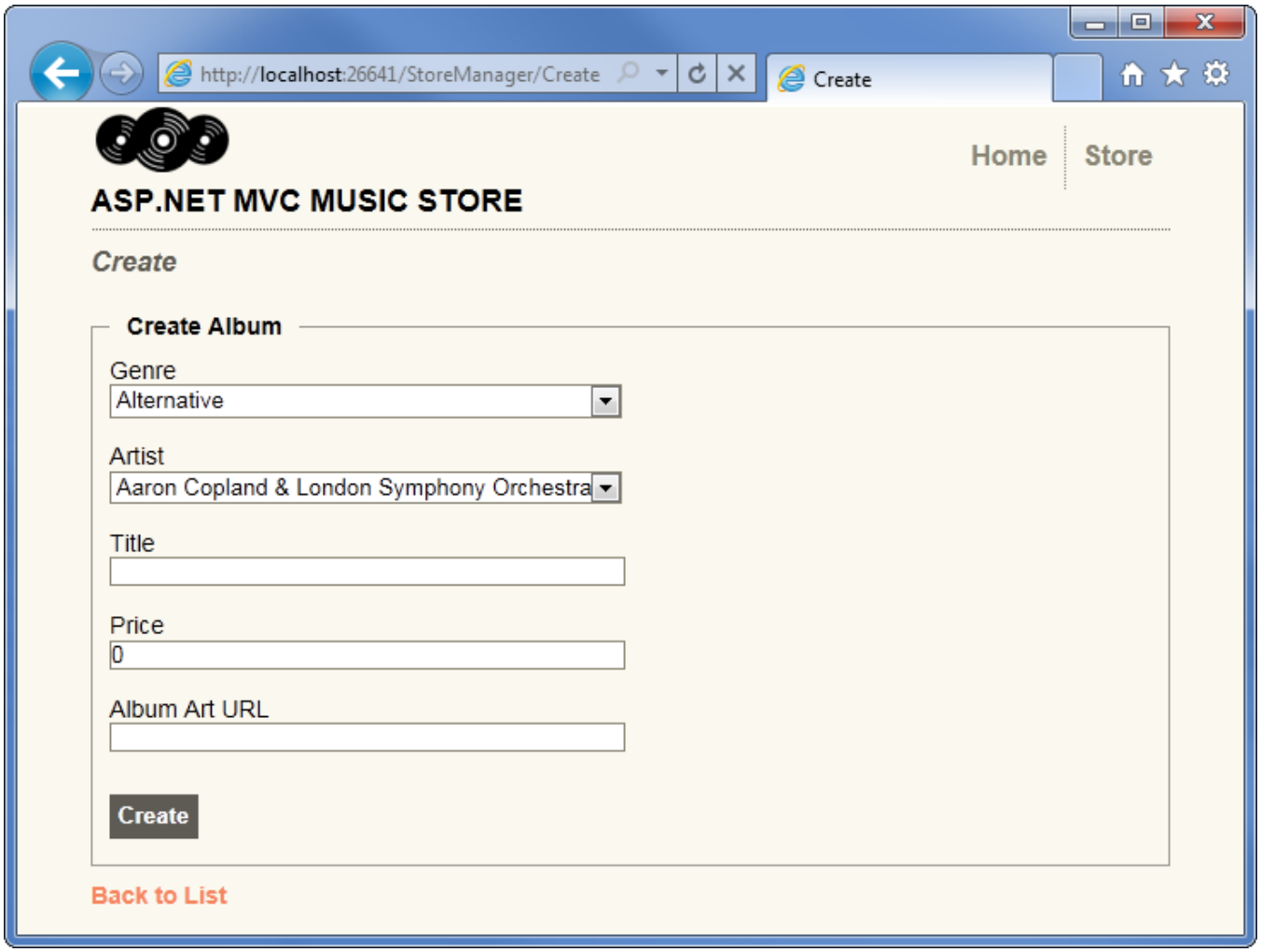
```
using System.ComponentModel;
using System.ComponentModel.DataAnnotations;
using System.Web.Mvc;
```

Next, update the properties to add display and validation attributes as shown below.

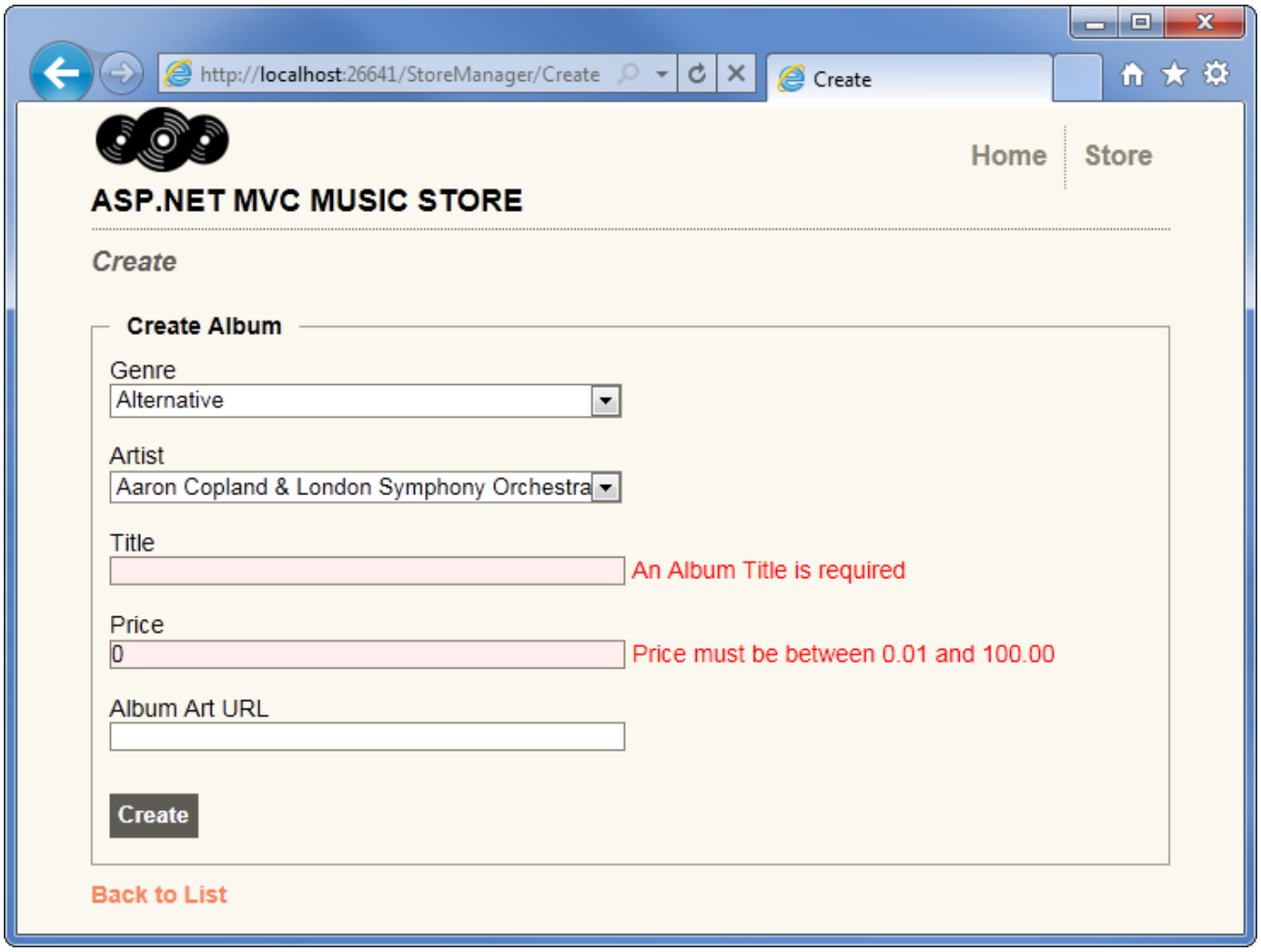
```
namespace MvcMusicStore.Models
{
    [Bind(Exclude = "AlbumId")]
    public class Album
    {
        [ScaffoldColumn(false)]
        public int      AlbumId    { get; set; }

        [DisplayName("Genre")]
        public int      GenreId    { get; set; }

        [DisplayName("Artist")]
        public int      ArtistId   { get; set; }

        [Required(ErrorMessage = "An Album Title is required")]
        [StringLength(160)]
        public string   Title      { get; set; }

        [Required(ErrorMessage = "Price is required")]
        [Range(0.01, 100.00,
            ErrorMessage = "Price must be between 0.01 and 100.00")]
```

```
        public decimal Price       { get; set; }

        [DisplayName("Album Art URL")]
        [StringLength(1024)]
        public string AlbumArtUrl { get; set; }

        public virtual Genre  Genre    { get; set; }
        public virtual Artist Artist   { get; set; }

    }
}
```

After having added this class, our Create and Edit screen immediately begin validating fields and using the Display Names we've chosen (e.g. Album Art Url instead of AlbumArtUrl). Run the application and browse to /StoreManager/Create.



When we click on the Save button, we will see the form displayed with validation error messages showing which fields did not meet the validation rules we have defined.



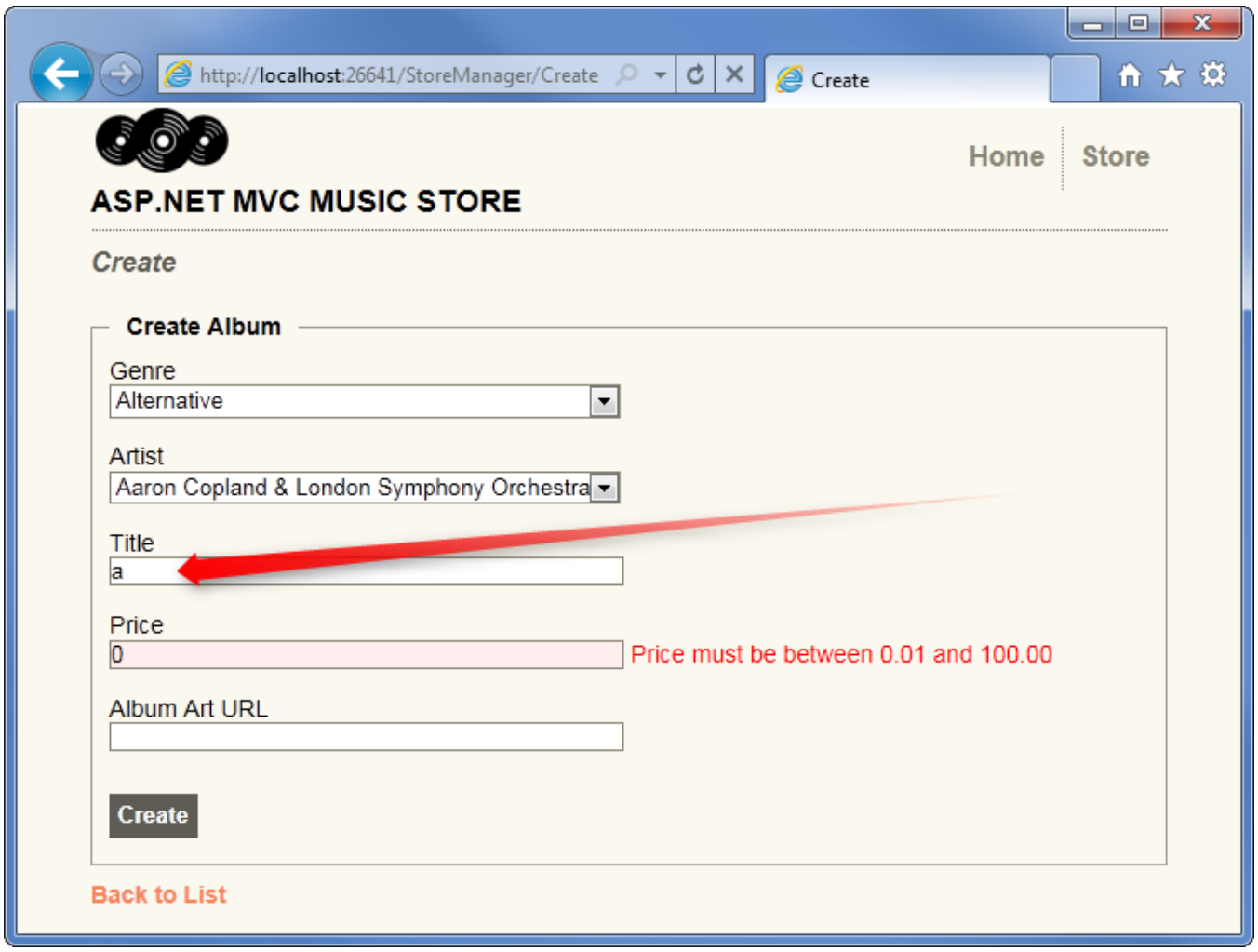
Testing the Client-Side Validation

Server-side validation is very important from an application perspective, because users can circumvent client-side validation. However, webpage forms which only implement server-side validation exhibit three significant problems.

1. The user has to wait for the form to be posted, validated on the server, and for the response to be sent to their browser.
2. The user doesn't get immediate feedback when they correct a field so that it now passes the validation rules.
3. We are wasting server resources to perform validation logic instead of leveraging the user's browser.

Fortunately, the ASP.NET MVC 3 scaffold templates have client-side validation built in, requiring no additional work whatsoever.

Typing a single letter in the Title field satisfies the validation requirements, so the validation message is immediately removed.


http://localhost:26641/StoreManager/Create
Create
Home
Store
ASP.NET MVC MUSIC STORE
Create
Create Album
Genre
Alternative
Artist
Aaron Copland & London Symphony Orchestra
Title
a
Price
0
Price must be between 0.01 and 100.00
Album Art URL
Create
Back to List

7. Membership and Authorization

Our Store Manager controller is currently accessible to anyone visiting our site. Let's change this to restrict permission to site administrators.

Adding the AccountController and Views

One difference between the full ASP.NET MVC 3 Web Application template and the ASP.NET MVC 3 Empty Web Application template is that the empty template doesn't include an Account Controller. We'll add an Account Controller by copying a few files from a new ASP.NET MVC application created from the full ASP.NET MVC 3 Web Application template.

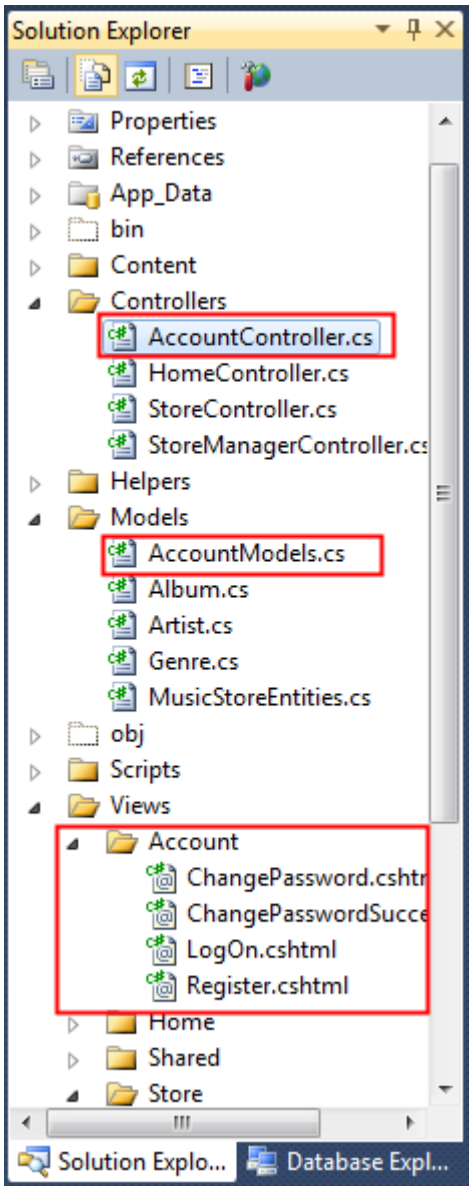
The MvcMusicStore-Assets.zip download - which included our site design files from the beginning of the tutorial - has all the AccountController files you'll need to add located in a folder named Code. Copy the following files into the same directories in our project:

1. Copy AccountController.cs in the Controllers directory
2. Copy AccountModels.cs in the Models directory
3. Create an Account directory inside the Views directory and copy all four views in

Change the namespace for the Controller and Model classes so they begin with MvcMusicStore. The AccountController class should use the MvcMusicStore.Controllers namespace, and the AccountModels class should use the MvcMusicStore.Models namespace.

Note: If you are copying these files from an empty website rather than the Assets zip, you will need to update the namespaces to match MvcMusicStore.Controllers and MvcMusicStore.Models.

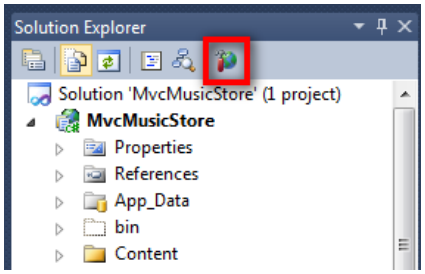
The updated solution should look like the following:



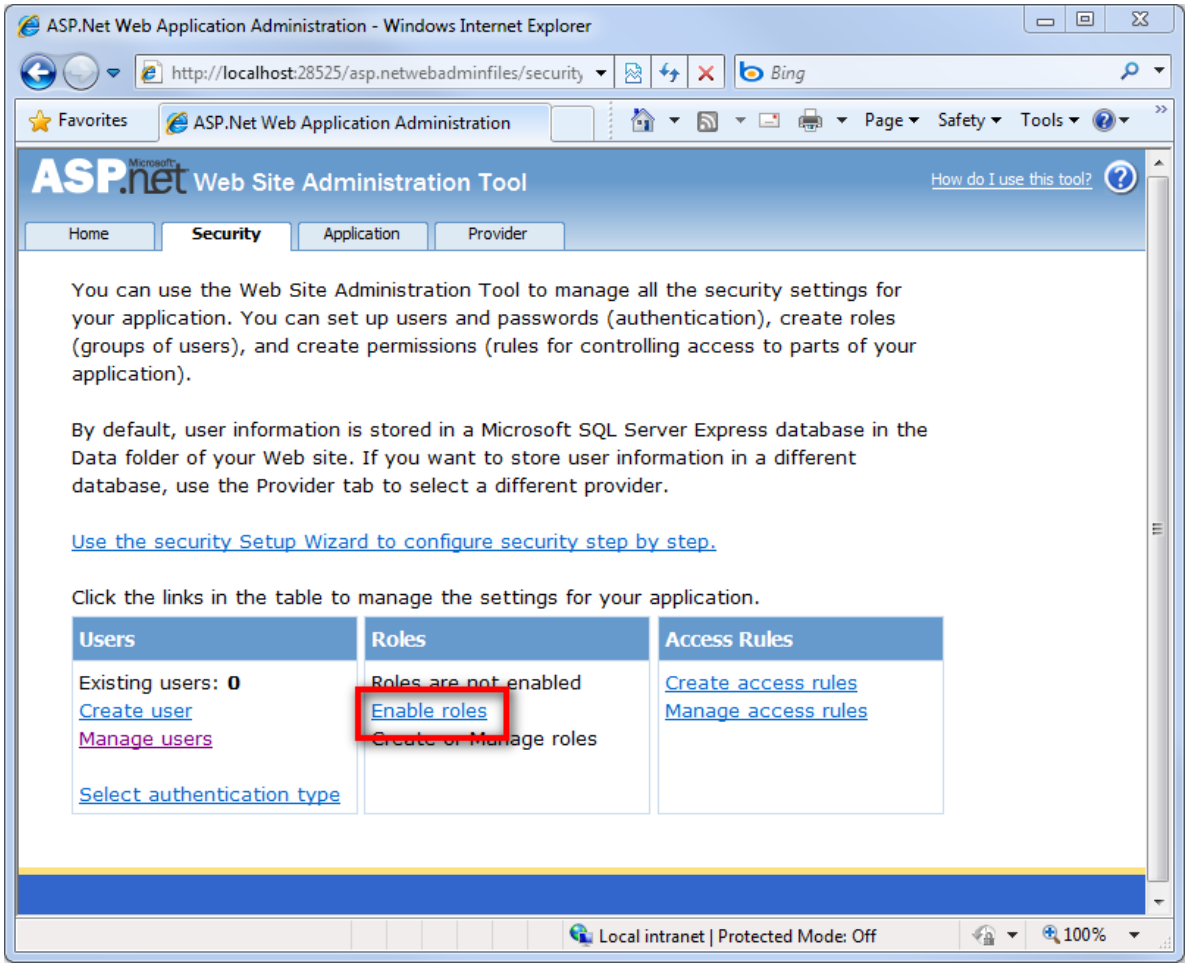
Adding an Administrative User with the ASP.NET Configuration site

Before we require Authorization in our website, we'll need to create a user with access. The easiest way to create a user is to use the built-in ASP.NET Configuration website.

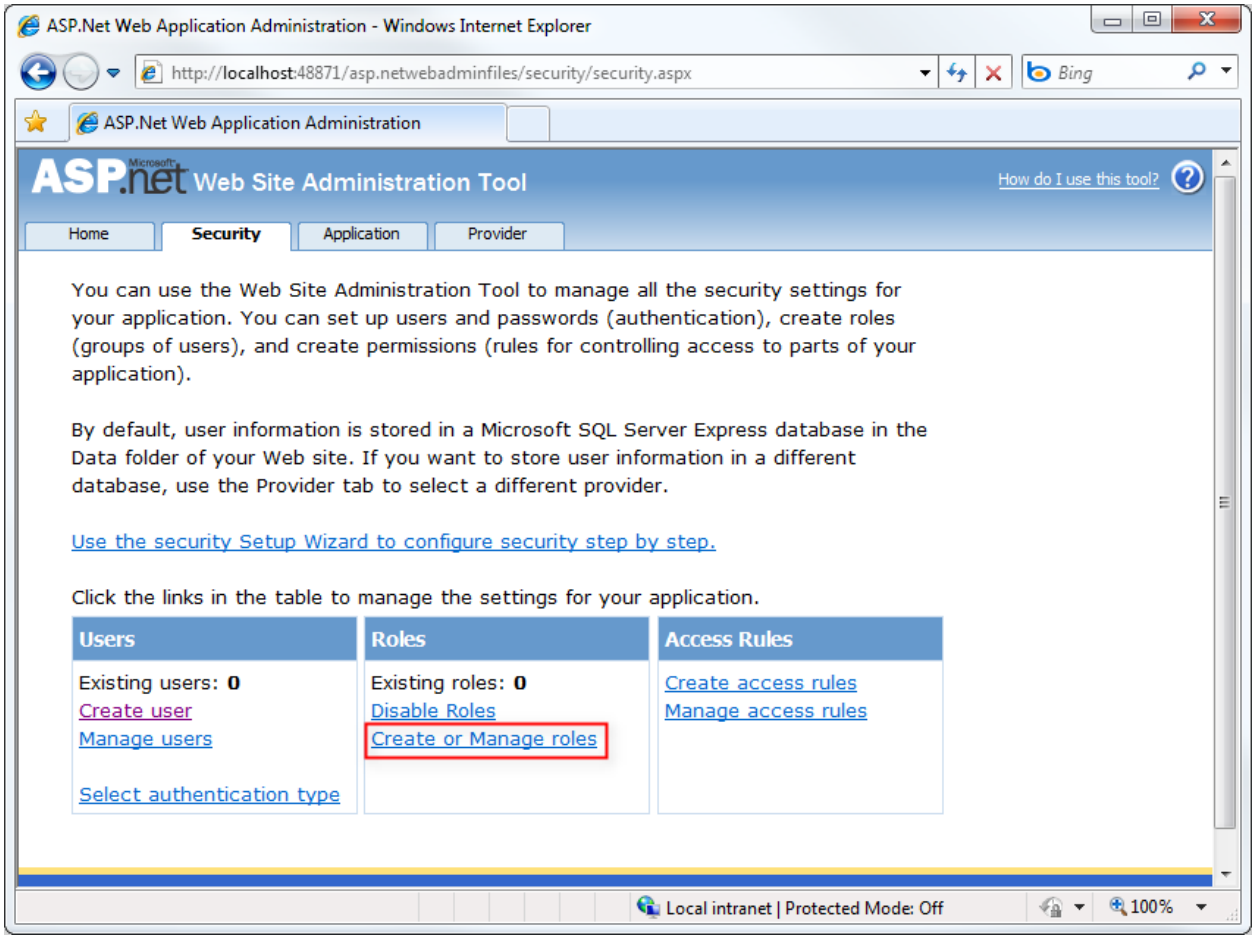
Launch the ASP.NET Configuration website by clicking following the icon in the Solution Explorer.



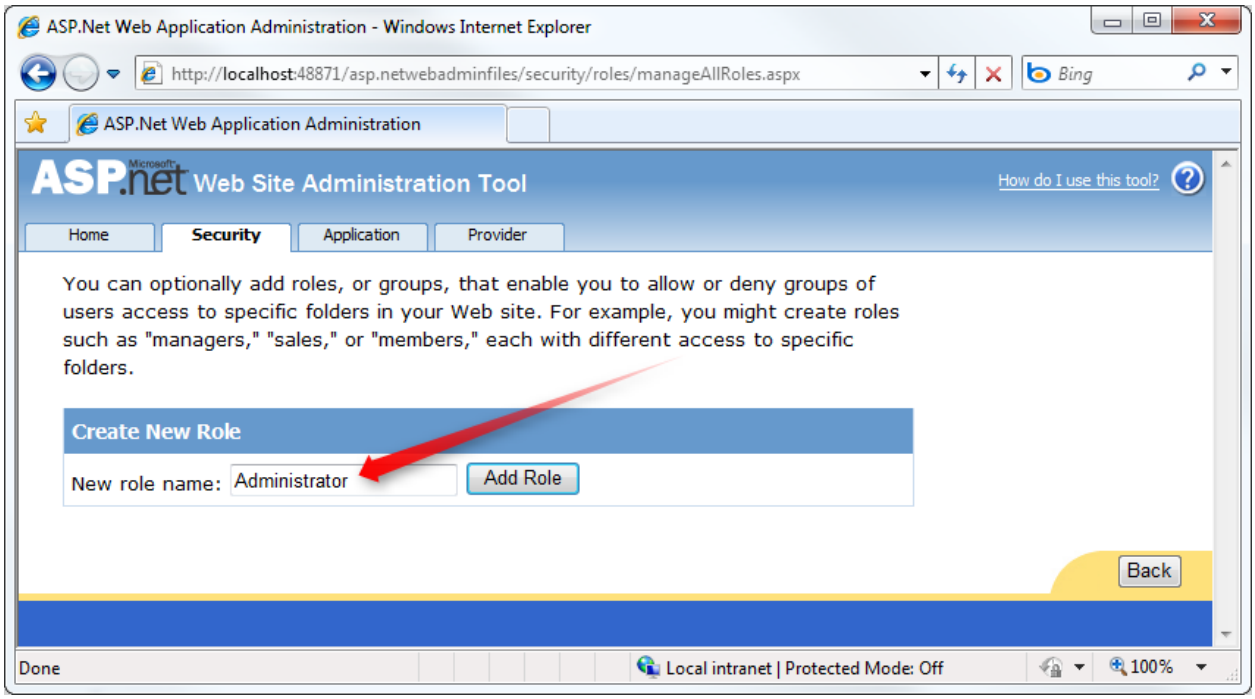
This launches a configuration website. Click on the Security tab on the home screen, then click the "Enable roles" link in the center of the screen.



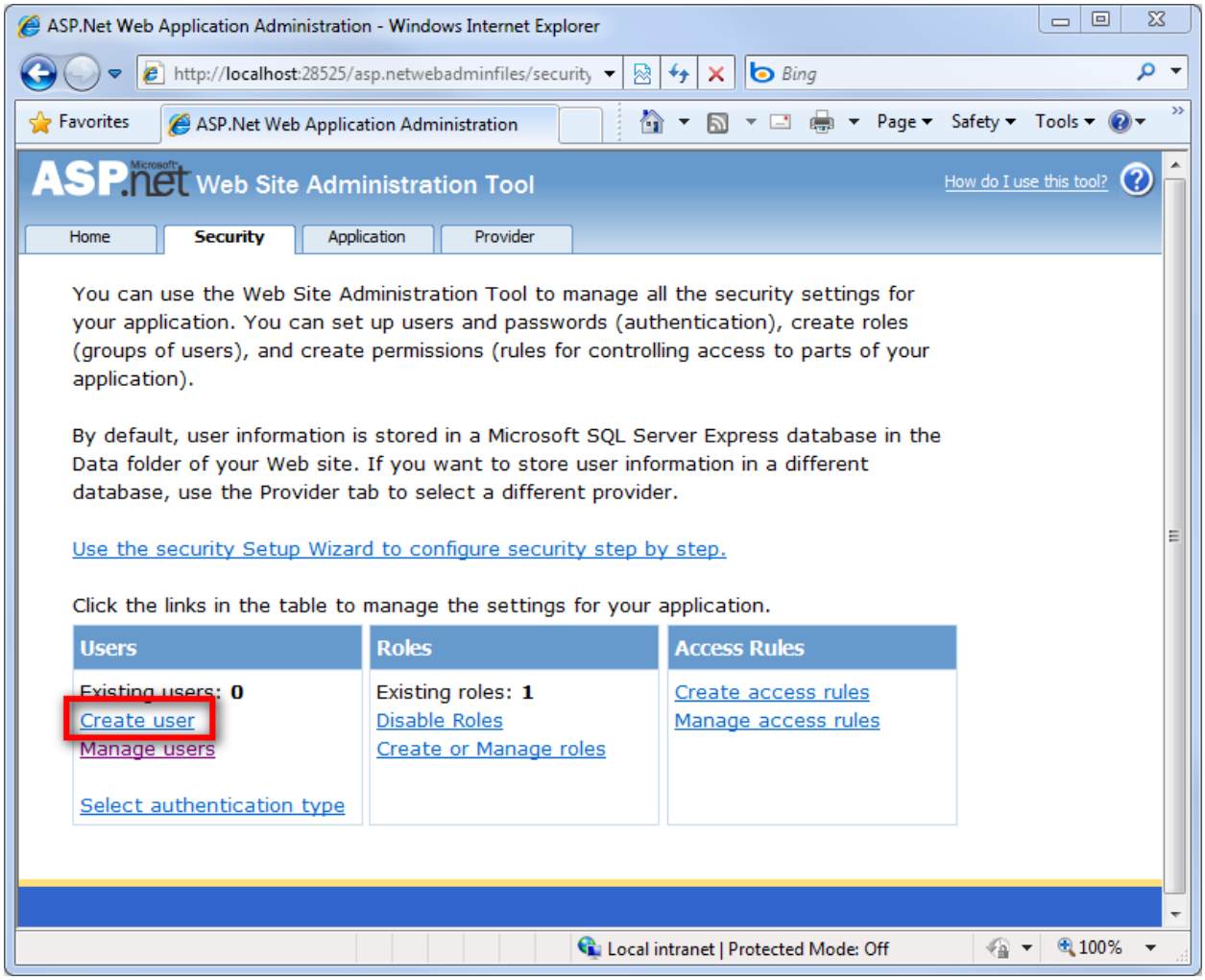
Click the "Create or Manage roles" link.



Enter “Administrator” as the role name and press the Add Role button.



Click the Back button, then click on the Create user link on the left side.

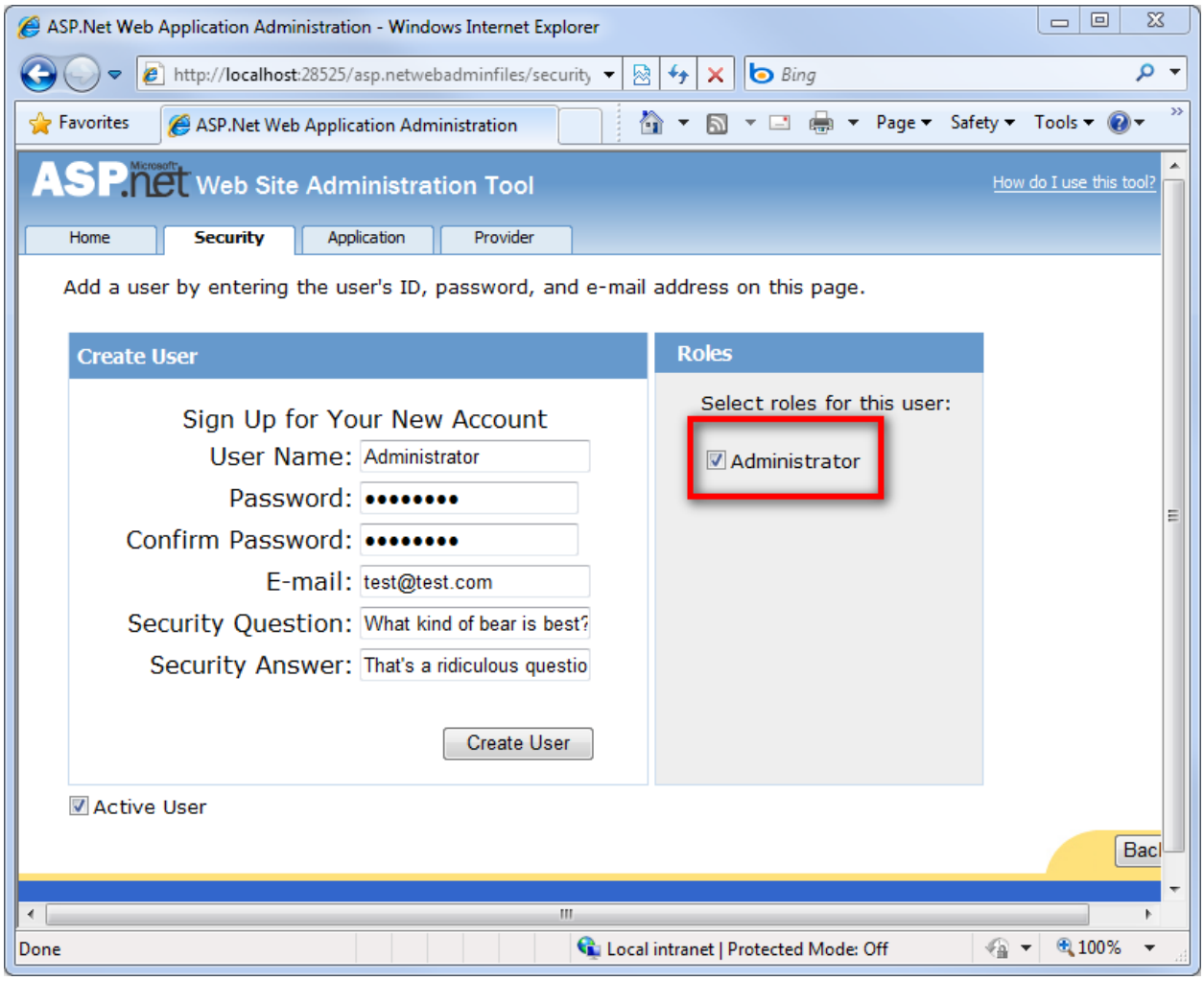


Fill in the user information fields on the left using the following information:

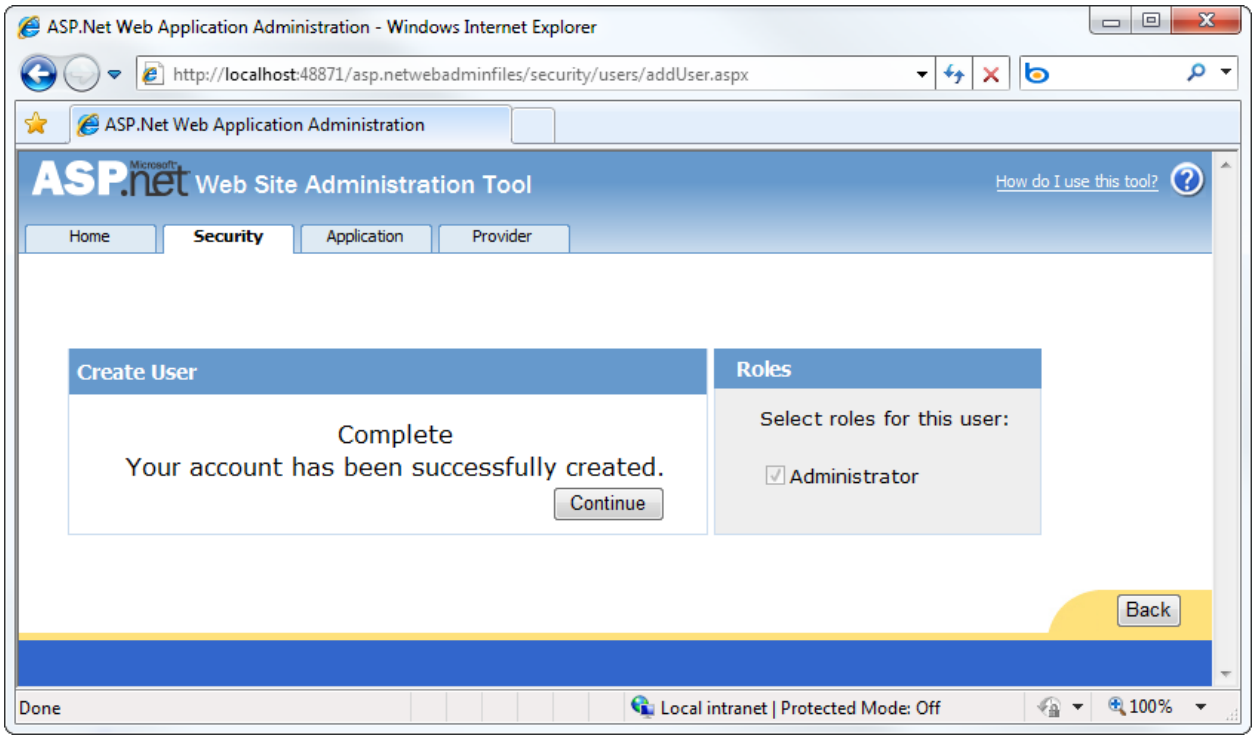
Field	Value
User Name	Administrator
Password	password123!
Confirm Password	password123!
E-mail	(any e-mail address will work)
Security Question	(whatever you like)
Security Answer	(whatever you like)

Note: You can of course use any password you'd like. The above password is shown as an example, and is assumed in the support forums on CodePlex. The default password security settings require a password that is 7 characters long and contains one non-alphanumeric character.

Select the Administrator role for this user, and click the Create User button.



At this point, you should see a message indicating that the user was created successfully.



You can now close the browser window.

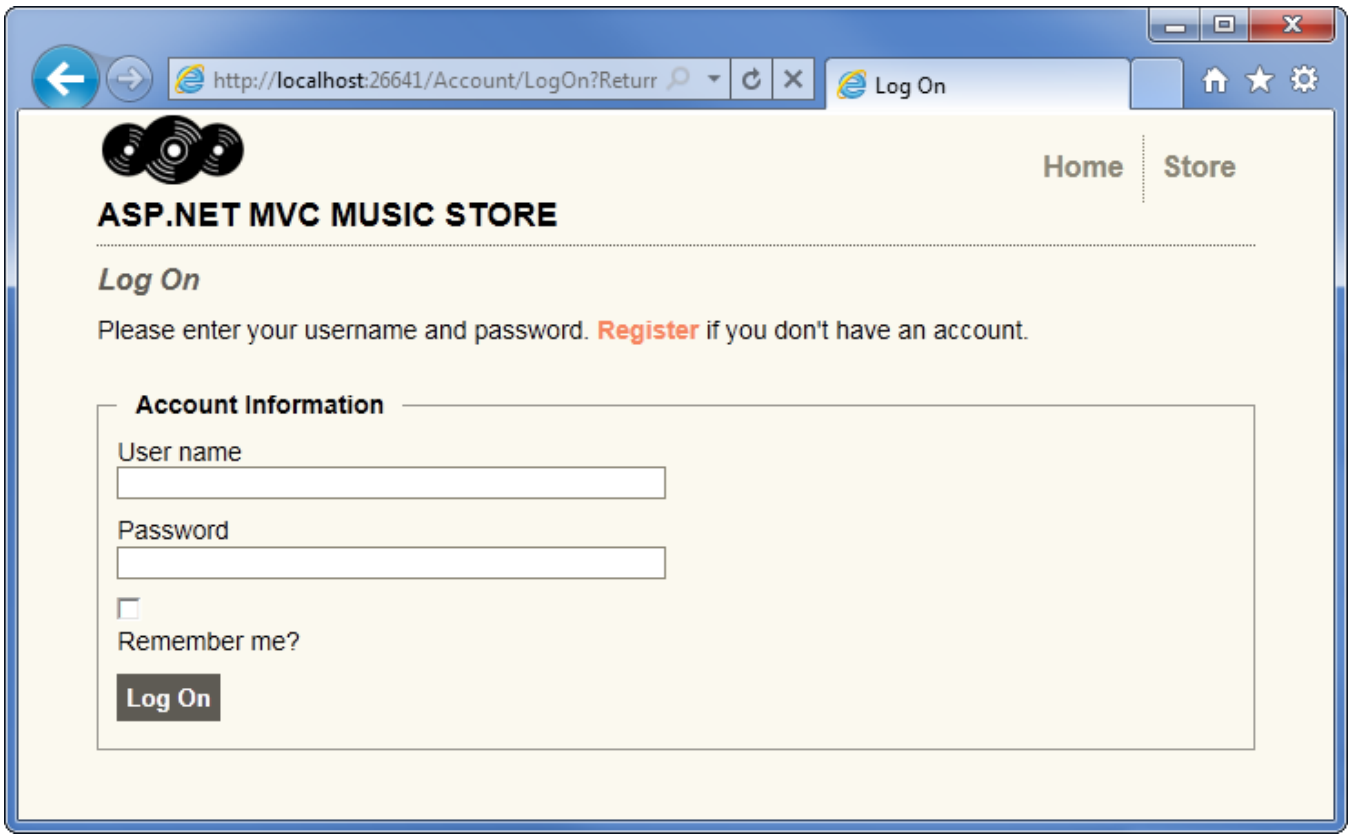
Role-based Authorization

Now we can restrict access to the StoreManagerController using the [Authorize] attribute, specifying that the user must be in the Administrator role to access any controller action in the class.

```
[Authorize(Roles = "Administrator")]
public class StoreManagerController : Controller
{
      // Controller code here
}
```

Note: The [Authorize] attribute can be placed on specific action methods as well as at the Controller class level.

Now browsing to /StoreManager brings up a Log On dialog:



After logging on with our new Administrator account, we're able to go to the Album Edit screen as before.

8. Shopping Cart with Ajax Updates

We'll allow users to place albums in their cart without registering, but they'll need to register as guests to complete checkout. The shopping and checkout process will be separated into two controllers: a ShoppingCart Controller which allows anonymously adding items to a cart, and a Checkout Controller which handles the checkout process. We'll start with the Shopping Cart in this section, then build the Checkout process in the following section.

Adding the Cart, Order, and OrderDetails model classes

Our Shopping Cart and Checkout processes will make use of some new classes. Right-click the Models folder and add a Cart class (Cart.cs) with the following code.

```
using System.ComponentModel.DataAnnotations;

namespace MvcMusicStore.Models
{
    public class Cart
    {
        [Key]
        public int      RecordId    { get; set; }
        public string   CartId      { get; set; }
        public int      AlbumId     { get; set; }
        public int      Count       { get; set; }
        public System.DateTime DateCreated { get; set; }

        public virtual Album Album  { get; set; }
    }
}
```

This class is pretty similar to others we've used so far, with the exception of the [Key] attribute for the RecordId property. Our Cart items will have a string identifier named CartID to allow anonymous shopping, but the table includes an integer primary key named RecordId. By convention, Entity Framework Code-First expects that the primary key for a table named Cart will be either CartId or ID, but we can easily override that via annotations or code if we want. This is an example of how we can use the simple conventions in Entity Framework Code-First when they suit us, but we're not constrained by them when they don't.

Next, add an Order class (Order.cs) with the following code.

```
using System.Collections.Generic;

namespace MvcMusicStore.Models
{
    public partial class Order
    {
        public int    OrderId    { get; set; }
        public string Username   { get; set; }
        public string FirstName  { get; set; }
        public string LastName   { get; set; }
        public string Address    { get; set; }
```

```
        public string City       { get; set; }
        public string State      { get; set; }
        public string PostalCode { get; set; }
        public string Country    { get; set; }
        public string Phone      { get; set; }
        public string Email      { get; set; }
        public decimal Total     { get; set; }
        public System.DateTime OrderDate      { get; set; }

        public List<OrderDetail> OrderDetails { get; set; }
    }
}
```

This class tracks summary and delivery information for an order. It won't compile yet, because it has an OrderDetails navigation property which depends on a class we haven't created yet. Let's fix that now by adding a class named OrderDetails.cs, adding the following code.

```
namespace MvcMusicStore.Models
{
    public class OrderDetail
    {
        public int OrderDetailId { get; set; }
        public int OrderId { get; set; }
        public int AlbumId { get; set; }
        public int Quantity { get; set; }
        public decimal UnitPrice { get; set; }

        public virtual Album Album { get; set; }
        public virtual Order Order { get; set; }
    }
}
```

We'll make one last update to our MusicStoreEntities class to include DbSet which expose those new Model classes. The updated MusicStoreEntities class appears as below.

```
using System.Data.Entity;

namespace MvcMusicStore.Models
{
    public class MusicStoreEntities : DbContext
    {
        public DbSet<Album>     Albums  { get; set; }
        public DbSet<Genre>     Genres  { get; set; }
        public DbSet<Artist>    Artists { get; set; }
        public DbSet<Cart>      Carts { get; set; }
        public DbSet<Order>     Orders { get; set; }
        public DbSet<OrderDetail> OrderDetails { get; set; }
    }
}
```

Managing the Shopping Cart business logic

Next, we'll create the ShoppingCart class in the Models folder. The ShoppingCart model handles data access to the Cart table. Additionally, it will handle the business logic to for adding and removing items from the shopping cart.

Since we don't want to require users to sign up for an account just to add items to their shopping cart, we will assign users a temporary unique identifier (using a GUID, or globally unique identifier) when they access the shopping cart. We'll store this ID using the ASP.NET Session class.

Note: The ASP.NET Session is a convenient place to store user-specific information which will expire after they leave the site. While misuse of session state can have performance implications on larger sites, our light use will work well for demonstration purposes.

The ShoppingCart class exposes the following methods:

AddToCart takes an Album as a parameter and adds it to the user's cart. Since the Cart table tracks quantity for each album, it includes logic to create a new row if needed or just increment the quantity if the user has already ordered one copy of the album.

RemoveFromCart takes an Album ID and removes it from the user's cart. If the user only had one copy of the album in their cart, the row is removed.

EmptyCart removes all items from a user's shopping cart.

GetCartItems retrieves a list of CartItems for display or processing.

GetCount retrieves a the total number of albums a user has in their shopping cart.

GetTotal calculates the total cost of all items in the cart.

CreateOrder converts the shopping cart to an order during the checkout phase.

GetCart is a static method which allows our controllers to obtain a cart object. It uses the **GetCartId** method to handle reading the CartId from the user's session. The GetCartId method requires the HttpContextBase so that it can read the user's CartId from user's session.

Here's the complete ShoppingCart class:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace MvcMusicStore.Models
{
    public partial class ShoppingCart
    {
        MusicStoreEntities storeDB = new MusicStoreEntities();
```

```
string ShoppingCartId { get; set; }

public const string CartSessionKey = "CartId";

public static ShoppingCart GetCart(HttpContextBase context)
{
    var cart = new ShoppingCart();
    cart.ShoppingCartId = cart.GetCartId(context);
    return cart;
}

// Helper method to simplify shopping cart calls
public static ShoppingCart GetCart(Controller controller)
{
    return GetCart(controller.HttpContext);
}

public void AddToCart(Album album)
{
    // Get the matching cart and album instances
    var cartItem = storeDB.Carts.SingleOrDefault(
            c => c.CartId == ShoppingCartId
            && c.AlbumId == album.AlbumId);

    if (cartItem == null)
    {
        // Create a new cart item if no cart item exists
        cartItem = new Cart
        {
            AlbumId = album.AlbumId,
            CartId = ShoppingCartId,
            Count = 1,
            DateCreated = DateTime.Now
        };

        storeDB.Carts.Add(cartItem);
    }
    else
    {
        // If the item does exist in the cart, then add one to the quantity
        cartItem.Count++;
    }

    // Save changes
    storeDB.SaveChanges();
}

public int RemoveFromCart(int id)
{
    // Get the cart
    var cartItem = storeDB.Carts.Single(
            cart => cart.CartId == ShoppingCartId
            && cart.RecordId == id);

    int itemCount = 0;
```

```
        if (cartItem != null)
        {
            if (cartItem.Count > 1)
            {
                cartItem.Count--;
                itemCount = cartItem.Count;
            }
            else
            {
                storeDB.Carts.Remove(cartItem);
            }

            // Save changes
            storeDB.SaveChanges();
        }

        return itemCount;
    }

    public void EmptyCart()
    {
        var cartItems = storeDB.Carts.Where(cart => cart.CartId == ShoppingCartId);

        foreach (var cartItem in cartItems)
        {
            storeDB.Carts.Remove(cartItem);
        }

        // Save changes
        storeDB.SaveChanges();
    }

    public List<Cart> GetCartItems()
    {
        return storeDB.Carts.Where(cart => cart.CartId == ShoppingCartId).ToList();
    }

    public int GetCount()
    {
        // Get the count of each item in the cart and sum them up
        int? count = (from cartItems in storeDB.Carts
                      where cartItems.CartId == ShoppingCartId
                      select (int?)cartItems.Count).Sum();

        // Return 0 if all entries are null
        return count ?? 0;
    }

    public decimal GetTotal()
    {
        // Multiply album price by count of that album to get
        // the current price for each of those albums in the cart
        // sum all album price totals to get the cart total
        decimal? total = (from cartItems in storeDB.Carts
```

```
                      where cartItems.CartId == ShoppingCartId
                      select (int?)cartItems.Count * cartItems.Album.Price).Sum();
    return total ?? decimal.Zero;
}

public int CreateOrder(Order order)
{
    decimal orderTotal = 0;

    var cartItems = GetCartItems();

    // Iterate over the items in the cart, adding the order details for each
    foreach (var item in cartItems)
    {
        var orderDetails = new OrderDetail
        {
            AlbumId = item.AlbumId,
            OrderId = order.OrderId,
            UnitPrice = item.Album.Price,
            Quantity = item.Count
        };

        // Set the order total of the shopping cart
        orderTotal += (item.Count * item.Album.Price);
    }

    // Set the order's total to the orderTotal count
    order.Total = orderTotal;

    // Save the order
    storeDB.SaveChanges();

    // Empty the shopping cart
    EmptyCart();

    // Return the OrderId as the confirmation number
    return order.OrderId;
}

// We're using HttpContextBase to allow access to cookies.
public string GetCartId(HttpContextBase context)
{
    if (context.Session[CartSessionKey] == null)
    {
        if (!string.IsNullOrWhiteSpace(context.User.Identity.Name))
        {
            context.Session[CartSessionKey] = context.User.Identity.Name;
        }
        else
        {
            // Generate a new random GUID using System.Guid class
            Guid tempCartId = Guid.NewGuid();

            // Send tempCartId back to client as a cookie
            context.Session[CartSessionKey] = tempCartId.ToString();
```

```
                }
            }

            return context.Session[CartSessionKey].ToString();
        }

        // When a user has logged in, migrate their shopping cart to
        // be associated with their username
        public void MigrateCart(string userName)
        {
            var shoppingCart = storeDB.Carts.Where(c => c.CartId == ShoppingCartId);

            foreach (Cart item in shoppingCart)
            {
                item.CartId = userName;
            }
            storeDB.SaveChanges();
        }
    }
}
```

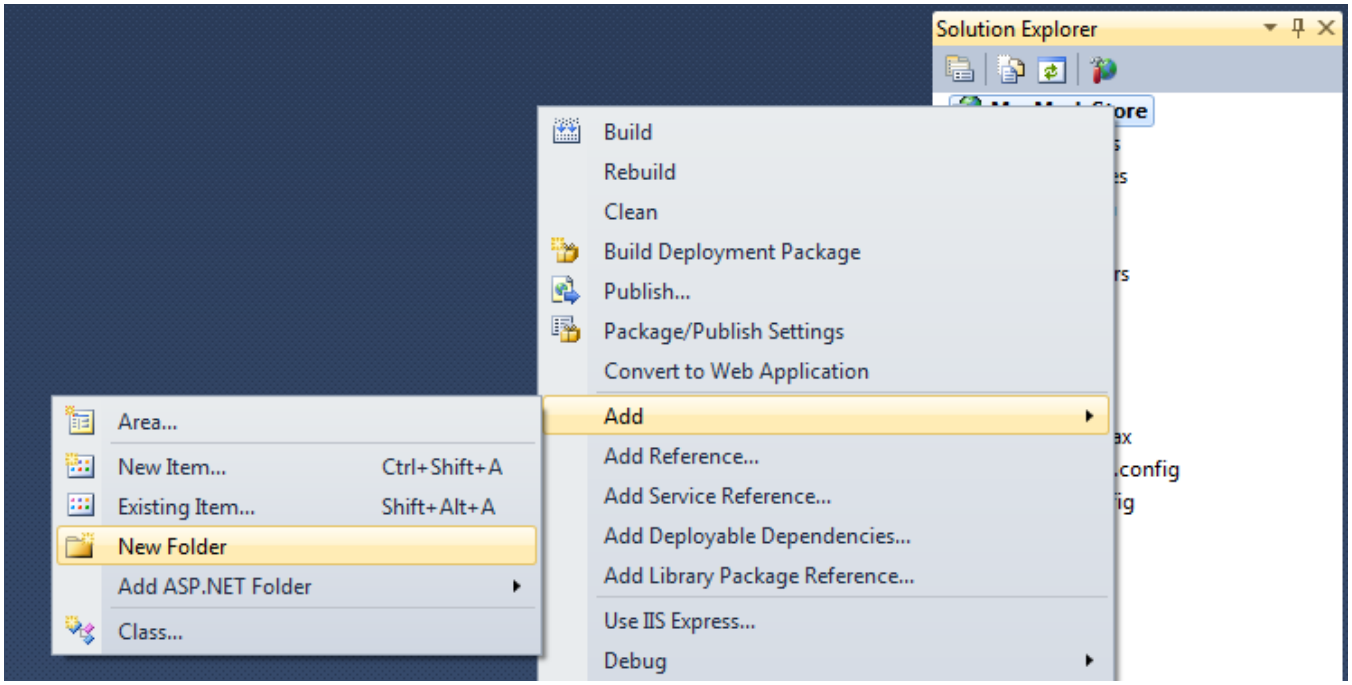
ViewModels

Our Shopping Cart Controller will need to communicate some complex information to its views which doesn't map cleanly to our Model objects. We don't want to modify our Models to suit our views; Model classes should represent our domain, not the user interface. One solution would be to pass the information to our Views using the ViewBag class, as we did with the Store Manager dropdown information, but passing a lot of information via ViewBag gets hard to manage.

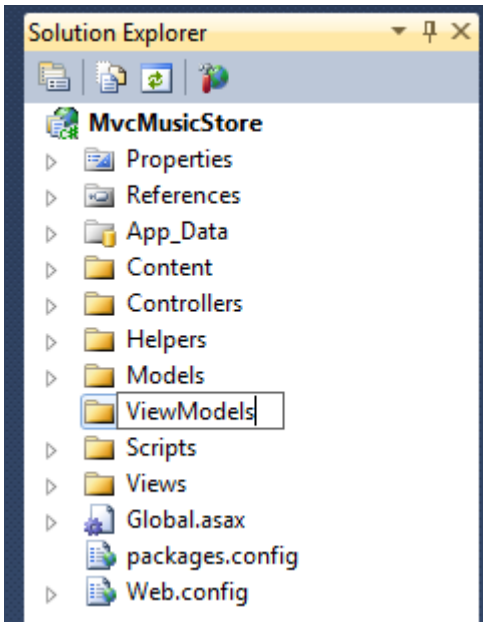
A solution to this is to use the *ViewModel* pattern. When using this pattern we create strongly-typed classes that are optimized for our specific view scenarios, and which expose properties for the dynamic values/content needed by our view templates. Our controller classes can then populate and pass these view-optimized classes to our view template to use. This enables type-safety, compile-time checking, and editor IntelliSense within view templates.

We'll create two View Models for use in our Shopping Cart controller: the ShoppingCartViewModel will hold the contents of the user's shopping cart, and the ShoppingCartRemoveViewModel will be used to display confirmation information when a user removes something from their cart.

Let's create a new ViewModels folder in the root of our project to keep things organized. Right-click the project, select Add / New Folder.



Name the folder ViewModels.



Next, add the ShoppingCartViewModel class in the ViewModels folder. It has two properties: a list of Cart items, and a decimal value to hold the total price for all items in the cart.

```
using System.Collections.Generic;
using MvcMusicStore.Models;

namespace MvcMusicStore.ViewModels
{
    public class ShoppingCartViewModel
```

```
    {
        public List<Cart> CartItems { get; set; }
        public decimal CartTotal { get; set; }
    }
}
```

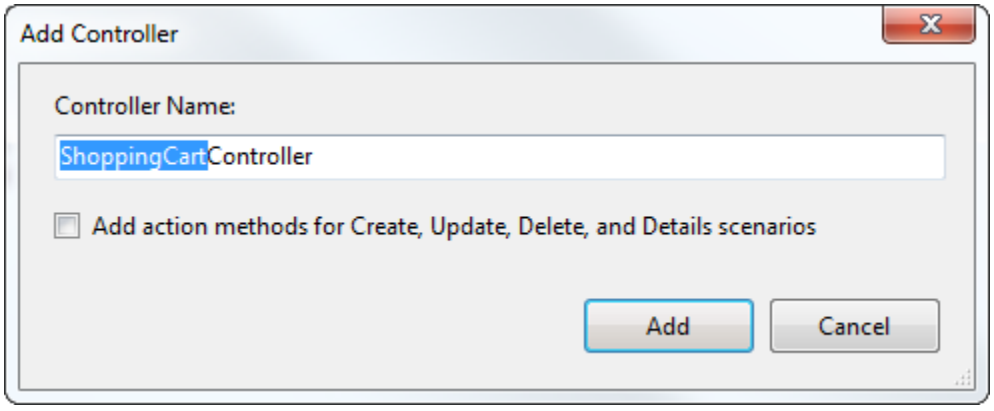
Now add the ShoppingCartRemoveViewModel to the ViewModels folder, with the following four properties.

```
namespace MvcMusicStore.ViewModels
{
    public class ShoppingCartRemoveViewModel
    {
        public string Message { get; set; }
        public decimal CartTotal { get; set; }
        public int CartCount { get; set; }
        public int ItemCount { get; set; }
        public int DeleteId { get; set; }
    }
}
```

The Shopping Cart Controller

The Shopping Cart controller has three main purposes: adding items to a cart, removing items from the cart, and viewing items in the cart. It will make use of the three classes we just created: ShoppingCartViewModel, ShoppingCartRemoveViewModel, and ShoppingCart. As in the StoreController and StoreManagerController, we'll add a field to hold an instance of MusicStoreEntities.

Add a new Shopping Cart controller to the project, leaving the checkbox for Create, Update, Delete, and Details action methods unchecked.



Here's the complete ShoppingCart Controller. The Index and Add Controller actions should look very familiar. The Remove and CartSummary controller actions handle two special cases, which we'll discuss in the following section.

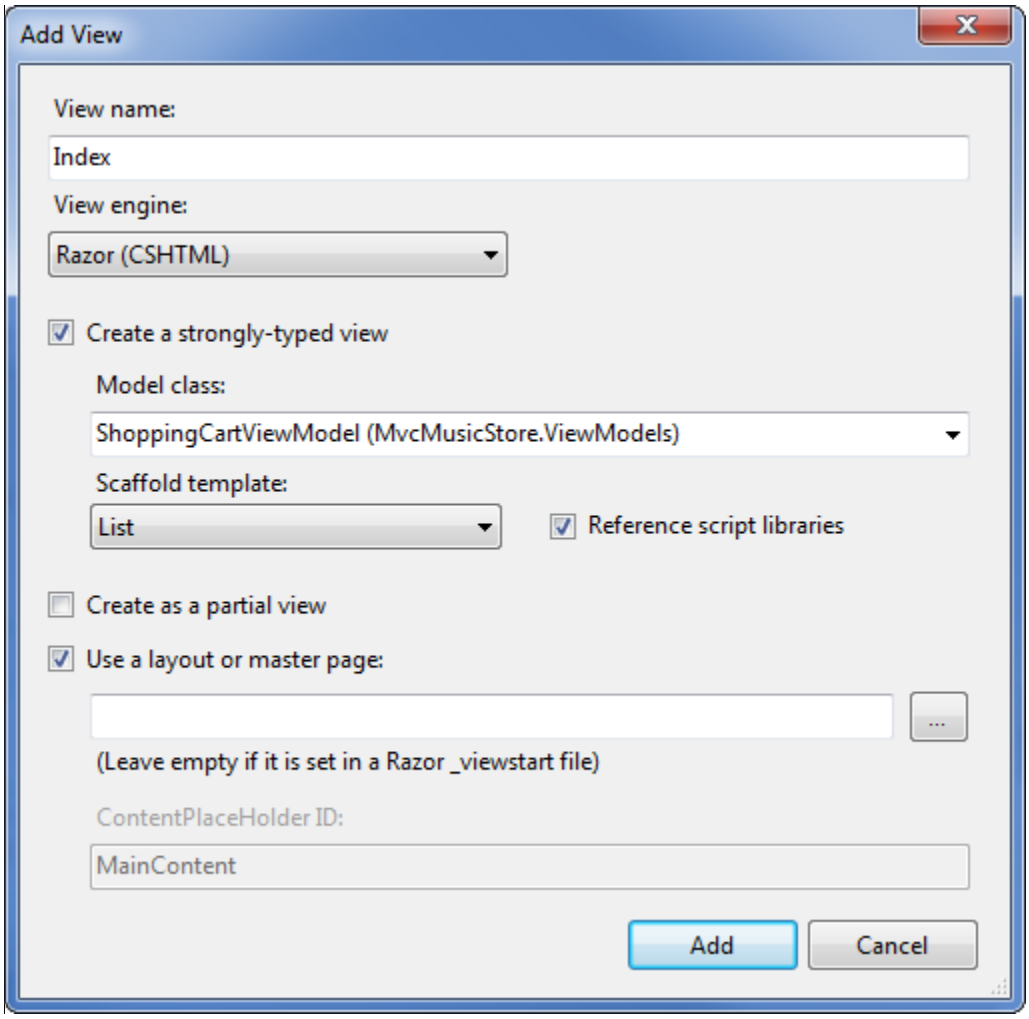
```
using System.Linq;
using System.Web.Mvc;
using MvcMusicStore.Models;
```

```
using MvcMusicStore.ViewModels;

namespace MvcMusicStore.Controllers
{
    public class ShoppingCartController : Controller
    {
        MusicStoreEntities storeDB = new MusicStoreEntities();

        //
        // GET: /ShoppingCart/

        public ActionResult Index()
        {
            var cart = ShoppingCart.GetCart(this.HttpContext);

            // Set up our ViewModel
            var viewModel = new ShoppingCartViewModel
            {
                CartItems = cart.GetCartItems(),
                CartTotal = cart.GetTotal()
            };

            // Return the view
            return View(viewModel);
        }

        //
        // GET: /Store/AddToCart/5

        public ActionResult AddToCart(int id)
        {

            // Retrieve the album from the database
            var addedAlbum = storeDB.Albums
                .Single(album => album.AlbumId == id);

            // Add it to the shopping cart
            var cart = ShoppingCart.GetCart(this.HttpContext);

            cart.AddToCart(addedAlbum);

            // Go back to the main store page for more shopping
            return RedirectToAction("Index");
        }

        //
        // AJAX: /ShoppingCart/RemoveFromCart/5

        [HttpPost]
        public ActionResult RemoveFromCart(int id)
        {
            // Remove the item from the cart
            var cart = ShoppingCart.GetCart(this.HttpContext);

            // Get the name of the album to display confirmation
```

```
            string albumName = storeDB.Carts
                .Single(item => item.RecordId == id).Album.Title;

            // Remove from cart
            int itemCount = cart.RemoveFromCart(id);

            // Display the confirmation message
            var results = new ShoppingCartRemoveViewModel
            {
                Message = Server.HtmlEncode(albumName) +
                    " has been removed from your shopping cart.",
                CartTotal = cart.GetTotal(),
                CartCount = cart.GetCount(),
                ItemCount = itemCount,
                DeleteId = id
            };

            return Json(results);
        }

        //
        // GET: /ShoppingCart/CartSummary

        [ChildActionOnly]
        public ActionResult CartSummary()
        {
            var cart = ShoppingCart.GetCart(this.HttpContext);

            ViewData["CartCount"] = cart.GetCount();

            return PartialView("CartSummary");
        }
    }
}
```

Ajax Updates using Ajax.ActionLink

We'll next create a Shopping Cart Index page that is strongly typed to the ShoppingCartViewModel and uses the List View template using the same method as before.



However, instead of using an Html.ActionLink to remove items from the cart, we'll use Ajax.ActionLink:

```
@Ajax.ActionLink("Remove from cart", "RemoveFromCart",
    new { id = item.RecordId }, new AjaxOptions { OnSuccess = "handleUpdate" })
```

This method works very similarly to the Html.ActionLink helper method, but instead of posting the form it just makes an AJAX callback to our RemoveFromCart. The RemoveFromCart returns a JSON serialized result, which is automatically passed to the JavaScript method specified in our AjaxOptions OnSuccess parameter – handleUpdate in this case. The handleUpdate Javascript function parses the JSON results and performs four quick updates to the page using jQuery:

1. Removes the deleted album from the list
2. Updates the cart count in the header
3. Displays an update message to the user
4. Updates the cart total price

Since the remove scenario is being handled by an Ajax callback within the Index view, we don't need an additional view for RemoveFromCart action. Here is the complete code for the /ShoppingCart/Index view:

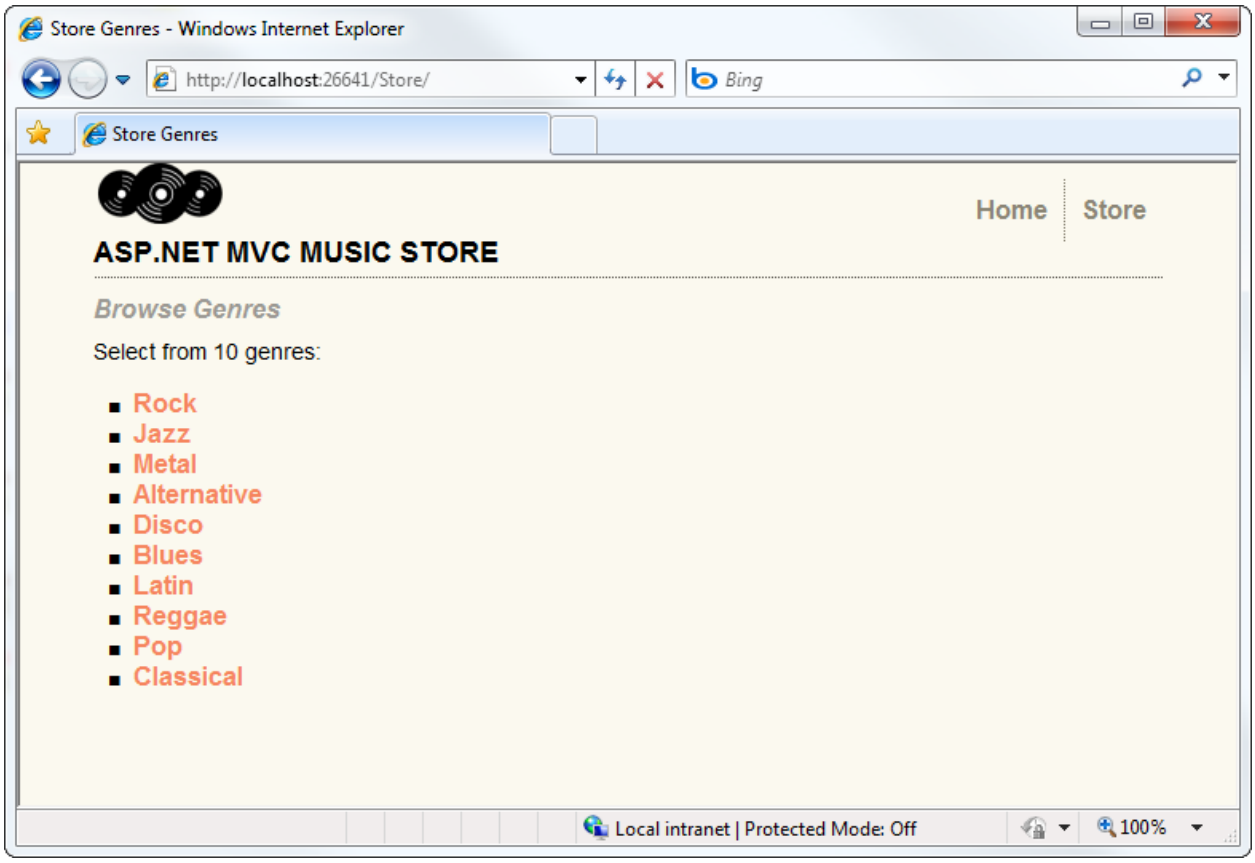
```
@model MvcMusicStore.ViewModels.ShoppingCartViewModel
@{
    ViewBag.Title = "Shopping Cart";
}
<script src="/Scripts/jquery-1.4.4.min.js" type="text/javascript"></script>
<script type="text/javascript">
    $(function () {
        // Document.ready -> link up remove event handler
        $(".RemoveLink").click(function () {
            // Get the id from the link
            var recordToDelete = $(this).attr("data-id");

            if (recordToDelete != '') {

                // Perform the ajax post
                $.post("/ShoppingCart/RemoveFromCart", { "id": recordToDelete },
                    function (data) {
                        // Successful requests get here
                        // Update the page elements
                        if (data.ItemCount == 0) {
                            $('#row-' + data.DeleteId).fadeOut('slow');
                        } else {
                            $('#item-count-' + data.DeleteId).text(data.ItemCount);
                        }

                        $('#cart-total').text(data.CartTotal);
                        $('#update-message').text(data.Message);
                        $('#cart-status').text('Cart (' + data.CartCount + ')');
                    });
            }
        });

    });


    function handleUpdate() {
        // Load and deserialize the returned JSON data
        var json = context.get_data();
        var data = Sys.Serialization.JavaScriptSerializer.deserialize(json);

        // Update the page elements
        if (data.ItemCount == 0) {
            $('#row-' + data.DeleteId).fadeOut('slow');
        } else {
            $('#item-count-' + data.DeleteId).text(data.ItemCount);
        }

        $('#cart-total').text(data.CartTotal);
        $('#update-message').text(data.Message);
        $('#cart-status').text('Cart (' + data.CartCount + ')');
    }
</script>
<h3>
    <em>Review</em> your cart:
</h3>
```

```
<p class="button">
    @Html.ActionLink("Checkout >>", "AddressAndPayment", "Checkout")
</p>
<div id="update-message">
</div>
<table>
    <tr>
        <th>
            Album Name
        </th>
        <th>
            Price (each)
        </th>
        <th>
            Quantity
        </th>
        <th></th>
    </tr>
    @foreach (var item in Model.CartItems)
    {
        <tr id="row-@item.RecordId">
            <td>
                @Html.ActionLink(item.Album.Title, "Details", "Store", new { id =
item.AlbumId }, null)
            </td>
            <td>
                @item.Album.Price
            </td>
            <td id="item-count-@item.RecordId">
                @item.Count
            </td>
            <td>
                <a href="#" class="RemoveLink" data-id="@item.RecordId">Remove from
cart</a>
            </td>
        </tr>
    }
    <tr>
        <td>
            Total
        </td>
        <td>
        </td>
        <td>
        </td>
        <td id="cart-total">
            @Model.CartTotal
        </td>
    </tr>
</table>
```

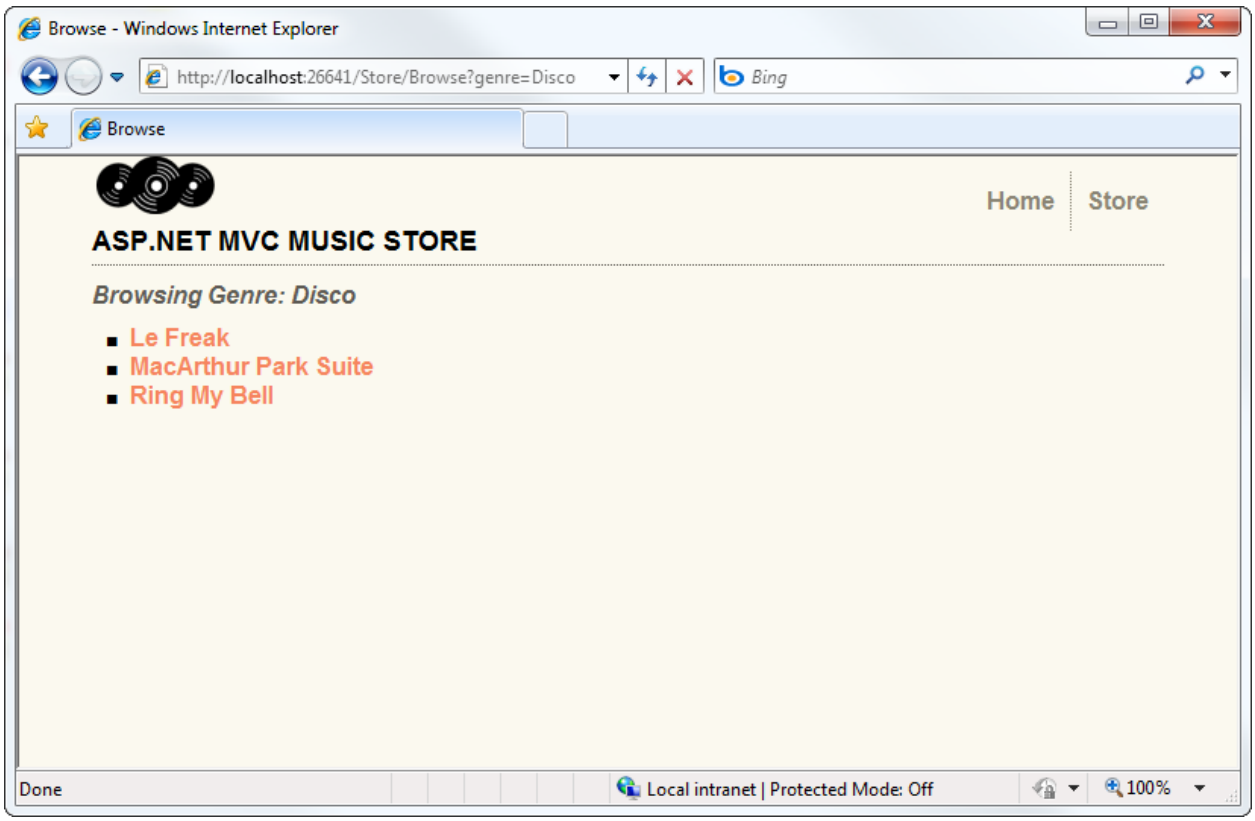
To test this out, let's update our Store Details view to include an "Add to cart" button. While we're at it, we can include some of the Album additional information which we've added since we last updated this view: Genre, Artist, Price, and Album Art. The updated Store Details view code appears as shown below.

```
@model MvcMusicStore.Models.Album

@{
    ViewBag.Title = "Album - " + Model.Title;
}

<h2>@Model.Title</h2>

<p>
    <img alt="@Model.Title" src="@Model.AlbumArtUrl" />
</p>

<div id="album-details">
    <p>
        <em>Genre:</em>
        @Model.Genre.Name
    </p>
    <p>
        <em>Artist:</em>
        @Model.Artist.Name
    </p>
    <p>
        <em>Price:</em>
        @String.Format("{0:F}", Model.Price)
    </p>
    <p class="button">
        @Html.ActionLink("Add to cart", "AddToCart",
        "ShoppingCart", new { id = Model.AlbumId }, "")
    </p>
</div>
```

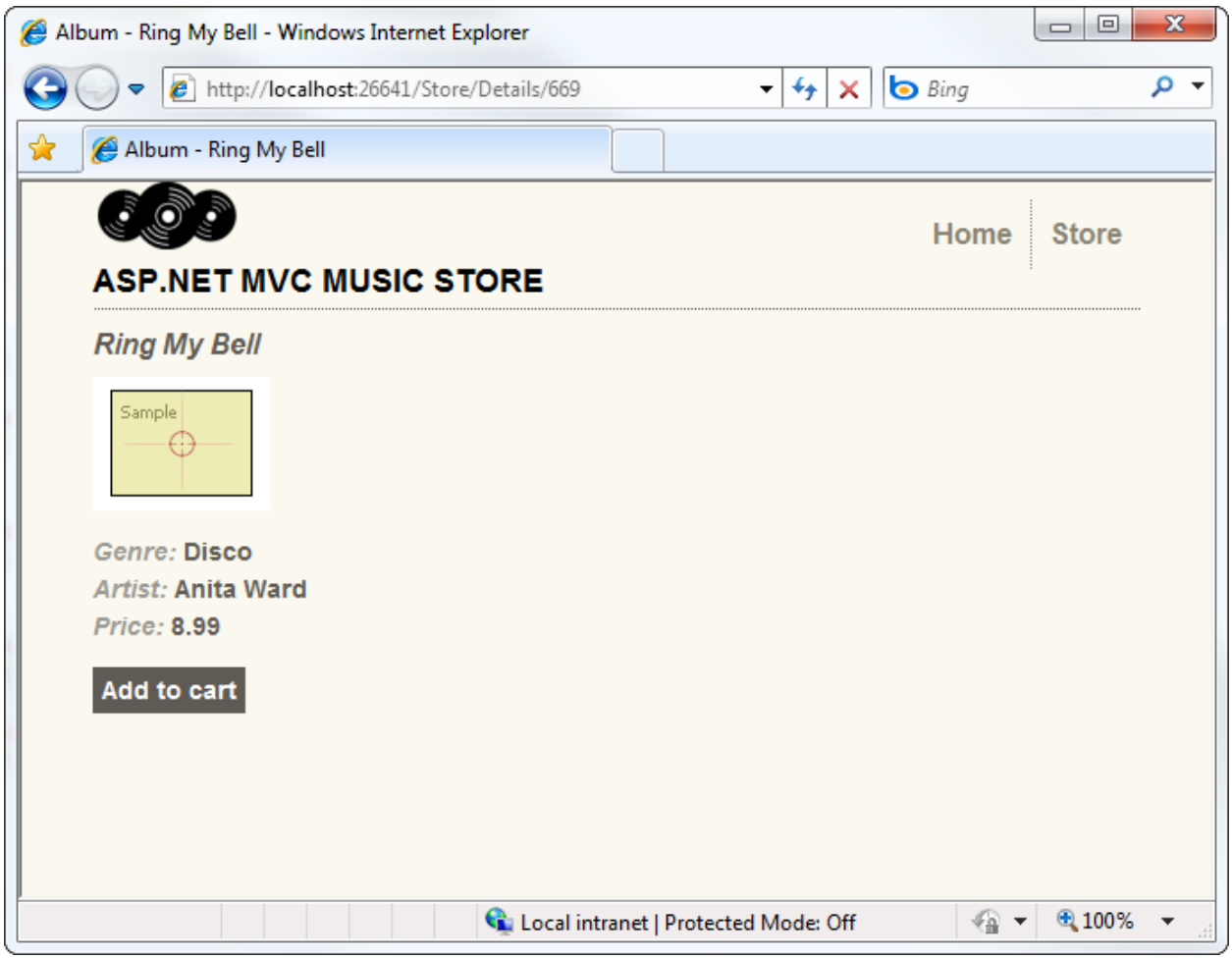
Now we can click through the store and test adding and removing Albums to and from our shopping cart. Run the application and browse to the Store Index.



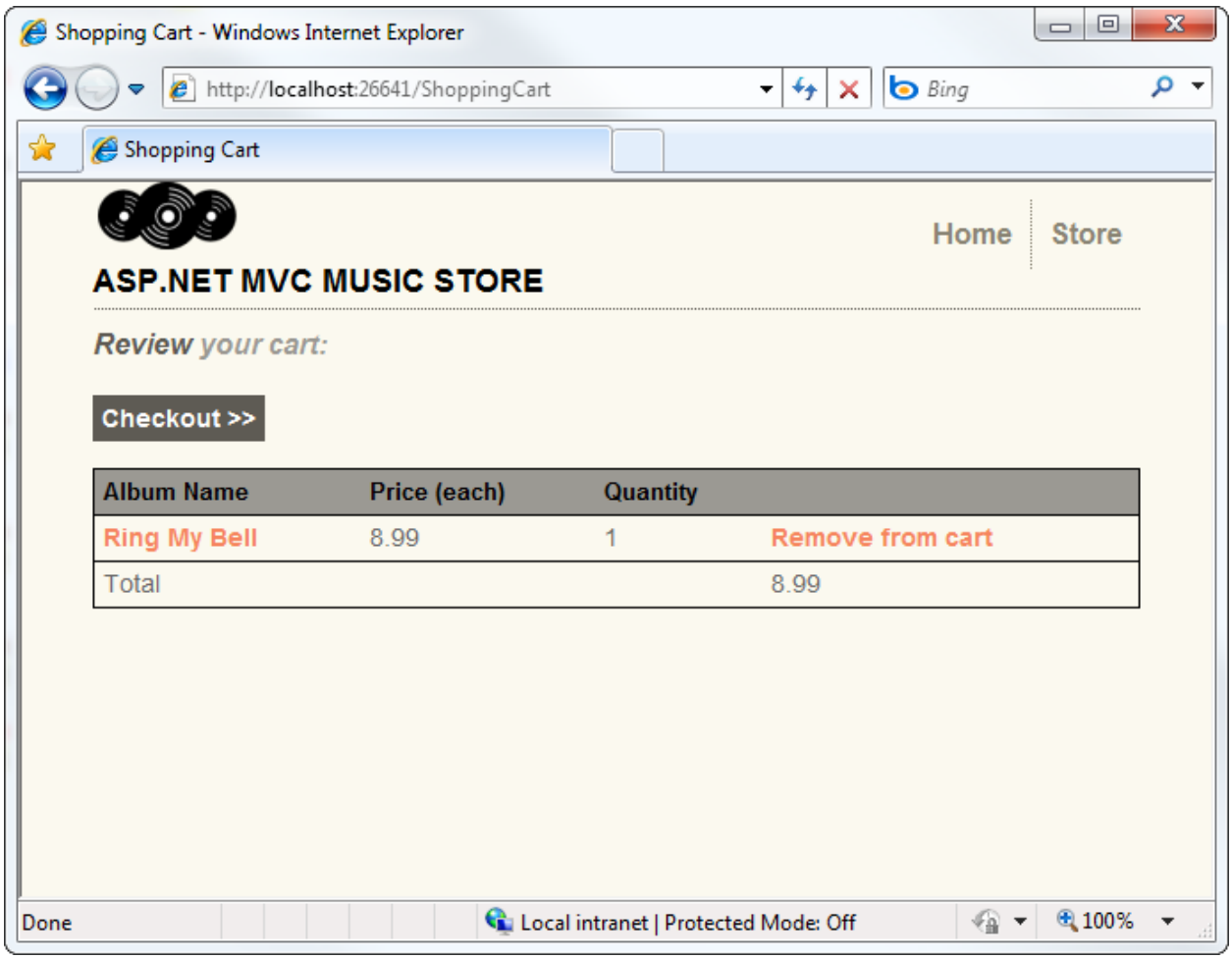
Next, click on a Genre to view a list of albums.



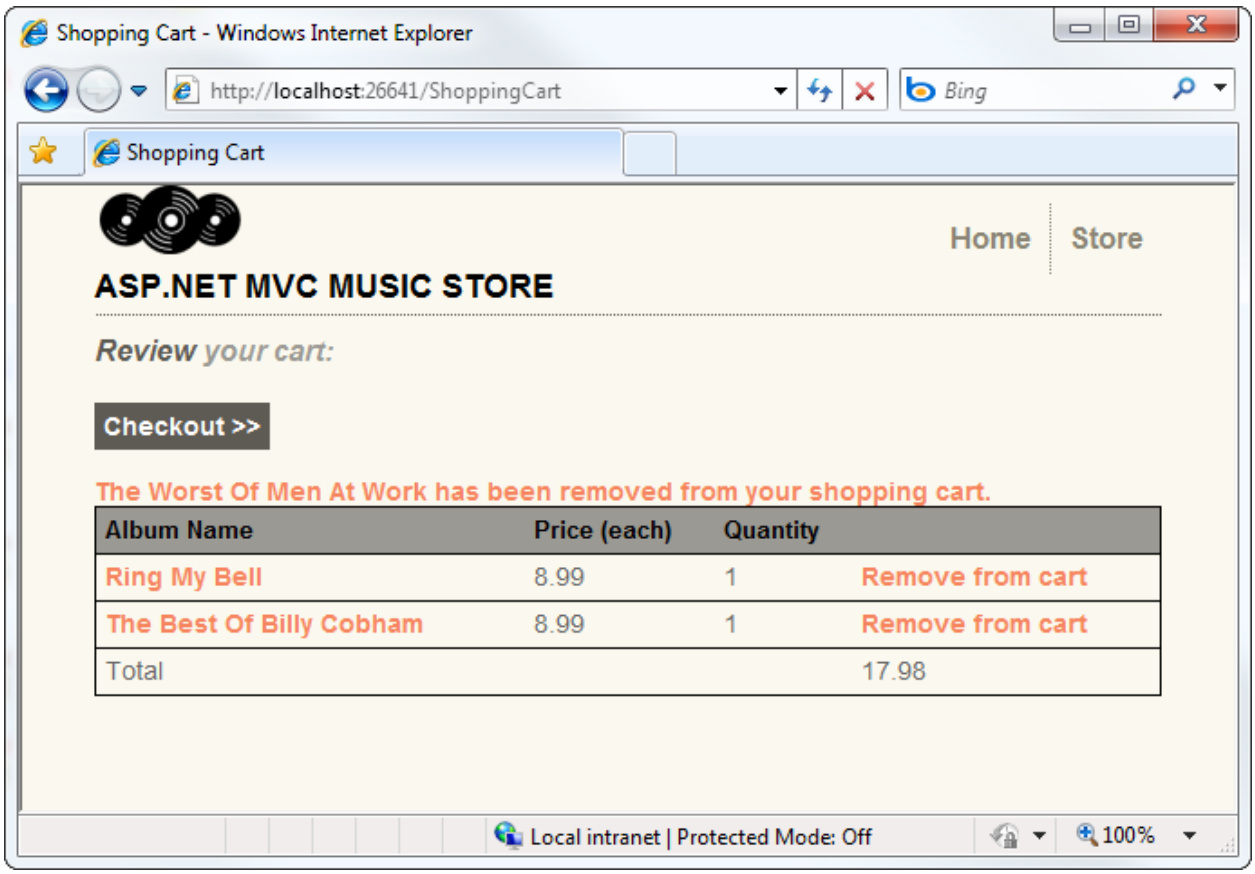
Clicking on an Album title now shows our updated Album Details view, including the "Add to cart" button.



Clicking the "Add to cart" button shows our Shopping Cart Index view with the shopping cart summary list.



After loading up your shopping cart, you can click on the Remove from cart link to see the Ajax update to your shopping cart.

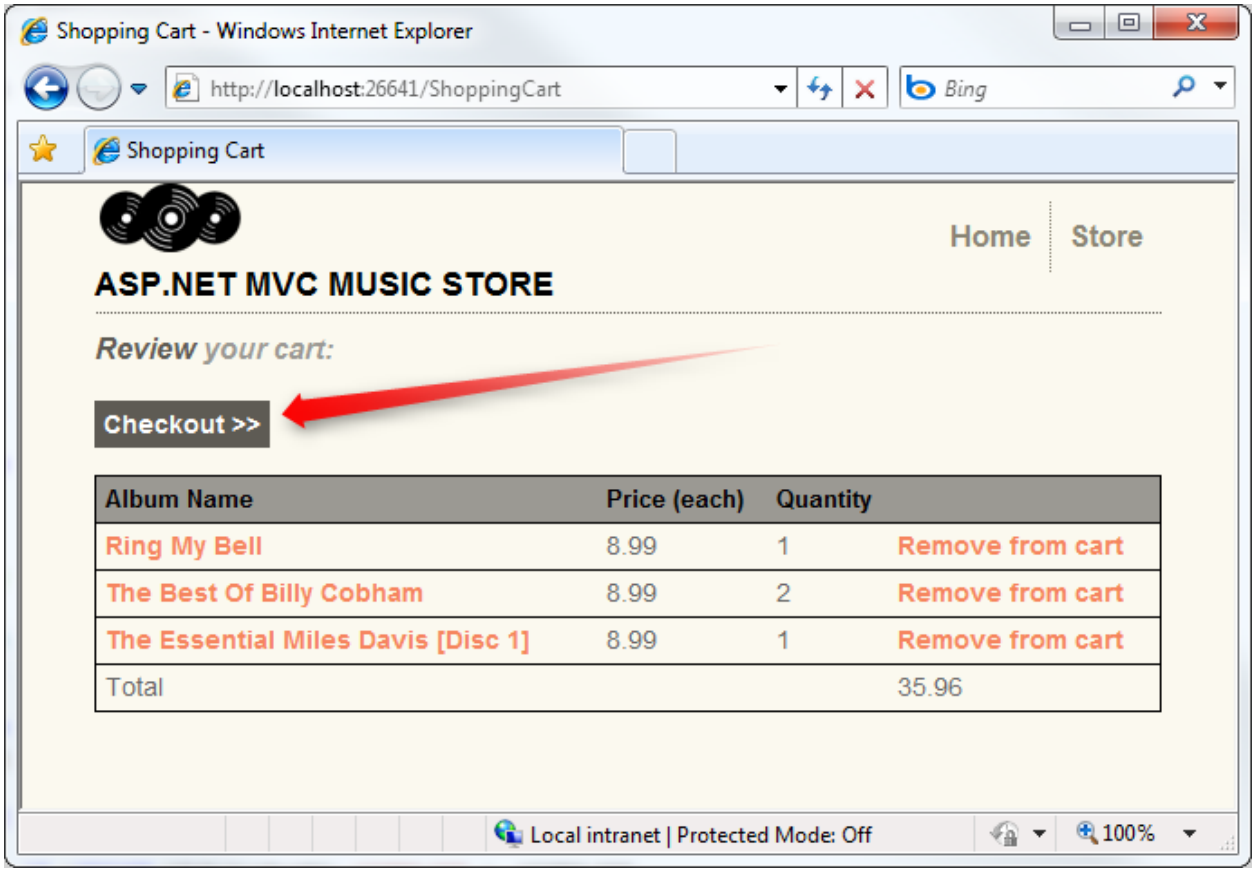


We've built out a working shopping cart which allows unregistered users to add items to their cart. In the following section, we'll allow them to register and complete the checkout process.

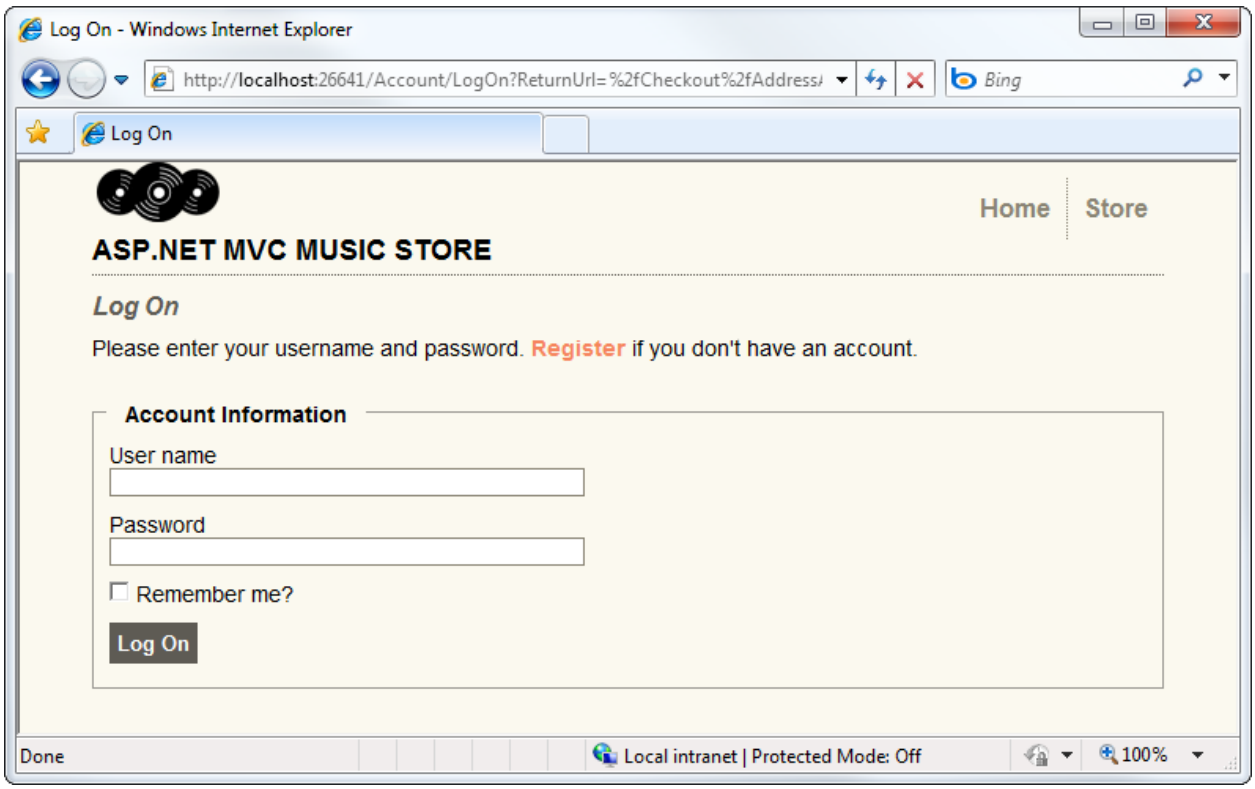
9. Registration and Checkout

In this section, we will be creating a CheckoutController which will collect the shopper's address and payment information. We will require users to register with our site prior to checking out, so this controller will require authorization.

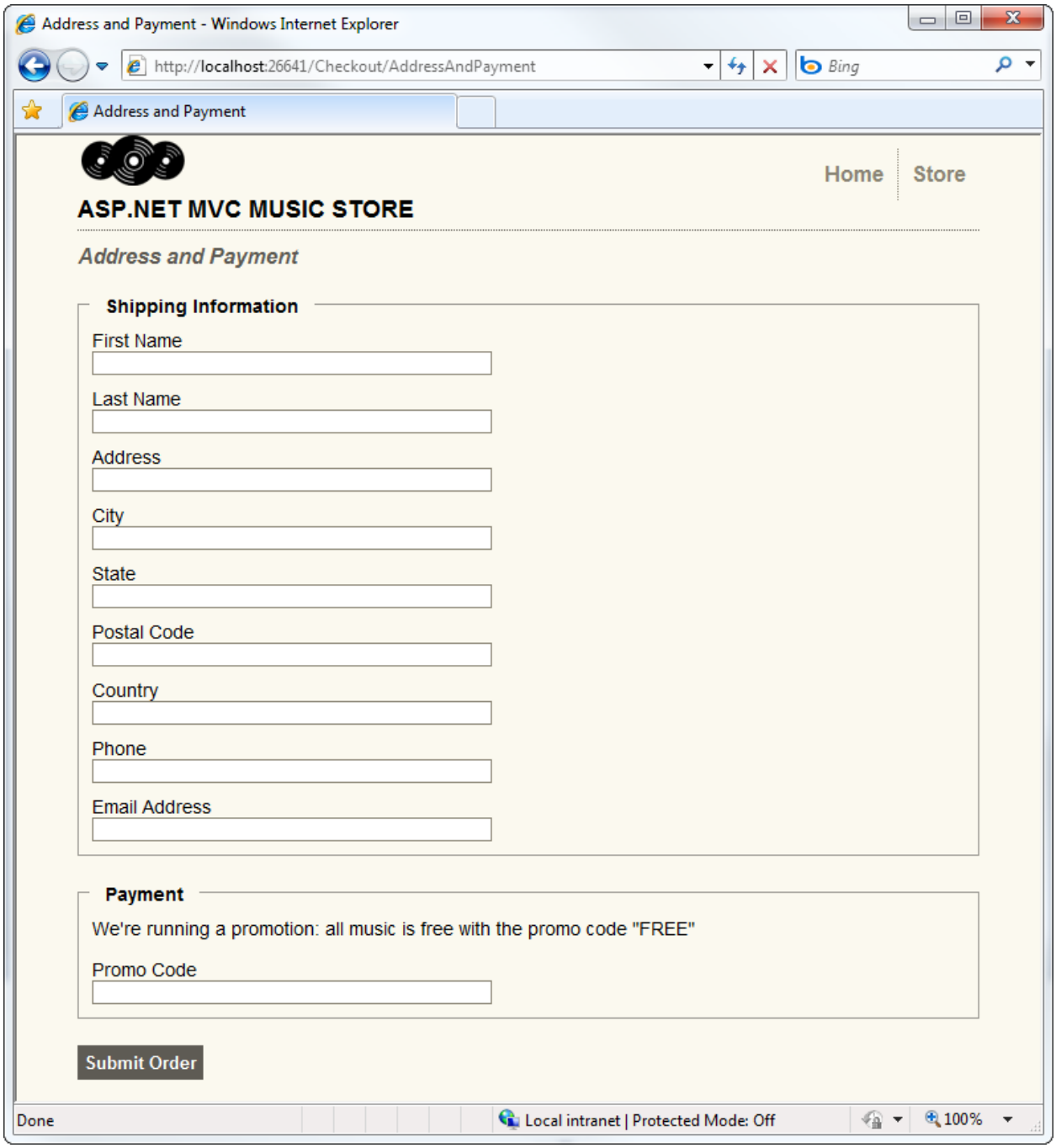
Users will navigate to the checkout process from their shopping cart by clicking the "Checkout" button.



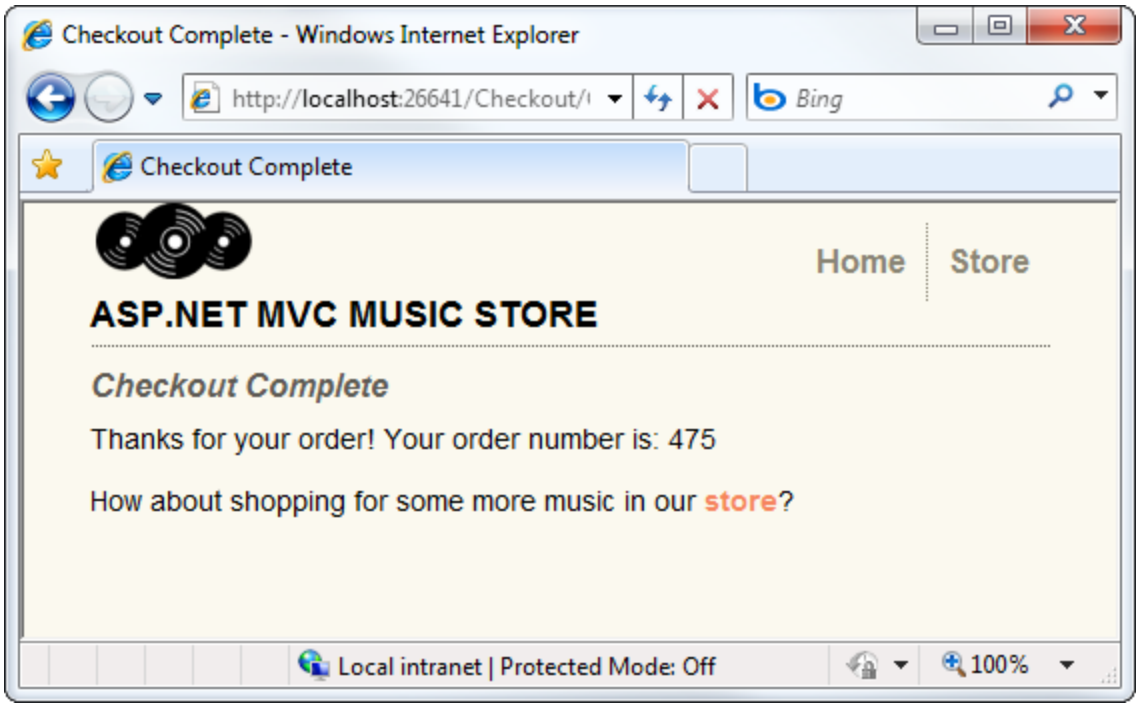
If the user is not logged in, they will be prompted to.



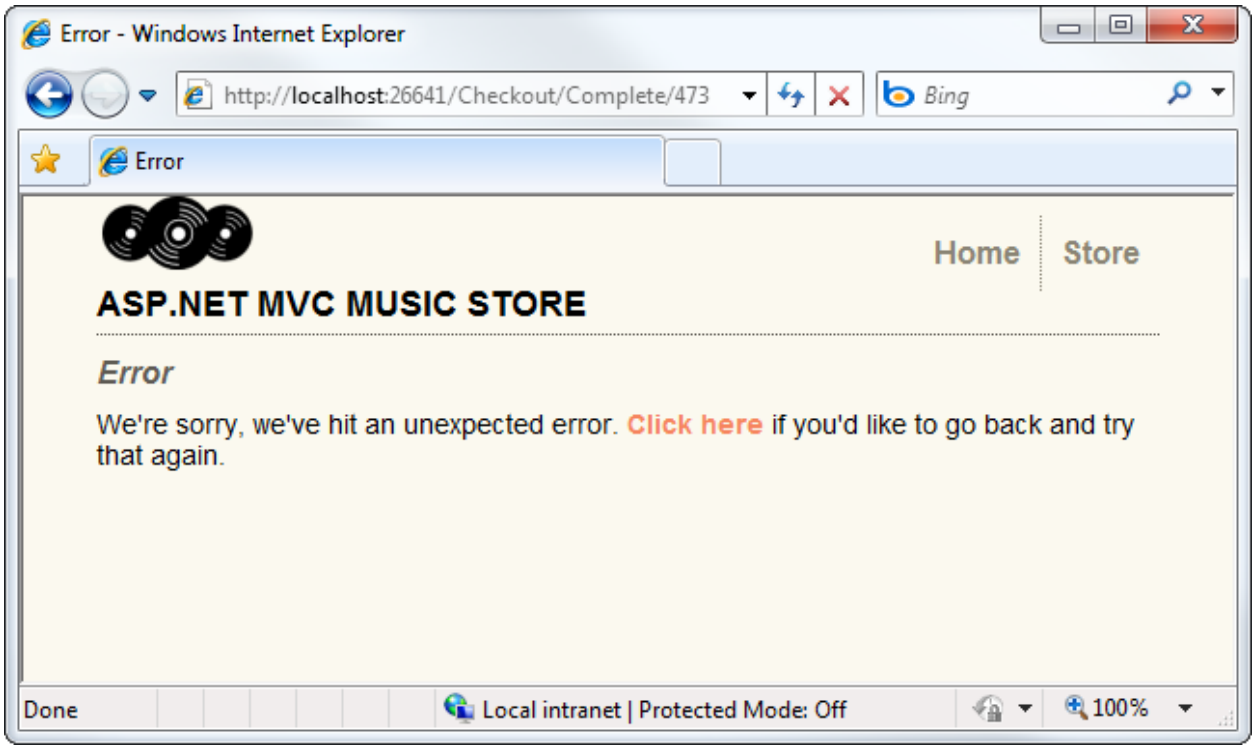
Upon successful login, the user is then shown the Address and Payment view.



Once they have filled the form and submitted the order, they will be shown the order confirmation screen.



Attempting to view either a non-existent order or an order that doesn't belong to you will show the Error view.



Migrating the Shopping Cart

While the shopping process is anonymous, when the user clicks on the Checkout button, they will be required to register and login. Users will expect that we will maintain their shopping cart information

between visits, so we will need to associate the shopping cart information with a user when they complete registration or login.

This is actually very simple to do, as our ShoppingCart class already has a method which will associate all the items in the current cart with a username. We will just need to call this method when a user completes registration or login.

Open the AccountController class that we added when we were setting up Membership and Authorization. Add the following MigrateShoppingCart method:

```
private void MigrateShoppingCart(string UserName)
{
    // Associate shopping cart items with logged-in user
    var cart = ShoppingCart.GetCart(this.HttpContext);

    cart.MigrateCart(UserName);
    Session[ShoppingCart.CartSessionKey] = UserName;
}
```

Next, modify the LogOn post action to call MigrateShoppingCart after the user has been validated, as shown below:

```
[HttpPost]
public ActionResult LogOn(LogOnModel model, string returnUrl)
{
    if (ModelState.IsValid)
    {
        if (MembershipService.ValidateUser(model.UserName, model.Password))
        {
            MigrateShoppingCart(model.UserName);

            FormsService.SignIn(model.UserName, model.RememberMe);
            if (Url.IsLocalUrl(returnUrl))
            {
                return Redirect(returnUrl);
            }
            else
            {
                return RedirectToAction("Index", "Home");
            }
        }
        else
        {
            ModelState.AddModelError("", "The user name or password provided is
incorrect.");
        }
    }

    // If we got this far, something failed, redisplay form
    return View(model);
}
```

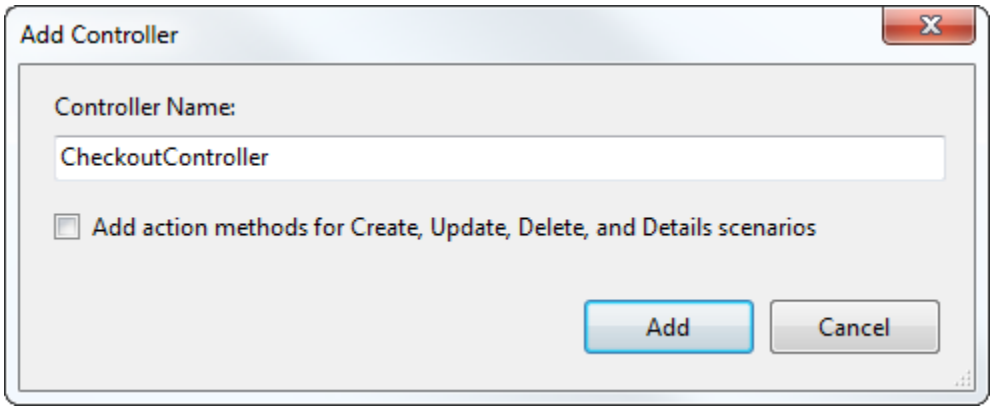
Make the same change to the Register post action, immediately after the user account is successfully created:

```
[HttpPost]
public ActionResult Register(RegisterModel model)
{
    if (ModelState.IsValid)
    {
        // Attempt to register the user
        MembershipCreateStatus createStatus = MembershipService.CreateUser(model.UserName,
model.Password, model.Email);

        if (createStatus == MembershipCreateStatus.Success)
        {
            MigrateShoppingCart(model.UserName);

            FormsService.SignIn(model.UserName, false /* createPersistentCookie */);
            return RedirectToAction("Index", "Home");
        }
        else
        {
            ModelState.AddModelError("", AccountValidation.ErrorCodeToString(createStatus));
        }
    }

    // If we got this far, something failed, redisplay form
    ViewBag.PasswordLength = MembershipService.MinPasswordLength;
    return View(model);
}
```

That's it - now an anonymous shopping cart will be automatically transferred to a user account upon successful registration or login.

Creating the CheckoutController

Right-click on the Controllers folder and add a new Controller to the project named CheckoutController. Do not select the checkbox to add Create, Update, Delete, or Details controller actions.



First, add the Authorize attribute above the Controller class declaration to require users to register before checkout:

```
namespace MvcMusicStore.Controllers
{
    [Authorize]
    public class CheckoutController : Controller
```

Note: This is similar to the change we previously made to the StoreManagerController, but in that case the Authorize attribute required that the user be in an Administrator role. In the Checkout Controller, we're requiring the user be logged in but aren't requiring that they be administrators.

For the sake of simplicity, we won't be dealing with payment information in this tutorial. Instead, we are allowing users to check out using a promotional code. We will store this promotional code using a constant named PromoCode.

As in the StoreController, we'll declare a field to hold an instance of the MusicStoreEntities class, named storeDB. In order to make use of the MusicStoreEntities class, we will need to add a using statement for the MvcMusicStore.Models namespace. The top of our Checkout controller appears below.

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

namespace MvcMusicStore.Controllers
{
    [Authorize]
    public class CheckoutController : Controller
    {
        MusicStoreEntities storeDB = new MusicStoreEntities();
        const string PromoCode = "FREE";
```

The CheckoutController will have the following controller actions:

AddressAndPayment (GET method) will display a form to allow the user to enter their information.

AddressAndPayment (POST method) will validate the input and process the order.

Complete will be shown after a user has successfully finished the checkout process. This view will include the user's order number, as confirmation.

First, let's rename the Index controller action (which was generated when we created the controller) to AddressAndPayment. This controller action just displays the checkout form, so it doesn't require any model information.

```
//
// GET: /Checkout/AddressAndPayment

public ActionResult AddressAndPayment()
{
```

```
    return View();
}
```

Our AddressAndPayment POST method will follow the same pattern we used in the StoreManagerController: it will try to accept the form submission and complete the order, and will re-display the form if it fails.

After validating the form input meets our validation requirements for an Order, we will check the PromoCode form value directly. Assuming everything is correct, we will save the updated information with the order, tell the ShoppingCart object to complete the order process, and redirect to the Complete action.

```
//
// POST: /Checkout/AddressAndPayment

[HttpPost]
public ActionResult AddressAndPayment(FormCollection values)
{
    var order = new Order();
    TryUpdateModel(order);

    try
    {
        if (string.Equals(values["PromoCode"], PromoCode,
            StringComparison.OrdinalIgnoreCase) == false)
        {
            return View(order);
        }
        else
        {
            order.Username = User.Identity.Name;
            order.OrderDate = DateTime.Now;

            //Save Order
            storeDB.AddToOrders(order);
            storeDB.SaveChanges();

            //Process the order
            var cart = ShoppingCart.GetCart(this.HttpContext);
            cart.CreateOrder(order);

            return RedirectToAction("Complete",
                new { id = order.OrderId });
        }

    }
    catch
    {
        //Invalid - redisplay with errors
        return View(order);
    }
}
```

Upon successful completion of the checkout process, users will be redirected to the Complete controller action. This action will perform a simple check to validate that the order does indeed belong to the logged-in user before showing the order number as a confirmation.

```
//
// GET: /Checkout/Complete

public ActionResult Complete(int id)
{
    // Validate customer owns this order
    bool isValid = storeDB.Orders.Any(
        o => o.OrderId == id &&
        o.Username == User.Identity.Name);

    if (isValid)
    {
        return View(id);
    }
    else
    {
        return View("Error");
    }
}
```

Note: The Error view was automatically created for us in the /Views/Shared folder when we began the project.

The complete CheckoutController code is as follows:

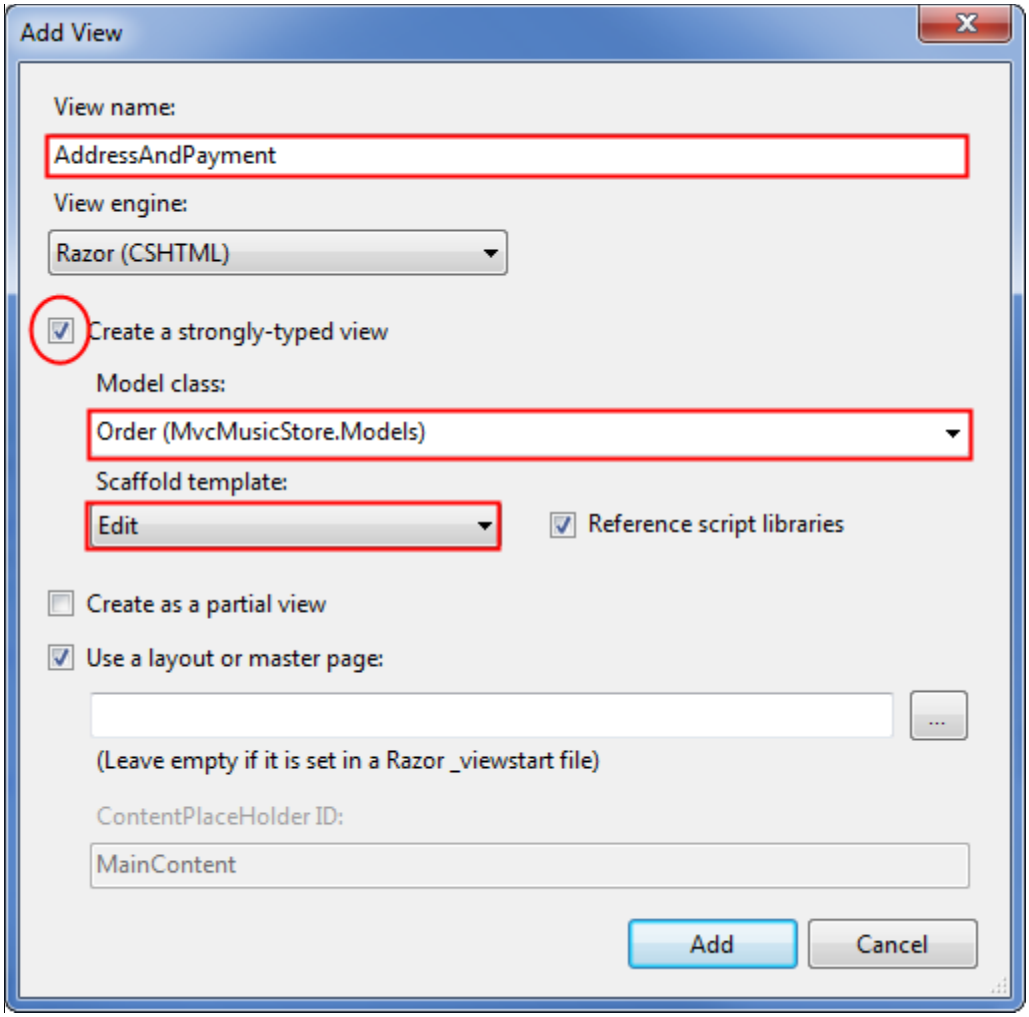
```
using System;
using System.Linq;
using System.Web.Mvc;
using MvcMusicStore.Models;

namespace MvcMusicStore.Controllers
{
    [Authorize]
    public class CheckoutController : Controller
    {
        MusicStoreEntities storeDB = new MusicStoreEntities();
        const string PromoCode = "FREE";

        //
        // GET: /Checkout/AddressAndPayment

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

        //
        // POST: /Checkout/AddressAndPayment

        [HttpPost]
```

```
public ActionResult AddressAndPayment(FormCollection values)
{
    var order = new Order();
    TryUpdateModel(order);

    try
    {
        if (string.Equals(values["PromoCode"], PromoCode,
            StringComparison.OrdinalIgnoreCase) == false)
        {
            return View(order);
        }
        else
        {
            order.Username = User.Identity.Name;
            order.OrderDate = DateTime.Now;

            //Save Order
            storeDB.AddToOrders(order);
            storeDB.SaveChanges();

            //Process the order
            var cart = ShoppingCart.GetCart(this.HttpContext);
            cart.CreateOrder(order);

            return RedirectToAction("Complete",
                new { id = order.OrderId });
        }

    }
    catch
    {
        //Invalid - redisplay with errors
        return View(order);
    }
}

//
// GET: /Checkout/Complete

public ActionResult Complete(int id)
{
    // Validate customer owns this order
    bool isValid = storeDB.Orders.Any(
        o => o.OrderId == id &&
        o.Username == User.Identity.Name);

    if (isValid)
    {
        return View(id);
    }
    else
    {
        return View("Error");
    }
```

```
        }
    }
}
```

Adding the AddressAndPayment view

Now, let's create the AddressAndPayment view. Right-click on one of the the AddressAndPayment controller actions and add a view named AddressAndPayment which is strongly typed as an Order and uses the Edit template, as shown below.



This view will make use of two of the techniques we looked at while building the StoreManagerEdit view:

- We will use Html.EditorForModel() to display form fields for the Order model
- We will leverage validation rules using an Order class with validation attributes

We'll start by updating the form code to use Html.EditorForModel(), followed by an additional textbox for the Promo Code. The complete code for the AddressAndPayment view is shown below.

```
@model MvcMusicStore.Models.Order
```

```
@{
    ViewBag.Title = "Address And Payment";
}

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

@using (Html.BeginForm()) {

    <h2>Address And Payment</h2>
    <fieldset>
        <legend>Shipping Information</legend>

        @Html.EditorForModel()
    </fieldset>
    <fieldset>
        <legend>Payment</legend>
        <p>We're running a promotion: all music is free with the promo code: "FREE"</p>

        <div class="editor-label">
            @Html.Label("Promo Code")
        </div>
        <div class="editor-field">
            @Html.TextBox("PromoCode")
        </div>
    </fieldset>

    <input type="submit" value="Submit Order" />
}
```

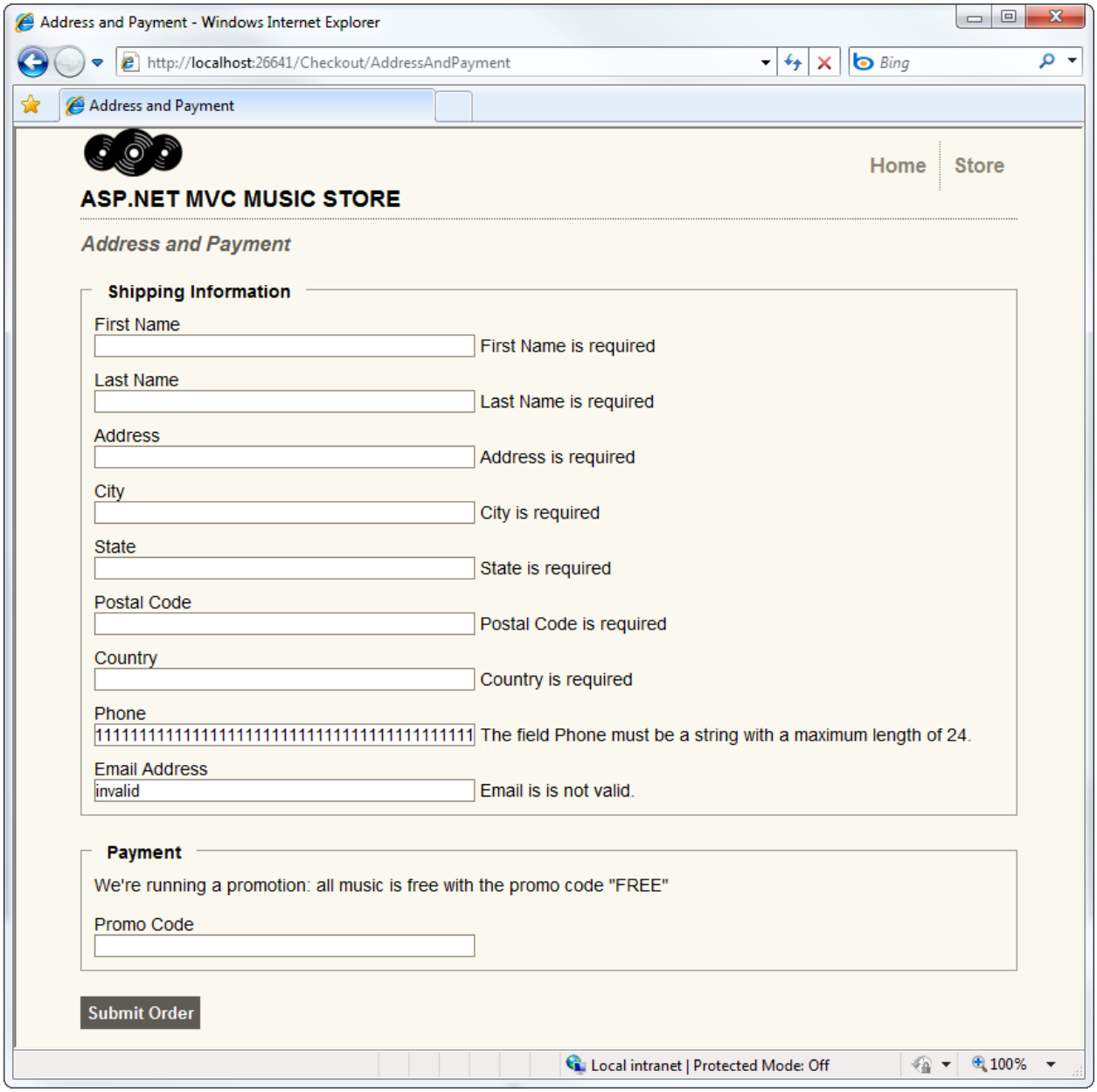
Defining validation rules for the Order

Now that our view is set up, we will set up the validation rules for our Order model as we did previously for the Album model. Right-click on the Models folder and add a class named Order. In addition to the validation attributes we used previously for the Album, we will also be using a Regular Expression to validate the user's e-mail address.

```
using System.Collections.Generic;
using System.ComponentModel;
using System.ComponentModel.DataAnnotations;
using System.Web.Mvc;

namespace MvcMusicStore.Models
{
    [Bind(Exclude = "OrderId")]
    public partial class Order
    {
        [ScaffoldColumn(false)]
        public int OrderId { get; set; }

        [ScaffoldColumn(false)]
```

```
        public System.DateTime OrderDate { get; set; }

        [ScaffoldColumn(false)]
        public string Username { get; set; }

        [Required(ErrorMessage = "First Name is required")]
        [DisplayName("First Name")]
        [StringLength(160)]
        public string FirstName { get; set; }

        [Required(ErrorMessage = "Last Name is required")]
        [DisplayName("Last Name")]
        [StringLength(160)]
        public string LastName { get; set; }

        [Required(ErrorMessage = "Address is required")]
        [StringLength(70)]
        public string Address { get; set; }

        [Required(ErrorMessage = "City is required")]
        [StringLength(40)]
        public string City { get; set; }

        [Required(ErrorMessage = "State is required")]
        [StringLength(40)]
        public string State { get; set; }

        [Required(ErrorMessage = "Postal Code is required")]
        [DisplayName("Postal Code")]
        [StringLength(10)]
        public string PostalCode { get; set; }

        [Required(ErrorMessage = "Country is required")]
        [StringLength(40)]
        public string Country { get; set; }

        [Required(ErrorMessage = "Phone is required")]
        [StringLength(24)]
        public string Phone { get; set; }

        [Required(ErrorMessage = "Email Address is required")]
        [DisplayName("Email Address")]
        [RegularExpression(@"[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}",
            ErrorMessage = "Email is is not valid.")]
        [DataType(DataType.EmailAddress)]
        public string Email { get; set; }

        [ScaffoldColumn(false)]
        public decimal Total { get; set; }

        public List<OrderDetail> OrderDetails { get; set; }
    }
}
```

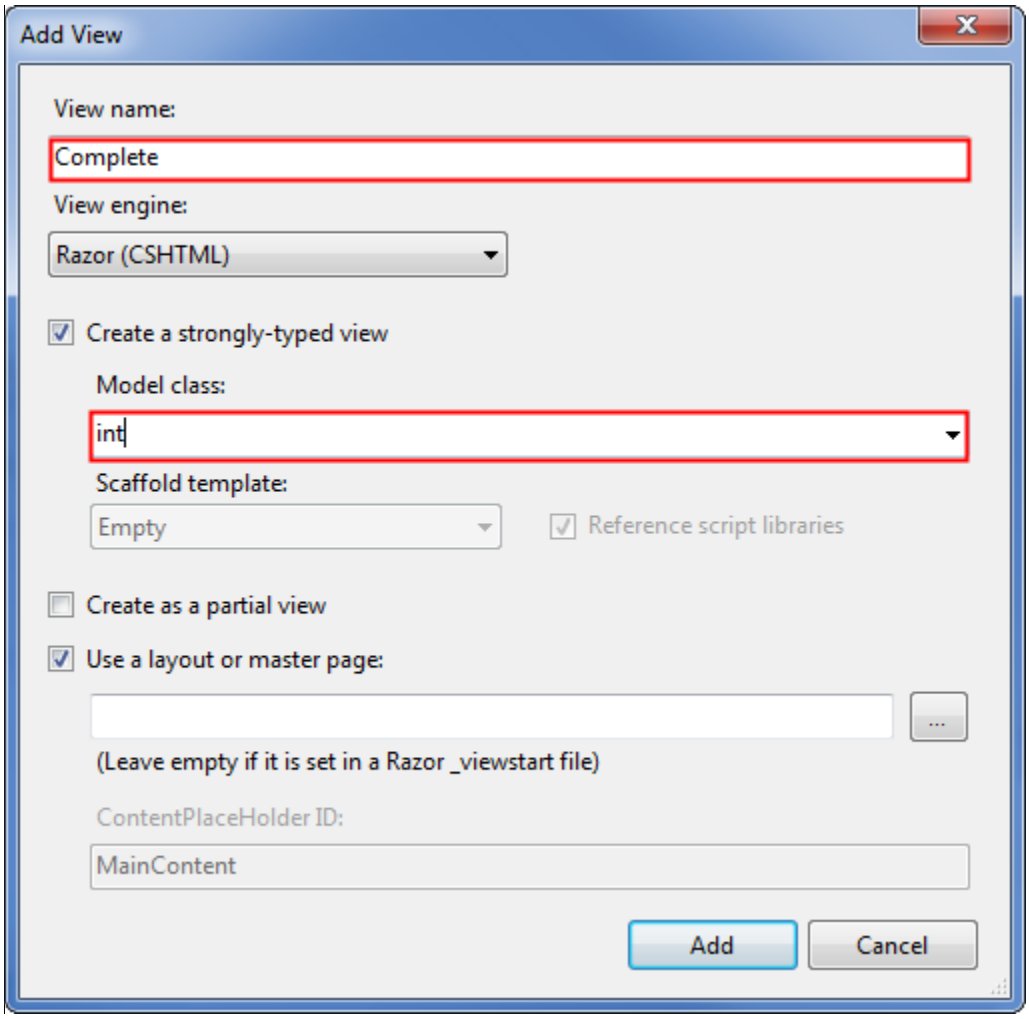
Attempting to submit the form with missing or invalid information will now show error message using client-side validation.



Okay, we've done most of the hard work for the checkout process; we just have a few odds and ends to finish. We need to add two simple views, and we need to take care of the handoff of the cart information during the login process.

Adding the Checkout Complete view

The Checkout Complete view is pretty simple, as it just needs to display the Order ID. Right-click on the Complete controller action and add a view named Complete which is strongly typed as an int.



Now we will update the view code to display the Order ID, as shown below.

```
@model int

@{
    ViewBag.Title = "Checkout Complete";
}

<h2>Checkout Complete</h2>

<p>Thanks for your order! Your order number is: @Model</p>

<p>How about shopping for some more music in our
    @Html.ActionLink("store", "Index", "Home")
</p>
```

Updating The Error view

The default template includes an Error view in the Shared views folder so that it can be re-used elsewhere in the site. This Error view contains a very simple error and doesn't use our site Layout, so we'll update it.

Since this is a generic error page, the content is very simple. We'll include a message and a link to navigate to the previous page in history if the user wants to re-try their action.

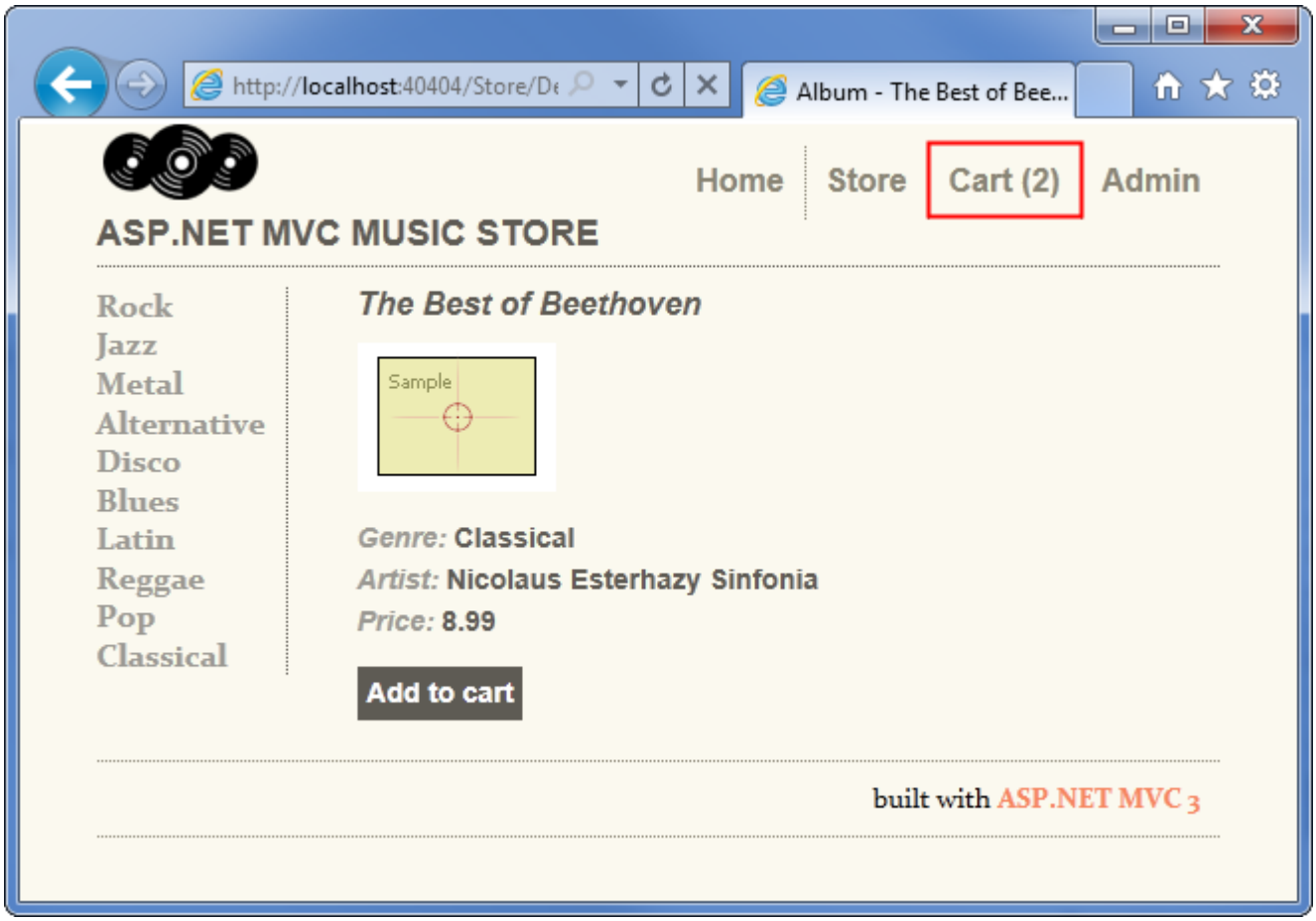
```
@{
    ViewBag.Title = "Error";
}

<h2>Error</h2>

<p>We're sorry, we've hit an unexpected error.
    <a href="javascript:history.go(-1)">Click here</a>
    if you'd like to go back and try that again.</p>
```

10. Final updates to Navigation and Site Design

We've completed all the major functionality for our site, but we still have some features to add to the site navigation, the home page, and the Store Browse page.

Creating the Shopping Cart Summary Partial View

We want to expose the number of items in the user's shopping cart across the entire site.



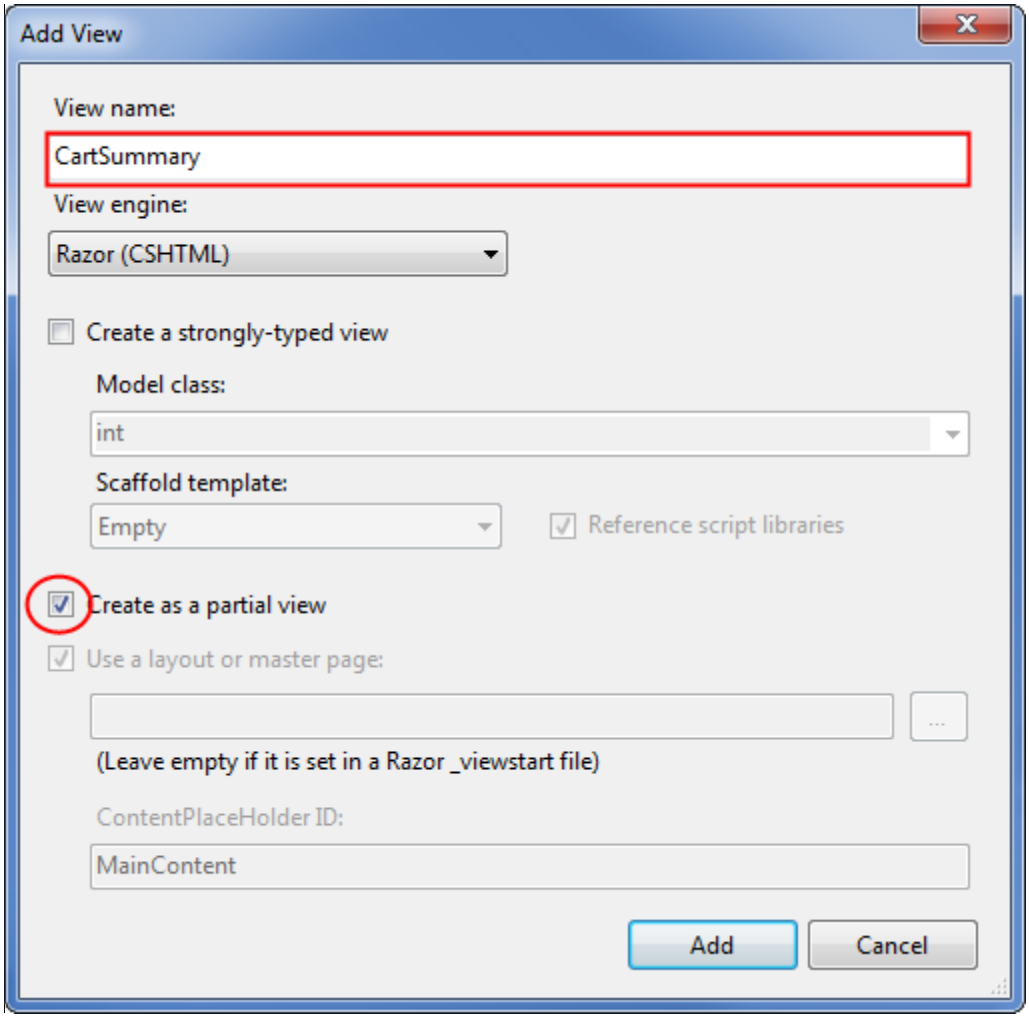
We can easily implement this by creating a partial view which is added to our Site.master.

As shown previously, the ShoppingCart controller includes a CartSummary action method which returns a partial view:

```
//
// GET: /ShoppingCart/CartSummary

[ChildActionOnly]
public ActionResult CartSummary()
{
    var cart = ShoppingCart.GetCart(this.HttpContext);

    ViewData["CartCount"] = cart.GetCount();
```

```
    return PartialView("CartSummary");
}
```

To create the CartSummary partial view, right-click on the Views/ShoppingCart folder and select Add View. Name the view CartSummary and check the "Create a partial view" checkbox as shown below.



The CartSummary partial view is really simple - it's just a link to the ShoppingCart Index view which shows the number of items in the cart. The complete code for CartSummary.cshtml is as follows:

```
@Html.ActionLink("Cart (" + ViewData["CartCount"] + ")",
    "Index",
    "ShoppingCart",
    new { id = "cart-status" })
```

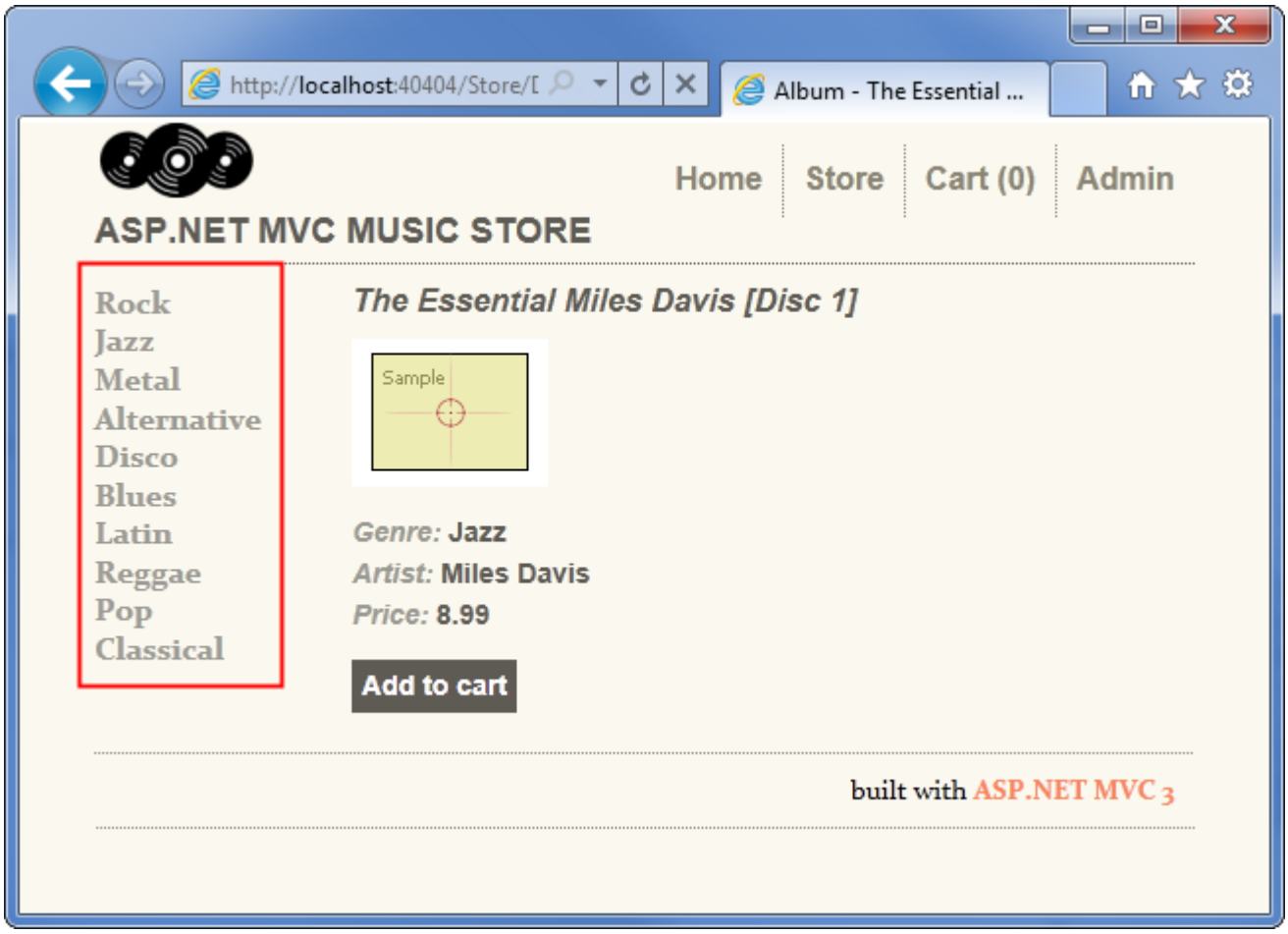
We can include a partial view in any page in the site, including the Site master, by using the Html.RenderAction method. RenderAction requires us to specify the Action Name ("CartSummary") and the Controller Name ("ShoppingCart") as below.

```
@Html.RenderAction("CartSummary", "ShoppingCart")
```

Before adding this to the site Layout, we will also create the Genre Menu so we can make all of our Site.master updates at one time.

Creating the Genre Menu Partial View

We can make it a lot easier for our users to navigate through the store by adding a Genre Menu which lists all the Genres available in our store.



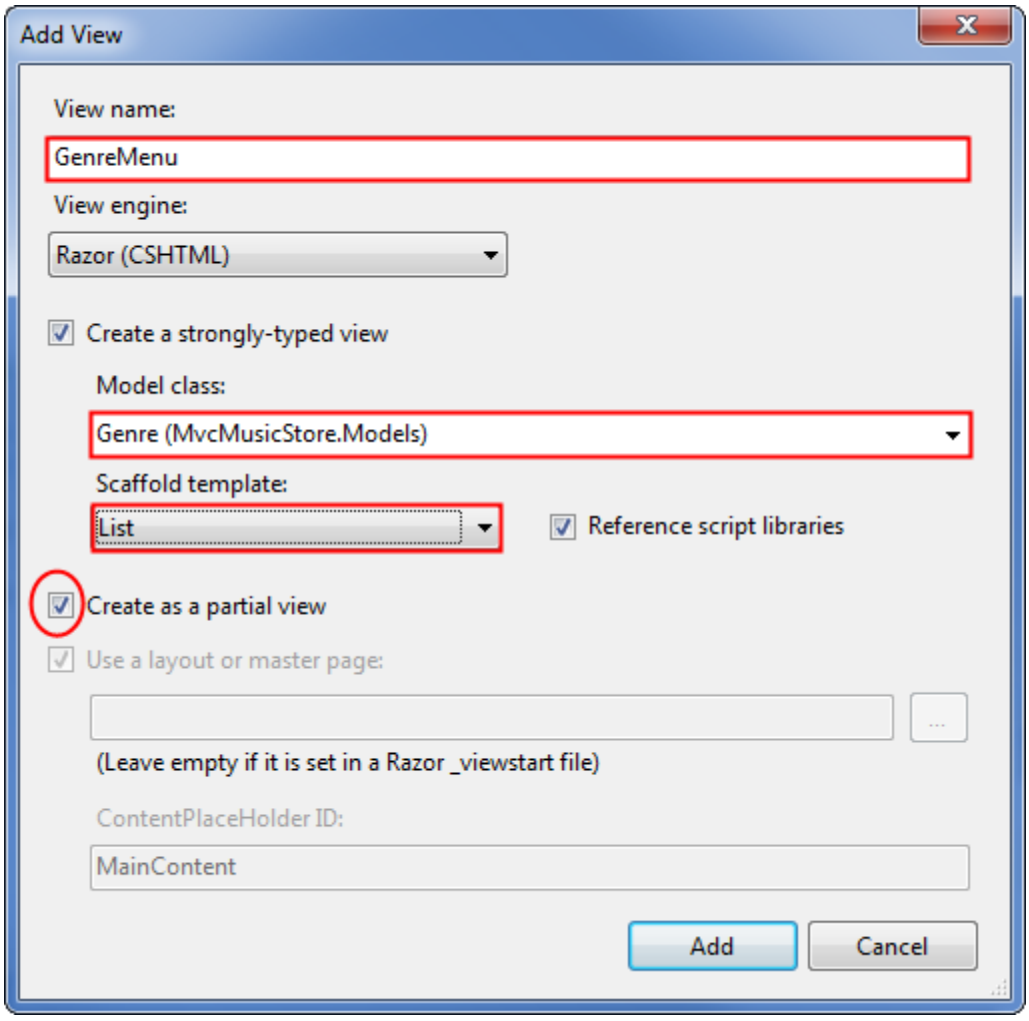
We will follow the same steps also create a GenreMenu partial view, and then we can add them both to the Site master. First, add the following GenreMenu controller action to the StoreController:

```
//
// GET: /Store/GenreMenu

[ChildActionOnly]
public ActionResult GenreMenu()
{
    var genres = storeDB.Genres.ToList();

    return PartialView(genres);
}
```

This action returns a list of Genres which will be displayed by the partial view, which we will create next.

Note: We have added the [ChildActionOnly] attribute to this controller action, which indicates that we only want this action to be used from a Partial View. This attribute will prevent the controller action from being executed by browsing to /Store/GenreMenu. This isn't required for partial views, but it is a good practice, since we want to make sure our controller actions are used as we intend. We are also returning PartialView rather than View, which lets the view engine know that it shouldn't use the Layout for this view, as it is being included in other views.

Right-click on the GenreMenu controller action and create a partial view named GenreMenu which is strongly typed using the Genre view data class as shown below.



Update the view code for the GenreMenu partial view to display the items using an unordered list as follows.

```
@model IEnumerable<MvcMusicStore.Models.Genre>

<ul id="categories">
```

```
    @foreach (var genre in Model)
    {
        <li>@Html.ActionLink(genre.Name,
                "Browse", "Store",
                new { Genre = genre.Name }, null)
        </li>
    }
</ul>
```

Updating Site Layout to display our Partial Views

We can add our partial views to the Site Layout by calling Html.RenderAction(). We'll add them both in, as well as some additional markup to display them, as shown below:

```
<!DOCTYPE html>
<html>
<head>
    <title>@ViewBag.Title</title>
    <link href="@Url.Content("~/Content/Site.css")" rel="stylesheet"
        type="text/css" />
    <script src="@Url.Content("~/Scripts/jquery-1.4.4.min.js")"
        type="text/javascript"></script>
</head>
<body>
    <div id="header">
        <h1><a href="/">ASP.NET MVC MUSIC STORE</a></h1>
        <ul id="navlist">
            <li class="first"><a href="@Url.Content("~")" id="current">Home</a></li>
            <li><a href="@Url.Content("~/Store/")">Store</a></li>
            <li>@{Html.RenderAction("CartSummary", "ShoppingCart");}</li>
            <li><a href="@Url.Content("~/StoreManager/")">Admin</a></li>
        </ul>
    </div>

    @{Html.RenderAction("GenreMenu", "Store");}

    <div id="main">
        @RenderBody()
    </div>

    <div id="footer">
        built with <a href="http://asp.net/mvc">ASP.NET MVC 3</a>
    </div>
</body>
</html>
```

Now when we run the application, we will see the Genre in the left navigation area and the Cart Summary at the top.

Update to the Store Browse page

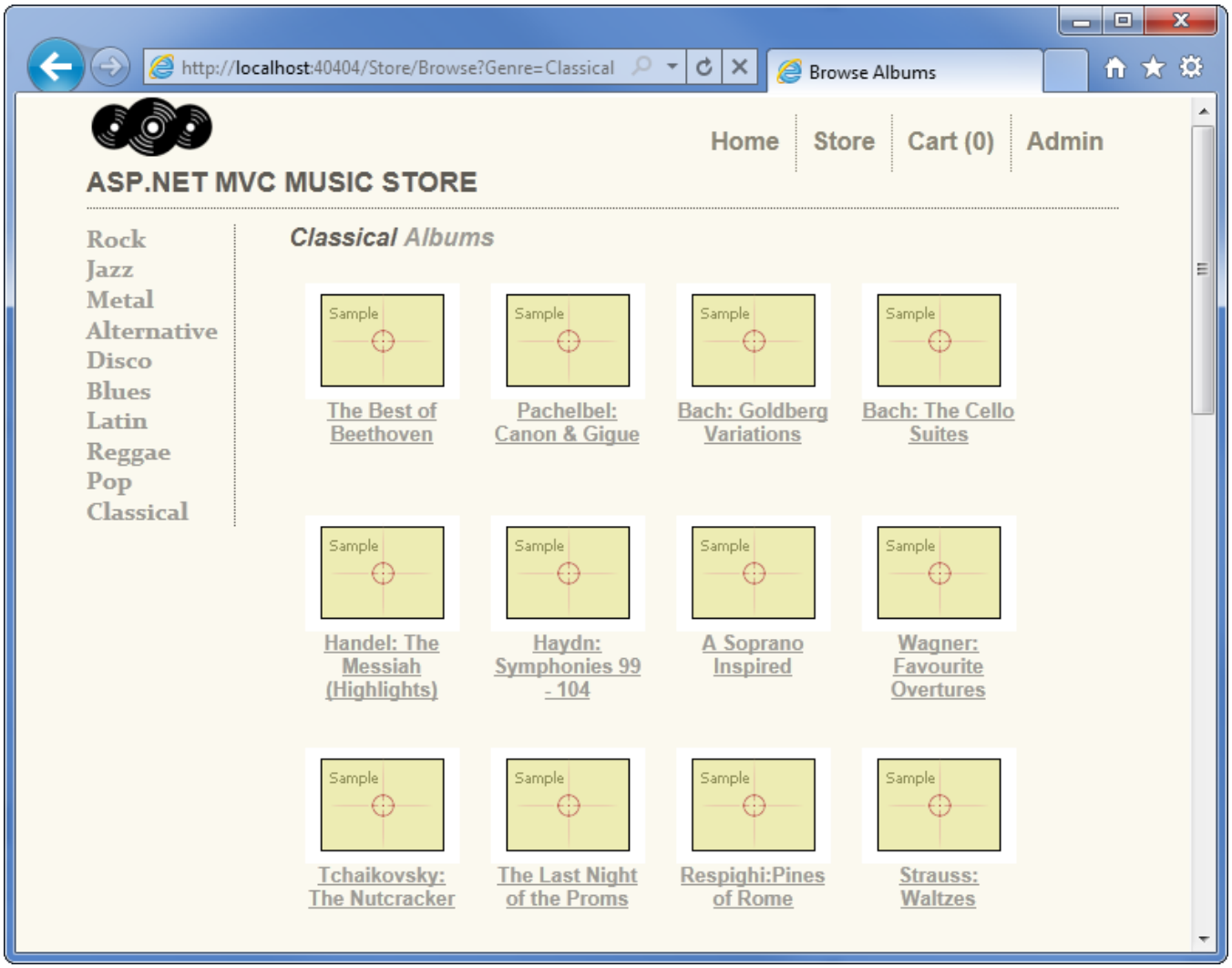
The Store Browse page is functional, but doesn't look very good. We can update the page to show the albums in a better layout by updating the view code (found in /Views/Store/Browse.cshtml) as follows:

```
@model MvcMusicStore.Models.Genre

@{
    ViewBag.Title = "Browse Albums";
}

<div class="genre">
    <h3><em>@Model.Name</em> Albums</h3>

    <ul id="album-list">
        @foreach (var album in Model.Albums)
        {
            <li>
                <a href="@Url.Action("Details", new { id = album.AlbumId })">
                    <img alt="@album.Title" src="@album.AlbumArtUrl" />
                    <span>@album.Title</span>
                </a>
            </li>
        }
    </ul>
</div>
```

Here we are making use of Url.Action rather than Html.ActionLink so that we can apply special formatting to the link to include the album artwork.

Note: We are displaying a generic album cover for these albums. This information is stored in the database and is editable via the Store Manager. You are welcome to add your own artwork.

Now when we browse to a Genre, we will see the albums shown in a grid with the album artwork.



Updating the Home Page to show Top Selling Albums

We want to feature our top selling albums on the home page to increase sales. We'll make some updates to our HomeController to handle that, and add in some additional graphics as well.

First, we'll add a navigation property to our Album class so that EntityFramework knows that they're associated. The last few lines of our Album class should now look like this:

```
        public virtual Genre  Genre                    { get; set; }
        public virtual Artist Artist                   { get; set; }
        public virtual List<OrderDetail> OrderDetails  { get; set; }
    }
}
```

First, we'll add a storeDB field and the MvcMusicStore.Models using statements, as in our other controllers. Next, we'll add the following method to the HomeController which queries our database to find top selling albums according to OrderDetails.

```
private List<Album> GetTopSellingAlbums(int count)
{
    // Group the order details by album and return
    // the albums with the highest count

    return storeDB.Albums
        .OrderByDescending(a => a.OrderDetails.Count())
        .Take(count)
        .ToList();
}
```

This is a private method, since we don't want to make it available as a controller action. We are including it in the HomeController for simplicity, but you are encouraged to move your business logic into separate service classes as appropriate.

With that in place, we can update the Index controller action to query the top 5 selling albums and return them to the view.

```
public ActionResult Index()
{
    // Get most popular albums
    var albums = GetTopSellingAlbums(5);

    return View(albums);
}
```

The complete code for the updated HomeController is as shown below.

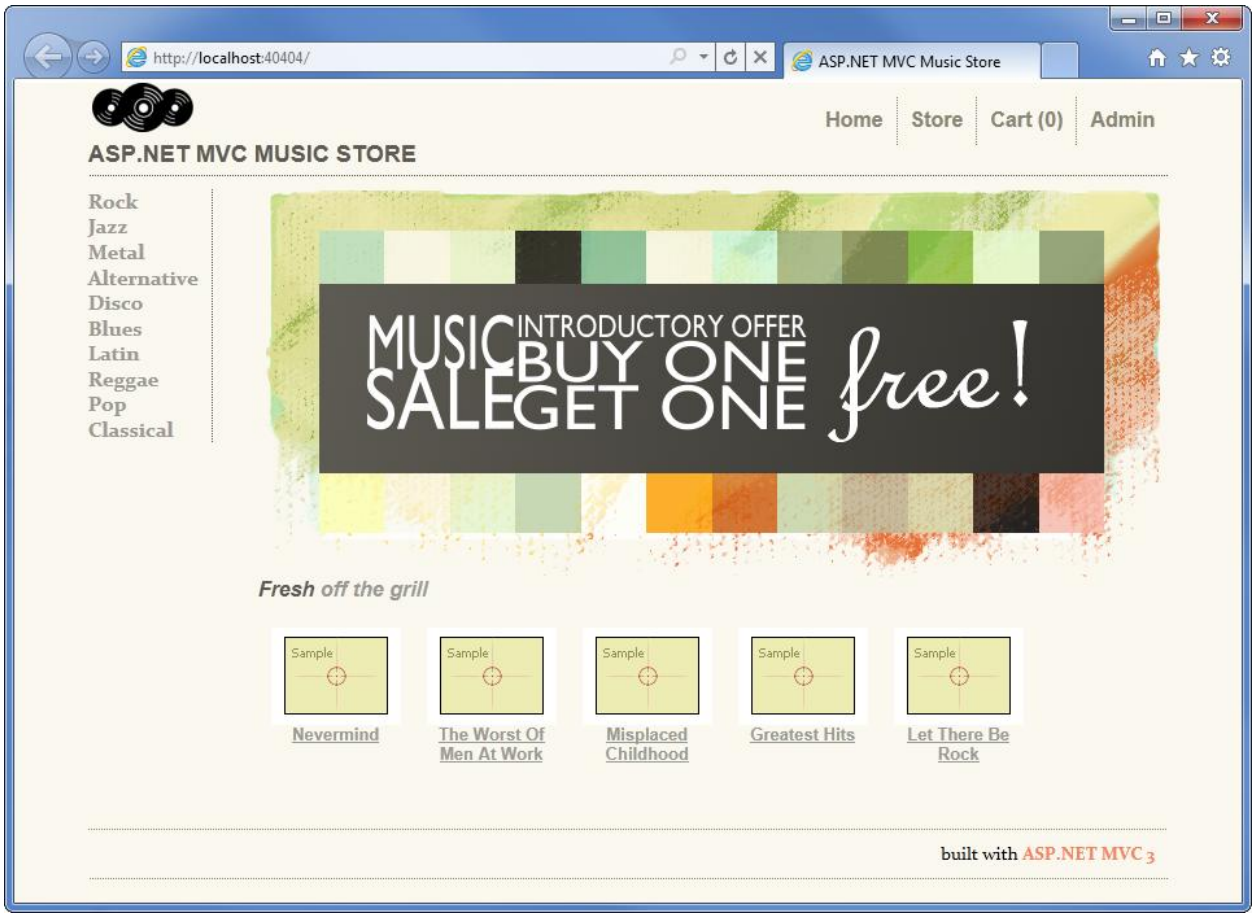
```
using System.Collections.Generic;
using System.Linq;
using System.Web.Mvc;
using MvcMusicStore.Models;

namespace MvcMusicStore.Controllers
{
    public class HomeController : Controller
    {
        //
        // GET: /Home/

        MusicStoreEntities storeDB = new MusicStoreEntities();

        public ActionResult Index()
        {
            // Get most popular albums
            var albums = GetTopSellingAlbums(5);

            return View(albums);
        }

        private List<Album> GetTopSellingAlbums(int count)
        {
            // Group the order details by album and return
            // the albums with the highest count
```

```
            return storeDB.Albums
                .OrderByDescending(a => a.OrderDetails.Count())
                .Take(count)
                .ToList();
        }
    }
}
```

Finally, we'll need to update our Home Index view so that it can display a list of albums by updating the Model type and adding the album list to the bottom. We will take this opportunity to also add a heading and a promotion section to the page.

```
@model List<MvcMusicStore.Models.Album>
@{
    ViewBag.Title = "ASP.NET MVC Music Store";
}
<div id="promotion">
</div>

<h3><em>Fresh</em> off the grill</h3>

<ul id="album-list">
    @foreach (var album in Model)
    {
        <li><a href="@Url.Action("Details", "Store",
                new { id = album.AlbumId })">

            <img alt="@album.Title" src="@album.AlbumArtUrl" />
            <span>@album.Title</span> </a>
        </li>
    }
</ul>
```

Now when we run the application, we'll see our updated home page with top selling albums and our promotional message.



Conclusion

We've seen that that ASP.NET MVC makes it easy to create a sophisticated website with database access, membership, AJAX, etc. pretty quickly. Hopefully this tutorial has given you the tools you need to get started building your own ASP.NET MVC applications!