ASP.NET MVC Music Store Tutorial

Version 3.0b

Jon Galloway - Microsoft

4/28/2011

http://mvcmusicstore.codeplex.com - Licensed under Creative Commons Attribution 3.0 License.

ASP.NET MVC Music Store Tutorial

Contents

Overview....4

1. File -> New Project....9

Installing the software....9

Creating a new ASP.NET MVC 3 project....11

2. Controllers....15

Adding a HomeController....15

Running the Application....17

Adding a StoreController....19

3. Views and Models....24

Adding a View template....24

Using a Layout for common site elements....27

Updating the StyleSheet....29

Using a Model to pass information to our View....31

Adding Links between pages....41

4. Data Access....44

Database access with Entity Framework Code-First....44

Changes to our Model Classes....44

Adding the App_Data folder....45

Creating a Connection String in the web.config file....46

Adding a Context Class....46

Adding our store catalog data....47

Querying the Database....48

Updating the Store Index to query the database....49

Updating Store Browse and Details to use live data....49

5. Edit Forms using Scaffolding....54

Creating the StoreManagerController....54

Modifying a Scaffolded View....55

A first look at the Store Manager....57

Tutorial under Creative Commons Attribution 3.0 License

Looking at the Store Manager Controller code 61
Store Manager Index and Details actions 62
The Create Action Methods 62
Passing information to a View using ViewBag 62
HTML Helpers to display the Drop Downs in the Create View 63
Handling the Posted Form values 64
Handling Edits 66
Handling Deletion 68
Using a custom HTML Helper to truncate text 72
6. Using Data Annotations for Model Validation 76
Adding Validation to our Album Forms 76
Testing the Client-Side Validation 79
7. Membership and Authorization 81
Adding the AccountController and Views 81
Adding an Administrative User with the ASP.NET Configuration site 82
Role-based Authorization 87
8. Shopping Cart with Ajax Updates 89
Adding the Cart, Order, and OrderDetail model classes 89
Managing the Shopping Cart business logic 91
ViewModels 95
The Shopping Cart Controller 97
Ajax Updates with jQuery 99
9. Registration and Checkout 109
Migrating the Shopping Cart 113
Creating the CheckoutController 114
Adding the AddressAndPayment view 119
Defining validation rules for the Order 121
Adding the Checkout Complete view 123
Updating The Error view 124
10. Final updates to Navigation and Site Design 126
Creating the Shopping Cart Summary Partial View 126
Creating the Genre Menu Partial View 128

Tutorial under Creative Commons Attribution 3.0 License

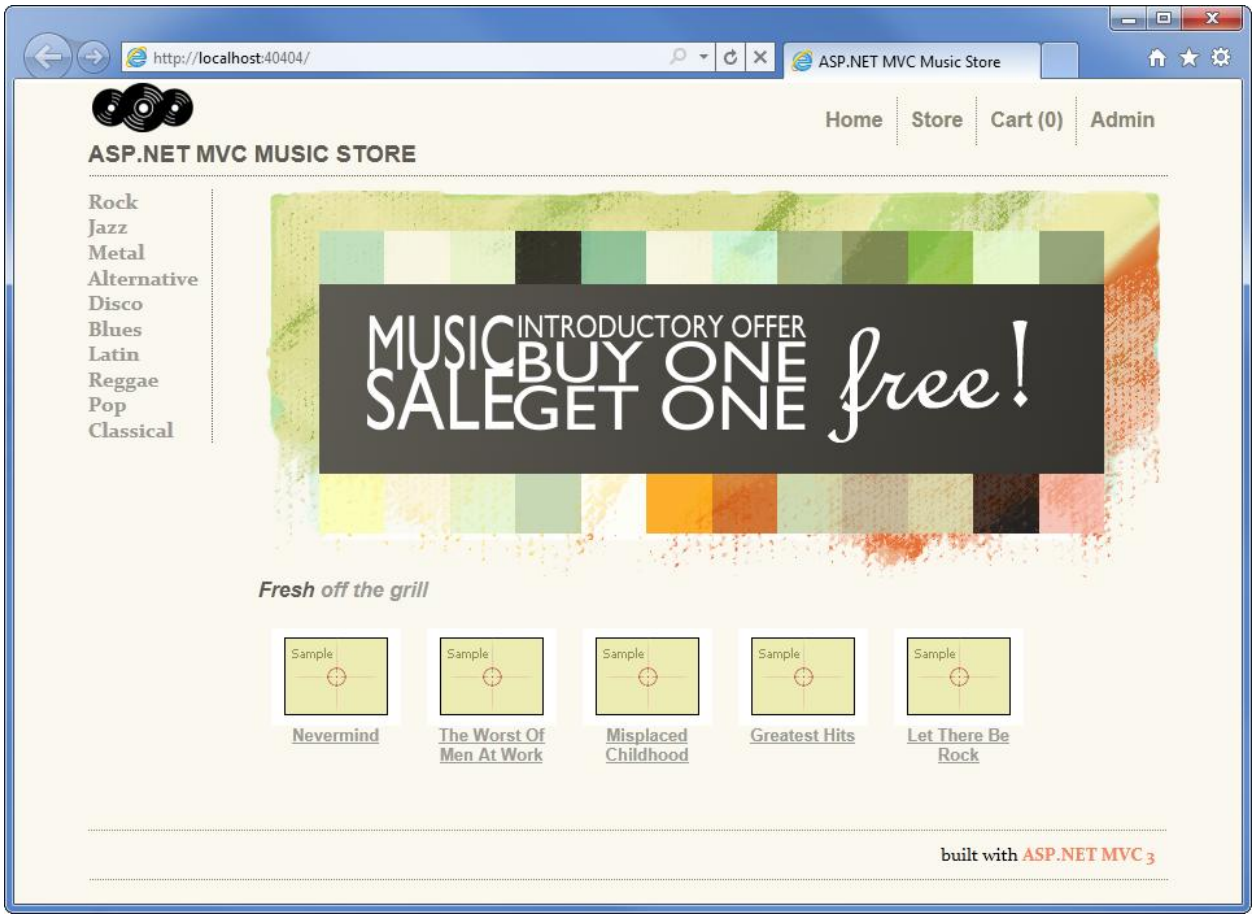
Updating Site Layout to display our Partial Views 130
Update to the Store Browse page 130
Updating the Home Page to show Top Selling Albums 132
Conclusion 135

Tutorial under Creative Commons Attribution 3.0 License

Overview

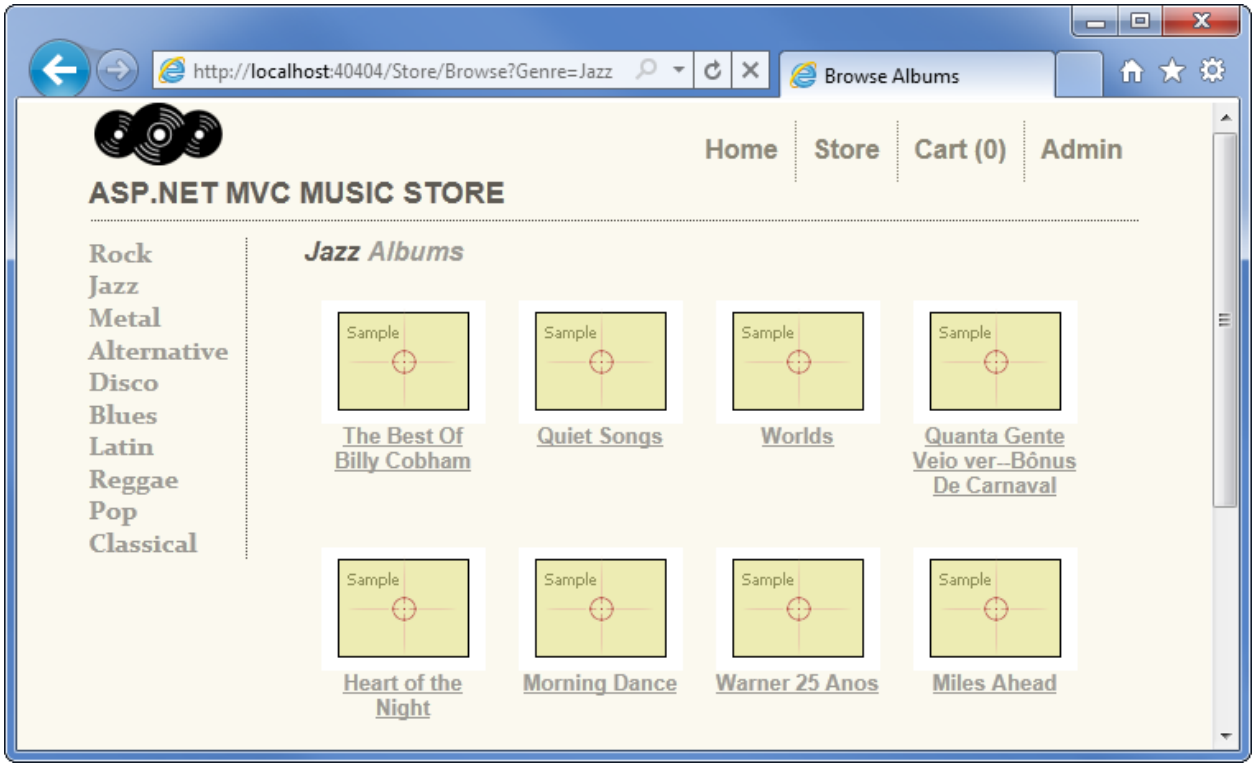
The MVC Music Store is a tutorial application that introduces and explains step-by-step how to use ASP.NET MVC and Visual Web Developer for web development. We'll be starting slowly, so beginner level web development experience is okay.

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

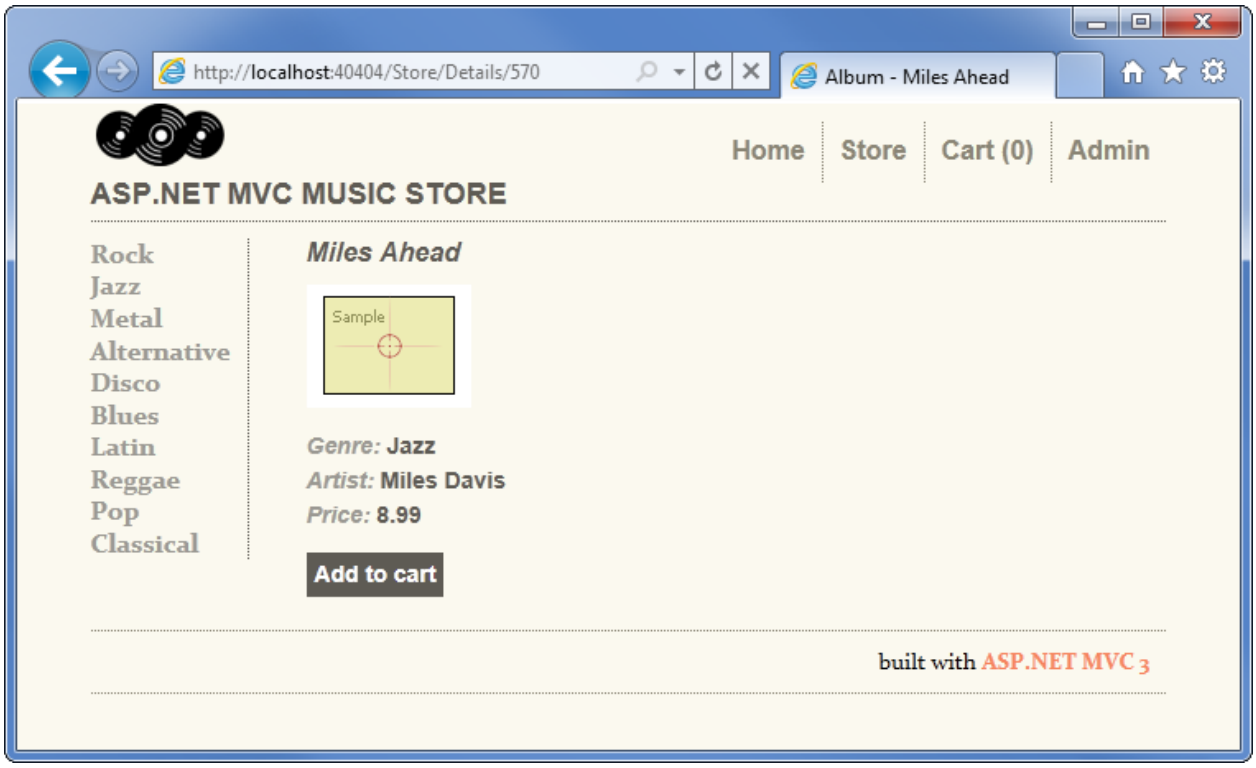


Tutorial under Creative Commons Attribution 3.0 License

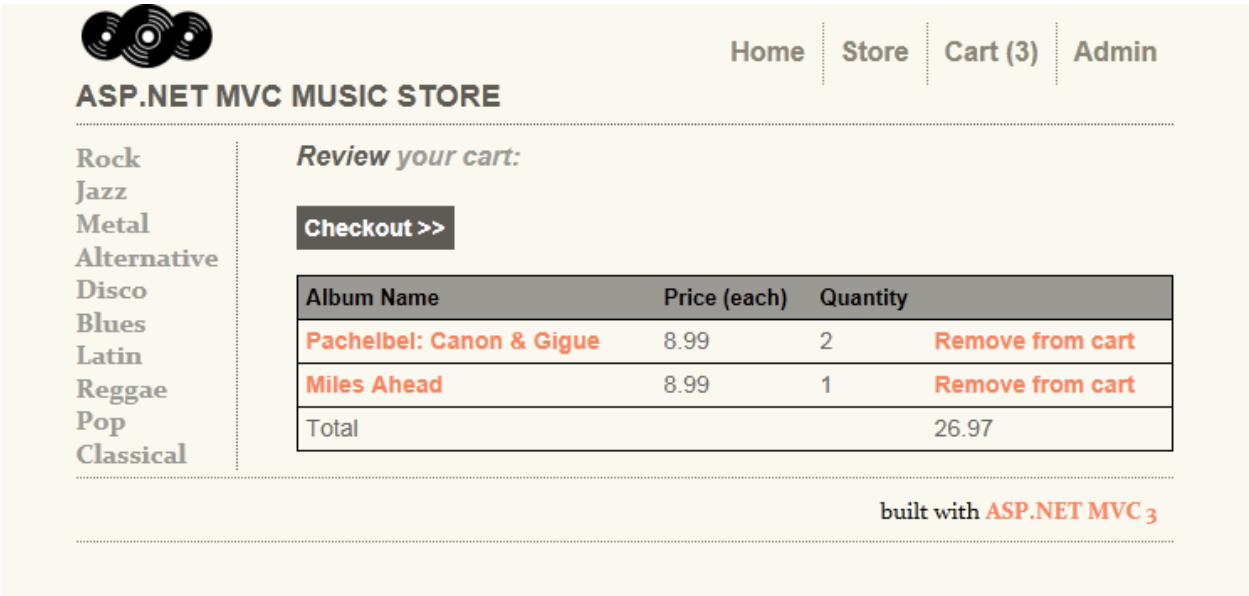
Visitors can browse Albums by Genre:



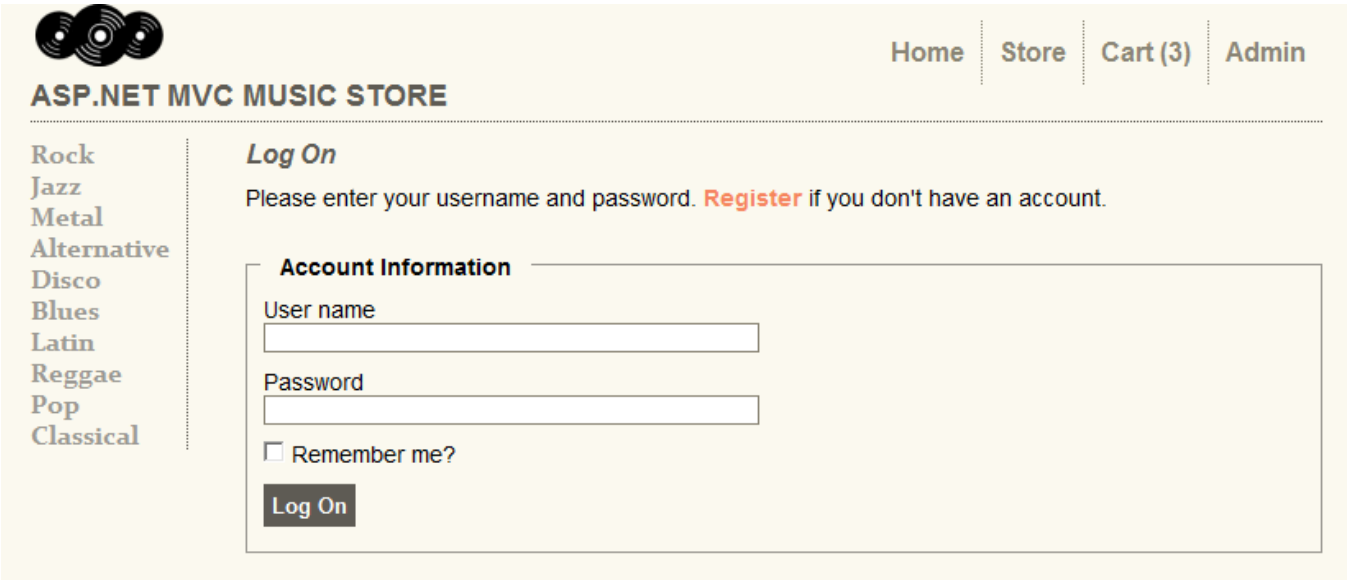
They can view a single album and add it to their cart:

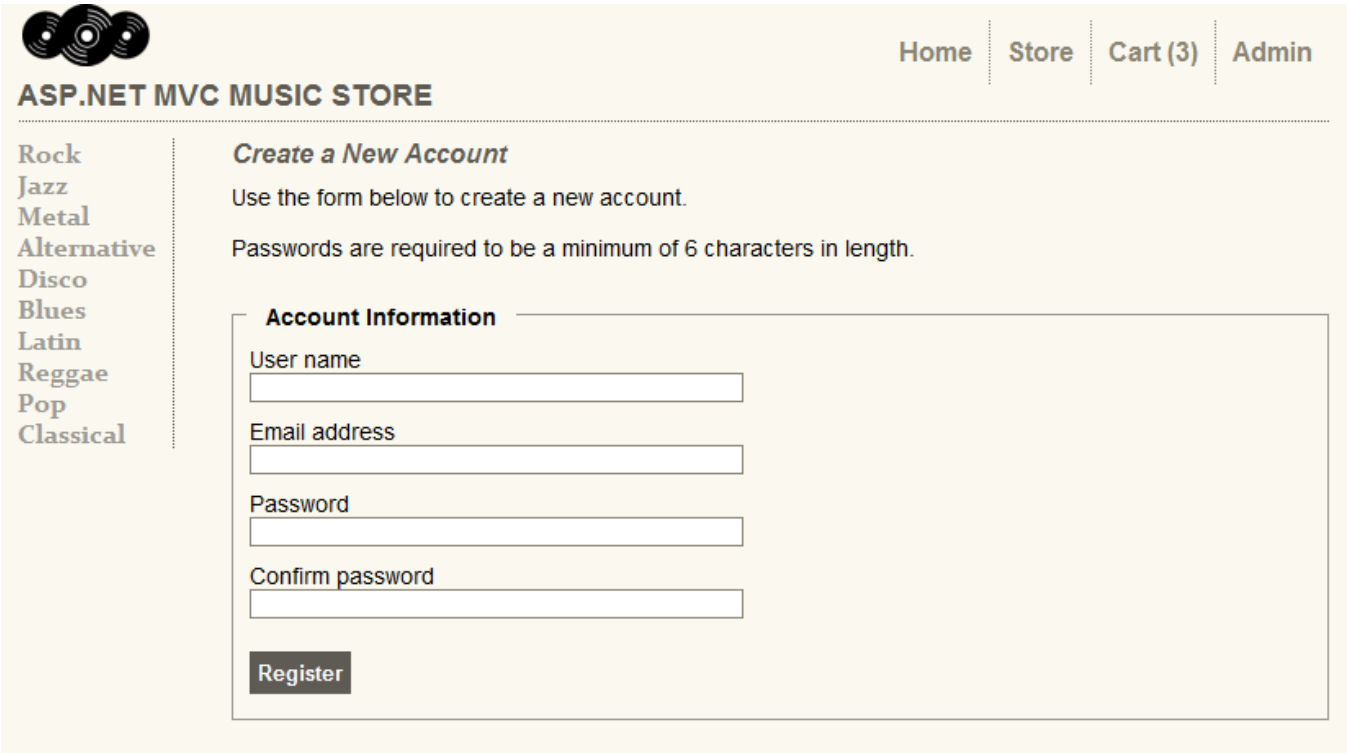


They can review their cart, removing any items they no longer want:



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"!

Rock
Jazz
Metal
Alternative
Disco
Blues
Latin
Reggae
Pop
Classical

Shipping Information

First Name	Jon
Last Name	Galloway
Address	123 Main
City	Denver
State	CO
Postal Code	12345
Country	USA
Phone	(123) 456-7890
Email Address	test@test.com

Payment

We're running a promotion: all music is free with the promo code "FREE"

Promo Code	FREE

Submit Order

After ordering, they see a simple confirmation screen:

Home | Store | Cart (0) | Admin

ASP.NET MVC MUSIC STORE

Rock
Jazz
Metal
Alternative
Disco
Blues
Latin
Reggae
Pop
Classical

Checkout Complete

Thanks for your order! Your order number is: 475

How about shopping for some more music in our store

built with ASP.NET MVC 3

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:

Albums

Create New Album

	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

1. File -> New Project

Installing the software

This tutorial will begin by creating a new ASP.NET MVC 3 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.

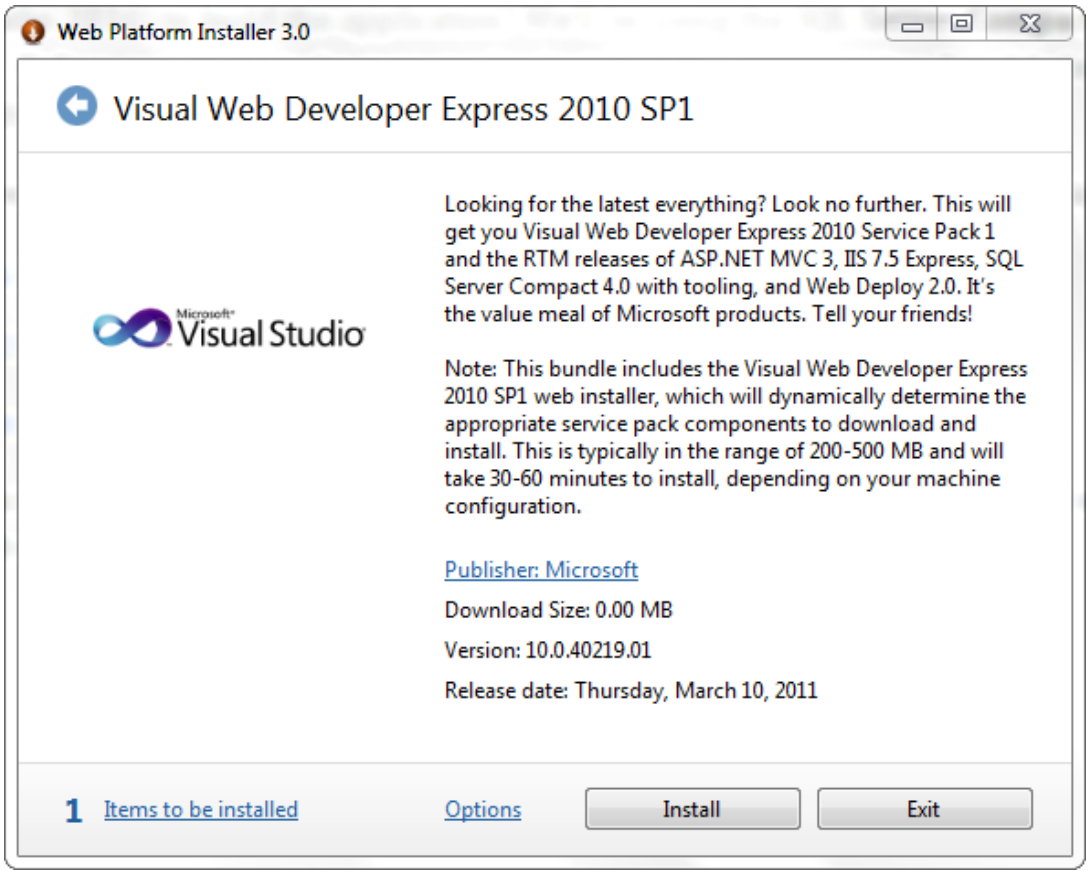
You can follow along step by step, or you can download the completed application from http://mvcmusicstore.codeplex.com.

You can use either Visual Studio 2010 SP1 or Visual Web Developer 2010 Express SP1 (a free version of Visual Studio 2010) to build the application. We'll be using the SQL Server Compact (also free) to host the database. Before you start, make sure you've installed the prerequisites listed below. You can install all of them using the following Web Platform Installer
link: http://www.microsoft.com/web/gallery/install.aspx?appid=VWD2010SP1Pack

Note: You can find this link on the big green button at this (easier to remember) link: http://asp.net/mvc

Tutorial under Creative Commons Attribution 3.0 License

The Web Platform Installer will check what you've got installed and just download what you need.



If you want to individually install the prerequisites using the following links instead of using the above link, use the following links (written out in case you're using a printed version of this tutorial):

- Visual Studio Web Developer Express SP1 prerequisites
  http://www.microsoft.com/web/gallery/install.aspx?appid=VWD2010SP1Pack
- ASP.NET MVC 3 Tools Update
  http://www.microsoft.com/web/gallery/install.aspx?appid=MVC3
- SQL Server Compact 4.0 - including both runtime and tools support
  http://www.microsoft.com/web/gallery/install.aspx?appid=SQLCE;SQLCEVSTools_4_0

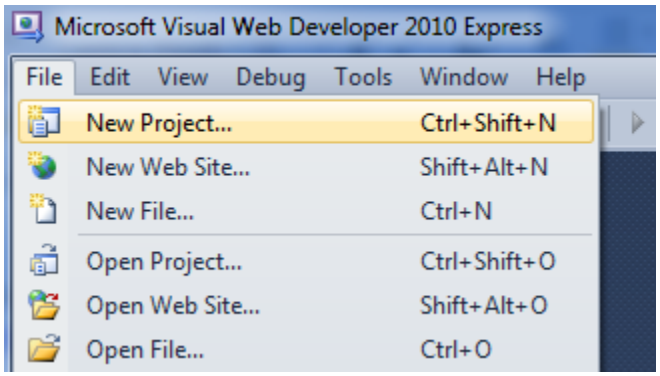
Note: If you're using Visual Studio 2010 instead of Visual Web Developer 2010, install the prerequisites with this link instead:

Visual Studio Web Developer Express SP1 prerequisites
http://www.microsoft.com/web/gallery/install.aspx?appsxml=&appid=VS2010SP1Pack

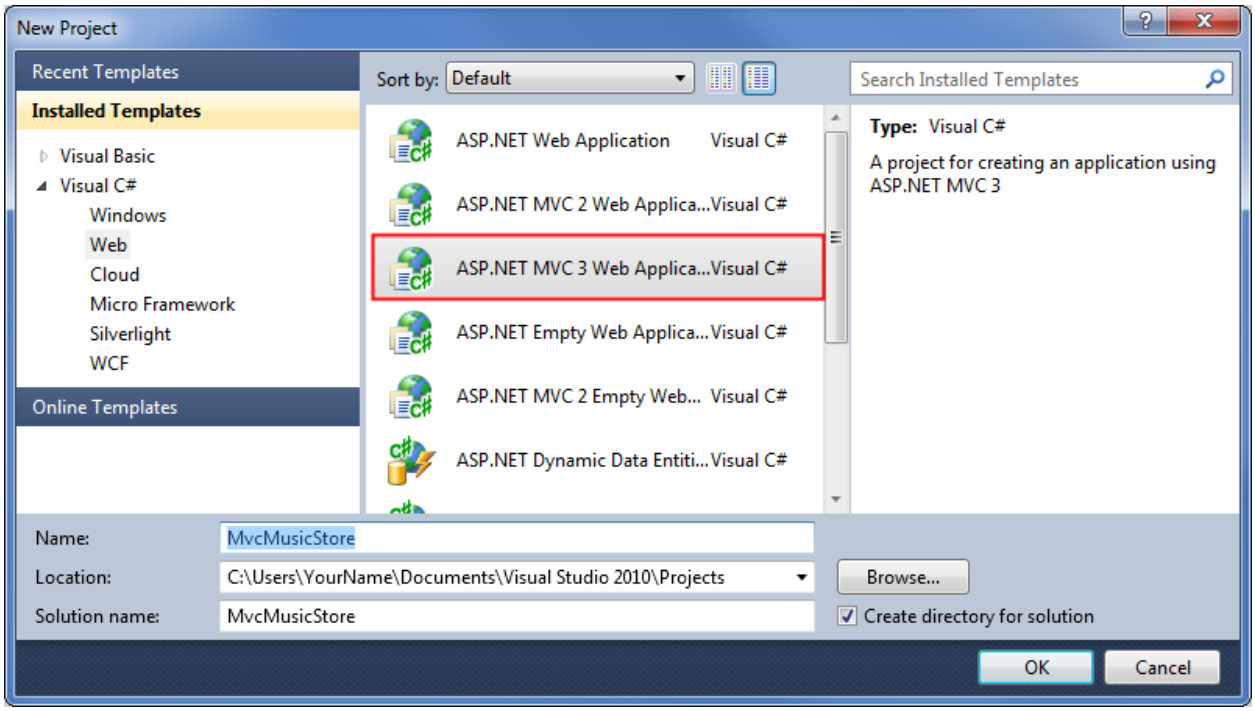
I highly recommend you use the first Web Platform Installer link, as it will make sure you've got everything set up correctly.

Creating a new ASP.NET MVC 3 project

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



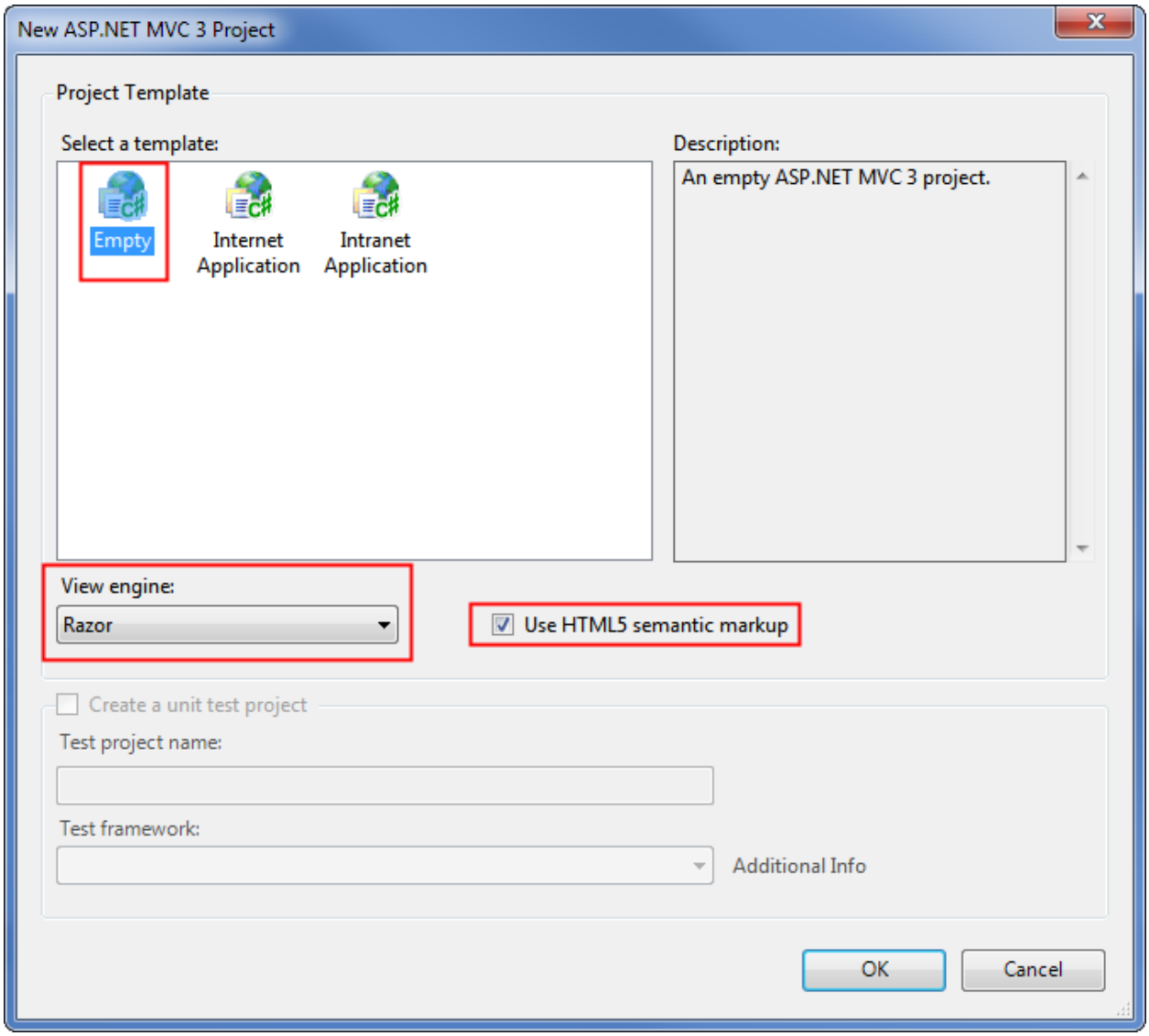
We'll select the Visual C# -> Web Templates group on the left, then choose the "ASP.NET MVC 3 Web Application" template in the center column. Name your project MvcMusicStore and press the OK button.



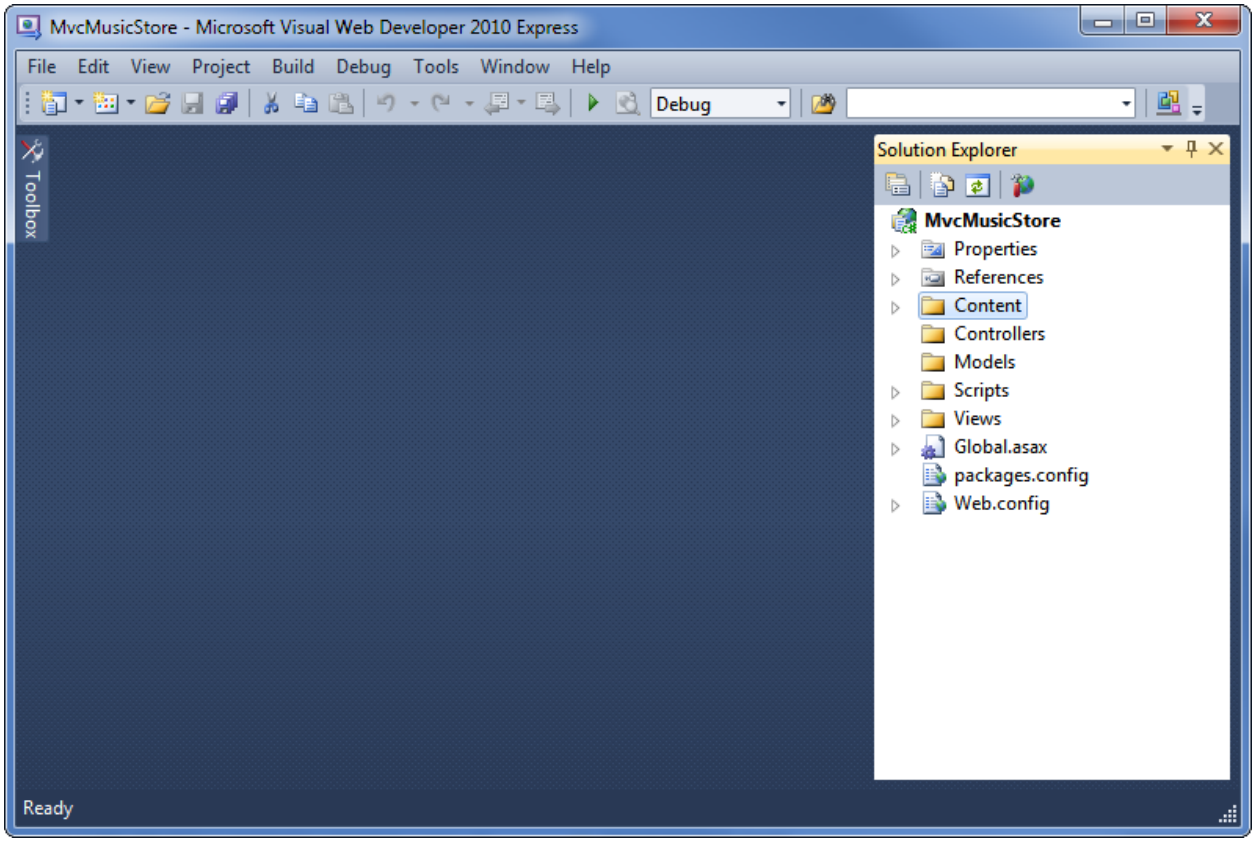
This will display a secondary dialog which allows us to make some MVC specific settings for our project. Select the following:

- Project Template - select Empty
- View Engine - select Razor
- Use HTML5 semantic markup - checked

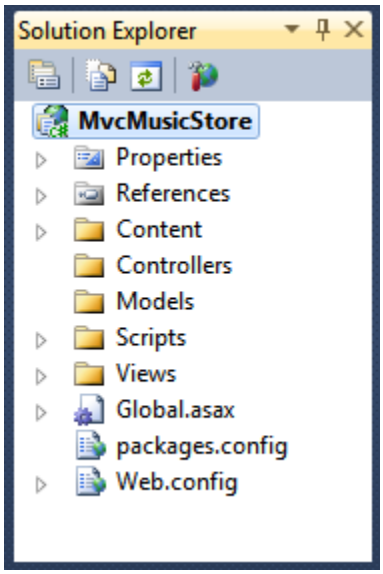
Verify that your settings are as shown below, then press the OK button.



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

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.

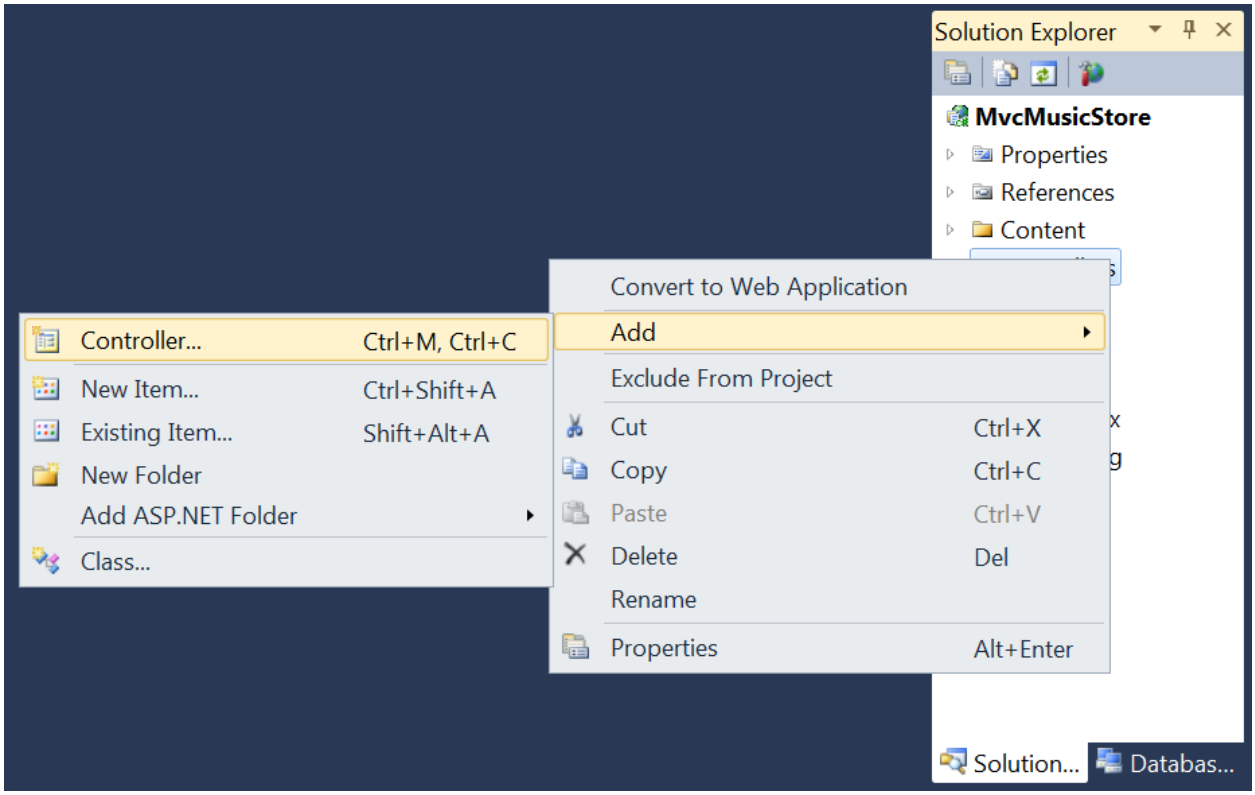
Web-based MVC frameworks map URLs to server code in a slightly different way. Instead of mapping incoming URLs to files, they instead map URLs to methods on classes. These classes are called "Controllers" and they are responsible for processing incoming HTTP requests, handling user input, retrieving 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

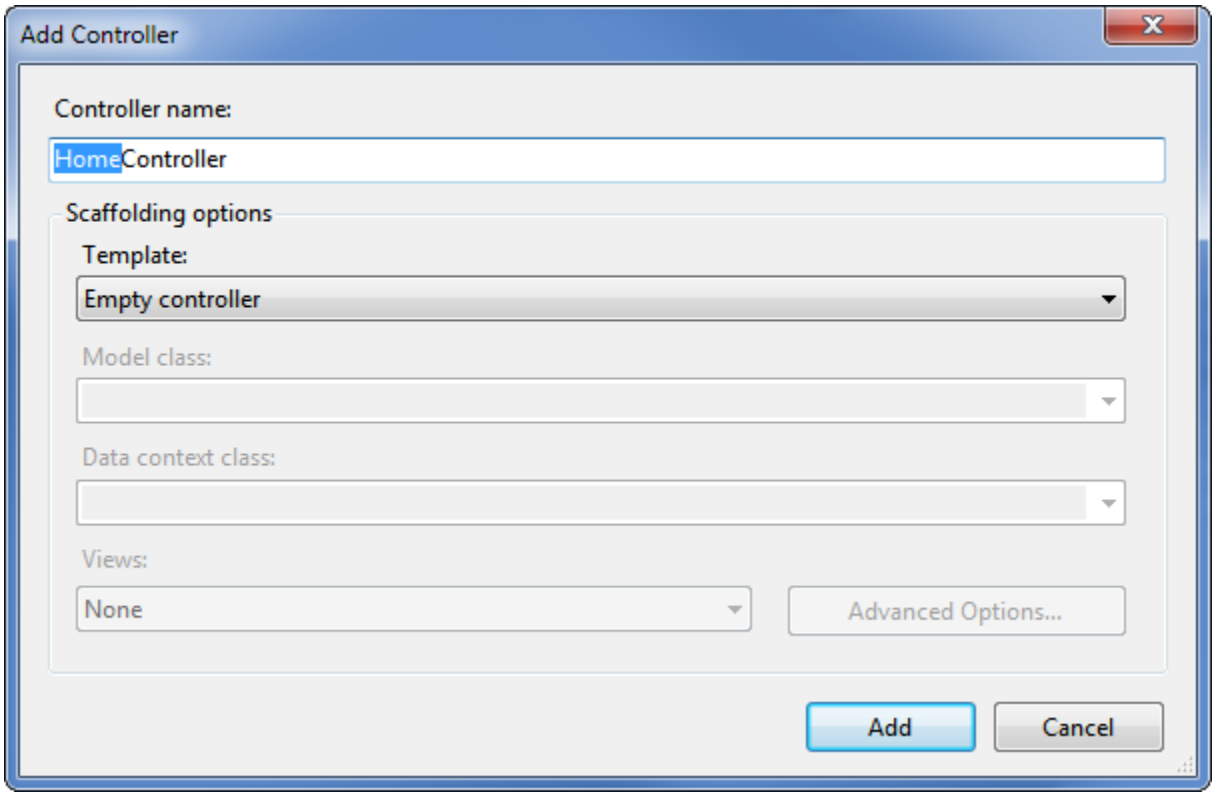
We’ll begin our MVC Music Store application by adding a Controller class that will handle URLs to the Home page of our site. We’ll follow the default naming conventions of ASP.NET MVC and call it HomeController.

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

Tutorial under Creative Commons Attribution 3.0 License



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 changes:

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

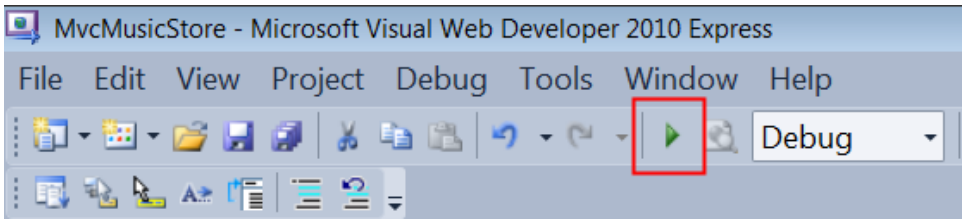
The method should now look like this:

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

Running the Application

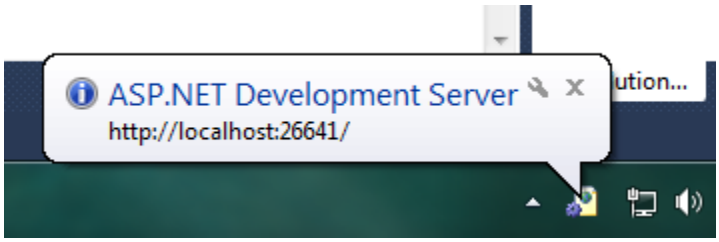
Now let's run the site. We can start our web-server and try out the site using any of the following::

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

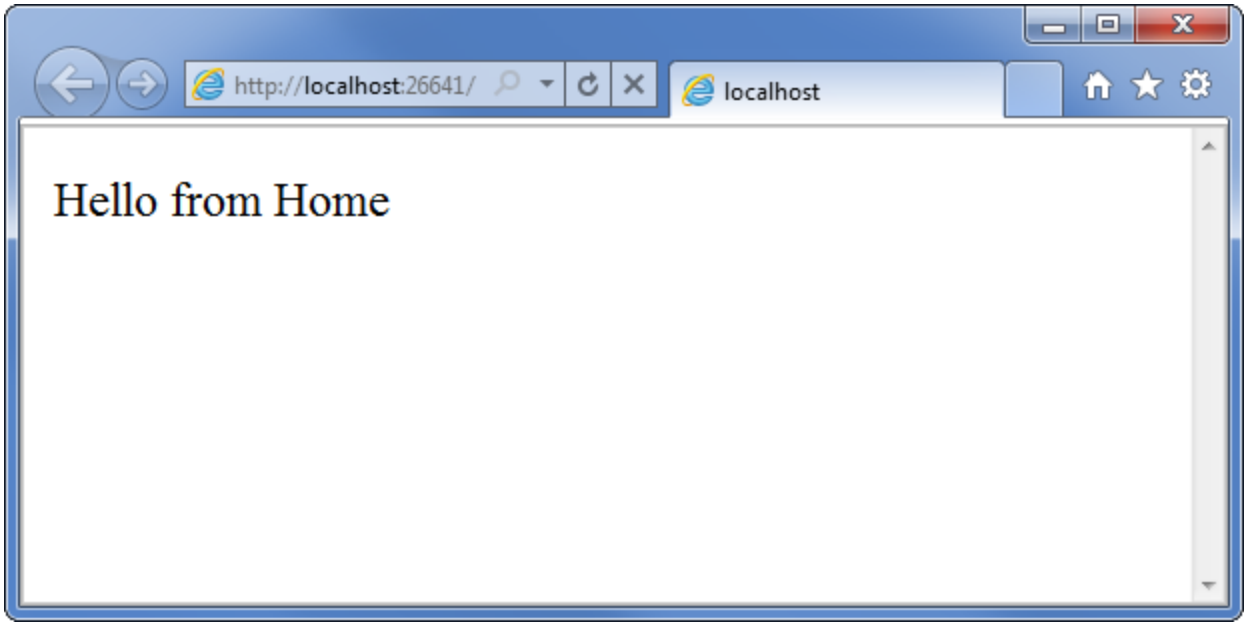


- Use the keyboard shortcut, F5.

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 Web Developer 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.

Tutorial under Creative Commons Attribution 3.0 License

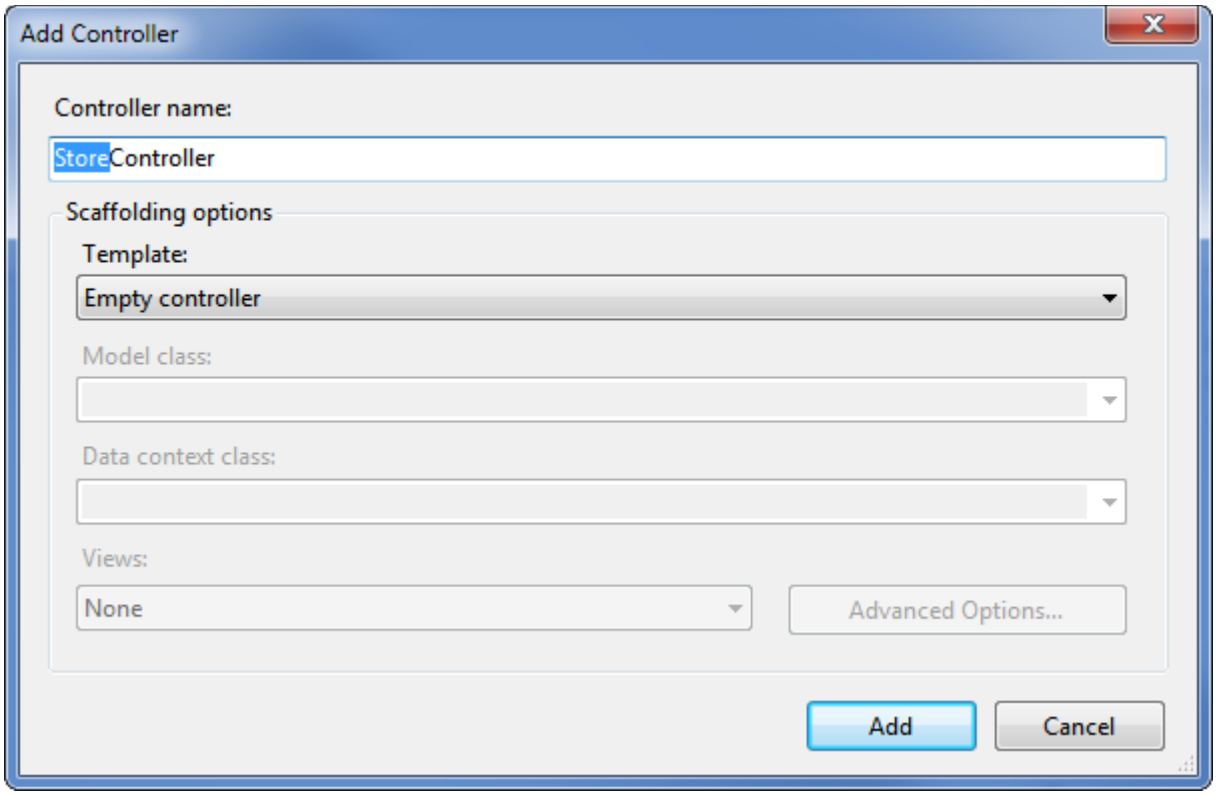
Adding a StoreController

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.

Tutorial under Creative Commons Attribution 3.0 License

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():

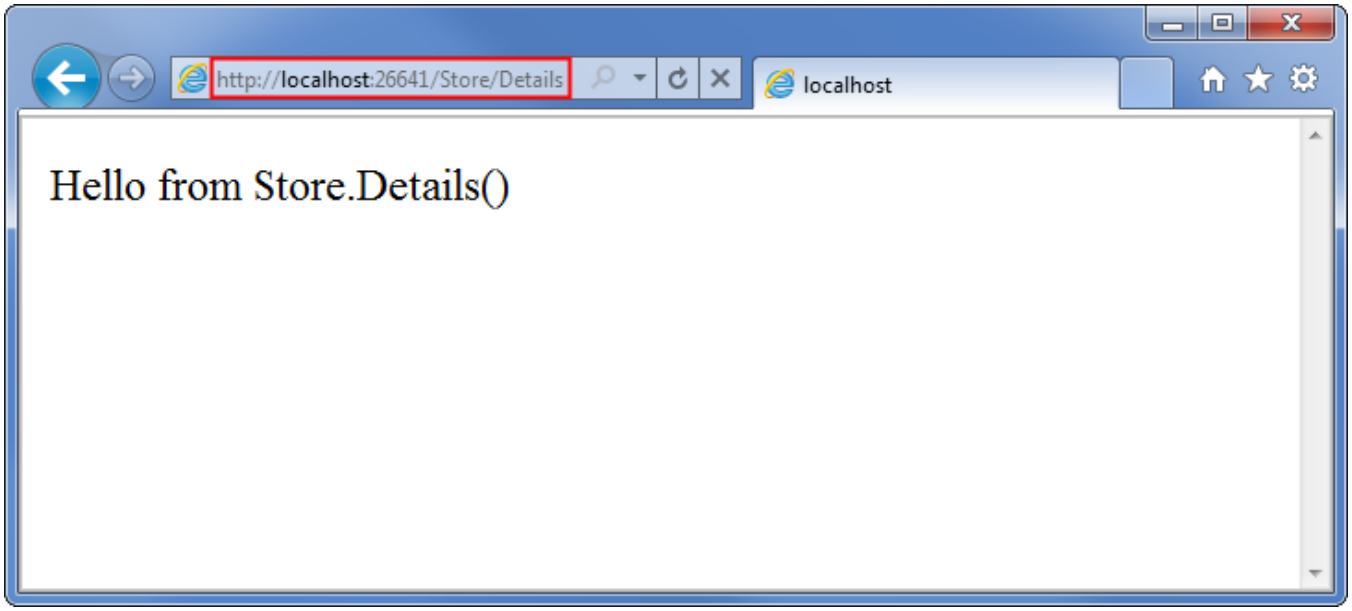
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

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:

Tutorial under Creative Commons Attribution 3.0 License



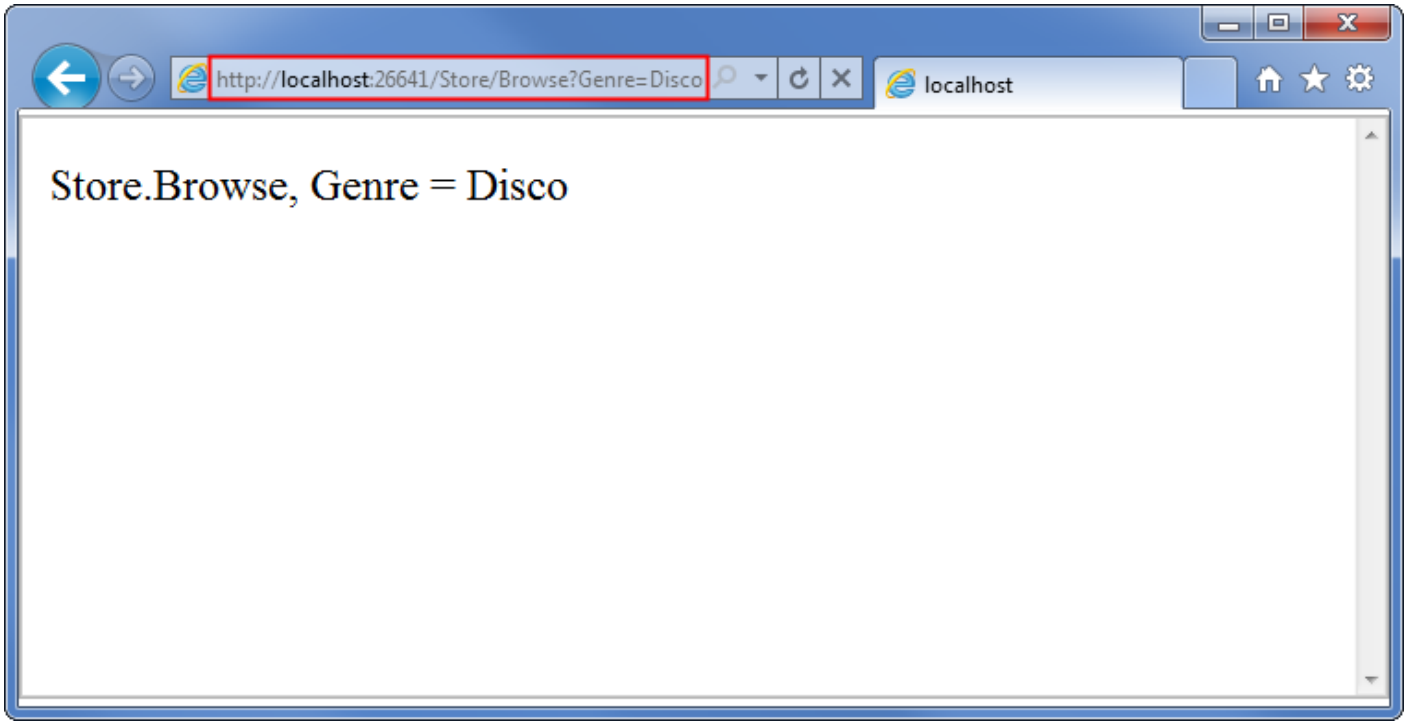
That's great, but these are just constant strings. Let's make them dynamic, so they take information from the URL and display it in the page output.

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

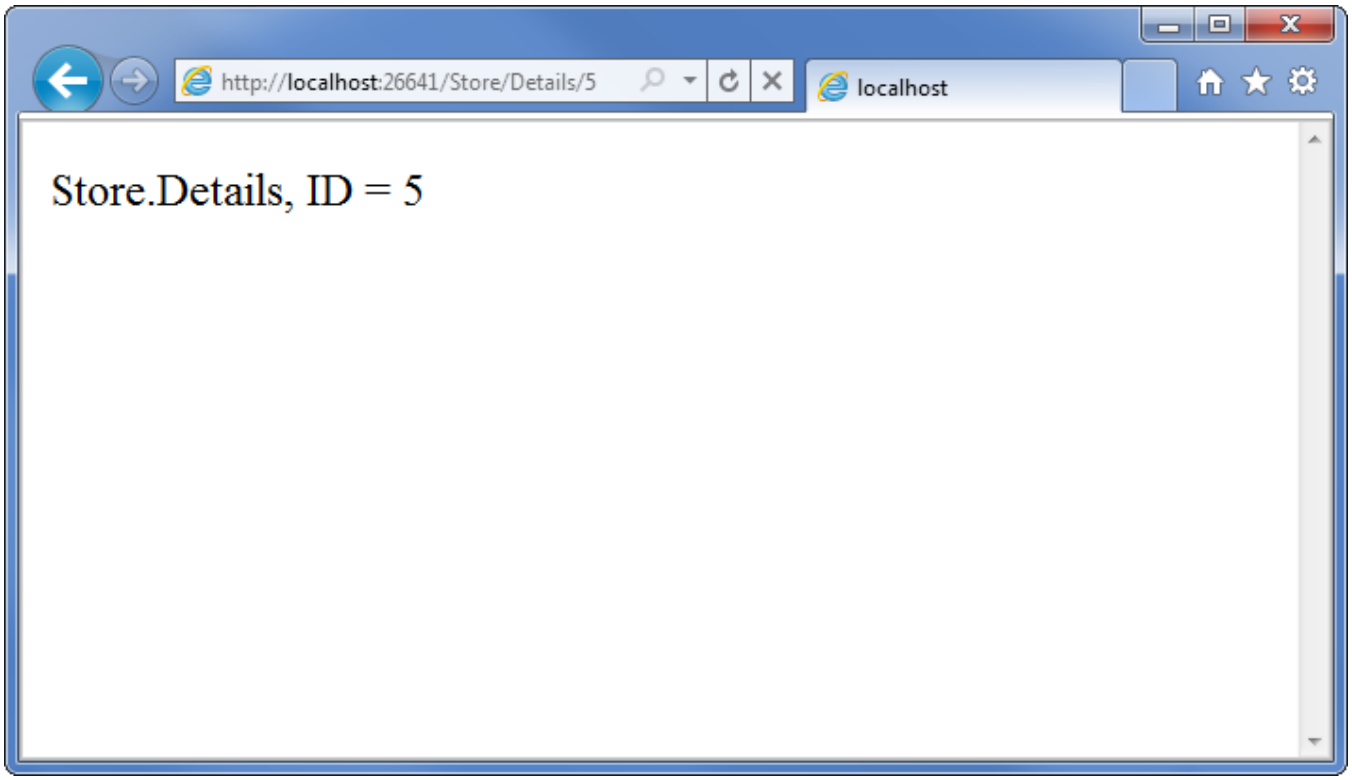


Let's next change the Details action to read and display an input parameter named ID. Unlike our previous method, we won't be embedding the ID value as a querystring parameter. Instead we'll embed it directly within the URL itself. For example: /Store/Details/5.

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 Web Developer
- 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.

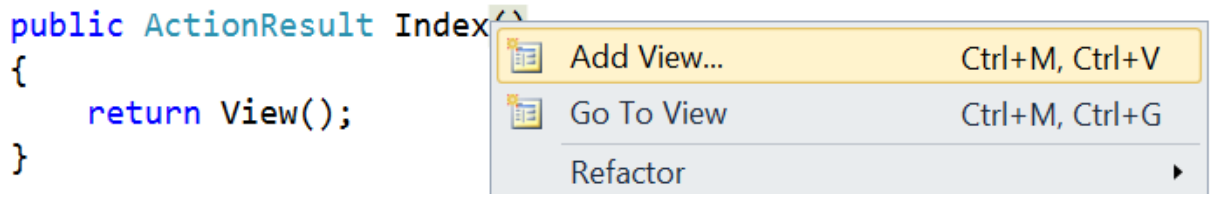
Adding a View template

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

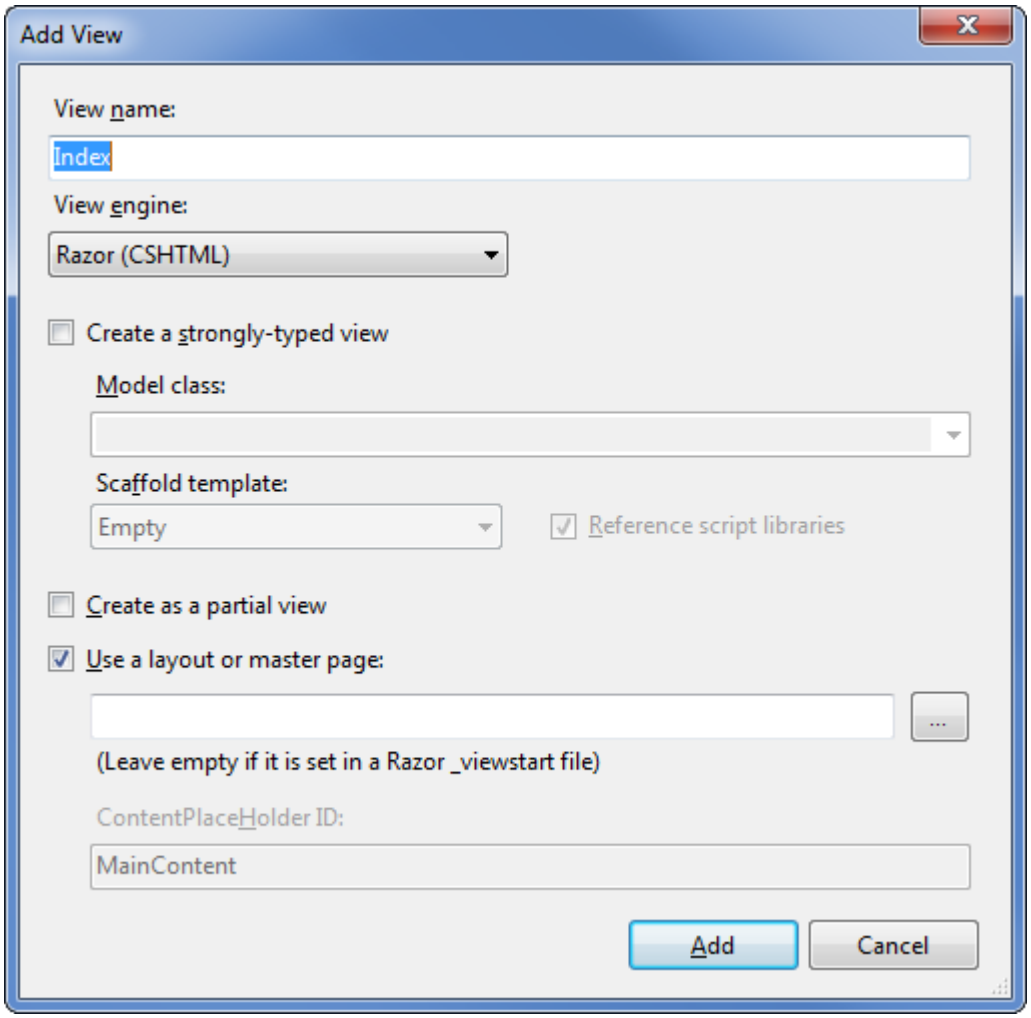
```
public class HomeController : Controller
{
    //
    // GET: /Home/

    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:



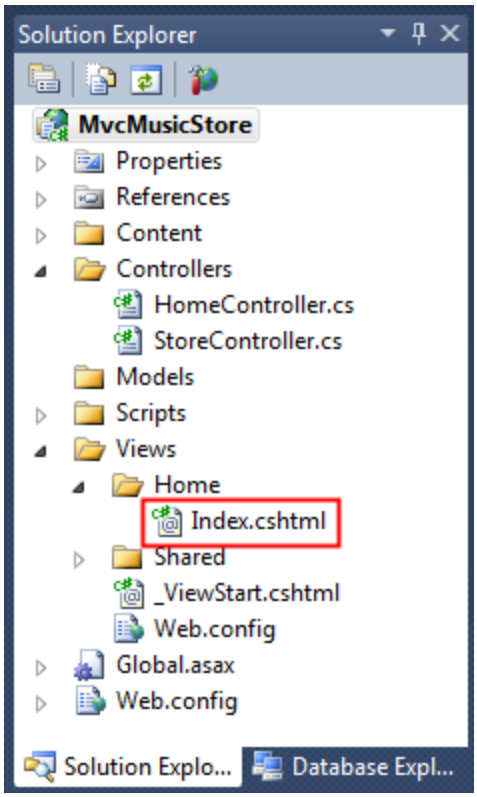
Tutorial under Creative Commons Attribution 3.0 License



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 Web Developer will create a new Index.cshtml view template for us in the \Views\Home directory, creating the folder if doesn't already exist.

Tutorial under Creative Commons Attribution 3.0 License



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 Web Developer 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.

```
@{
```

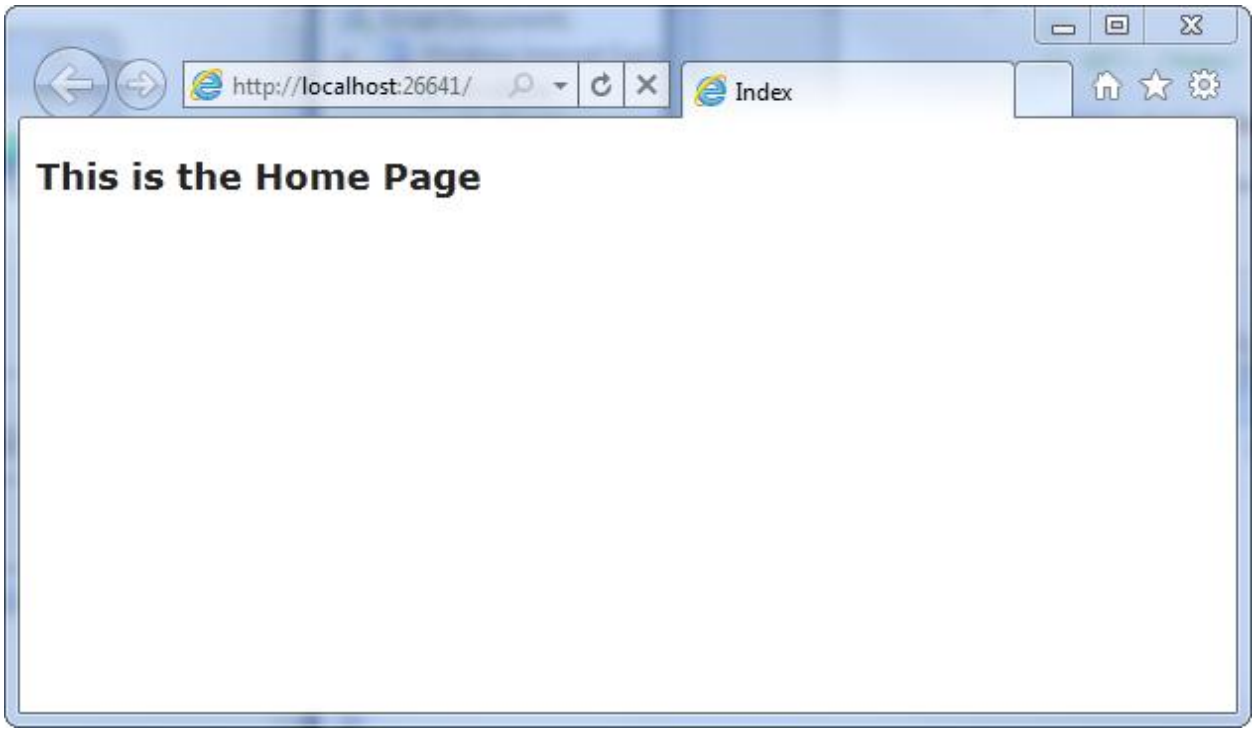
```
    ViewBag.Title = "Index";
}

<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.

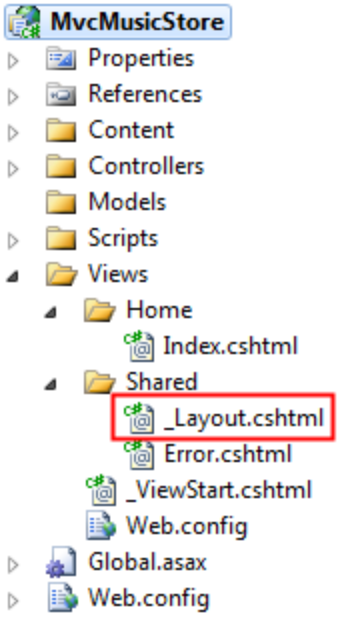
```
@{
    ViewBag.Title = "Index";
}

<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 which 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>
    <meta charset="utf-8" />
    <title>@ViewBag.Title</title>
    <link href="@Url.Content("~/Content/Site.css")" rel="stylesheet" type="text/css" />
    <script src="@Url.Content("~/Scripts/jquery-1.5.1.min.js")"
            type="text/javascript"></script>
    <script src="@Url.Content("~/Scripts/modernizr-1.7.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>
    <meta charset="utf-8" />
    <title>@ViewBag.Title</title>
    <link href="@Url.Content("~/Content/Site.css")" rel="stylesheet" type="text/css" />
    <script src="@Url.Content("~/Scripts/jquery-1.5.1.min.js")"
            type="text/javascript"></script>
```

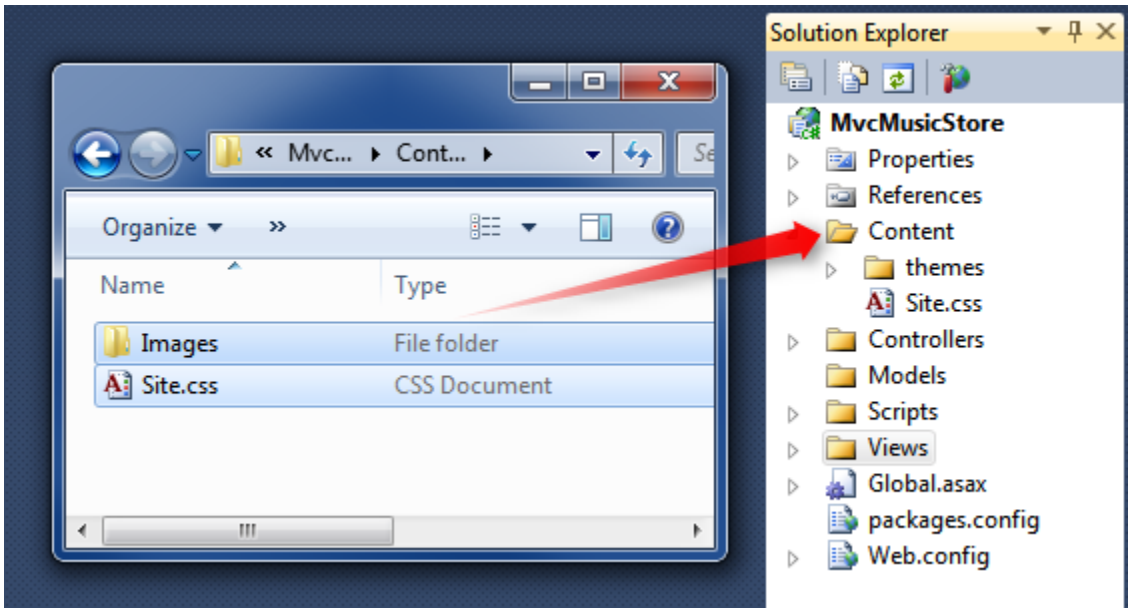

Tutorial under Creative Commons Attribution 3.0 License

```
    <script src="@Url.Content("~/Scripts/modernizr-1.7.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>
```

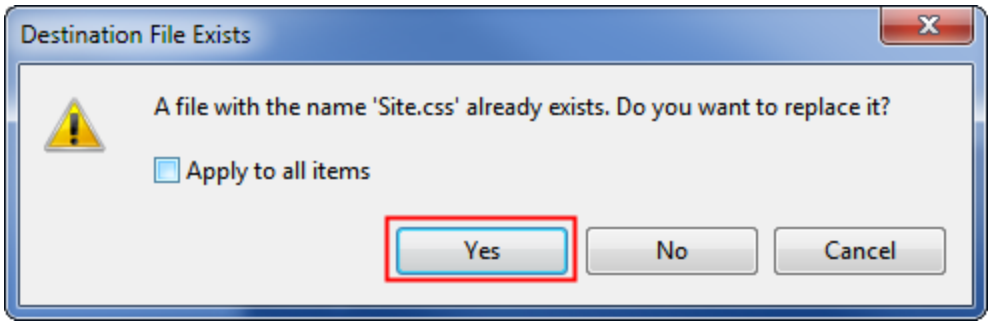
Updating the StyleSheet

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.

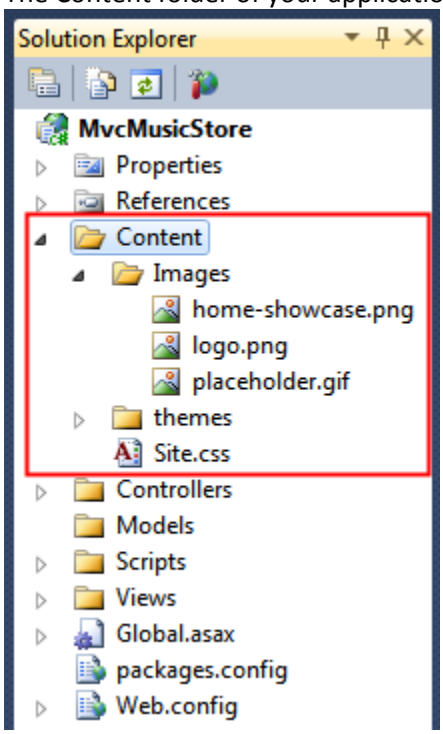
The updated CSS file and Images are included in the Content directory of MvcMusicStore-Assets.zip which is available at http://mvcmusicstore.codeplex.com. We'll select both of them in Windows Explorer and drop them into our Solution's Content folder in Visual Web Developer, as shown below:



You'll be asked to confirm if you want to overwrite the existing Site.css file. Click Yes.

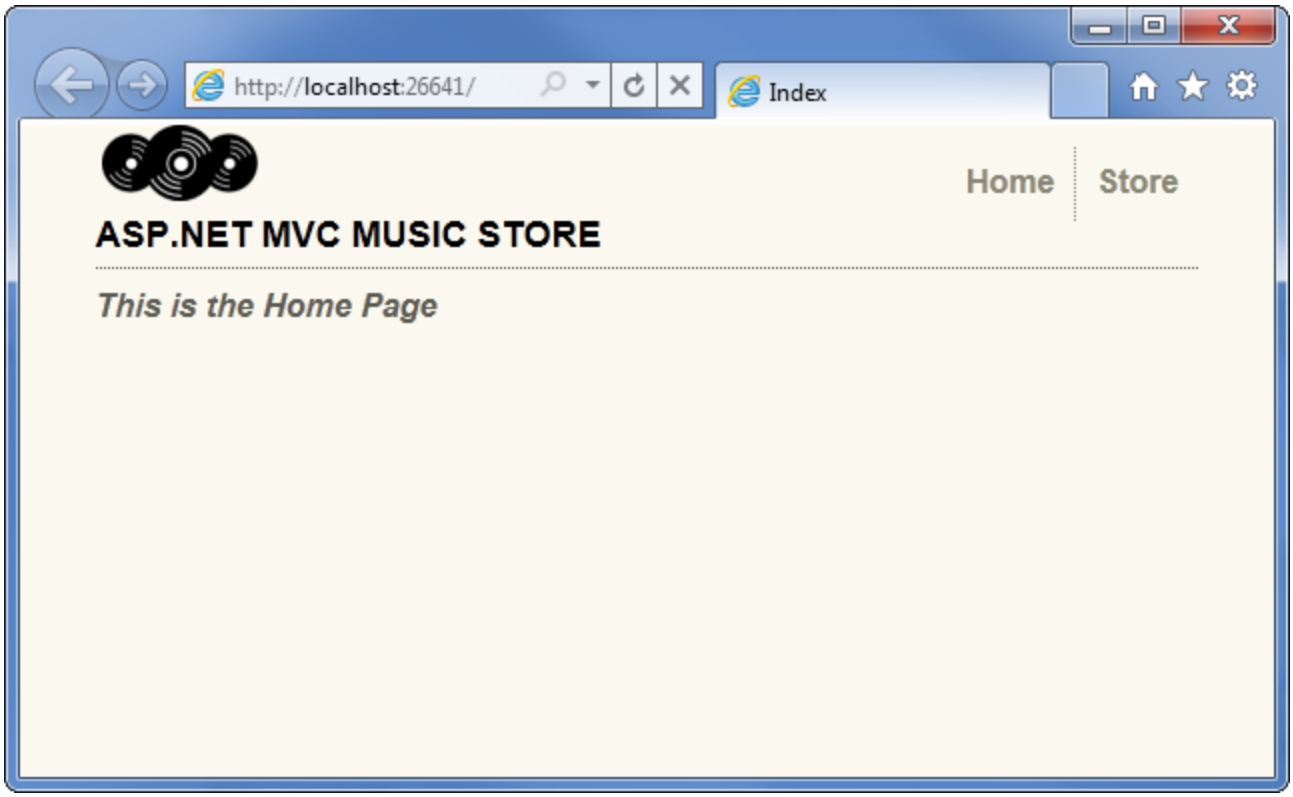


The Content folder of your application will now appear as follows:



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

Tutorial under Creative Commons Attribution 3.0 License



- 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.

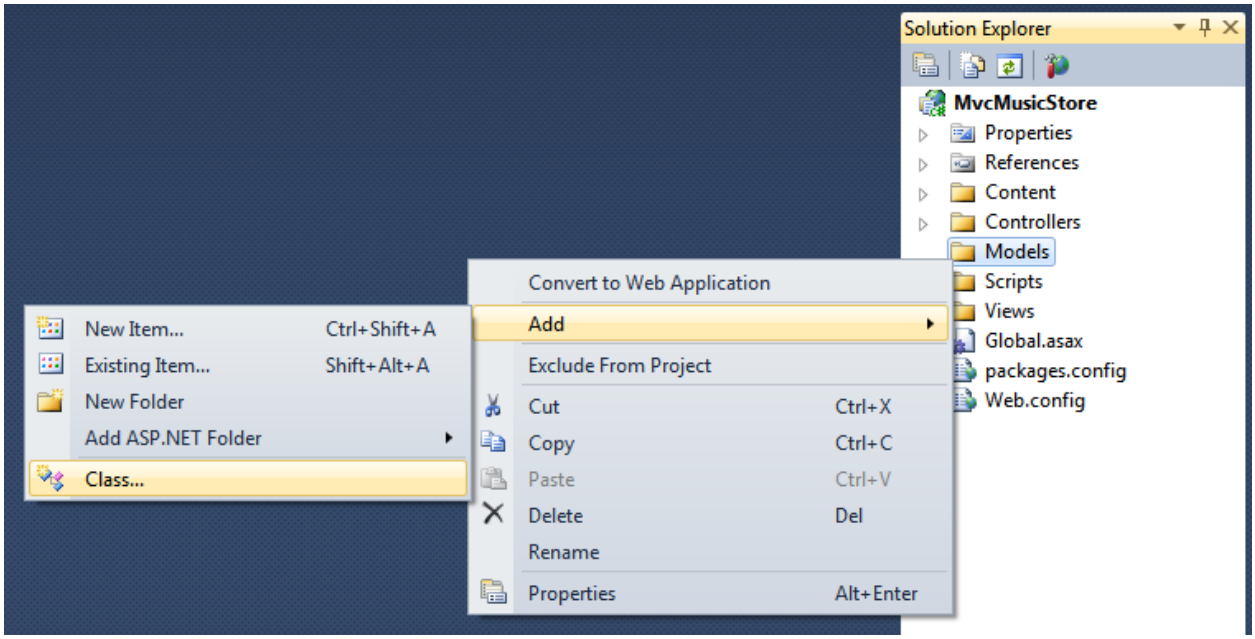
Using a Model to pass information to our View

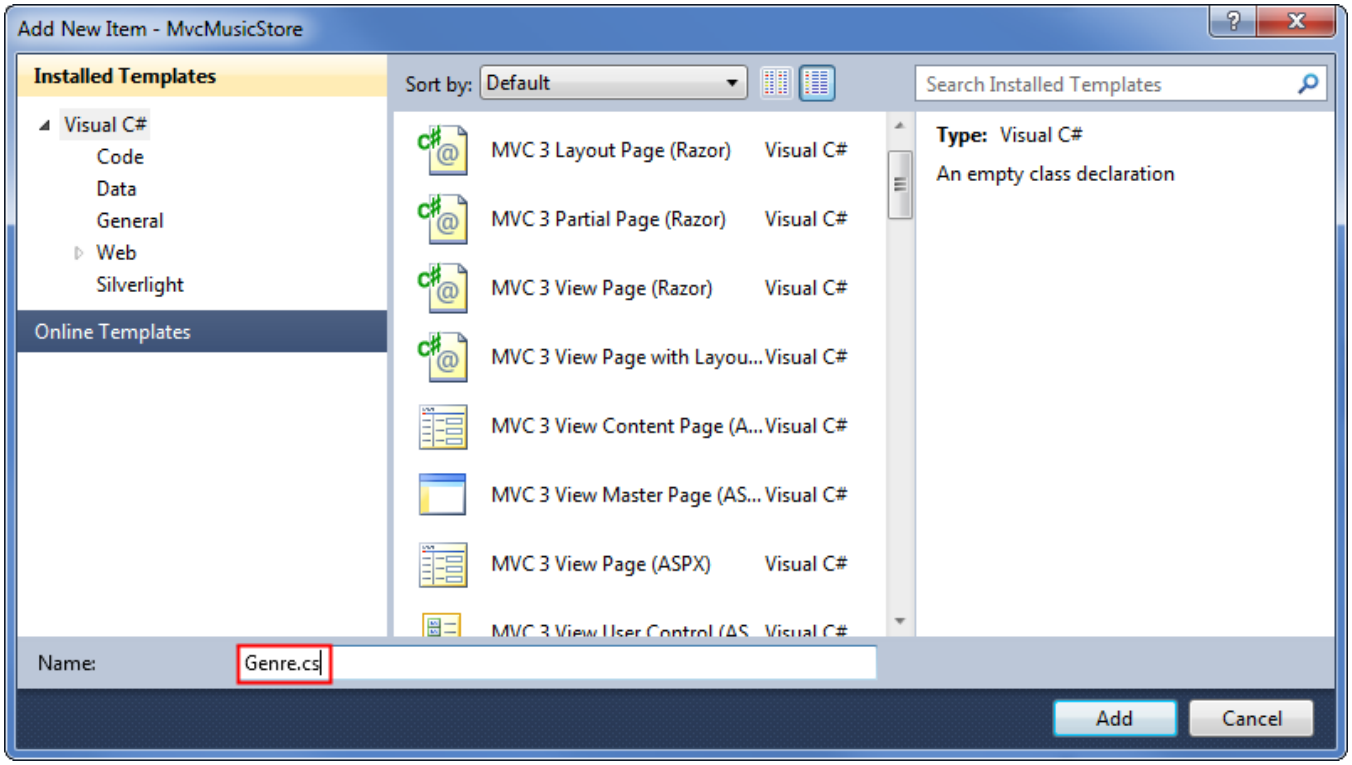
A View template that just displays hardcoded HTML isn't going to make a very interesting web site. To create a dynamic web site, we'll instead want to pass information from our controller actions to our view templates.

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:

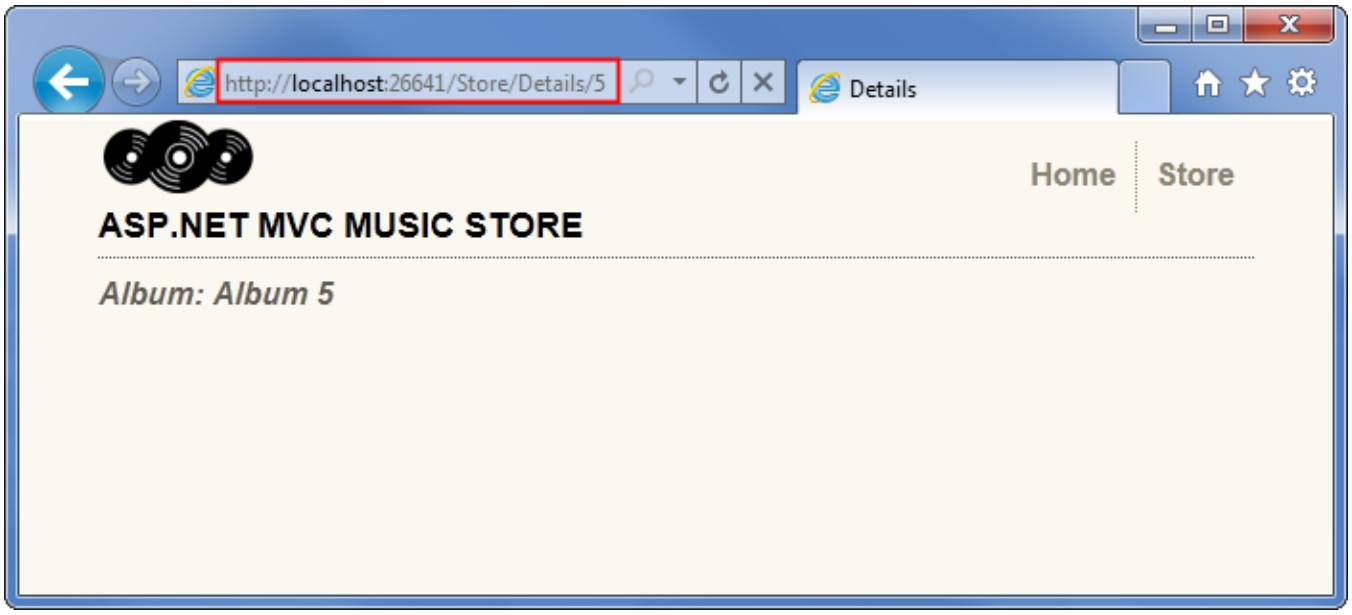
```
public class Genre
{
    public string Name { get; set; }
}
```

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 dynamic information from our Model. If - for demonstration purposes right now - we named our Albums based on the request ID, we could display that information as in the view 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.

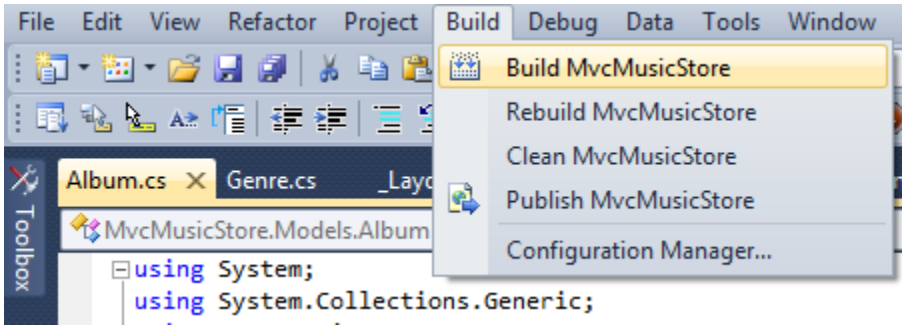
```
public ActionResult Details(int id)
```

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.

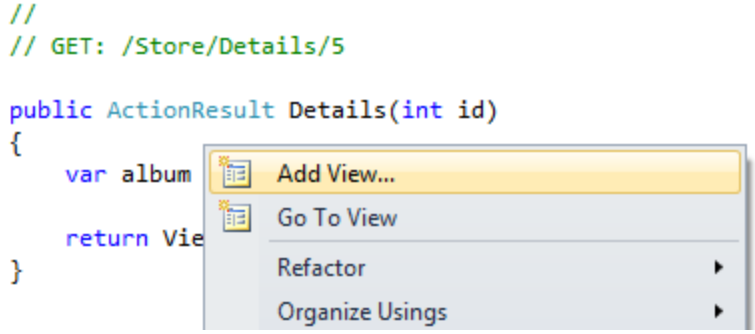
```
public ActionResult Details(int id)
{
    var album = new Album { Title = "Album " + id };

    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 (for extra credit, you can use the Ctrl-Shift-B shortcut to build the project).

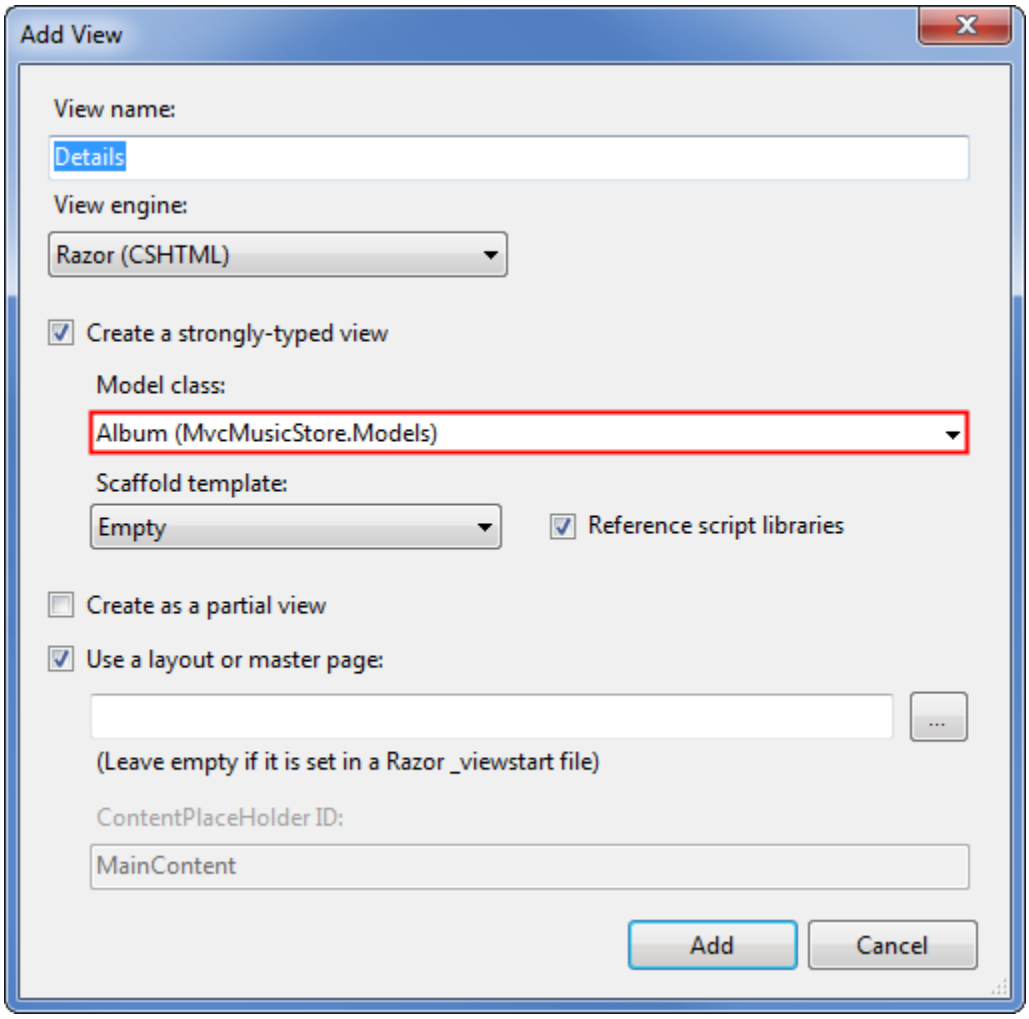


Now that we've set up our supporting classes, we're ready to build our View template. Right-click within the Details method and select "Add View…" from the context menu.



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.

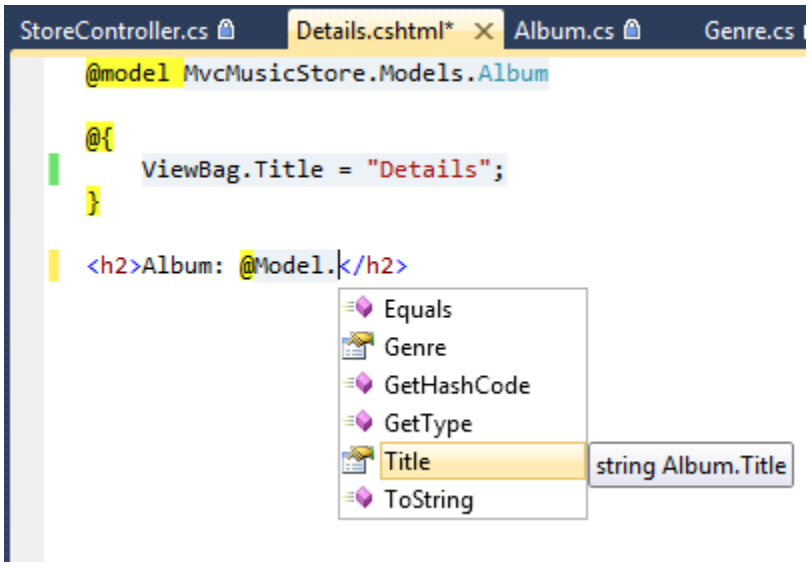
```
@model MvcMusicStore.Models.Album

@{
    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 Web Developer editor.

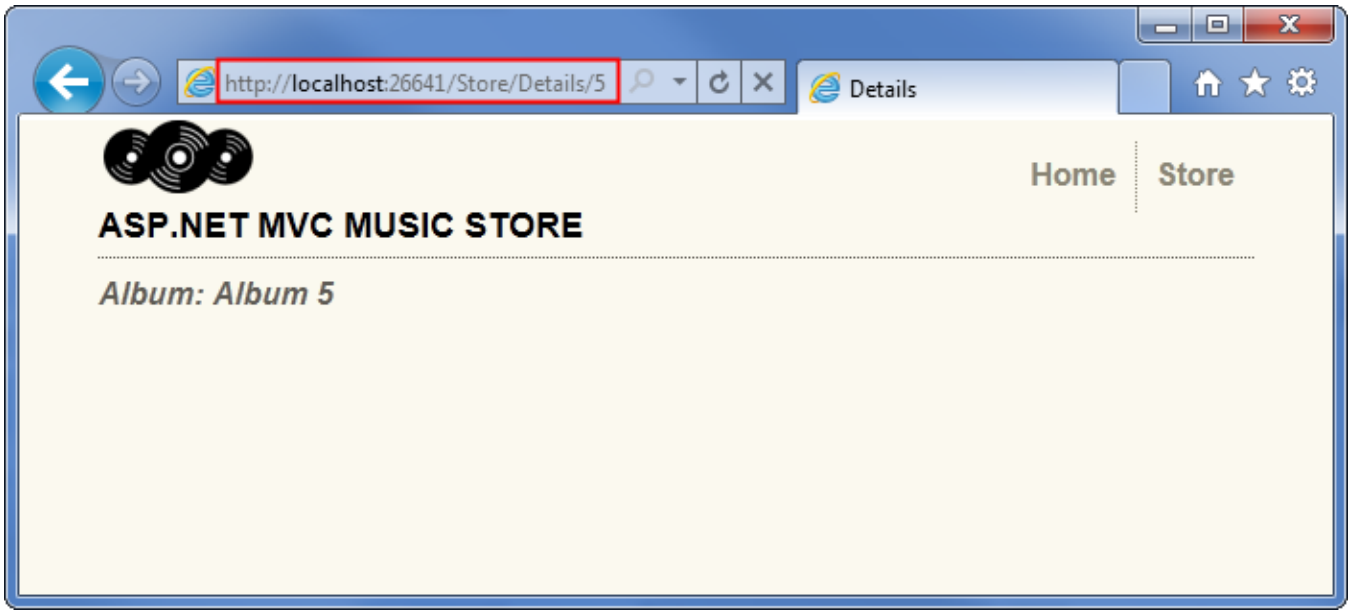
Update the <h2> tag so it displays the Album's Title property by modifying that line to appear as follows.

```
<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/5 URL. We'll see details of an Album like below.

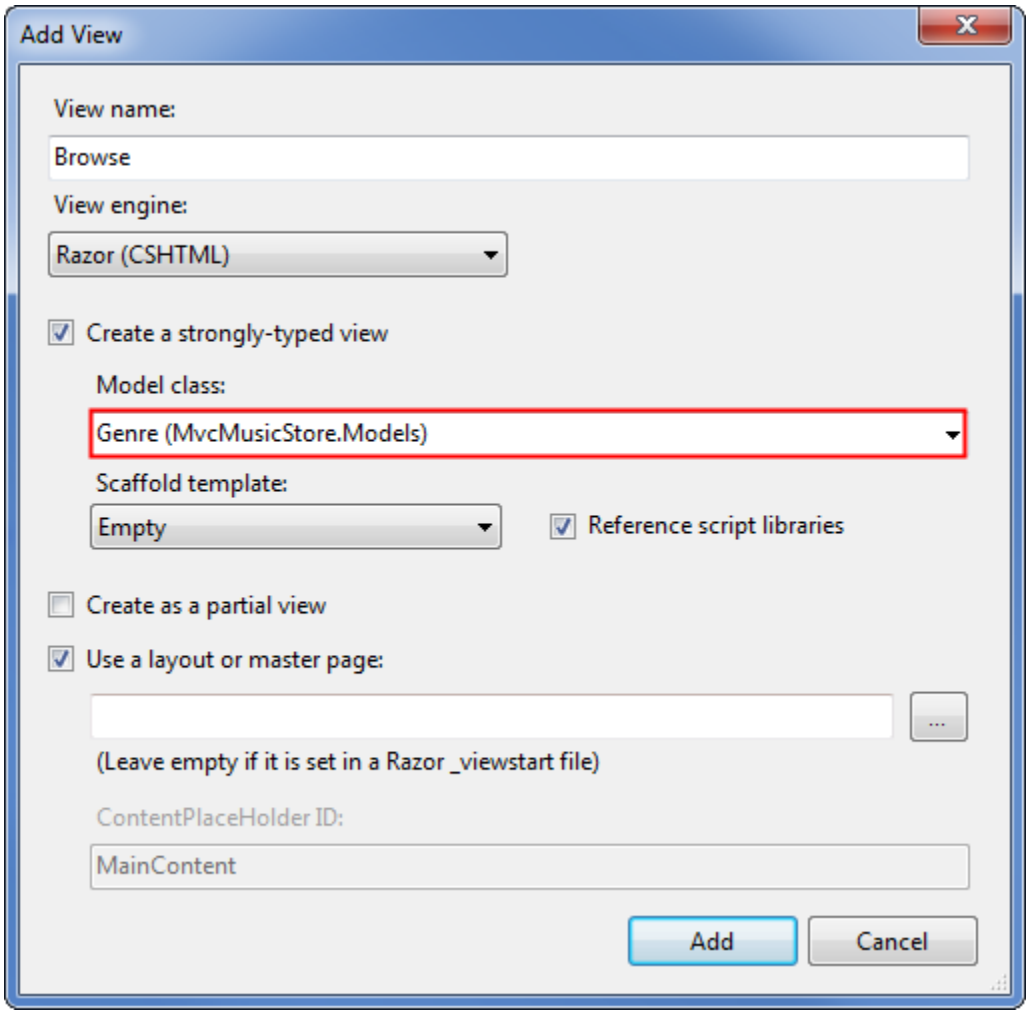


Now we'll make a similar update for the Store Browse action method. Update the method so it returns an ActionResult, and modify the method logic so it creates a new Genre object and returns it to the View.

```
public ActionResult Browse(string genre)
```

```
{
    var genreModel = new Genre { Name = genre };

    return View(genreModel);
}
```

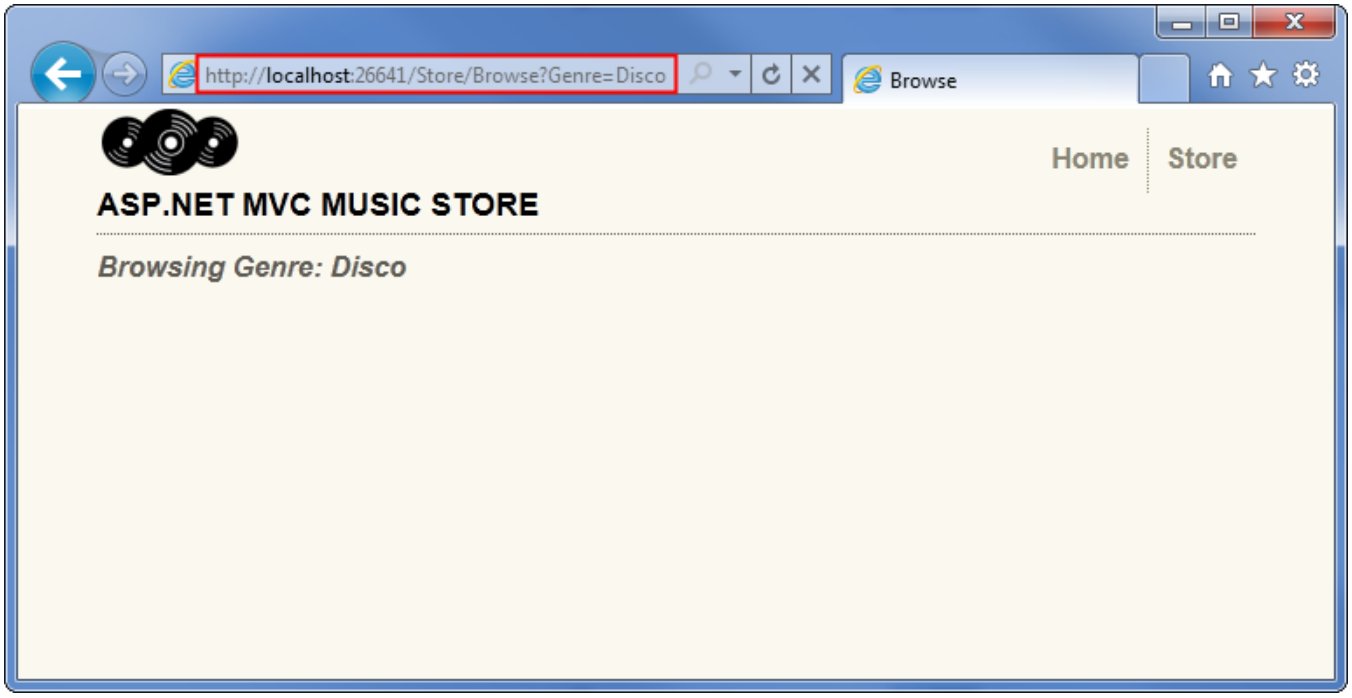
Right-click in the Browse method and select “Add View…” from the context menu, then add a View that is strongly-typed add a strongly typed to the Genre class.



Update the <h2> element in 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>
```

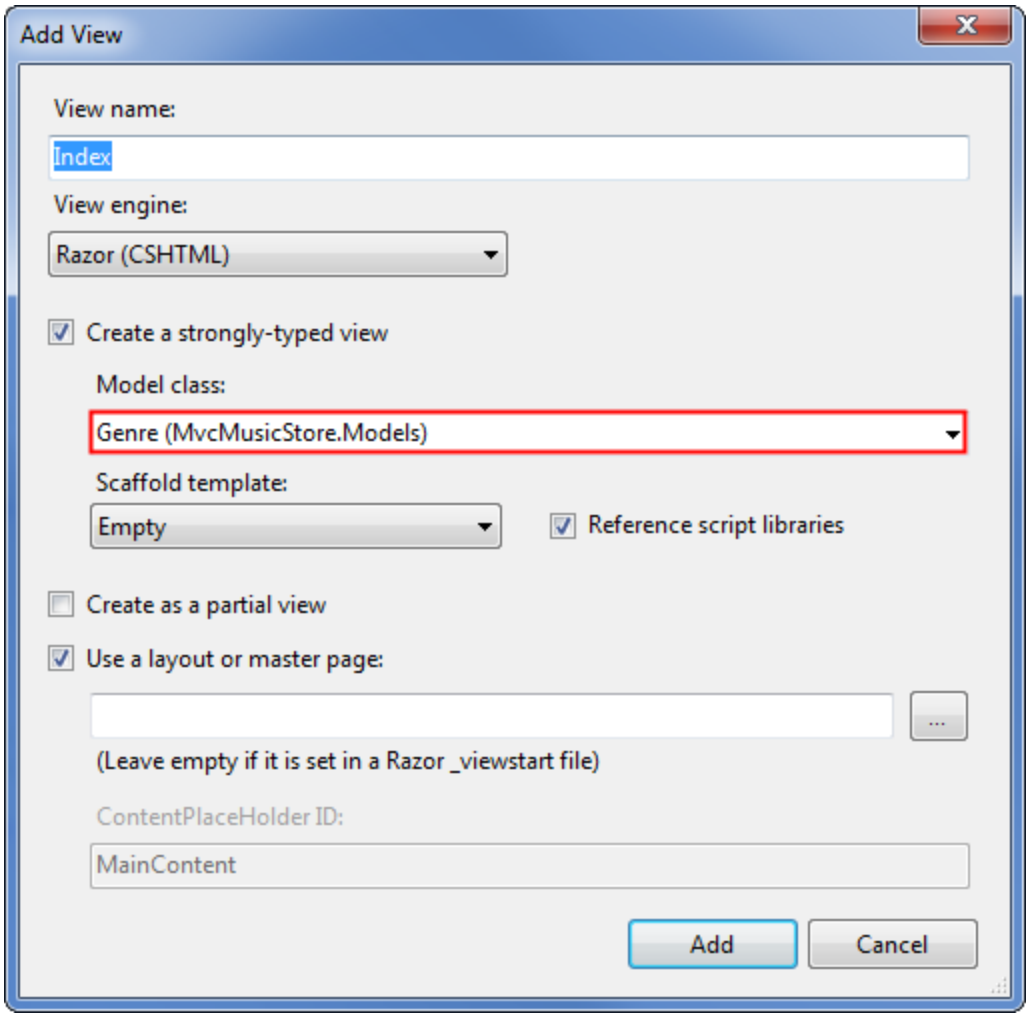
Now let's re-run our project and browse to the /Store/Browse?Genre=Disco URL. We'll see the Browse page displayed like below.



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 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.

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

<p>
    Select from @Model.C genres:</p>
<ul>
    @foreach (var genre
    {
        <li>@genre.Name
    }
</ul>
```

AsQueryable<>
Average<>
Cast<>
Concat<>
Contains<>
Count<>
DefaultIfEmpty<>
Distinct<>
ElementAt<>

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

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

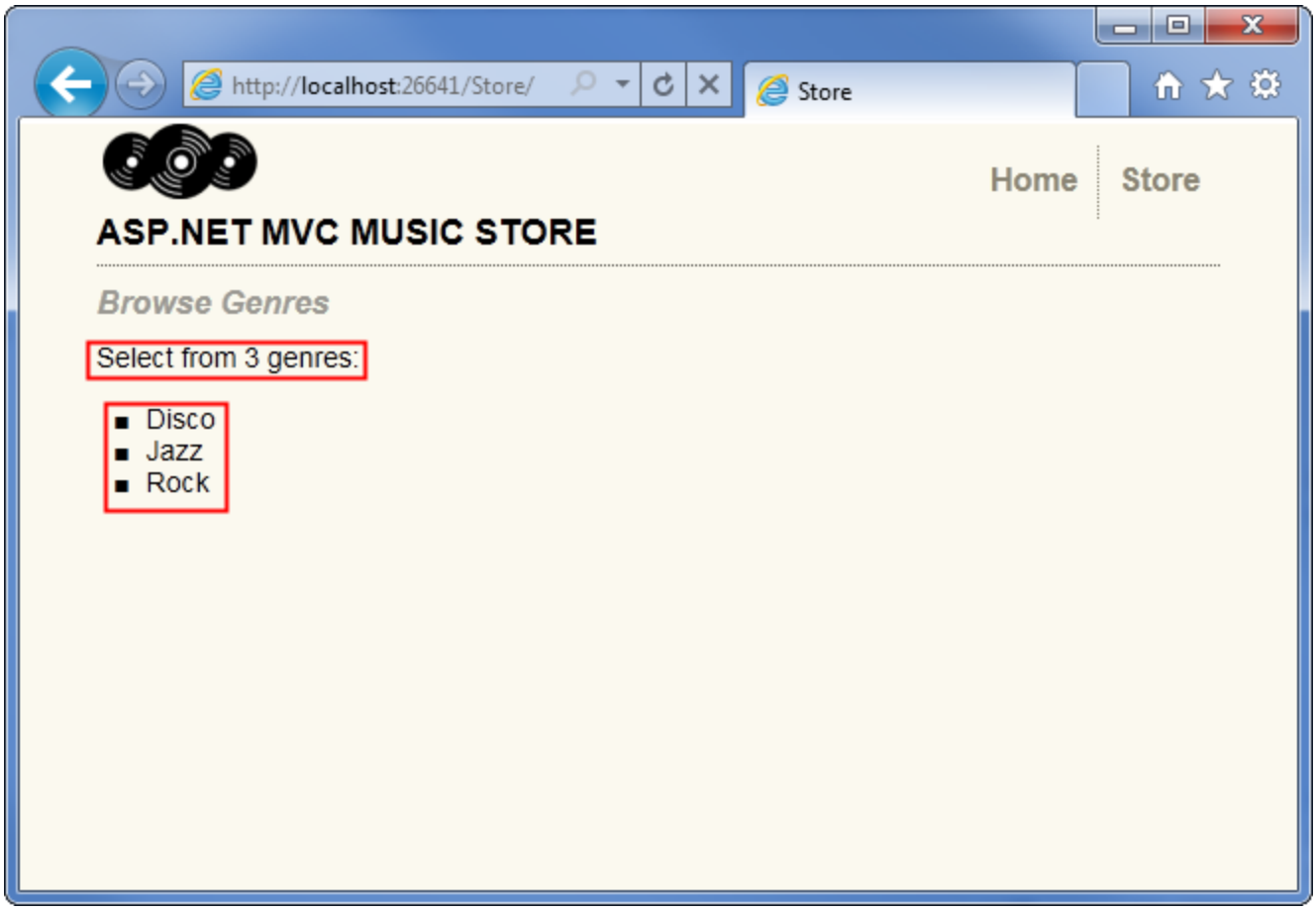
Equals
GetHashCode
GetType
Name
ToString

string Genre.Name

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.

Tutorial under Creative Commons Attribution 3.0 License



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 **(don't type this in - we're going to improve on it)**:

```
<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:

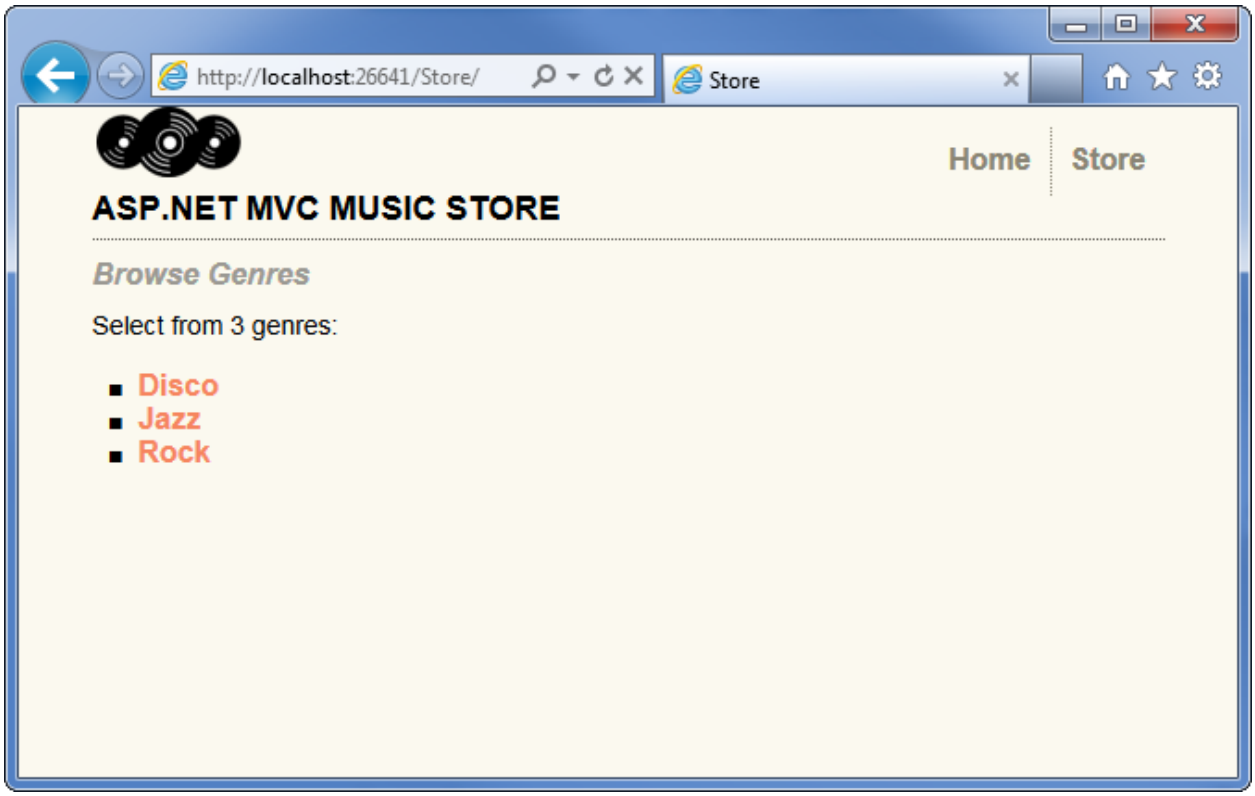
1. Link text, which will display the Genre name
2. Controller action name (Browse)
3. Route parameter values, specifying both the name (Genre) and the value (Genre name)

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.

Tutorial under Creative Commons Attribution 3.0 License



The HTML for the genre list looks like this:

```
<ul>
    <li><a href="/Store/Browse?genre=Disco">Disco</a> </li>
    <li><a href="/Store/Browse?genre=Jazz">Jazz</a> </li>
    <li><a href="/Store/Browse?genre=Rock">Rock</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 SQL Server Compact Edition (often called SQL CE) as our database engine. SQL CE is a free, embedded, file based database that doesn't require any installation or configuration, which makes it really convenient for local development.

Database access with Entity Framework Code-First

We'll use the Entity Framework (EF) support that is included in ASP.NET MVC 3 projects to query and update the database. 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.

Changes to our Model Classes

We will be leveraging the database creation feature in Entity Framework in this tutorial. Before we do that, though, let's make a few minor changes to our model classes to add in some things we'll be using later on.

Adding the Artist Model Classes

Our Albums will be associated with Artists, so we'll add a simple model class to describe an Artist. Add a new class to the Models folder named Artist.cs using the code shown below.

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

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


Tutorial under Creative Commons Attribution 3.0 License

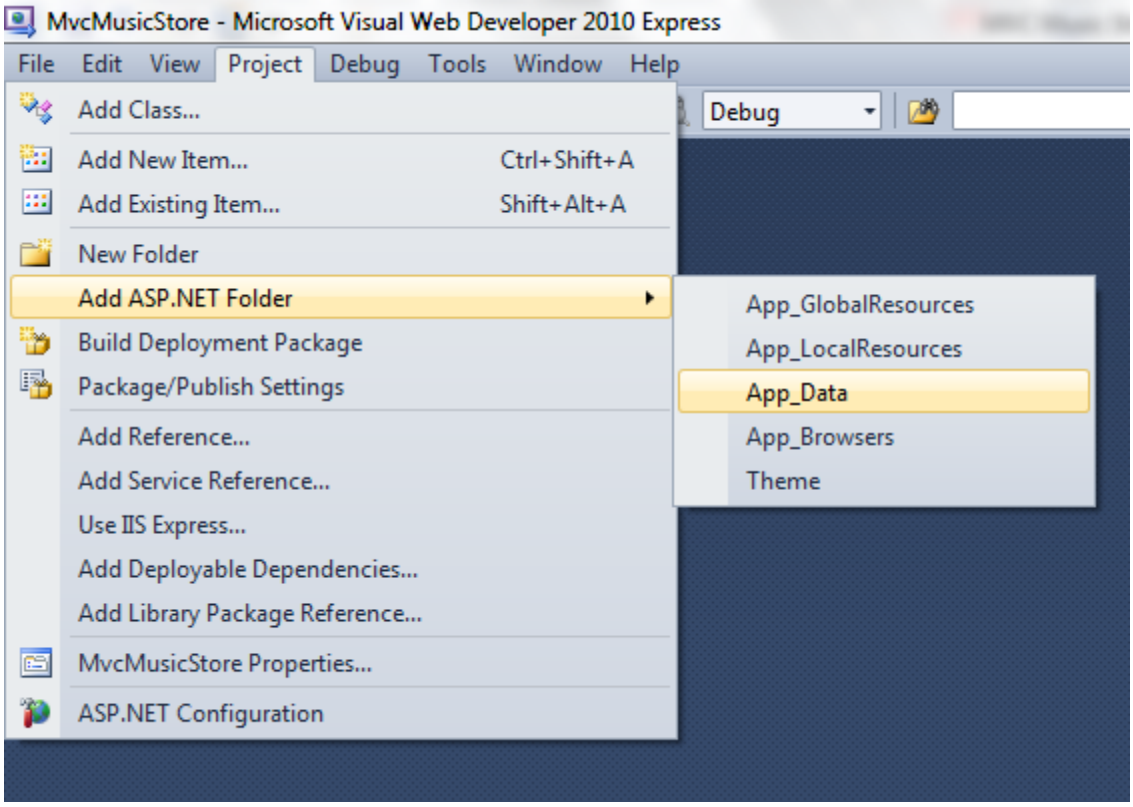
```
    }
}
```

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; }
    }
}
```

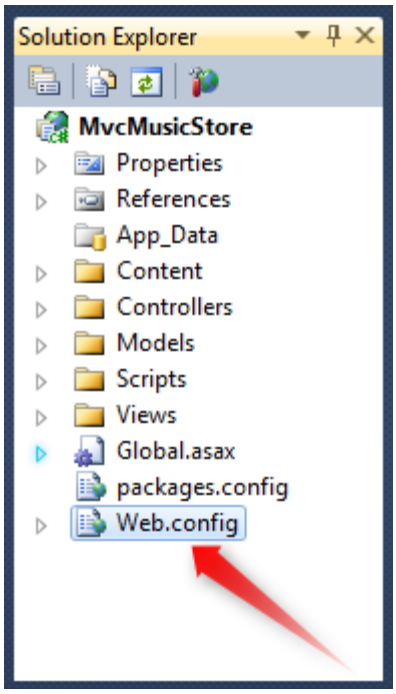
Adding the App_Data folder

We'll add 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. From the Project menu, select Add ASP.NET Folder, then App_Data.



Creating a Connection String in the web.config file

We will add a few 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.



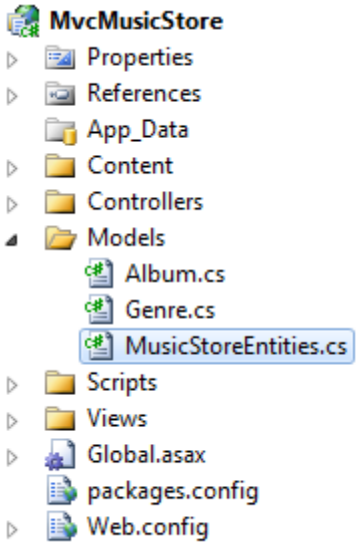
Scroll to the bottom of this file and add a <connectionStrings> section directly above the last line, as shown below.

```
  <connectionStrings>
    <add name="MusicStoreEntities"
     connectionString="Data Source=|DataDirectory|MvcMusicStore.sdf"
     providerName="System.Data.SqlServerCe.4.0"/>
  </connectionStrings>
</configuration>
```

Adding a Context Class

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

Tutorial under Creative Commons Attribution 3.0 License



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; }
    }
}
```

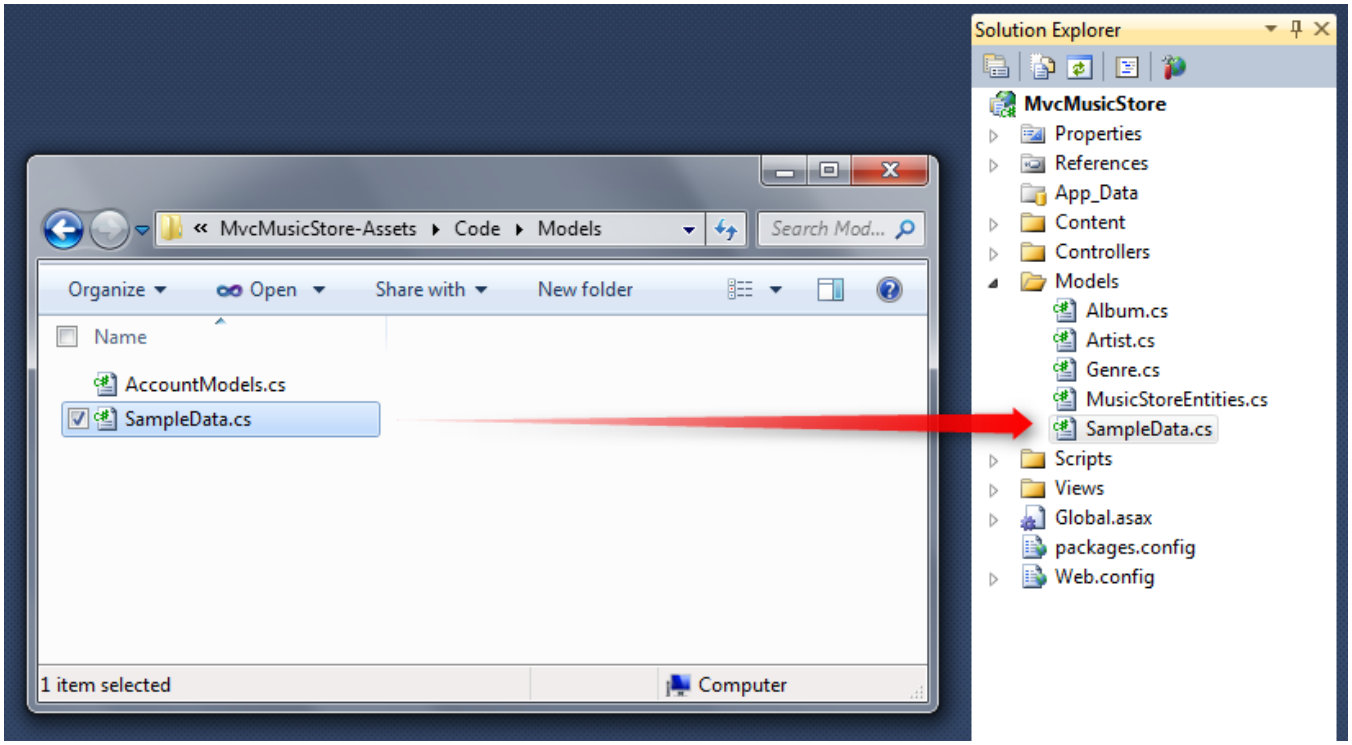
That's it - there's no other configuration, special interfaces, etc. By extending the DbContext base class, our MusicStoreEntities class is able to handle our database operations for us. Now that we've got that hooked up, let's add a few more properties to our model classes to take advantage of some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

Adding our store catalog data

We will take advantage of a feature in Entity Framework which adds "seed" data to a newly created database. This will pre-populate our store catalog with a list of Genres, Artists, and Albums. The MvcMusicStore-Assets.zip download - which included our site design files used earlier in this tutorial - has a class file with this seed data, located in a folder named Code.

Within the Code / Models folder, locate the SampleData.cs file and drop it into the Models folder in our project, as shown below.

Tutorial under Creative Commons Attribution 3.0 License



Now we need to add one line of code to tell Entity Framework about that SampleData class. Double-click on the Global.asax file in the root of the project to open it and add the following line to the top the Application_Start method.

```
protected void Application_Start()
{
    System.Data.Entity.Database.SetInitializer(new MvcMusicStore.Models.SampleData());

    AreaRegistration.RegisterAllAreas();

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

At this point, we've completed the work necessary to configure Entity Framework for our project.

Querying the Database

Now let's update our StoreController so that instead of using "dummy data" it instead calls 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();
```

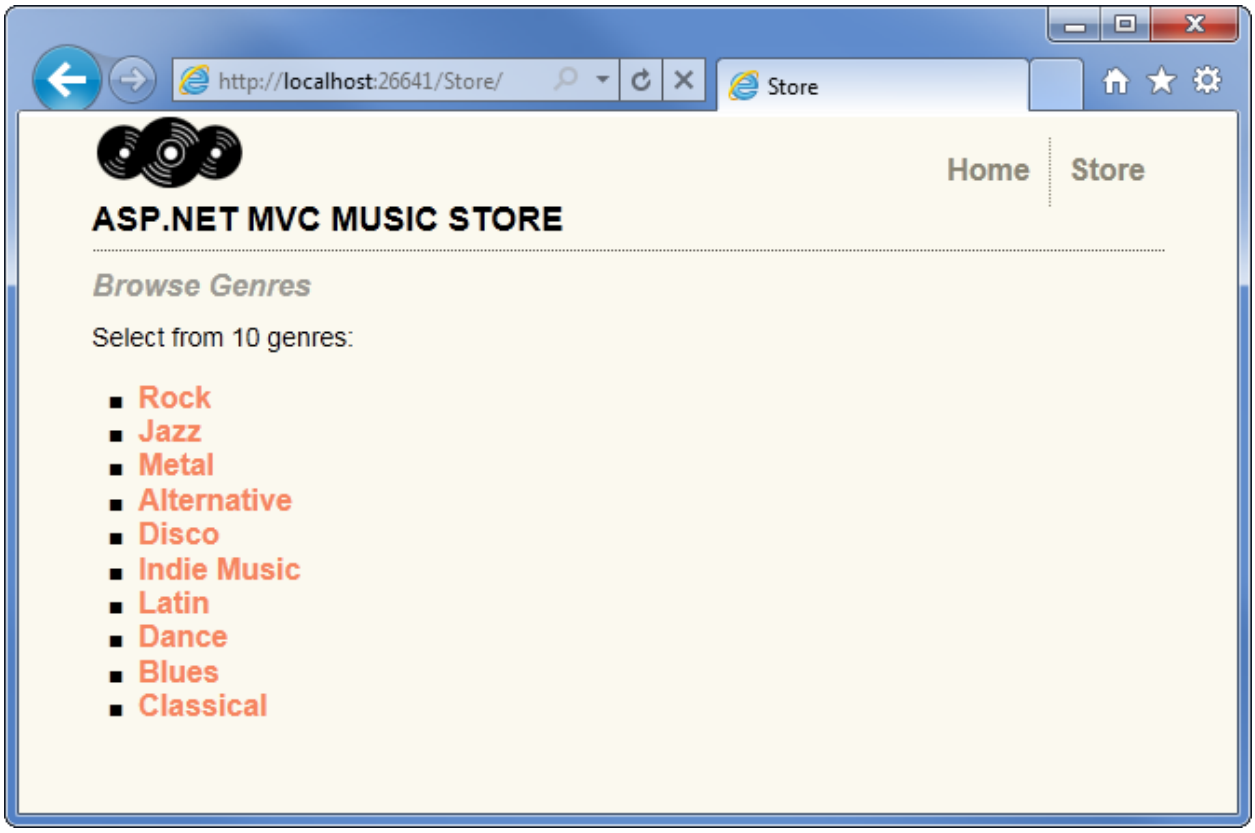
Updating the Store Index to query the database

The MusicStoreEntities class is maintained by the Entity Framework and exposes a collection property for each table in our database. Let's update our StoreController's Index action to retrieve all Genres in our database. Previously we did this by hard-coding string data. Now we can instead just use the Entity Framework context Generes collection:

```
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:



Updating Store Browse and Details to use live data

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 (don't type this yet):

```
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:

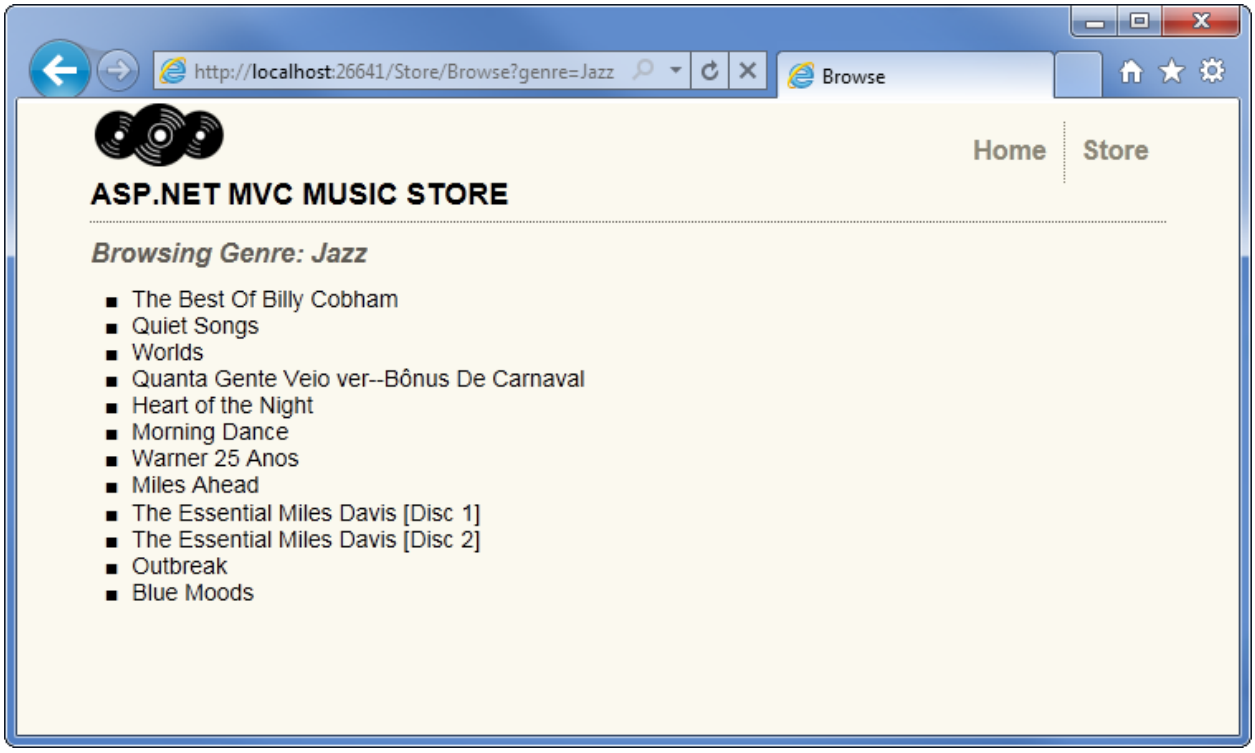
```
public ActionResult Browse(string genre)
{
    // Retrieve Genre and its Associated Albums from database
    var genreModel = storeDB.Genres.Include("Albums")
        .Single(g => g.Name == genre);

    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.

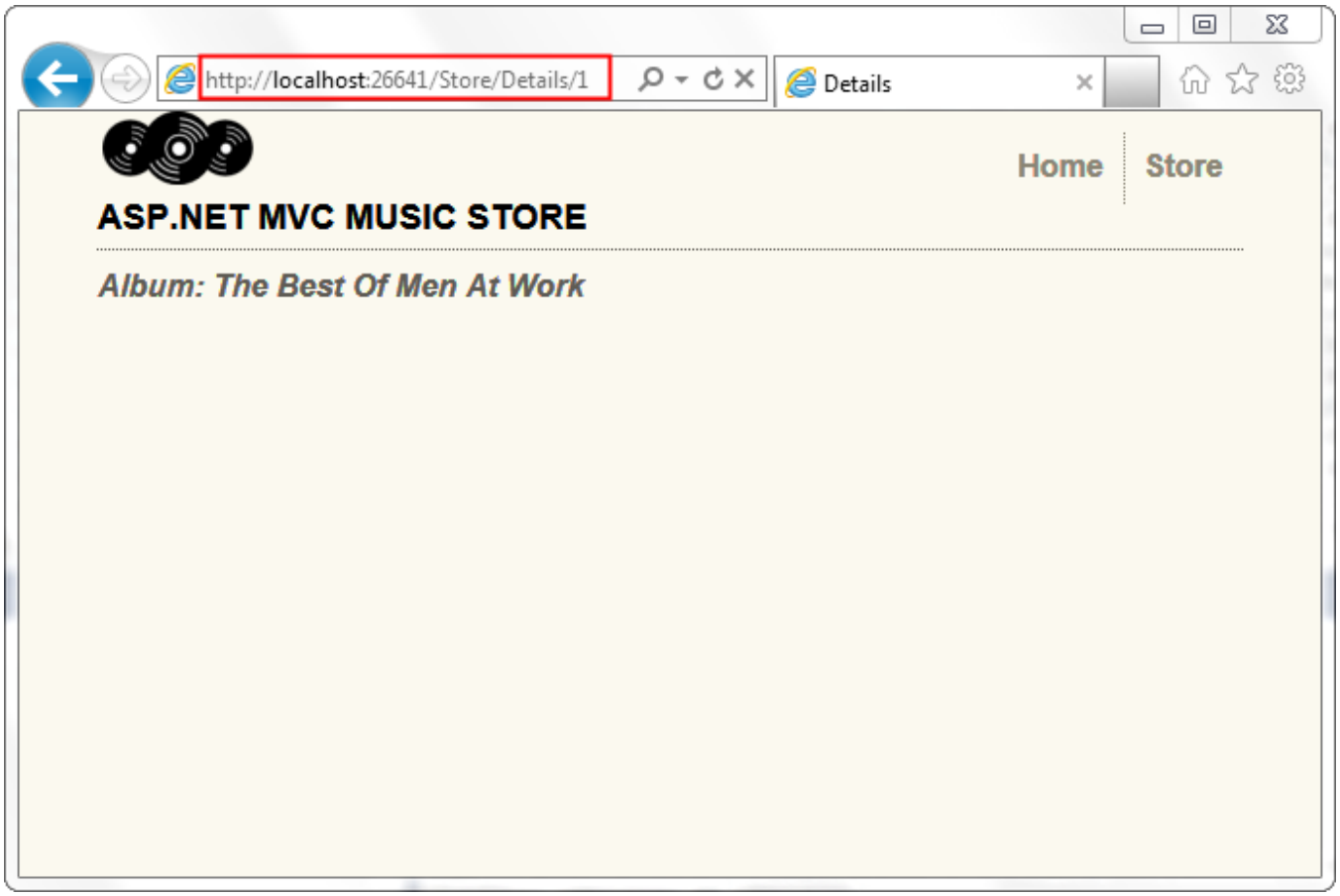
Tutorial under Creative Commons Attribution 3.0 License



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);
}
```

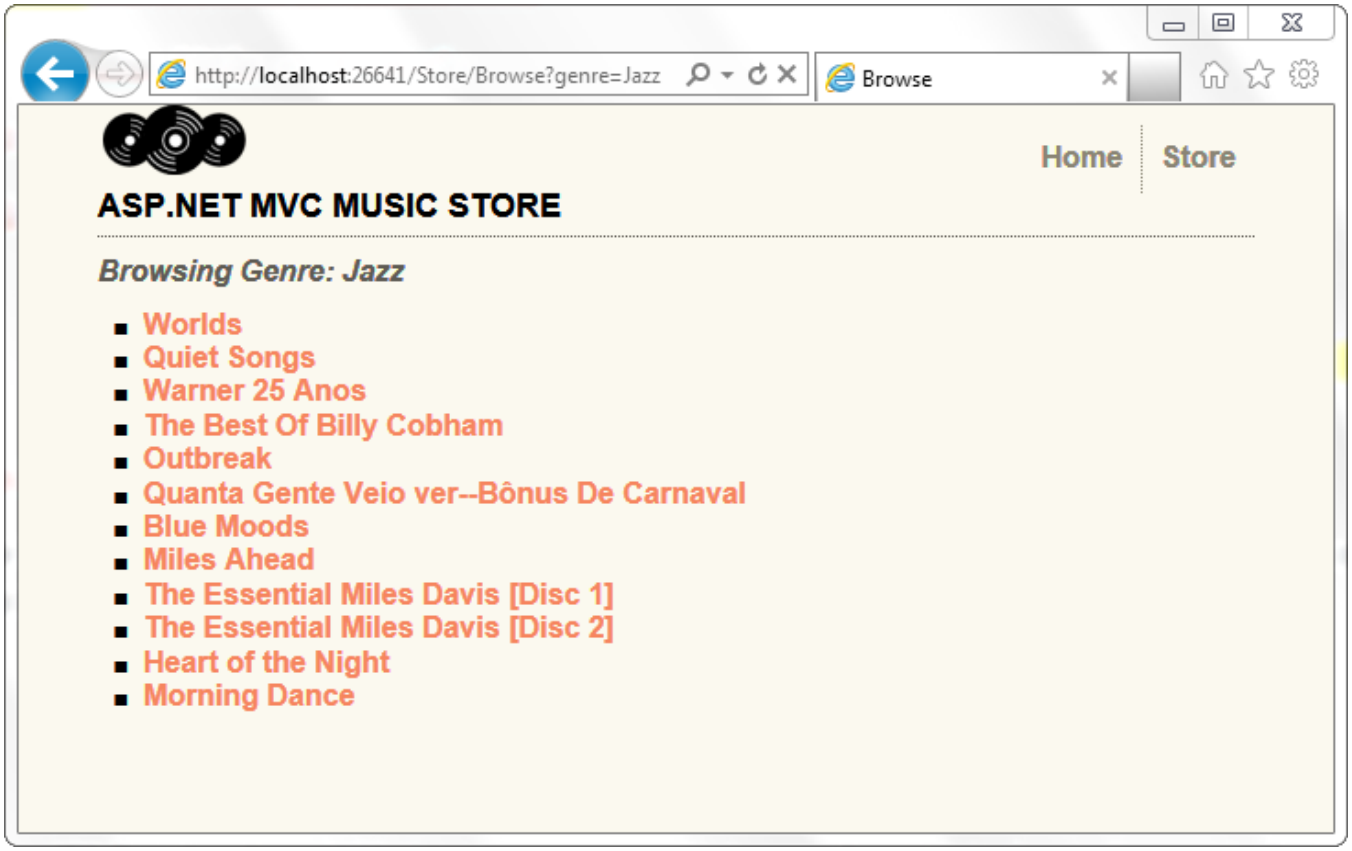
Running our application and browsing to /Store/Details/1 shows that our results are now being pulled from the database.



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.

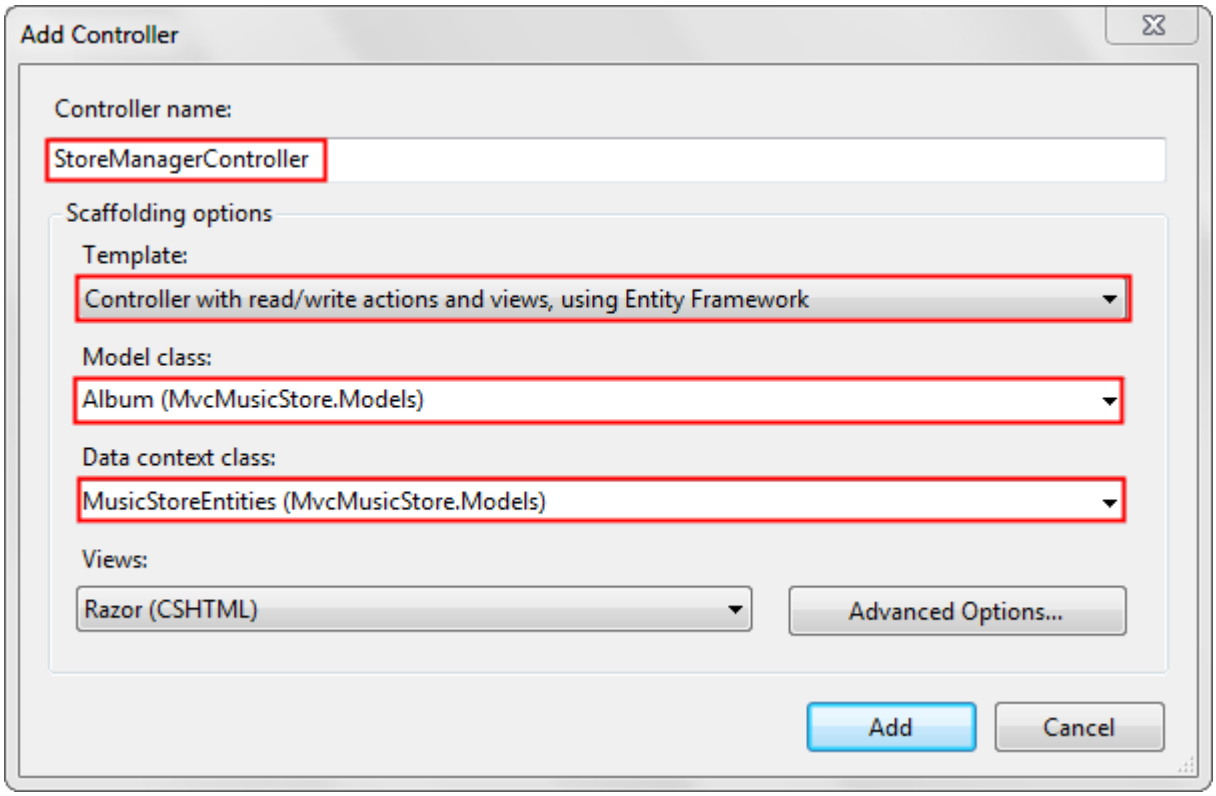

http://localhost:26641/Store/Browse?genre=Jazz
Browse
Home
Store
ASP.NET MVC MUSIC STORE
Browsing Genre: Jazz
Worlds
Quiet Songs
Warner 25 Anos
The Best Of Billy Cobham
Outbreak
Quanta Gente Veio ver--Bônus De Carnaval
Blue Moods
Miles Ahead
The Essential Miles Davis [Disc 1]
The Essential Miles Davis [Disc 2]
Heart of the Night
Morning Dance

5. Edit Forms using Scaffolding

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

Creating the StoreManagerController

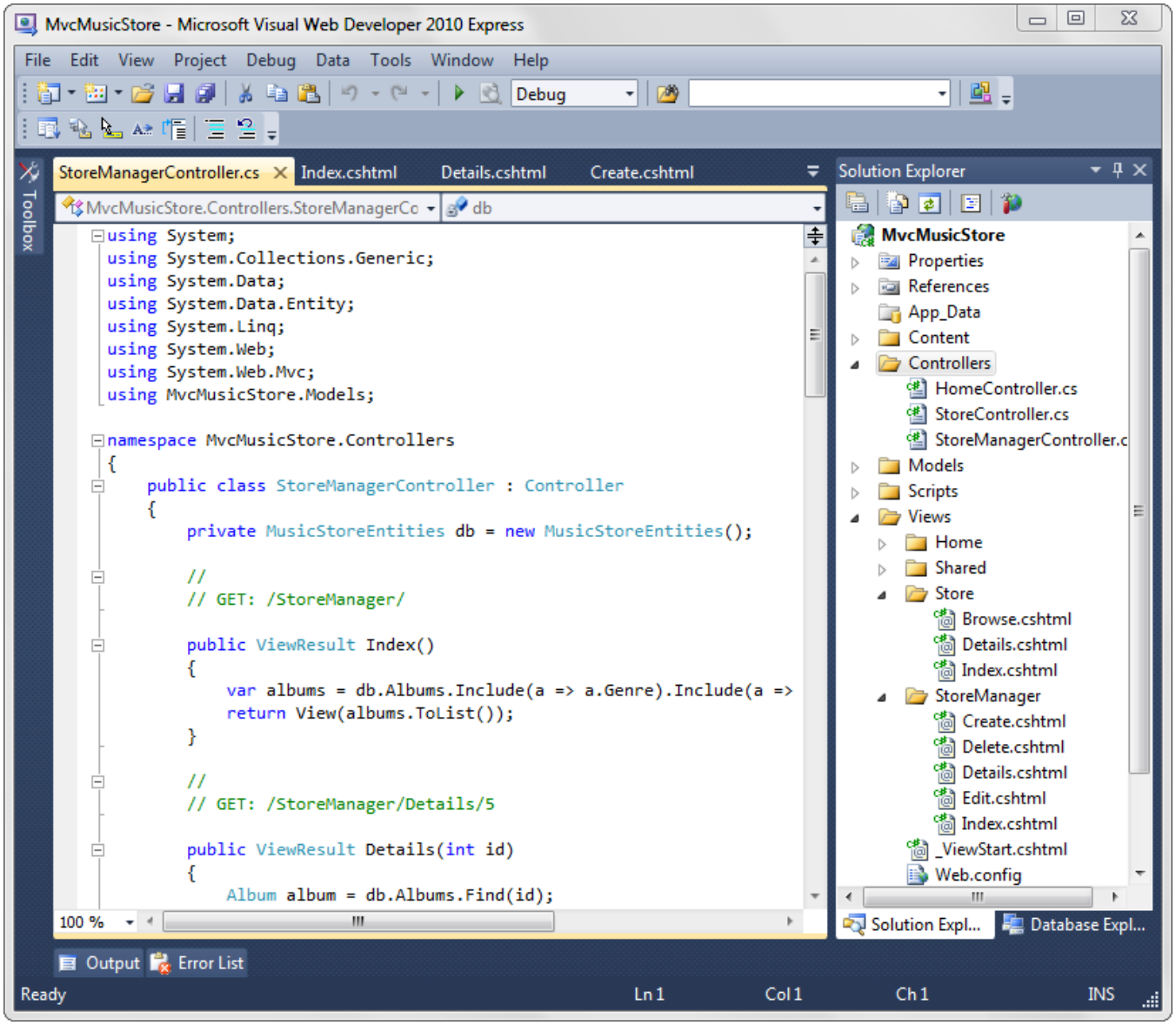
We'll begin by creating a new controller called **StoreManagerController**. For this controller, we will be taking advantage of the Scaffolding features available in the ASP.NET MVC 3 Tools Update. Set the options for the Add Controller dialog as shown below.



When you click the Add button, you'll see that the ASP.NET MVC 3 scaffolding mechanism does a good amount of work for you:

- It creates the new StoreManagerController with a local Entity Framework variable
- It adds a StoreManager folder to the project's Views folder
- It adds Create.cshtml, Delete.cshtml, Details.cshtml, Edit.cshtml, and Index.cshtml view, strongly typed to the Album class

Tutorial under Creative Commons Attribution 3.0 License



The new StoreManager controller class includes CRUD (create, read, update, delete) controller actions which know how to work with the Album model class and use our Entity Framework context for database access.

Modifying a Scaffolded View

It's important to remember that, while this code was generated for us, it's standard ASP.NET MVC code, just like we've been writing throughout this tutorial. It's intended to save you the time you'd spend on writing boilerplate controller code and creating the strongly typed views manually, but this isn't the kind of generated code you may have seen prefaced with dire warnings in comments about how you mustn't change the code. This is your code, and you're expected to change it.

So, let's start with a quick edit to the StoreManager Index view (/Views/StoreManager/Index.cshtml). This view will display a table which lists the Albums in our store with Edit / Details / Delete links, and includes the Album's public properties. We'll remove the AlbumArtUrl field, as it's not very useful in this display. In <table> section of

the view code, remove the <th> and <td> elements surrounding AlbumArtUrl references, as indicated by the highlighted lines below:

```
<table>
    <tr>
        <th>
            Genre
        </th>
        <th>
            Artist
        </th>
        <th>
            Title
        </th>
        <th>
            Price
        </th>
        <th>
            AlbumArtUrl
        </th>
        <th></th>
    </tr>

@foreach (var item in Model) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.Genre.Name)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Artist.Name)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Title)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Price)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.AlbumArtUrl)
        </td>
        <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>
    </tr>
}

</table>
```

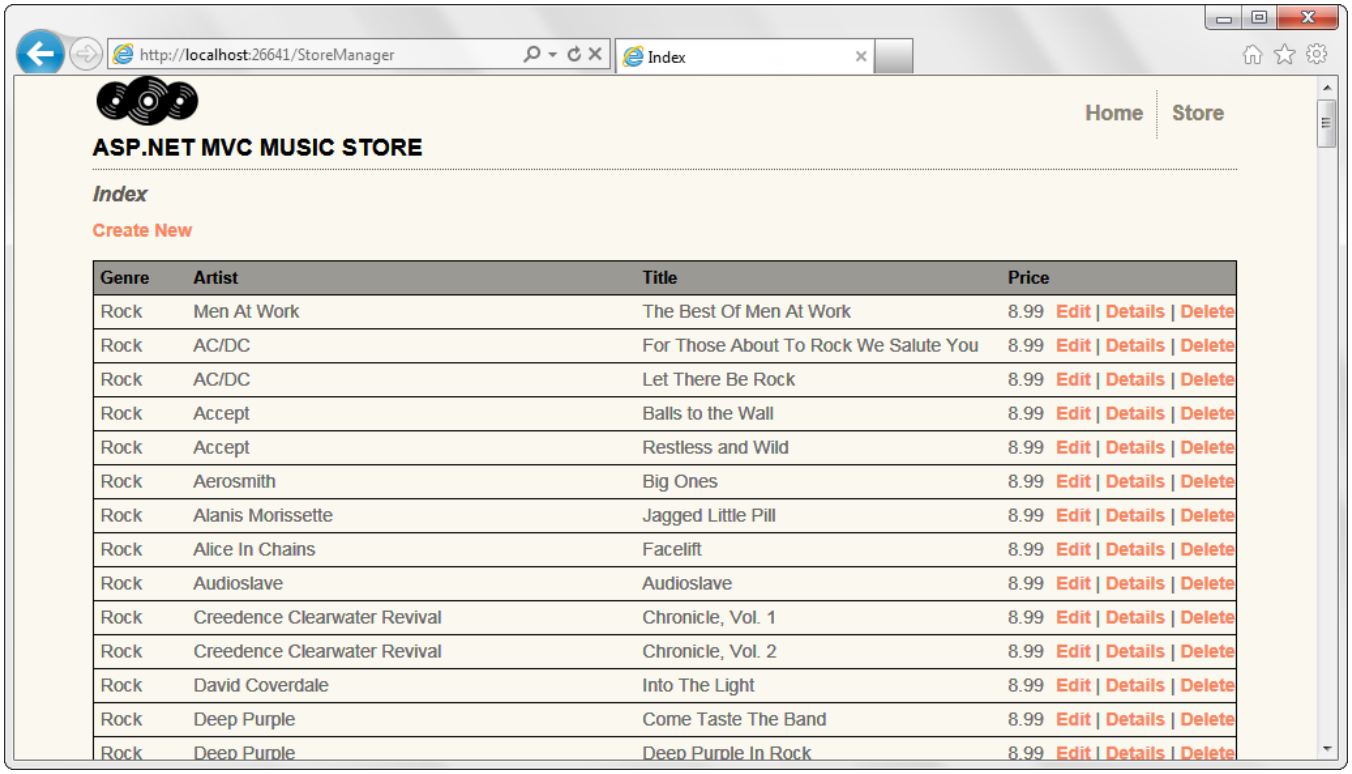
The modified view code will appear as follows:

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

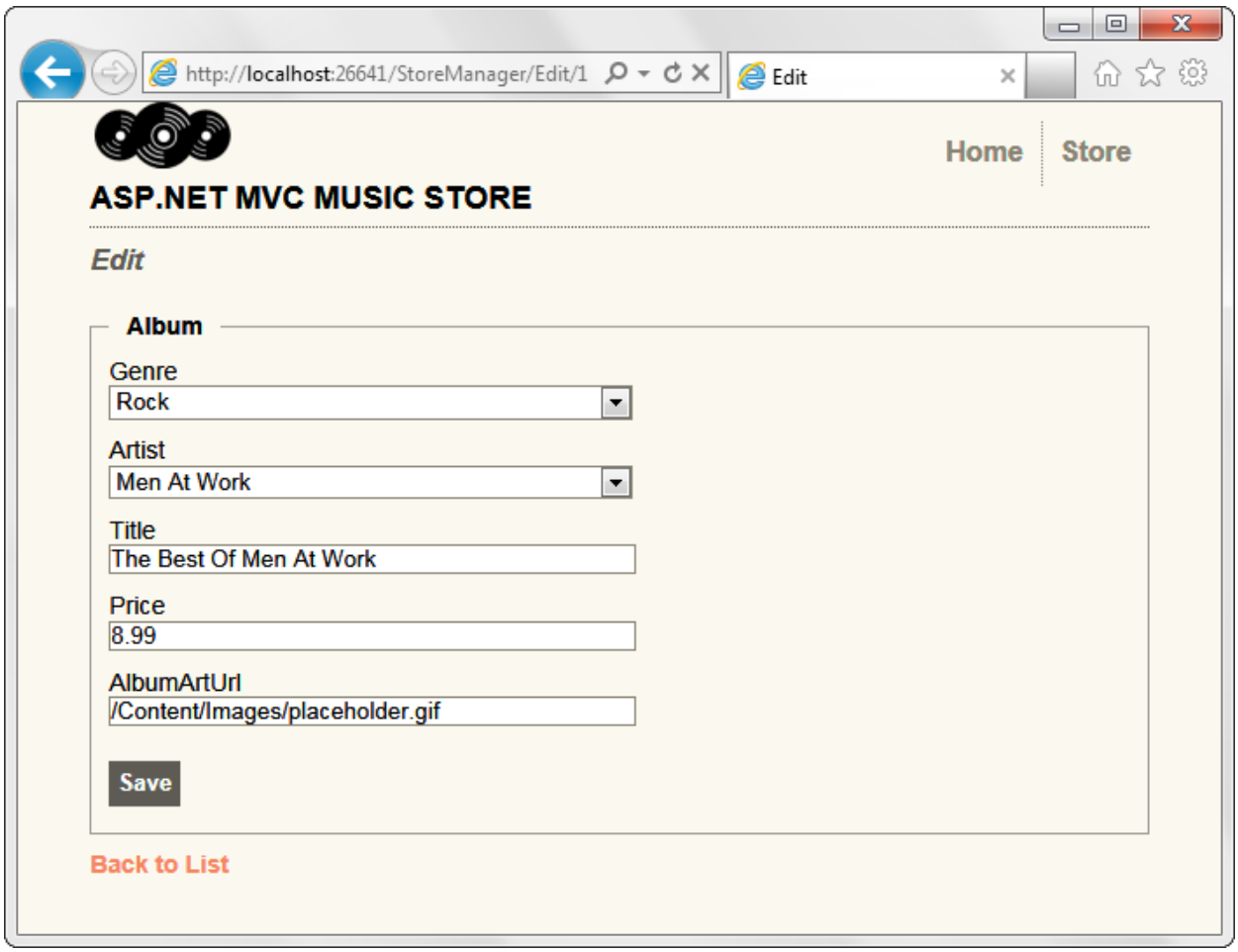
```
@{
    ViewBag.Title = "Index";
}

<h2>Index</h2>

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

@foreach (var item in Model) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.Genre.Name)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Artist.Name)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Title)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Price)
        </td>
        <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>
    </tr>
}

</table>
```

A first look at the Store Manager

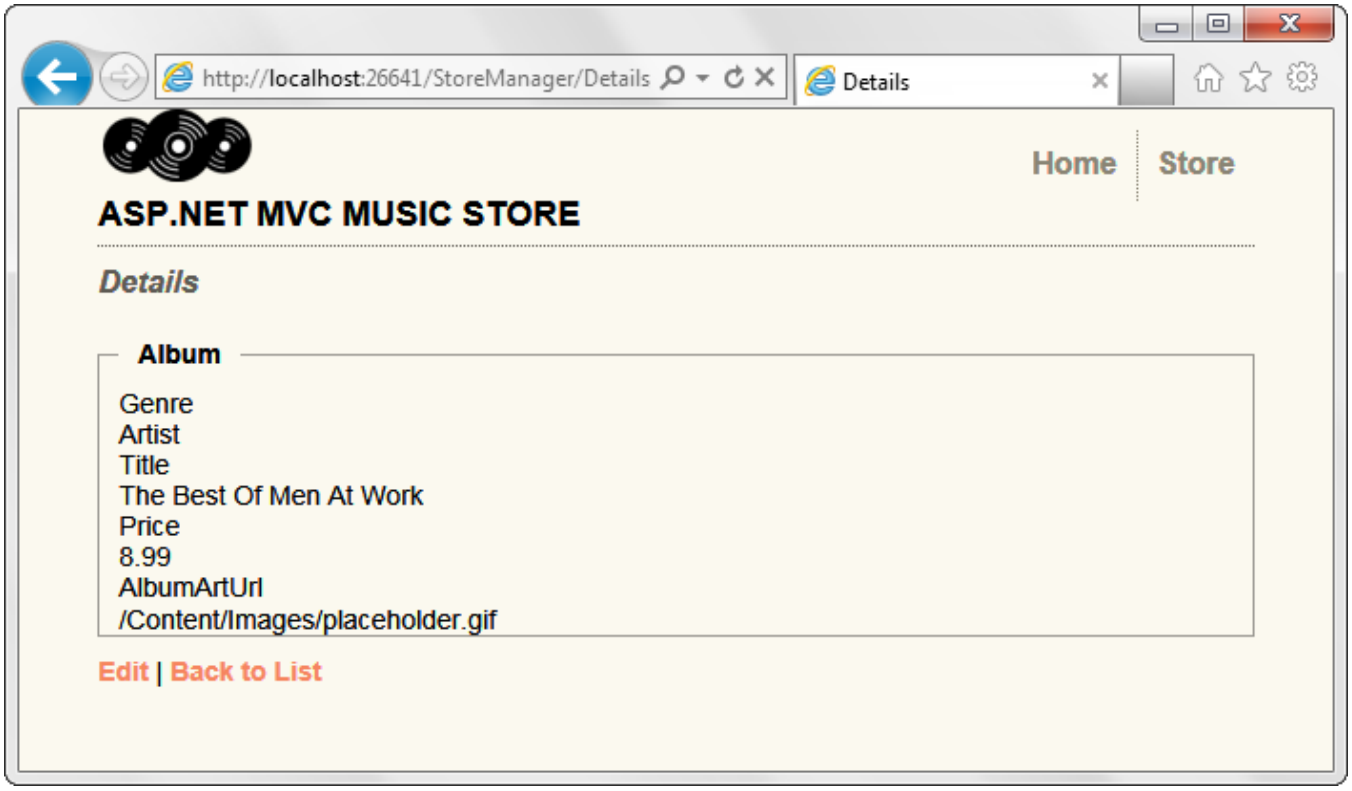
Now run the application and browse to /StoreManager/. This displays the Store Manager Index we just modified, showing a list of the albums in the store with links to Edit, Details, and Delete.



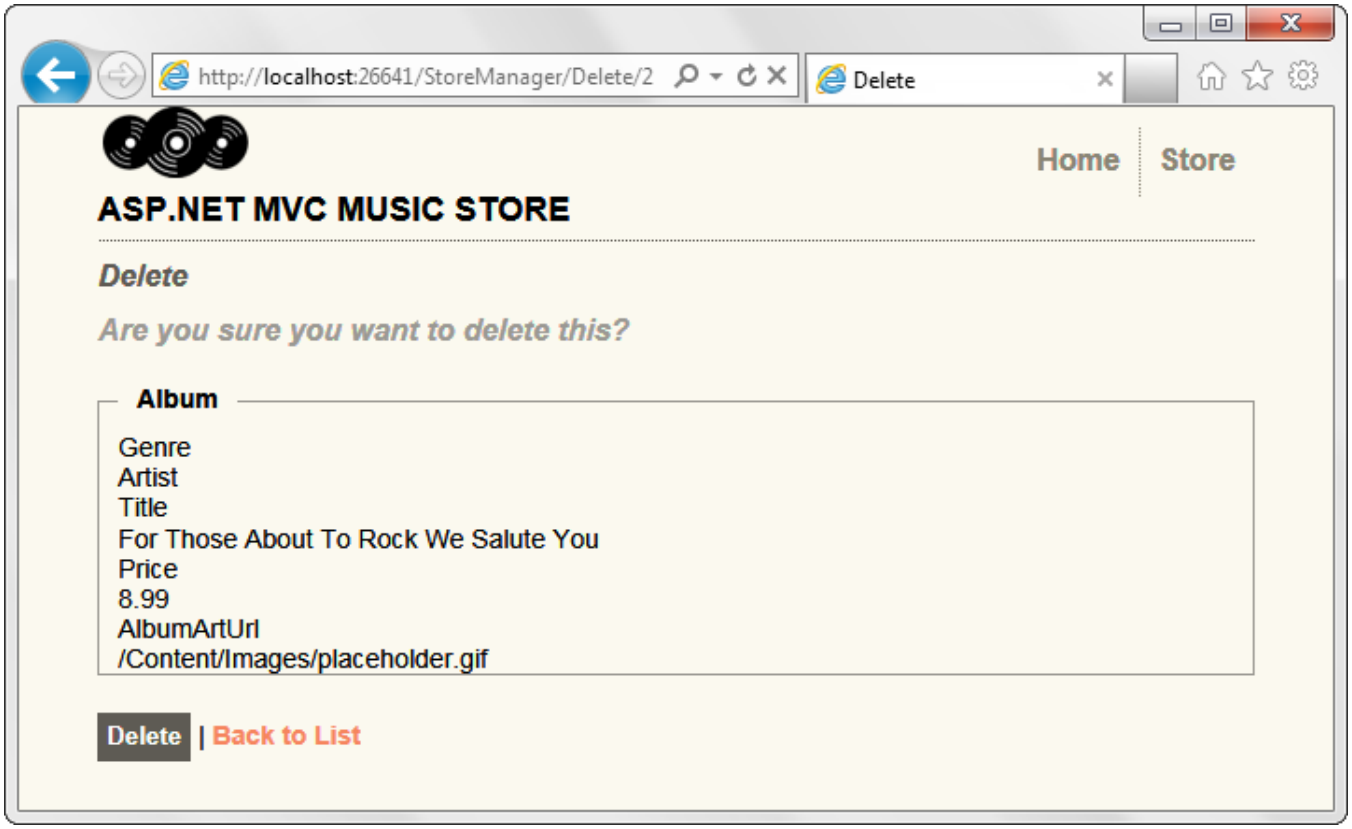
Clicking the Edit link displays an edit form with fields for the Album, including dropdowns for Genre and Artist.



Click the "Back to List" link at the bottom, then click on the Details link for an Album. This displays the detail information for an individual Album.



Again, click the Back to List link, then click on a Delete link. This displays a confirmation dialog, showing the album details and asking if we're sure we want to delete it.



Clicking the Delete button at the bottom will delete the album and return you to the Index page, which shows the album deleted.

We're not done with the Store Manager, but we have working controller and view code for the CRUD operations to start from.

Looking at the Store Manager Controller code

The Store Manager Controller contains a good amount of code. Let's go through this from top to bottom. The controller includes some standard namespaces for an MVC controller, as well as a reference to our Models namespace. The controller has a private instance of MusicStoreEntities, used by each of the controller actions for data access.

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

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

```
        private MusicStoreEntities db = new MusicStoreEntities();
```

Store Manager Index and Details actions

The index view retrieves a list of Albums, including each album's referenced Genre and Artist information, as we previously saw when working on the Store Browse method. The Index view is following the references to the linked objects so that it can display each album's Genre name and Artist name, so the controller is being efficient and querying for this information in the original request.

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

public ViewResult Index()
{
  var albums = db.Albums.Include(a => a.Genre).Include(a => a.Artist);
  return View(albums.ToList());
}
```

The StoreManager Controller's Details controller action works exactly the same as the Store Controller Details action we wrote previously - it queries for the Album by ID using the Find() method, then returns it to the view.

```
  //
  // GET: /StoreManager/Details/5

  public ViewResult Details(int id)
  {
    Album album = db.Albums.Find(id);
    return View(album);
  }
```

The Create Action Methods

The Create action methods are a little different from ones we've seen so far, because they handle form input. When a user first visits /StoreManager/Create/ they will be shown an empty form. This HTML page will contain a <form> element that contains dropdown and textbox input elements where they can enter the album's details.

After the user fills in the Album form values, they can 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/Create/ 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 "Create" action methods within our StoreManagerController class – one to handle the initial HTTP-GET browse to the /StoreManager/Create/ URL, and the other to handle the HTTP-POST of the submitted changes.

Passing information to a View using ViewBag

We've used the ViewBag earlier in this tutorial, but haven't talked much about it. The ViewBag allows us to pass information to the view without using a strongly typed model object. In this case, our Edit HTTP-GET controller

Tutorial under Creative Commons Attribution 3.0 License

action needs to pass both a list of Genres and Artists to the form to populate the dropdowns, and the simplest way to do that is to return them as ViewBag items.

The ViewBag is a dynamic object, meaning that you can type ViewBag.Foo or ViewBag.YourNameHere without writing code to define those properties. In this case, the controller code uses ViewBag.GenreId and ViewBag.ArtistId so that the dropdown values submitted with the form will be GenreId and ArtistId, which are the Album properties they will be setting.

These dropdown values are returned to the form using the SelectList object, which is built just for that purpose. This is done using code like this:

```
ViewBag.GenreId = new SelectList(db.Genres, "GenreId", "Name");
```

As you can see from the action method code, three parameters are being used to create this object:

- The list of items the dropdown will be displaying. Note that this isn't just a string - we're passing a list of Genres.
- The next parameter being passed to the SelectList is the Selected Value. This how the SelectList knows how to pre-select an item in the list. This will be easier to understand when we look at the Edit form, which is pretty similar.
- The final parameter is the property to be displayed. In this case, this is indicating that the Genre.Name property is what will be shown to the user.

With that in mind, then, the HTTP-GET Create action is pretty simple - two SelectLists are added to the ViewBag, and no model object is passed to the form (since it hasn't been created yet).

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

public ActionResult Create()
{
    ViewBag.GenreId = new SelectList(db.Genres, "GenreId", "Name");
    ViewBag.ArtistId = new SelectList(db.Artists, "ArtistId", "Name");
    return View();
}
```

HTML Helpers to display the Drop Downs in the Create View

Since we've talked about how the drop down values are passed to the view, let's take a quick look at the view to see how those values are displayed. In the view code (/Views/StoreManager/Create.cshtml), you'll see the following call is made to display the Genre drop down.

```
@Html.DropDownList("GenreId", String.Empty)
```


Tutorial under Creative Commons Attribution 3.0 License

This is known as an HTML Helper - a utility method which performs a common view task. HTML Helpers are very useful in keeping our view code concise and readable. The Html.DropDownList helper is provided by ASP.NET MVC, but as we'll see later it's possible to create our own helpers for view code we'll reuse in our application.

The Html.DropDownList call just needs to be told two things - where to get the list to display, and what value (if any) should be pre-selected. The first parameter, GenreId, tells the DropDownList to look for a value named GenreId in either the model or ViewBag. The second parameter is used to indicate the value to show as initially selected in the drop down list. Since this form is a Create form, there's no value to be preselected and String.Empty is passed.

Handling the Posted Form values

As we discussed before, there are two action methods associated with each form. The first handles the HTTP-GET request and displays the form. The second handles the HTTP-POST request, which contains the submitted form values. Notice that controller action has an [HttpPost] attribute, which tells ASP.NET MVC that it should only respond to HTTP-POST requests.

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

[HttpPost]
public ActionResult Create(Album album)
{
    if (ModelState.IsValid)
    {
        db.Albums.Add(album);
        db.SaveChanges();
        return RedirectToAction("Index");
    }

    ViewBag.GenreId = new SelectList(db.Genres, "GenreId", "Name", album.GenreId);
    ViewBag.ArtistId = new SelectList(db.Artists, "ArtistId", "Name", album.ArtistId);
    return View(album);
}
```

This action has four responsibilities:

1. Read the form values
2. Check if the form values pass any validation rules
3. If the form submission is valid, save the data and display the updated list
4. If the form submission is not valid, redisplay the form with validation errors

Reading Form Values with Model Binding

The controller action is processing a form submission that includes values for GenreId and ArtistId (from the drop down list) and textbox values for Title, Price, and AlbumArtUrl. While it's possible to directly access form values, a better approach is to use the Model Binding capabilities built into ASP.NET MVC. When a controller action takes a model type as a parameter, ASP.NET MVC will attempt to populate an object of that type using form inputs (as well as route and querystring values). It does this by looking for values whose names match properties of the model object, e.g. when setting the new Album object's GenreId value, it looks for an input

Tutorial under Creative Commons Attribution 3.0 License

with the name GenreId. When you create views using the standard methods in ASP.NET MVC, the forms will always be rendered using property names as input field names, so this the field names will just match up.

Validating the Model

The model is validated with a simple call to ModelState.IsValid. We haven't added any validation rules to our Album class yet - we'll do that in a bit - so right now this check doesn't have much to do. What's important is that this ModelStat.IsValid check will adapt to the validation rules we put on our model, so future changes to validation rules won't require any updates to the controller action code.

Saving the submitted values

If the form submission passes validation, it's time to save the values to the database. With Entity Framework, that just requires adding the model to the Albums collection and calling SaveChanges.

```
db.Albums.Add(album);
db.SaveChanges();
```

Entity Framework generates the appropriate SQL commands to persist the value. After saving the data, we redirect back to the list of Albums so we can see our update. This is done by returning RedirectToAction with the name of the controller action we want displayed. In this case, that's the Index method.

Displaying invalid form submissions with Validation Errors

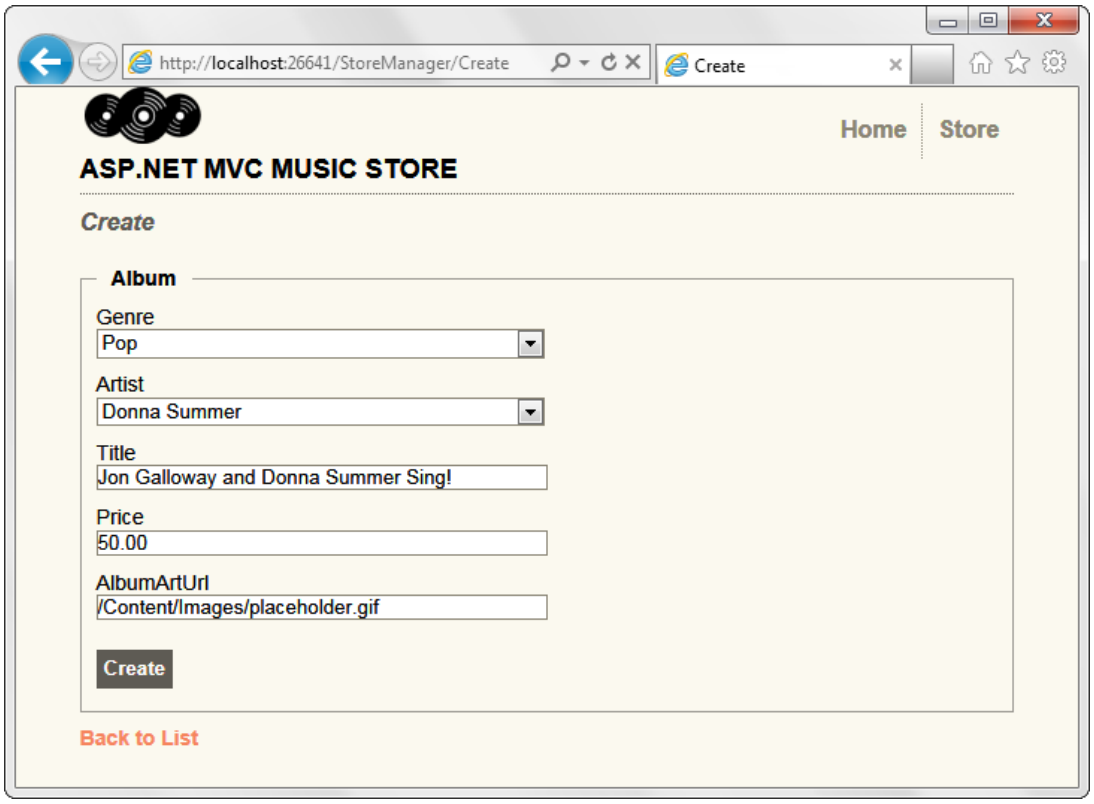
In the case of invalid form input, the dropdown values are added to the ViewBag (as in the HTTP-GET case) and the bound model values are passed back to the view for display. Validation errors are automatically displayed using the @Html.ValidationMessageFor HTML Helper.

Testing the Create Form

To test this out, run the application and browse to /StoreManager/Create/ - this will show you the blank form which was returned by the StoreController Create HTTP-GET method.

Fill in some values and click the Create button to submit the form.

Tutorial under Creative Commons Attribution 3.0 License



Jazz	Miles Davis	Miles Ahead	8.99	Edit | Details | Delete
Jazz	Miles Davis	The Essential Miles Davis [Disc 1]	8.99	Edit | Details | Delete
Jazz	Miles Davis	The Essential Miles Davis [Disc 2]	8.99	Edit | Details | Delete
Jazz	Spyro Gyra	Heart of the Night	8.99	Edit | Details | Delete
Jazz	Spyro Gyra	Morning Dance	8.99	Edit | Details | Delete
Pop	Donna Summer	Jon Galloway and Donna Summer Sing!	50.00	Edit | Details | Delete
Pop	Amy Winehouse	Frank	8.99	Edit | Details | Delete

Handling Edits

The Edit action pair (HTTP-GET and HTTP-POST) are very similar to the Create action methods we just looked at. Since the edit scenario involves working with an existing album, the Edit HTTP-GET method loads the Album based on the "id" parameter, passed in via the route. This code for retrieving an album by AlbumId is the same as we've previously looked at in the Details controller action. As with the Create / HTTP-GET method, the drop down values are returned via the ViewBag. This allows us to return an Album as our model object to the view (which is strongly typed to the Album class) while passing additional data (e.g. a list of Genres) via the ViewBag.

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

public ActionResult Edit(int id)
{
    Album album = db.Albums.Find(id);
    ViewBag.GenreId = new SelectList(db.Genres, "GenreId", "Name", album.GenreId);
    ViewBag.ArtistId = new SelectList(db.Artists, "ArtistId", "Name", album.ArtistId);
    return View(album);
```

```
}
```

The Edit HTTP-POST action is very similar to the Create HTTP-POST action. The only difference is that instead of adding a new album to the db.Albums collection, we're finding the current instance of the Album using db.Entry(album) and setting its state to Modified. This tells Entity Framework that we are modifying an existing album as opposed to creating a new one.

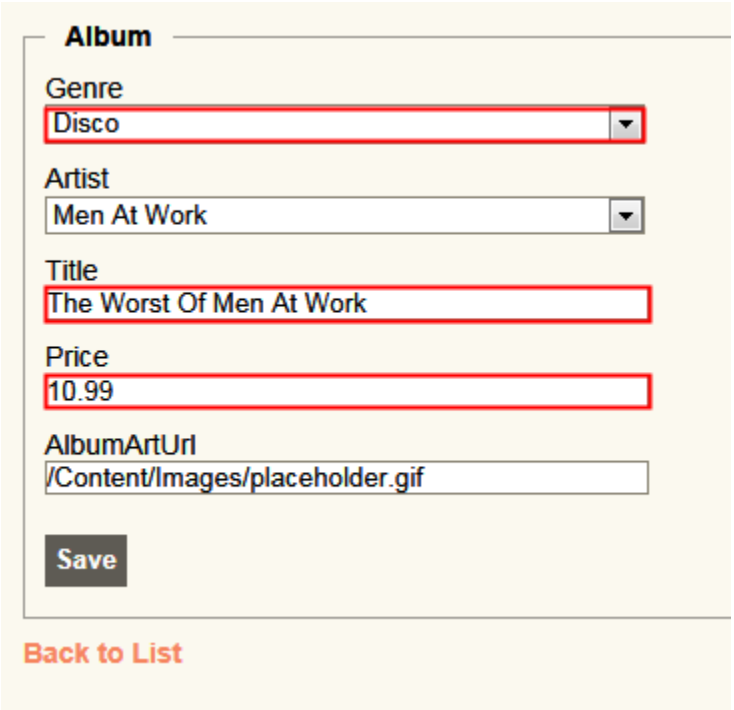
```
//
// POST: /StoreManager/Edit/5

[HttpPost]
public ActionResult Edit(Album album)
{
    if (ModelState.IsValid)
    {
        db.Entry(album).State = EntityState.Modified;
        db.SaveChanges();
        return RedirectToAction("Index");
    }
    ViewBag.GenreId = new SelectList(db.Genres, "GenreId", "Name", album.GenreId);
    ViewBag.ArtistId = new SelectList(db.Artists, "ArtistId", "Name", album.ArtistId);
    return View(album);
}
```

We can test this out by running the application and browsing to /StoreManger/, then clicking the Edit link for an album.

Genre	Artist	Title	Price	
Rock	Men At Work	The Best Of Men At Work	8.99	Edit | Details | Delete
Rock	AC/DC	For Those About To Rock We Salute You	8.[illegible]	Edit | Details | Delete
Rock	AC/DC	Let There Be Rock	8.99	Edit | Details | Delete
Rock	Accept	Balls to the Wall	8.99	Edit | Details | Delete
Rock	Accept	Restless and Wild	8.99	Edit | Details | Delete
Rock	Aerosmith	Big Ones	8.99	Edit | Details | Delete

This displays the Edit form shown by the Edit HTTP-GET method. Fill in some values and click the Save button.

Tutorial under Creative Commons Attribution 3.0 License



This posts the form, saves the values, and returns us to the Album list, showing that the values were updated.

Jazz	Spyro Gyra	Heart of the Night	8.99	Edit | Details | Delete
Jazz	Spyro Gyra	Morning Dance	8.99	Edit | Details | Delete
Pop	Amy Winehouse	Frank	8.99	Edit | Details | Delete
Pop	Various Artists	Axé Bahia 2001	8.99	Edit | Details | Delete
Disco	Men At Work	The Worst Of Men At Work	10.99	Edit | Details | Delete
Disco	Anita Ward	Ring My Bell	8.99	Edit | Details | Delete

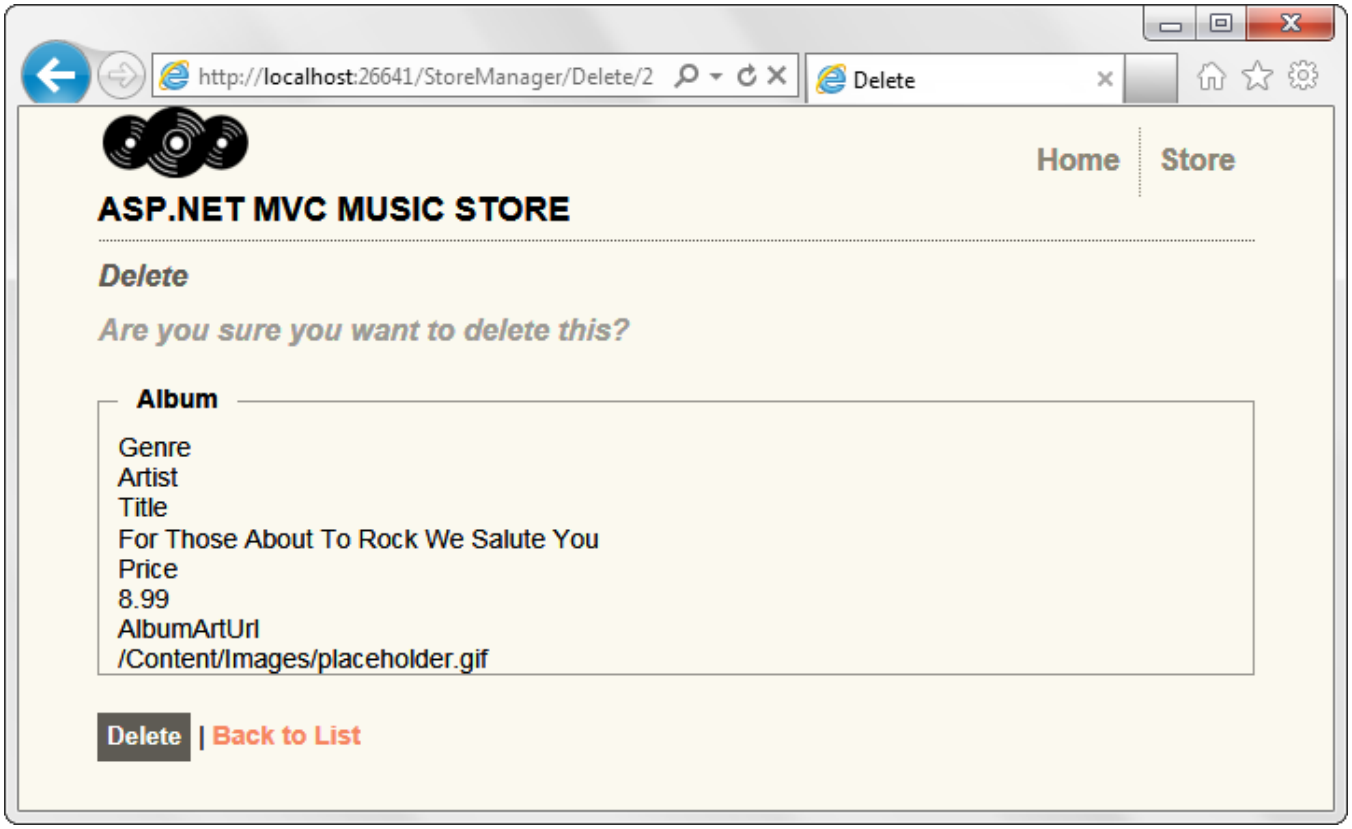
Handling Deletion

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 Manager Details controller action.

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

public ActionResult Delete(int id)
{
    Album album = db.Albums.Find(id);
    return View(album);
}
```

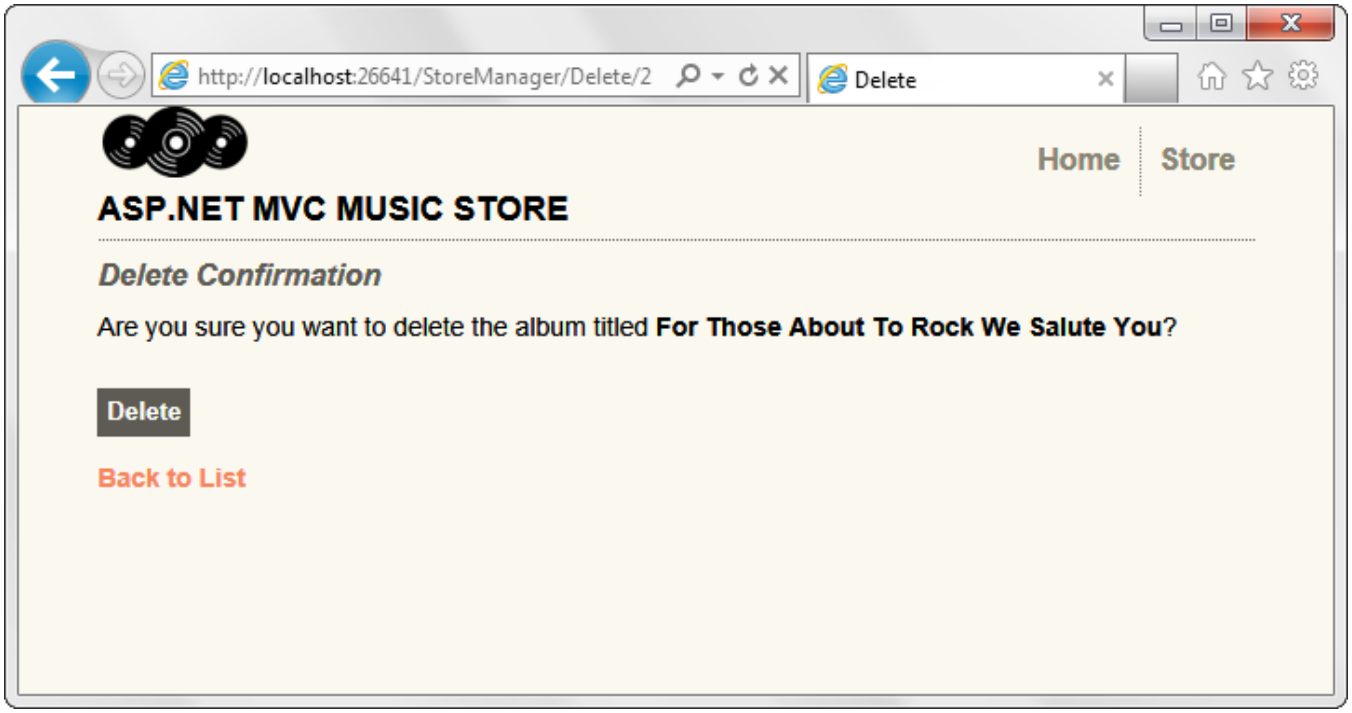
We display a form that's strongly typed to an Album type, using the Delete view content template.



The Delete template shows all the fields for the model, but we can simplify that down quite a bit. Change the view code in /Views/StoreManager/Delete.cshtml to the following.

```
@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>

}
```

This displays a simplified Delete confirmation.



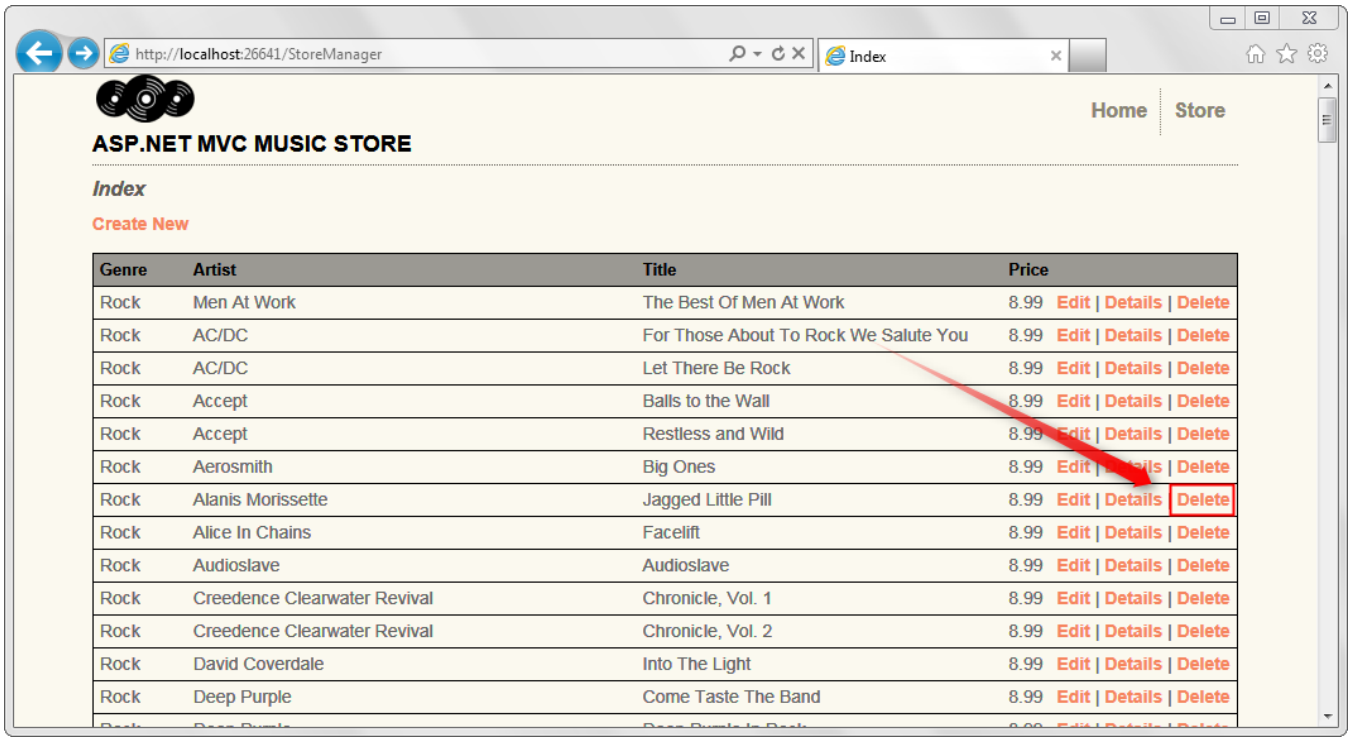
Clicking the Delete button causes the form to be posted back to the server, which executes the DeleteConfirmed action.

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

[HttpPost, ActionName("Delete")]
public ActionResult DeleteConfirmed(int id)
{
    Album album = db.Albums.Find(id);
    db.Albums.Remove(album);
    db.SaveChanges();
    return RedirectToAction("Index");
}
```

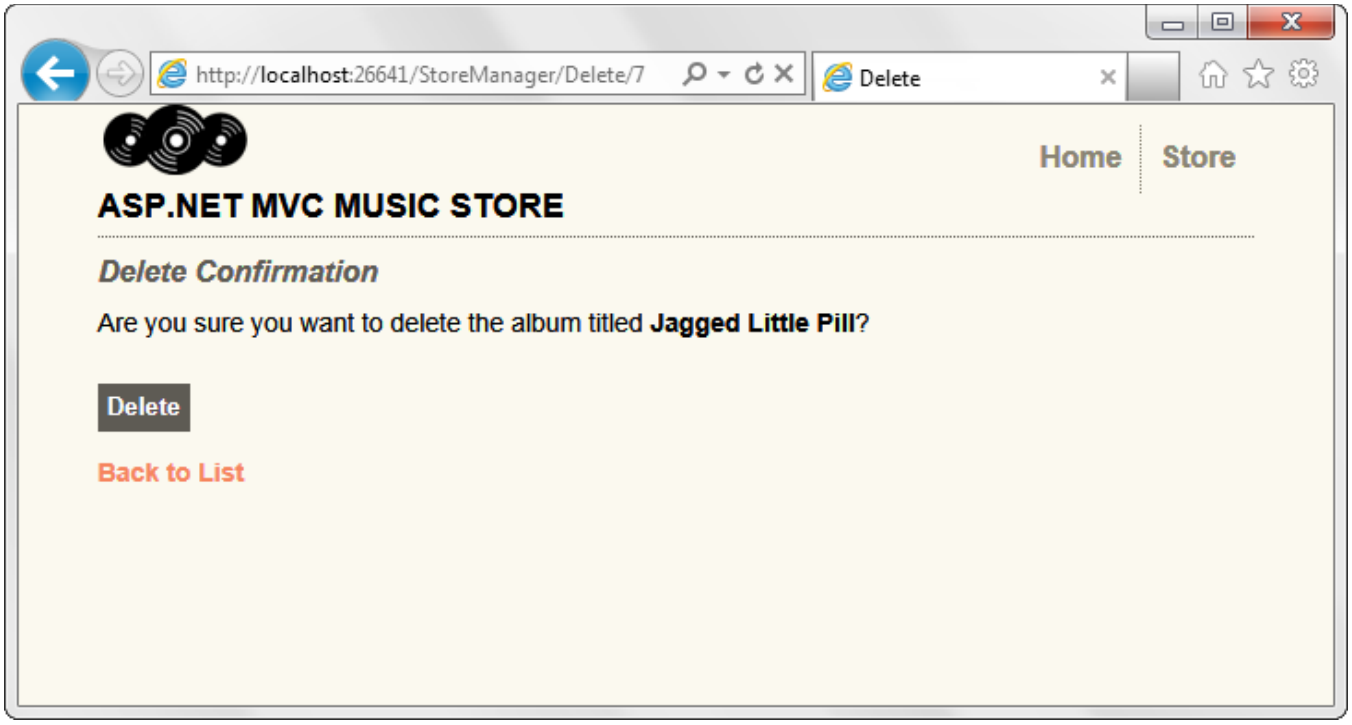
Our HTTP-POST Delete Controller Action takes the following actions:

1. Loads the Album by ID
2. Deletes it the album and save changes
3. Redirects to the Index, showing that the Album was removed from the list

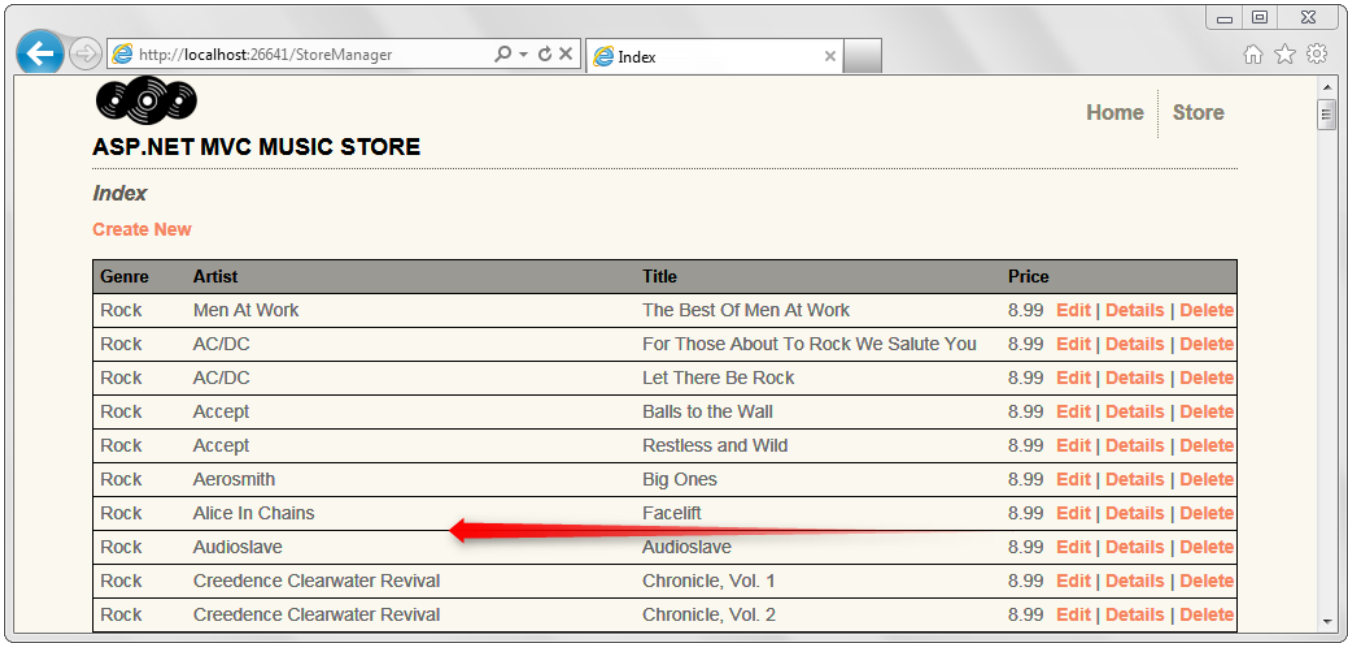
To test this, run the application and browse to /StoreManager. Select an album from the list and click the Delete link.



This displays our Delete confirmation screen.

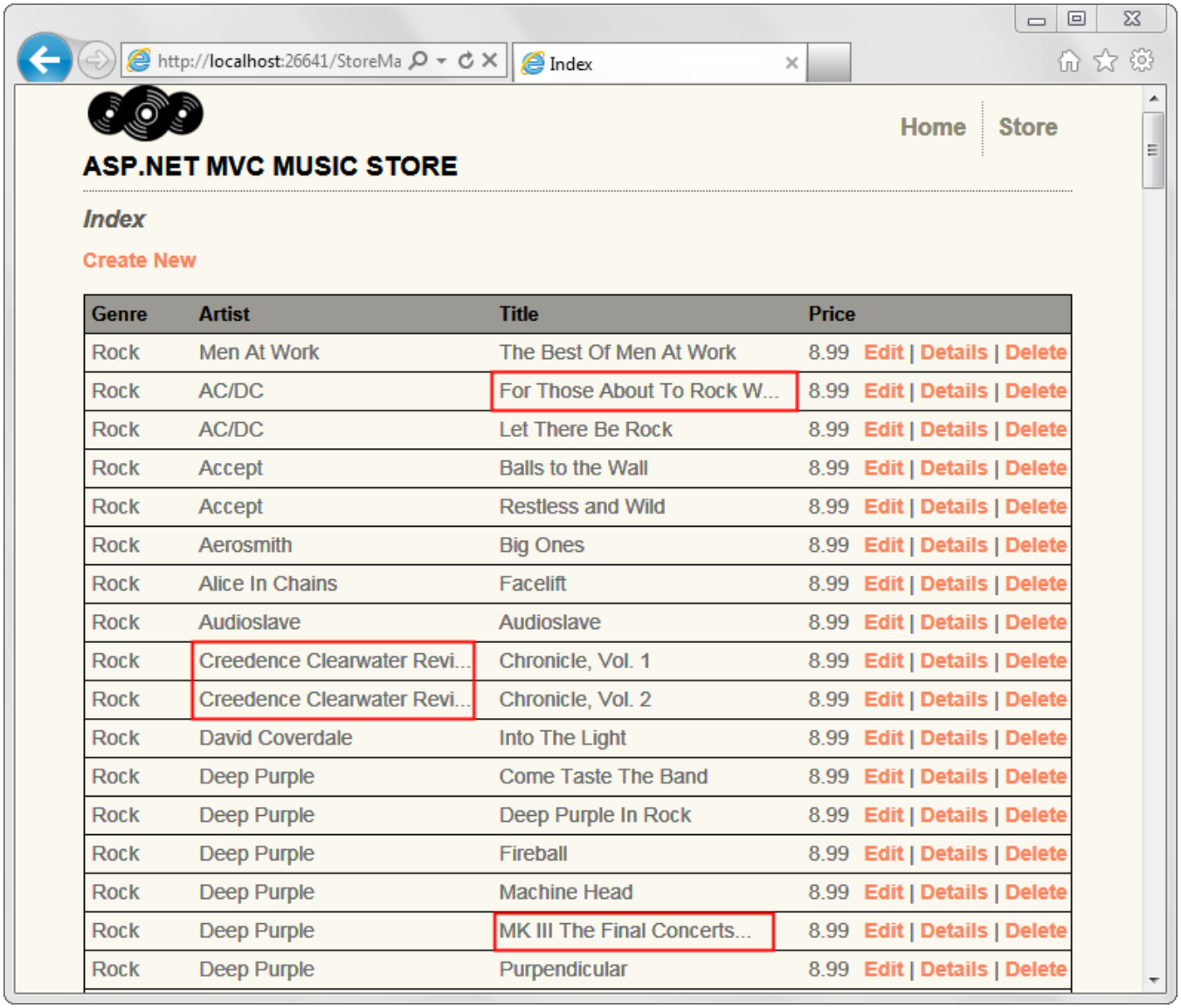


Clicking the Delete button removes the album and returns us to the Store Manager Index page, which shows that the album has been deleted.



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.



Razor's @helper syntax has made it pretty easy to create your own helper functions for use in your views. Open the /Views/StoreManager/Index.cshtml view and add the following code directly after the @model line.

```
@helper Truncate(string input, int length)
{
    if (input.Length <= length) {
        @input
    } else {
        @input.Substring(0, length)<text>...</text>
    }
}
```

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.

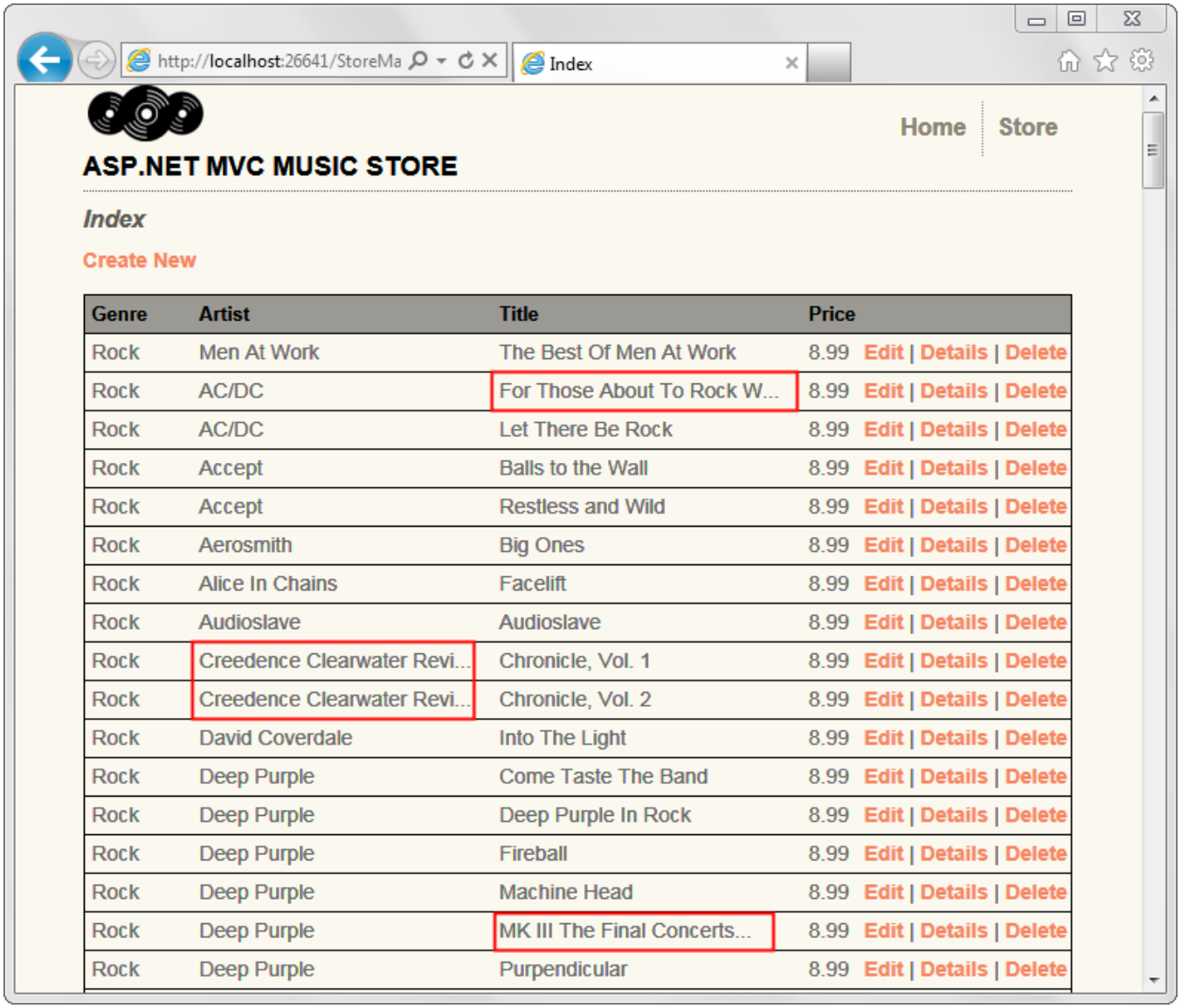
Now we can use our Truncate helper to ensure that both the Album Title and Artist Name properties are less than 25 characters. The complete view code using our new Truncate helper appears below.

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

@helper Truncate(string input, int length)
{
    if (input.Length <= length) {
        @input
    } else {
        @input.Substring(0, length)<text>...</text>
    }
}

@{
    ViewBag.Title = "Index";
}

<h2>Index</h2>

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

@foreach (var item in Model) {
    <tr>
        <td>
            @Html.DisplayFor(modelItem => item.Genre.Name)
        </td>
        <td>
            @Truncate(item.Artist.Name, 25)
        </td>
        <td>
            @Truncate(item.Title, 25)
        </td>
        <td>
            @Html.DisplayFor(modelItem => item.Price)
        </td>
```


Tutorial under Creative Commons Attribution 3.0 License

```
        <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>
    </tr>
}

</table>
```

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



Note: This shows the simple case of creating and using a helper in one view. To learn more about creating helpers that you can use throughout your site, see my blog post: http://bit.ly/mvc3-helper-options

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; }
```


Tutorial under Creative Commons Attribution 3.0 License

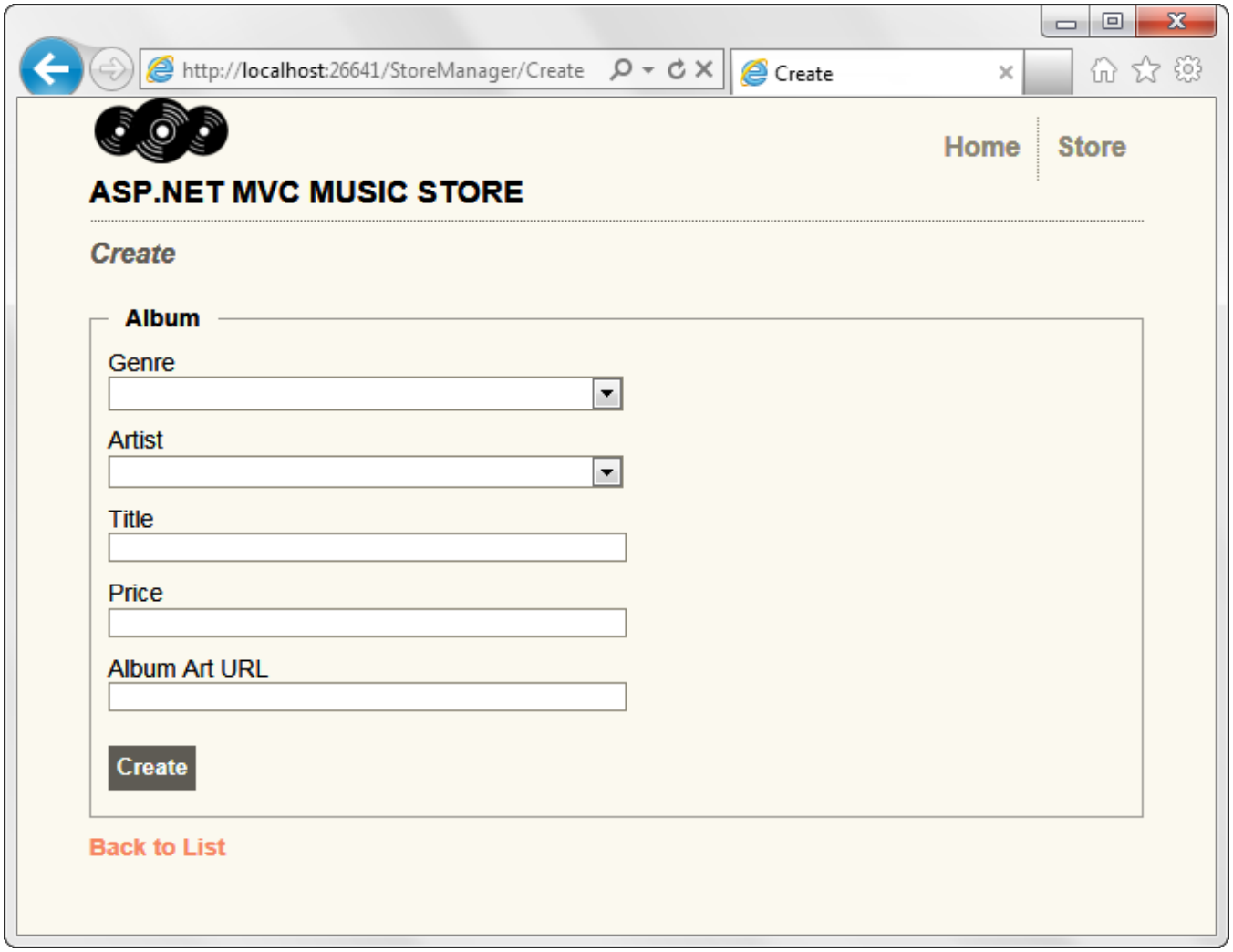
```
        [DisplayName("Album Art URL")]
        [StringLength(1024)]
        public string AlbumArtUrl { get; set; }

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

    }
}
```

While we're there, we've also changed the Genre and Artist to virtual properties. This allows Entity Framework to lazy-load them as necessary.

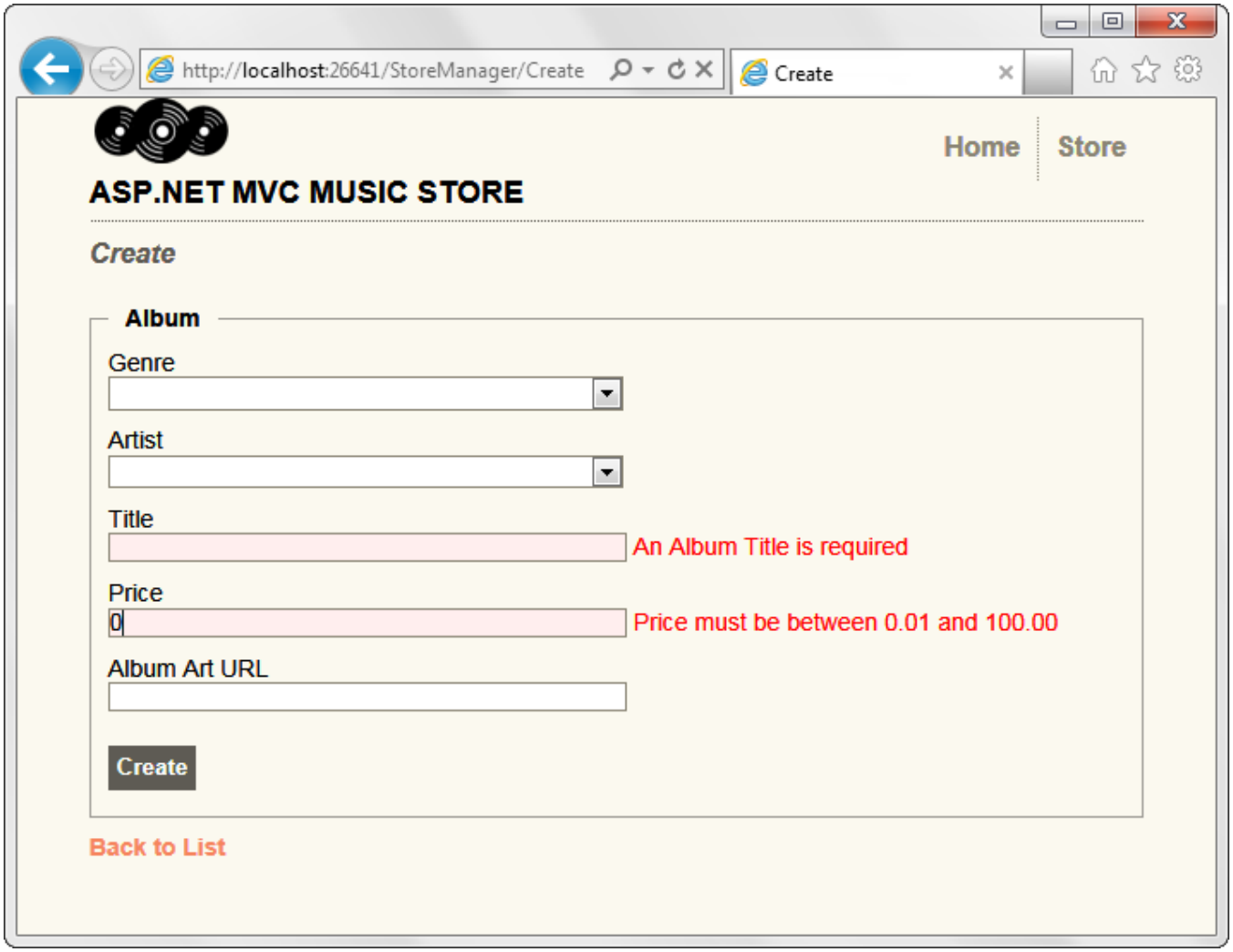
```
public virtual Genre    Genre       { get; set; }
public virtual Artist   Artist      { get; set; }
```

After having added these attributes to our Album model, 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.

Tutorial under Creative Commons Attribution 3.0 License



Next, we'll break some validation rules. Enter a price of 0 and leave the Title blank. When we click on the Create button, we will see the form displayed with validation error messages showing which fields did not meet the validation rules we have defined.



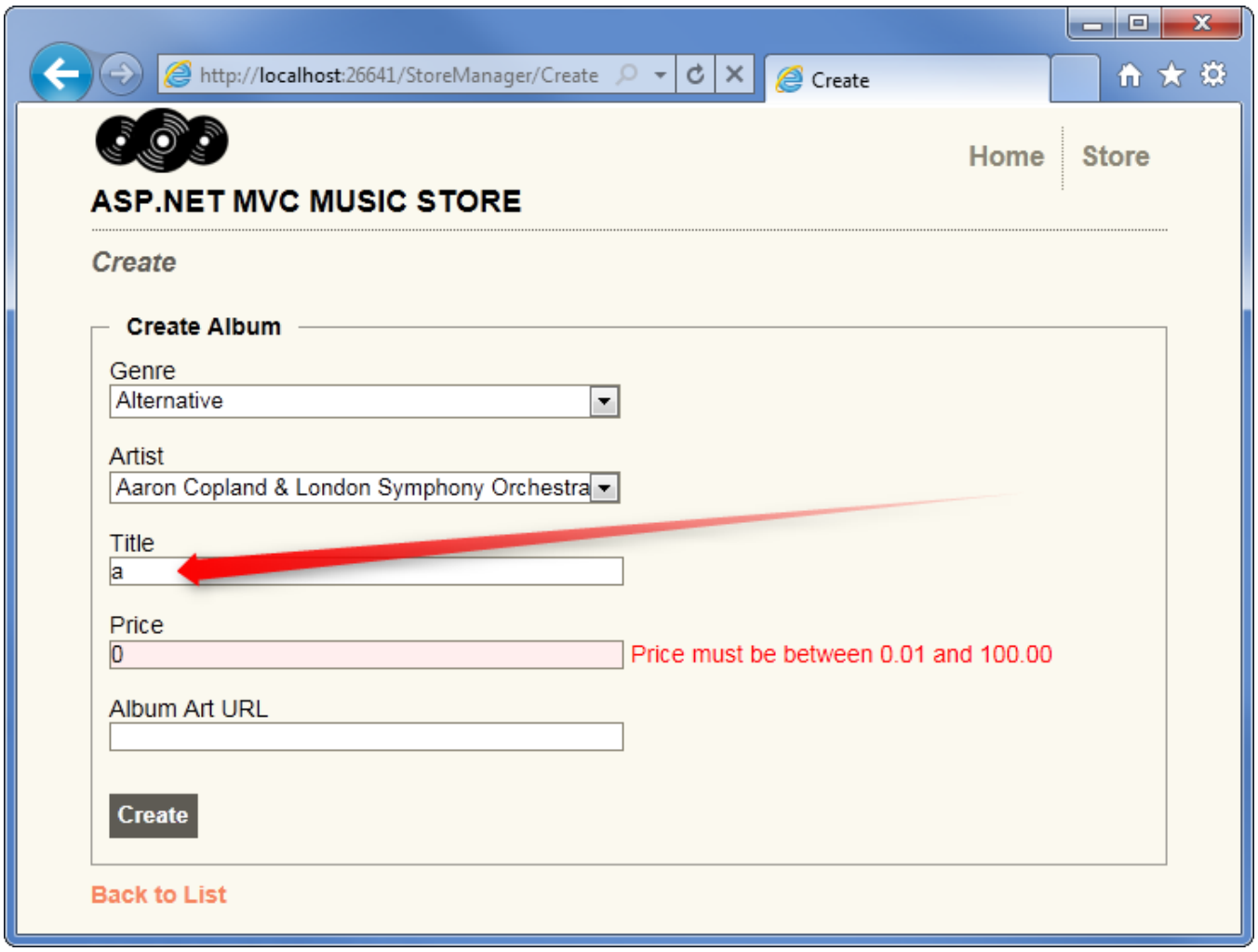
Testing the Client-Side Validation

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:

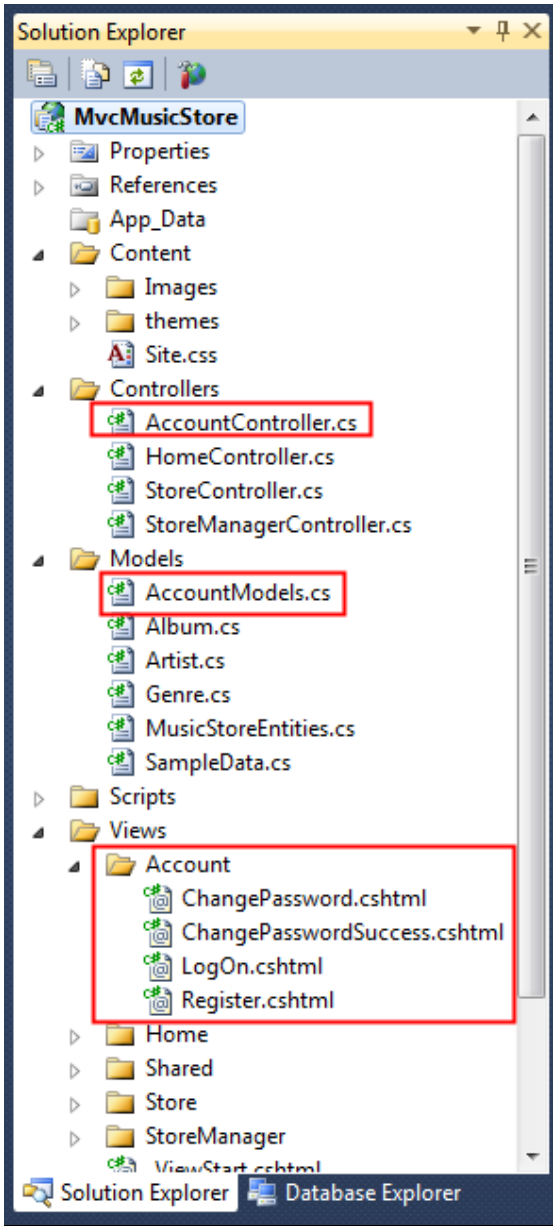
1. Copy AccountController.cs in the Controllers directory
2. Copy AccountModels.cs in the Models directory
3. Create an Account directory inside the Views directory and copy all four views in

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:

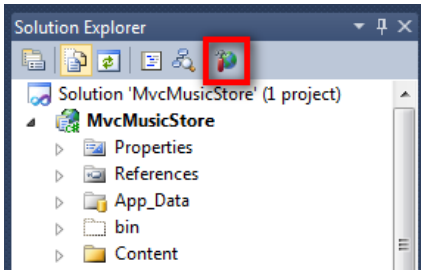
Tutorial under Creative Commons Attribution 3.0 License



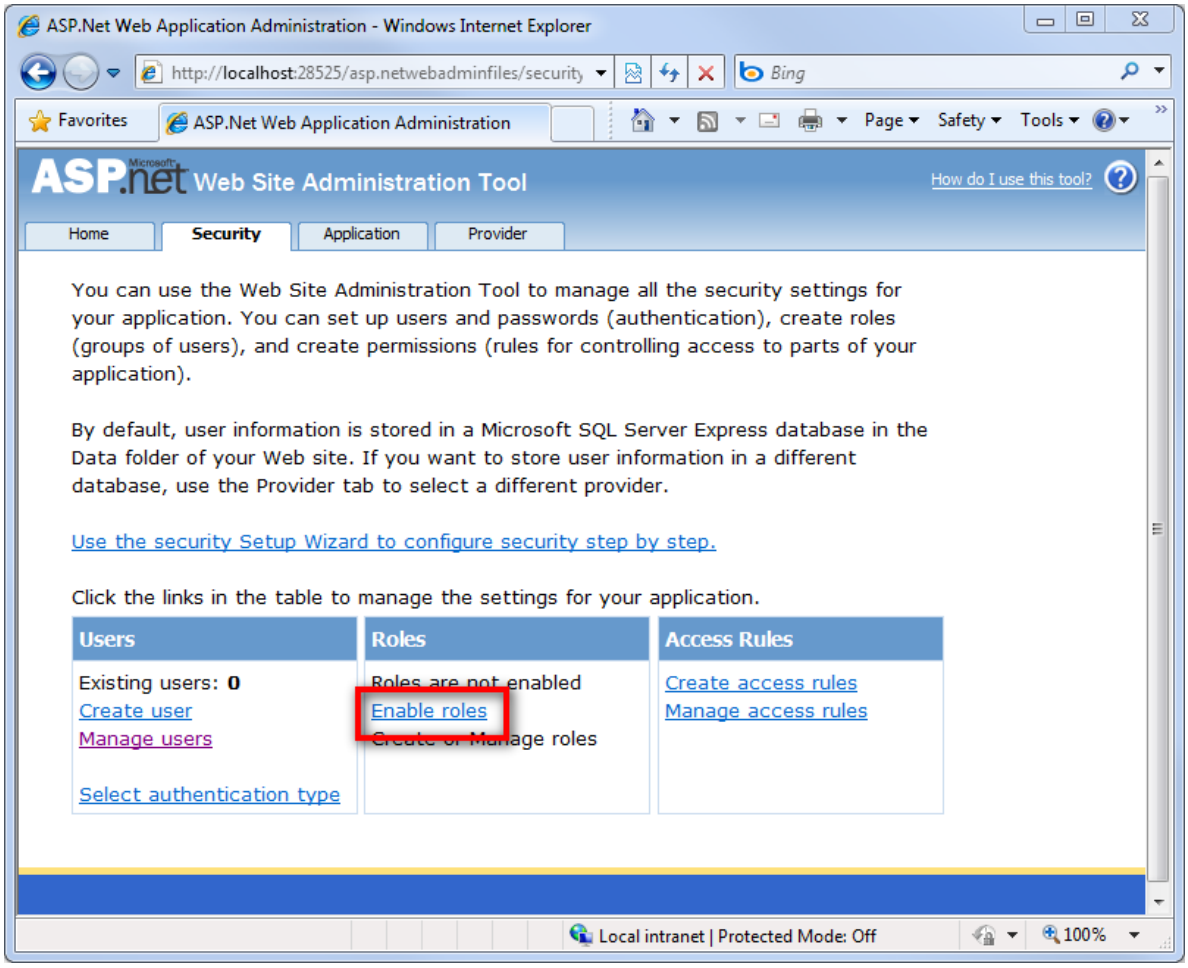
Adding an Administrative User with the ASP.NET Configuration site

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.

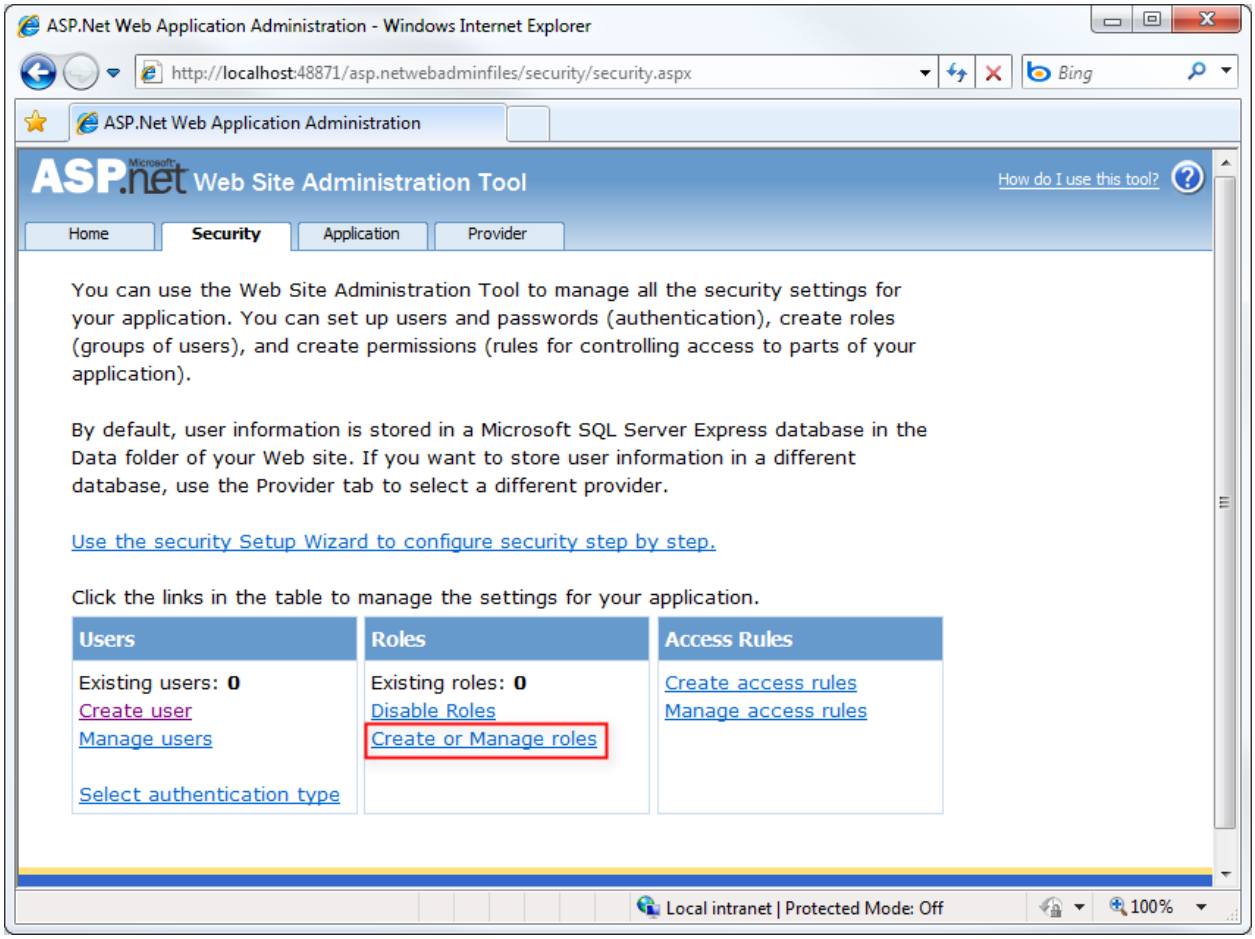
Launch the ASP.NET Configuration website by clicking following the icon in the Solution Explorer.



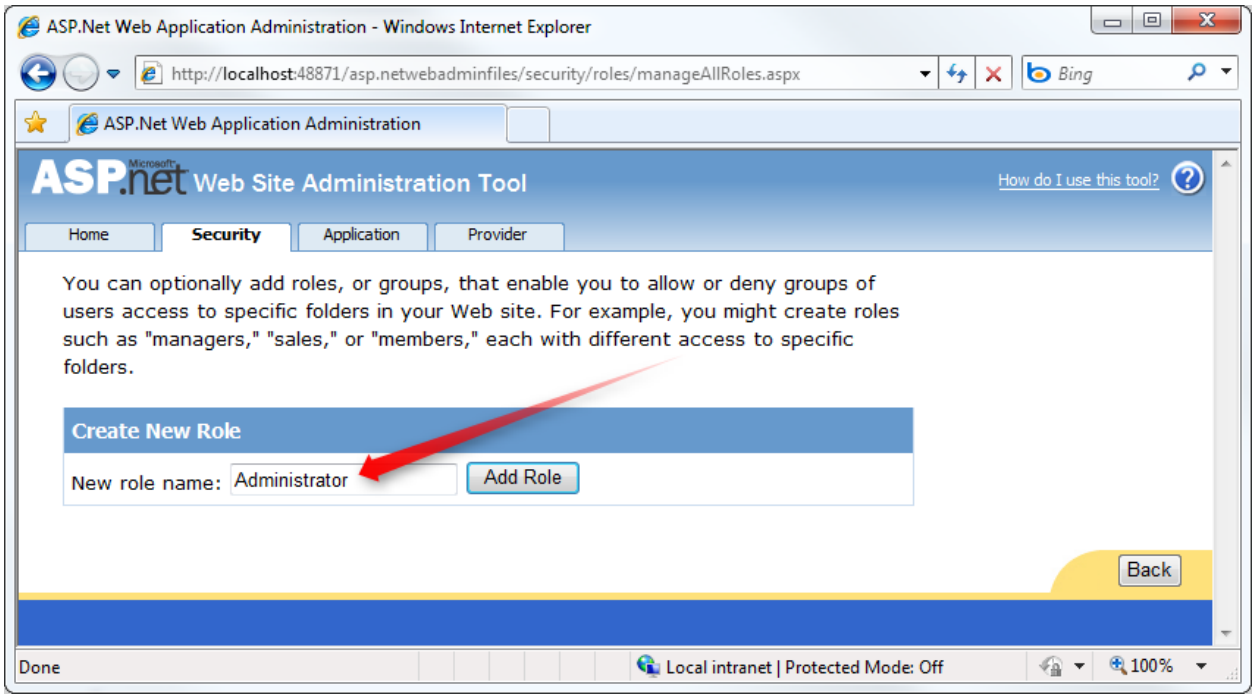
This launches a configuration website. Click on the Security tab on the home screen, then click the "Enable roles" link in the center of the screen.



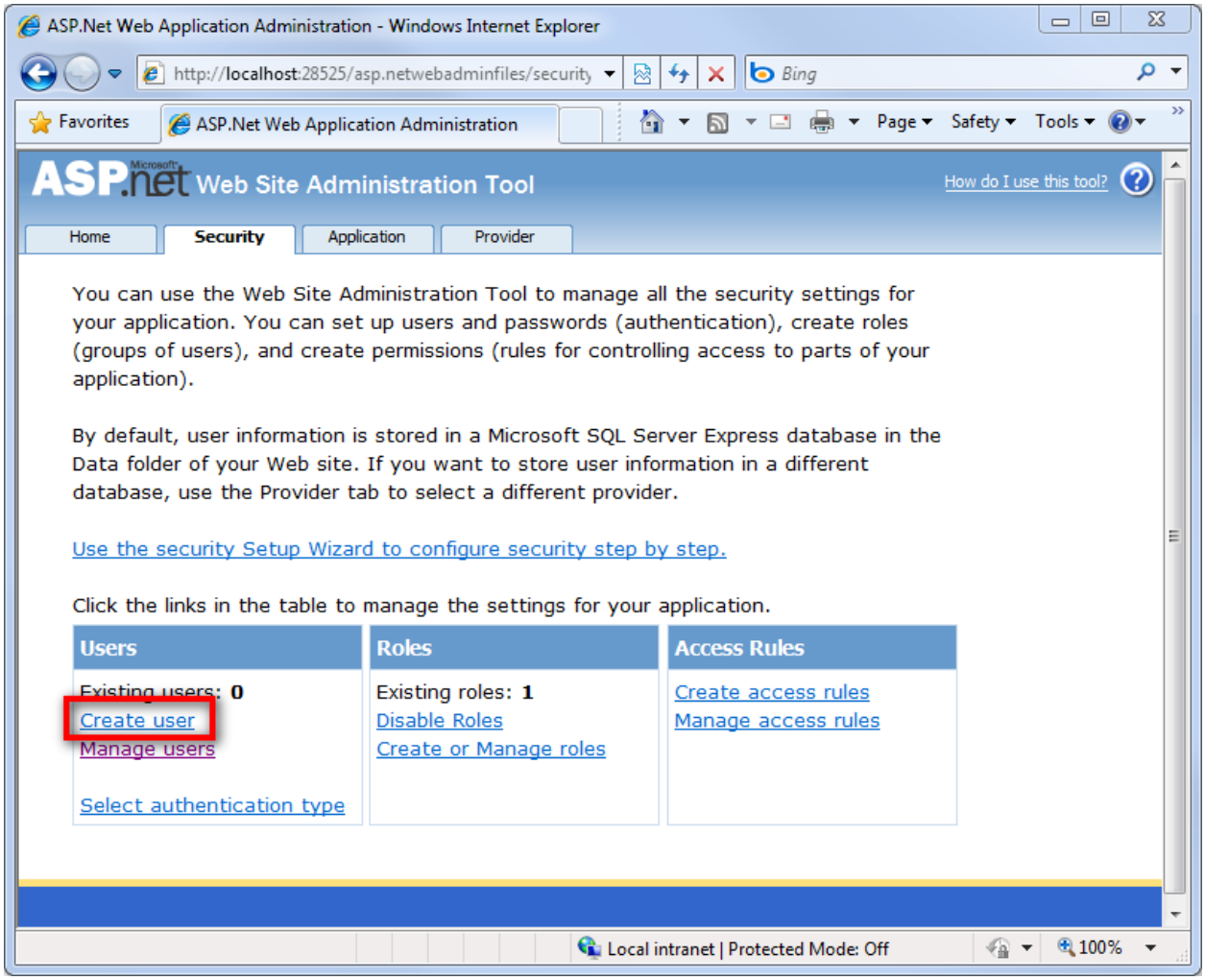
Click the "Create or Manage roles" link.



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.



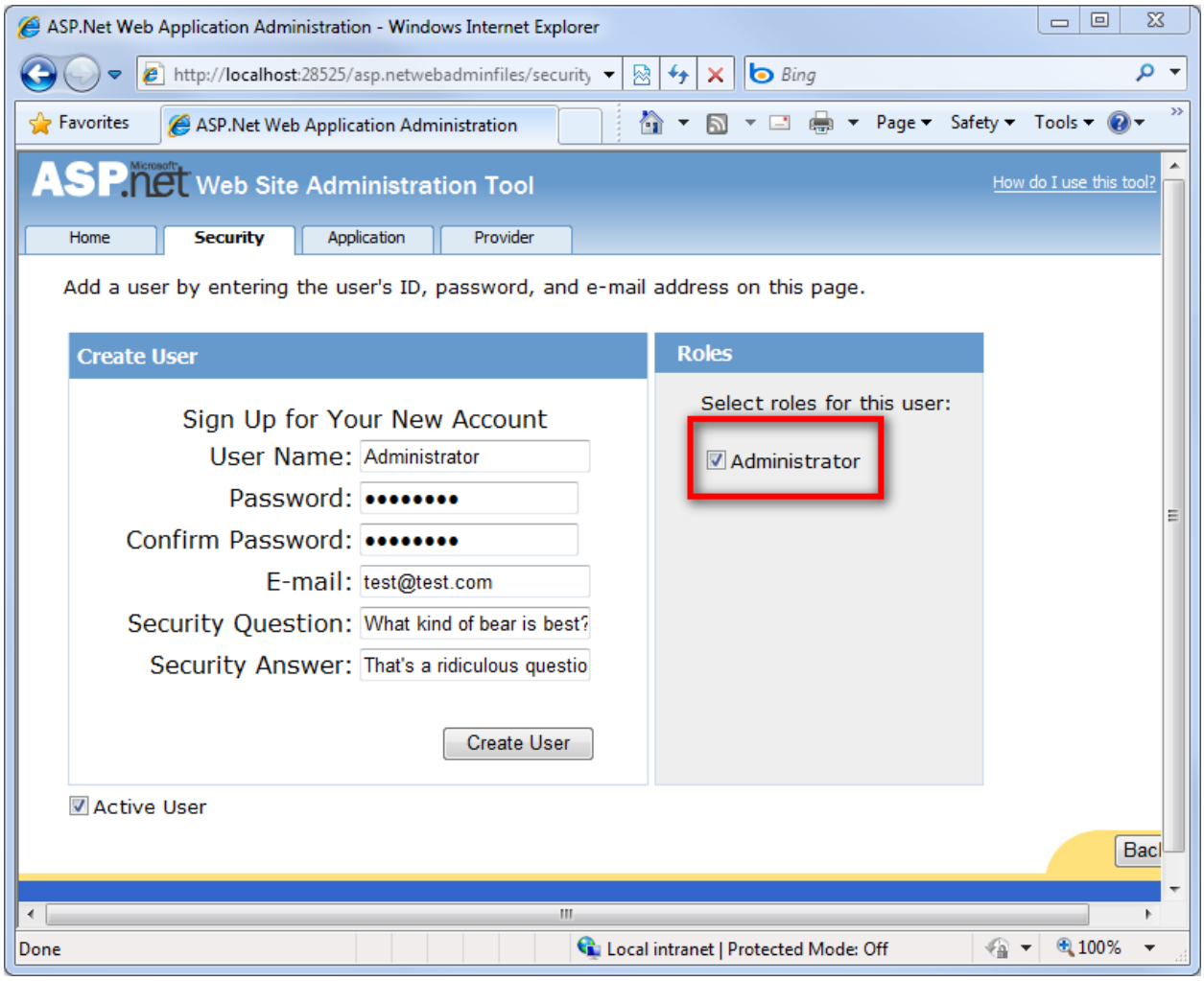
Fill in the user information fields on the left using the following information:

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)

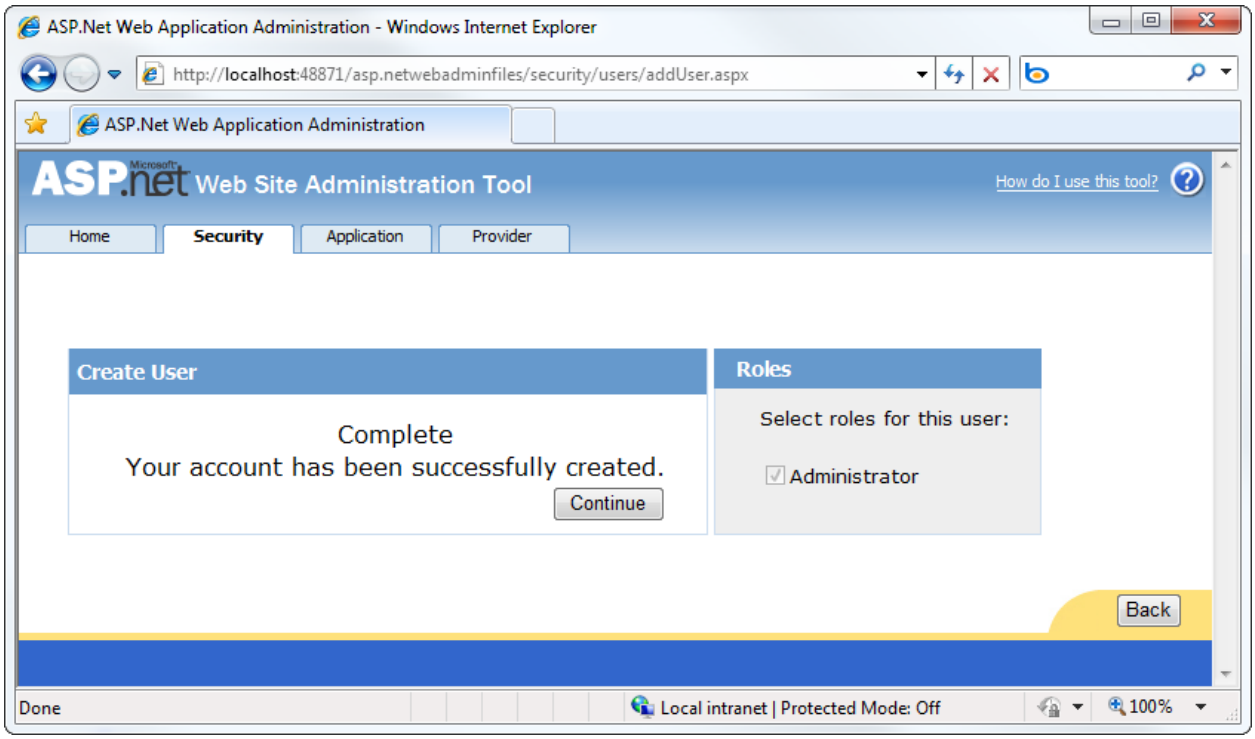
Note: You can of course use any password you'd like. The above password is shown as an example, and is assumed in the support forums on CodePlex. The default password security settings require a password that is 7 characters long and contains one non-alphanumeric character.

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

Tutorial under Creative Commons Attribution 3.0 License



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



You can now close the browser window.

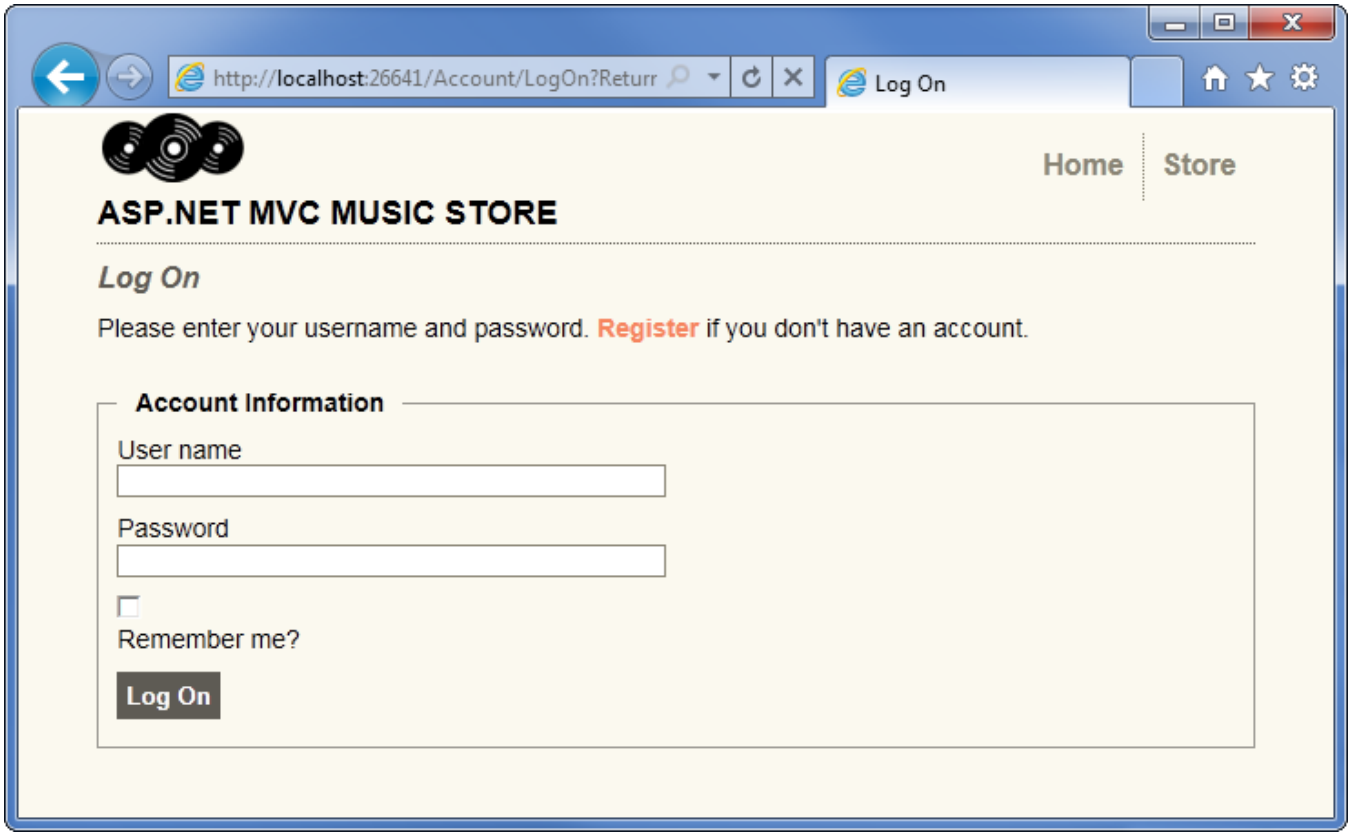
Role-based Authorization

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 OrderDetail 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; }
```


Tutorial under Creative Commons Attribution 3.0 License

```
        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 OrderDetail.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 DbSets which expose those new Model classes, also including a DbSet<Artist>. 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; }
    }
}
```


Tutorial under Creative Commons Attribution 3.0 License

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();
```


Tutorial under Creative Commons Attribution 3.0 License

```
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);
```


Tutorial under Creative Commons Attribution 3.0 License

```
        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
```


Tutorial under Creative Commons Attribution 3.0 License

```
    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 orderDetail = 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);

       storeDB.OrderDetails.Add(orderDetail);

    }

    // 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
```


Tutorial under Creative Commons Attribution 3.0 License

```
                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

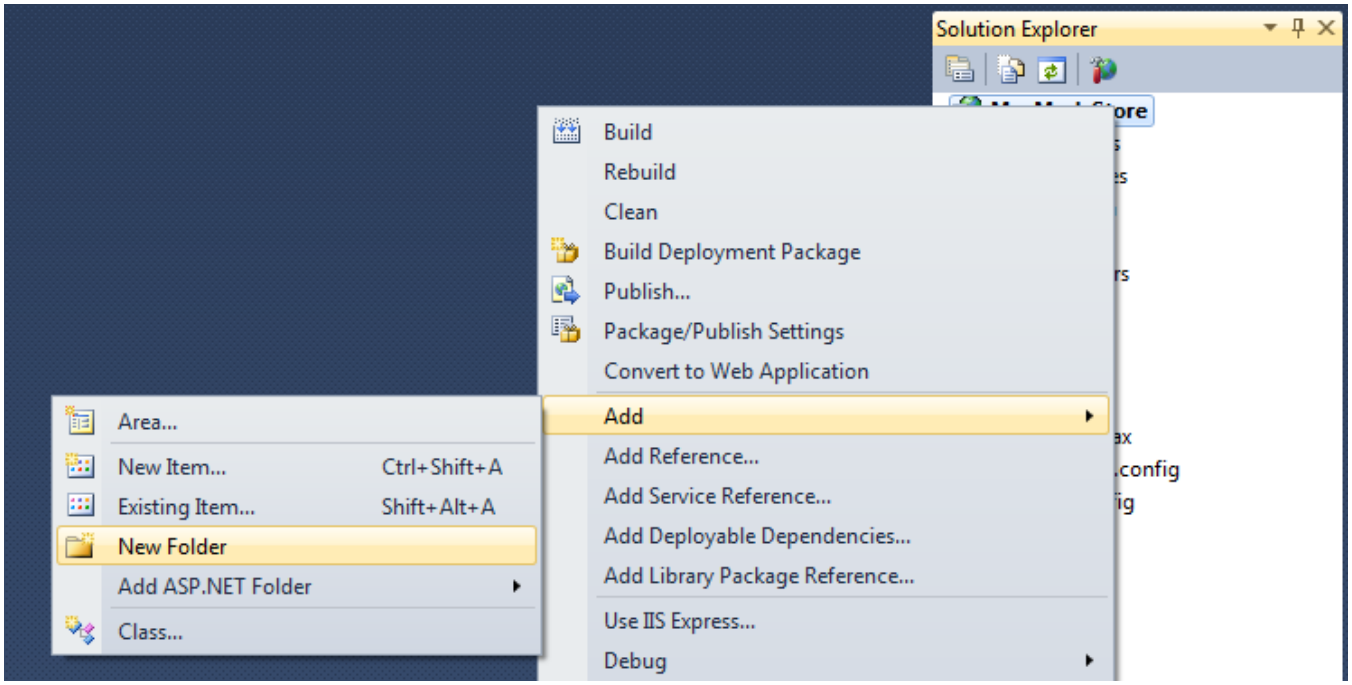
Our Shopping Cart Controller will need to communicate some complex information to its views which doesn't map cleanly to our Model objects. We don't want to modify our Models to suit our views; Model classes should represent our domain, not the user interface. One solution would be to pass the information 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.

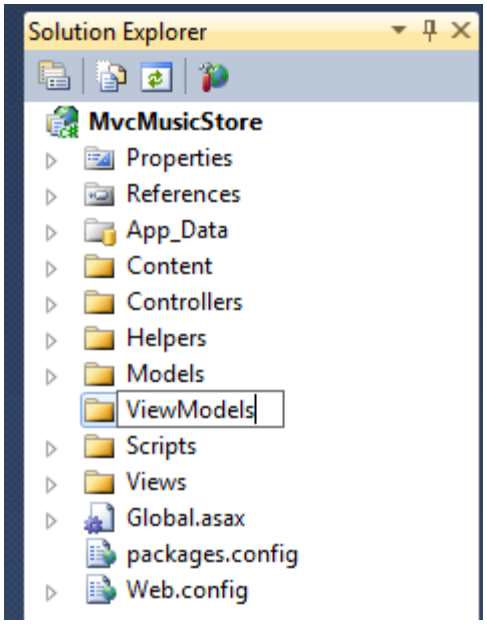
We'll create two View Models for use in our Shopping Cart controller: the ShoppingCartViewModel will hold the contents of the user's shopping cart, and the ShoppingCartRemoveViewModel will be used to display confirmation information when a user removes something from their cart.

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.

Tutorial under Creative Commons Attribution 3.0 License



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; }
    }
}
```

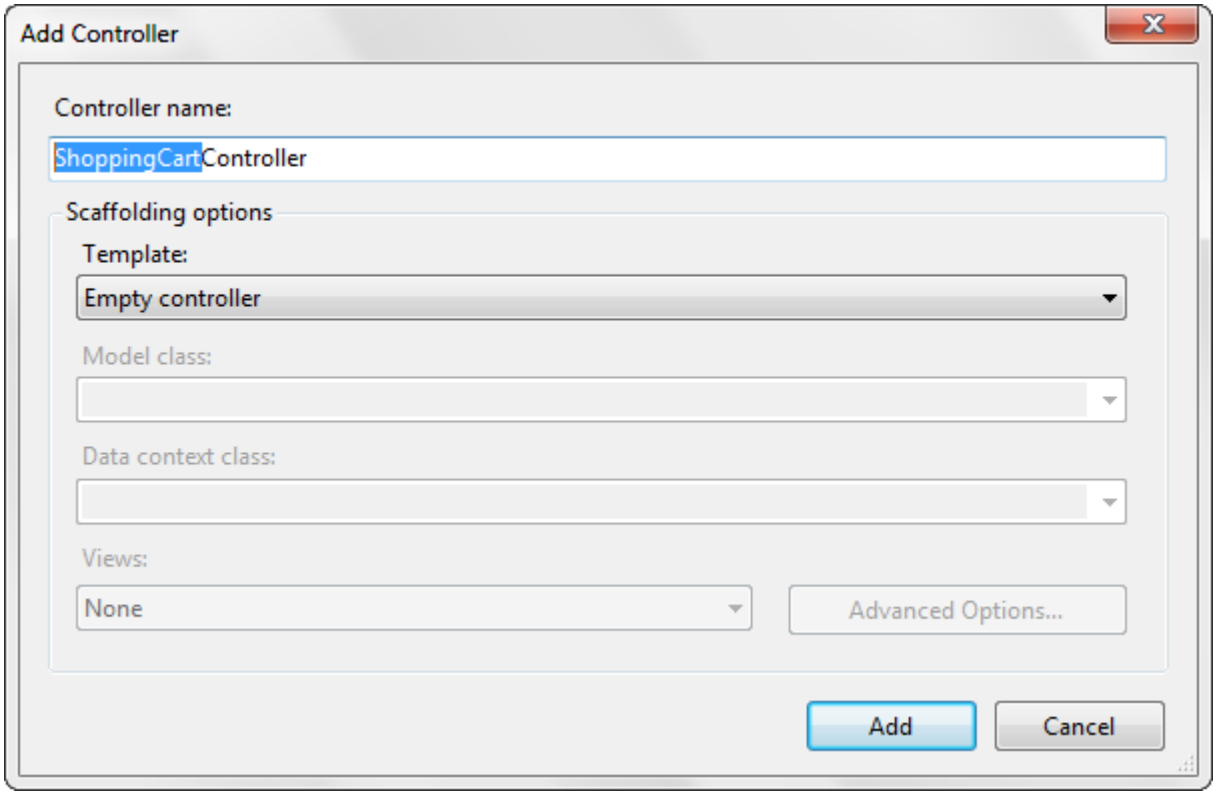
Now add the ShoppingCartRemoveViewModel to the ViewModels folder, with the following four properties.

```
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 using the Empty controller template.



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
```

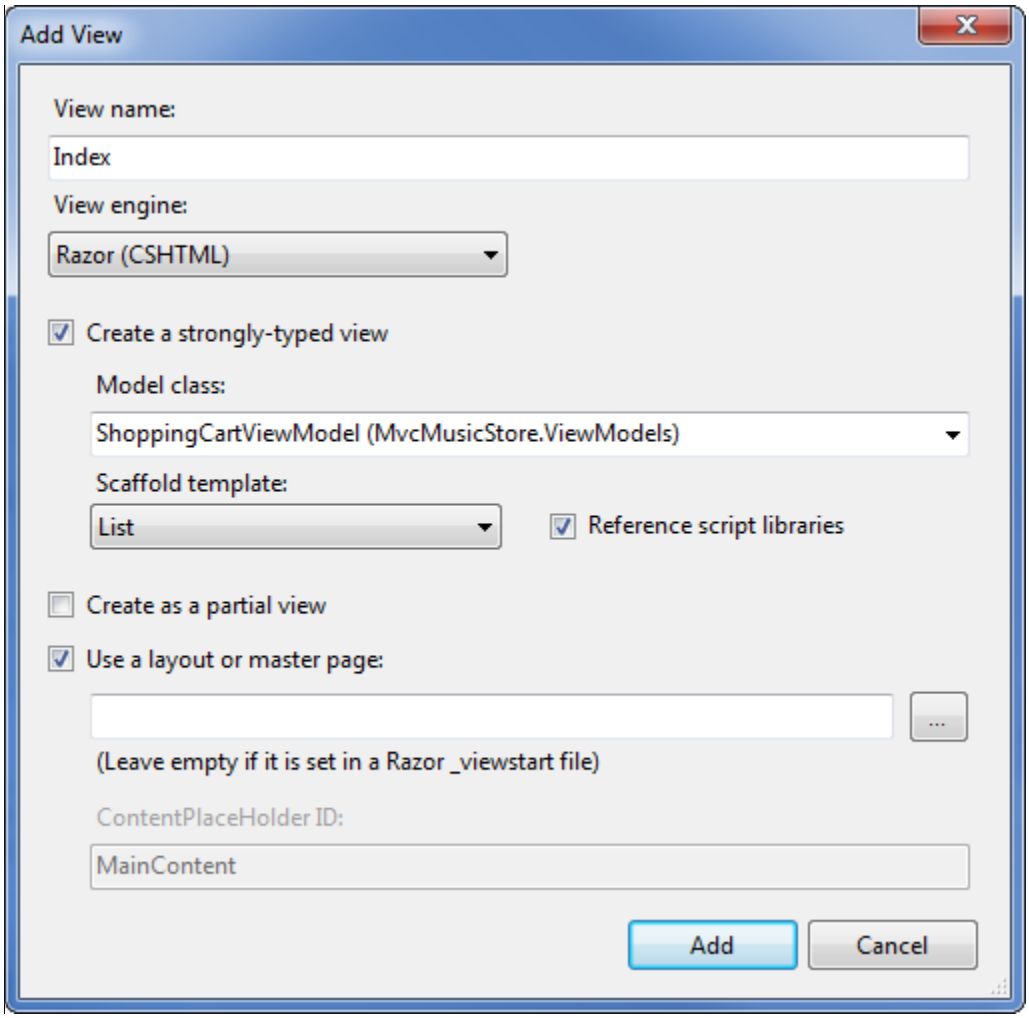

Tutorial under Creative Commons Attribution 3.0 License

```
        [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 with jQuery

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.

Tutorial under Creative Commons Attribution 3.0 License



However, instead of using an Html.ActionLink to remove items from the cart, we'll use jQuery to "wire up" the click event for all links in this view which have the HTML class RemoveLink. Rather than posting the form, this click event handler will just make an AJAX callback to our RemoveFromCart controller action. 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";
```

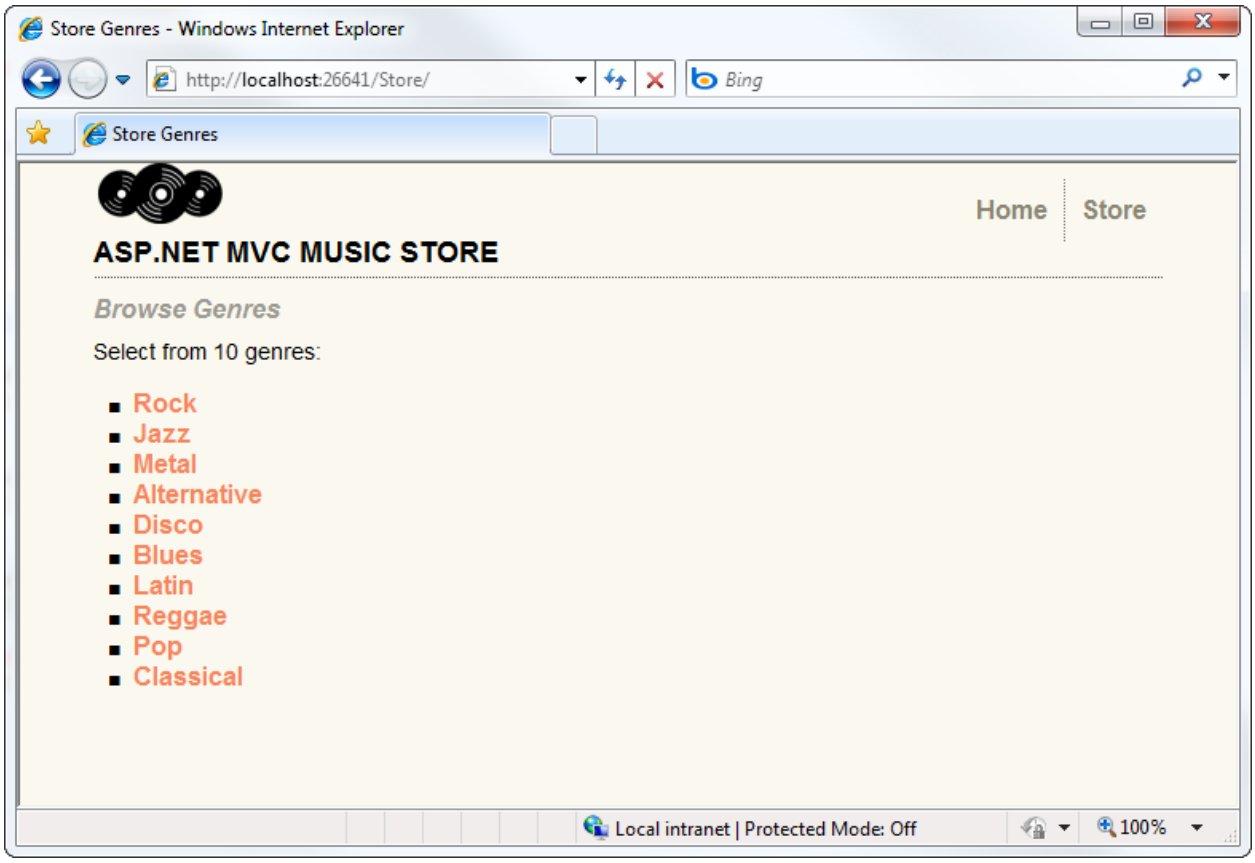
```
}
<script src="/Scripts/jquery-1.4.4.min.js" type="text/javascript"></script>
<script type="text/javascript">
    $(function () {
        // Document.ready -> link up remove event handler
        $(".RemoveLink").click(function () {
            // Get the id from the link
            var recordToDelete = $(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>
```

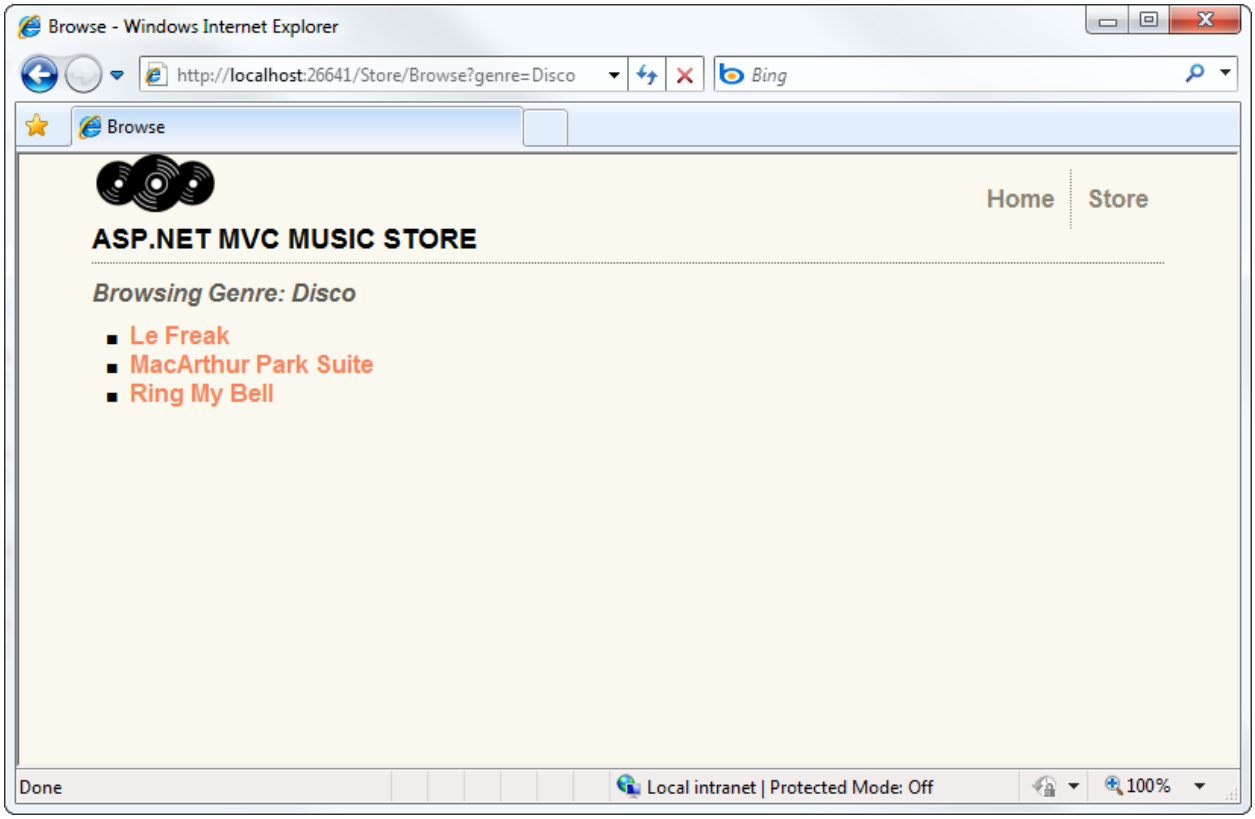
In order to test this out, we need to be able to add items to our shopping cart. We'll update our **Store Details** view to include an "Add to cart" button. While we're at it, we can include some of the Album additional information which we've added since we last updated this view: Genre, Artist, Price, 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>
```

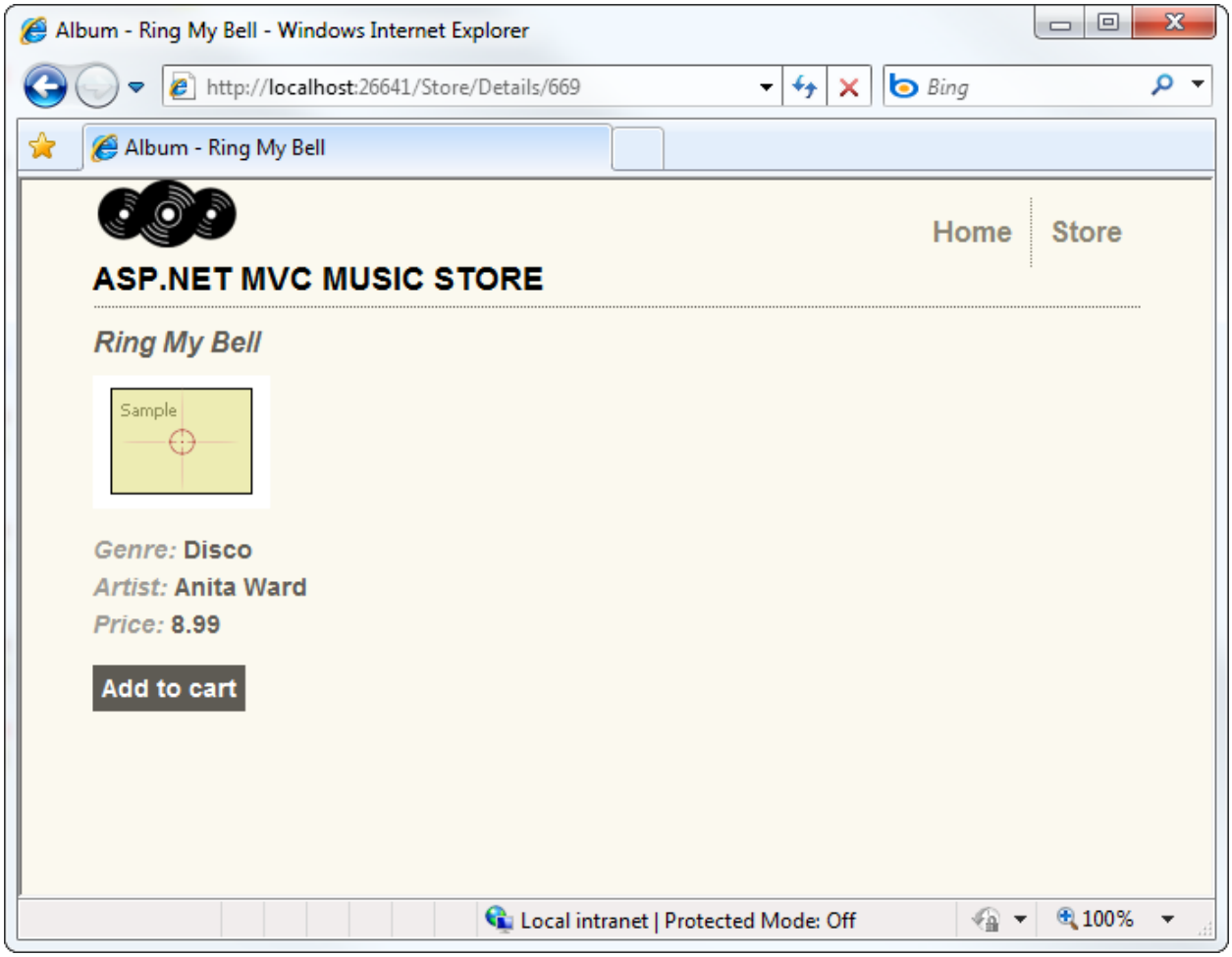
Now we can click through the store and test adding and removing Albums to and from our shopping cart. Run the application and browse to the Store Index.



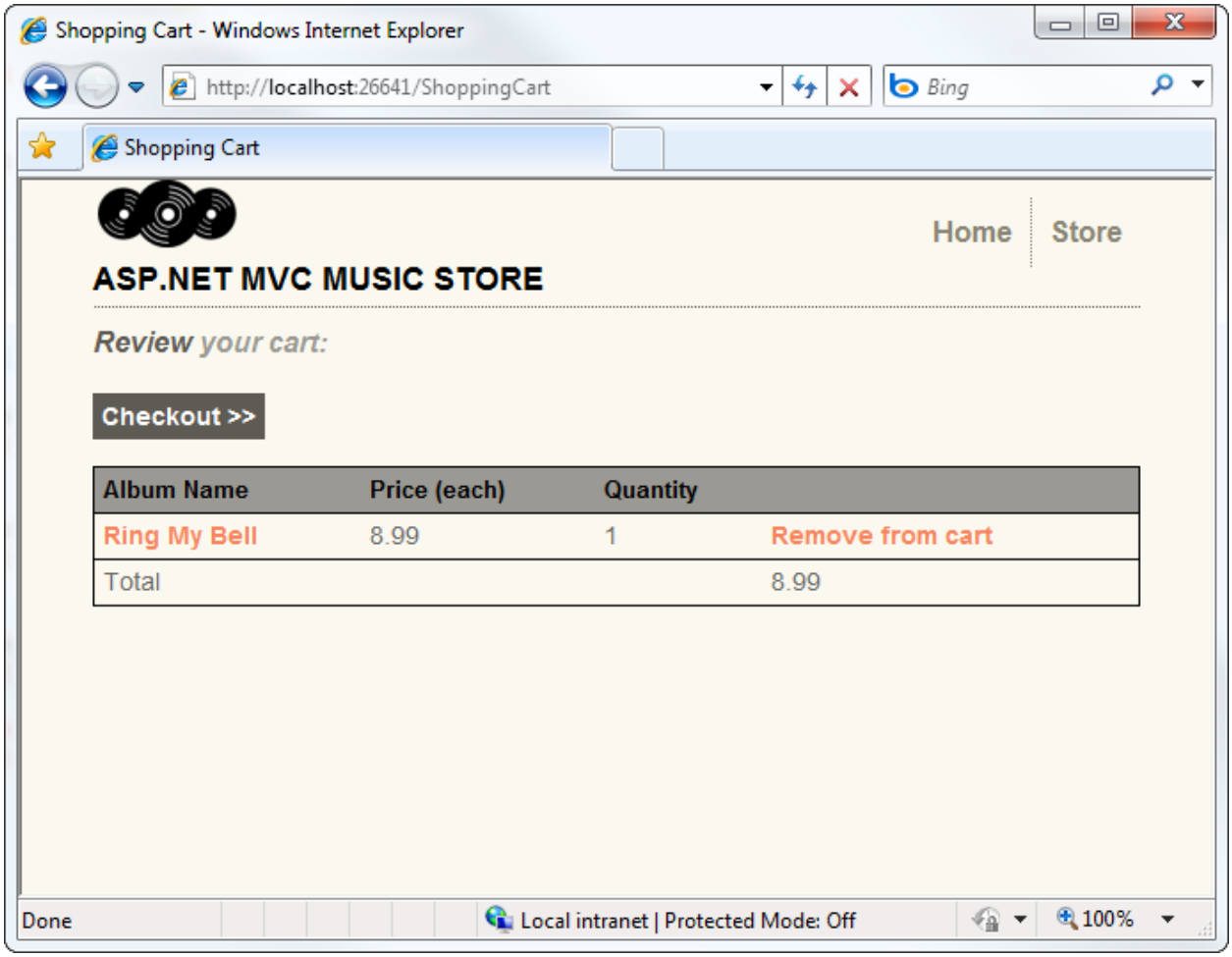
Next, click on a Genre to view a list of albums.



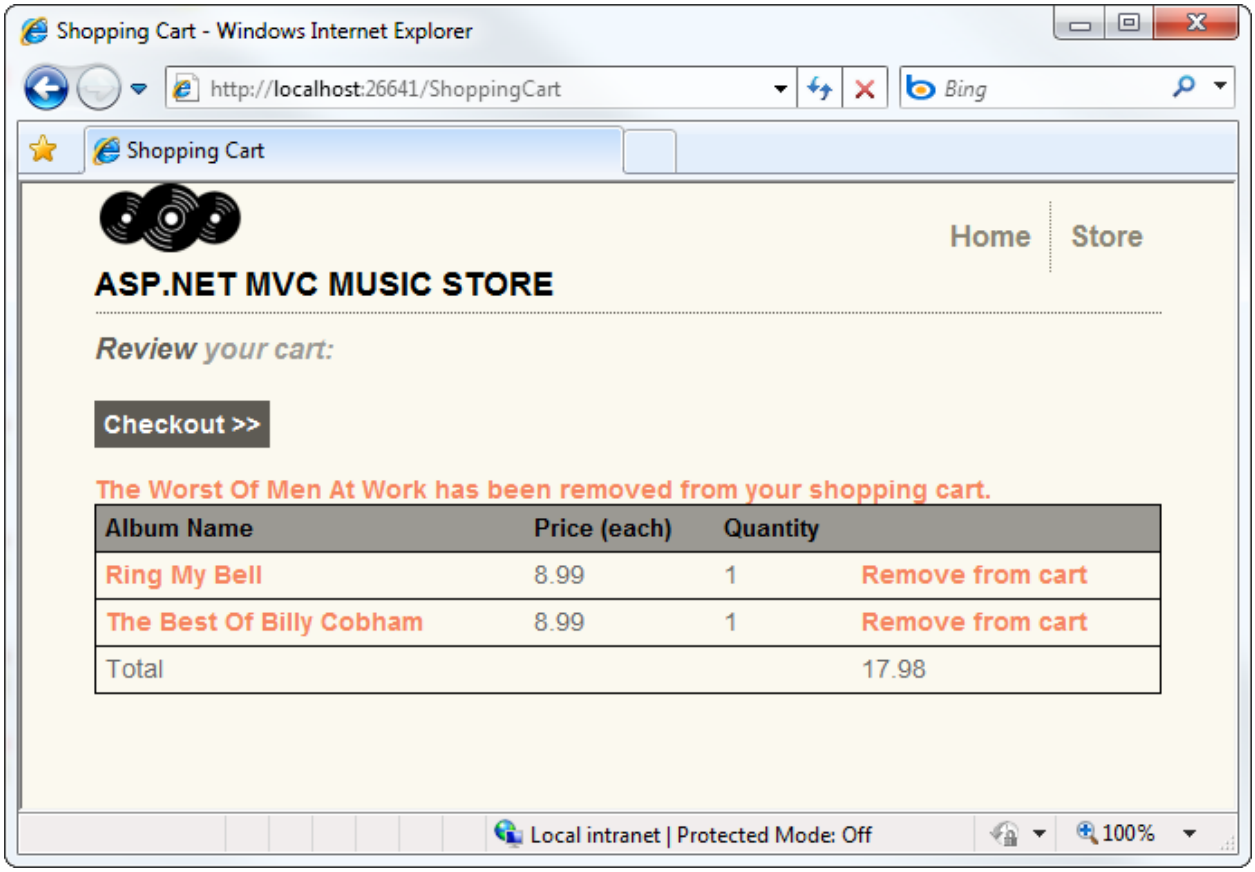
Clicking on an Album title now shows our updated Album Details view, including the “Add to cart” button.



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.

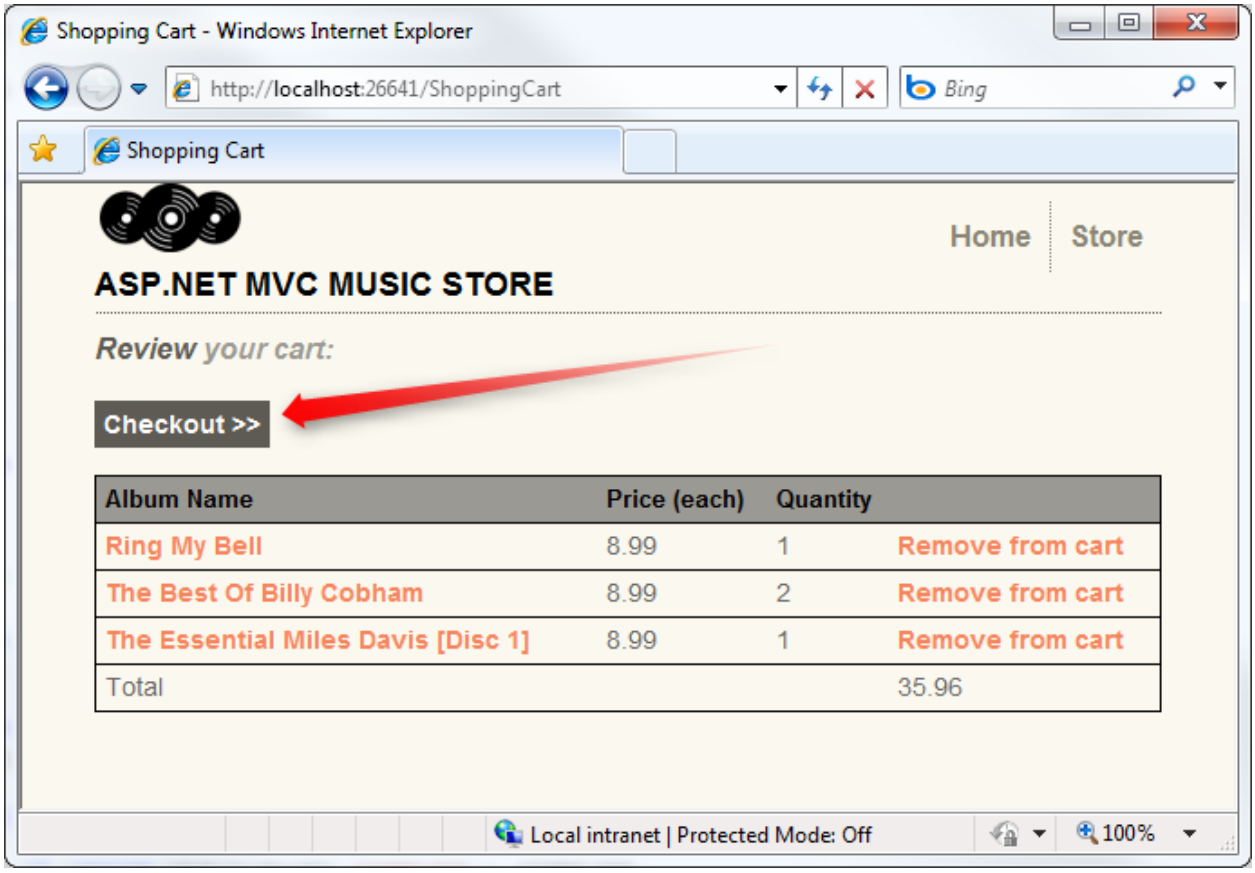


We've built out a working shopping cart which allows unregistered users to add items to their cart. In the following section, we'll allow them to register and complete the checkout process.

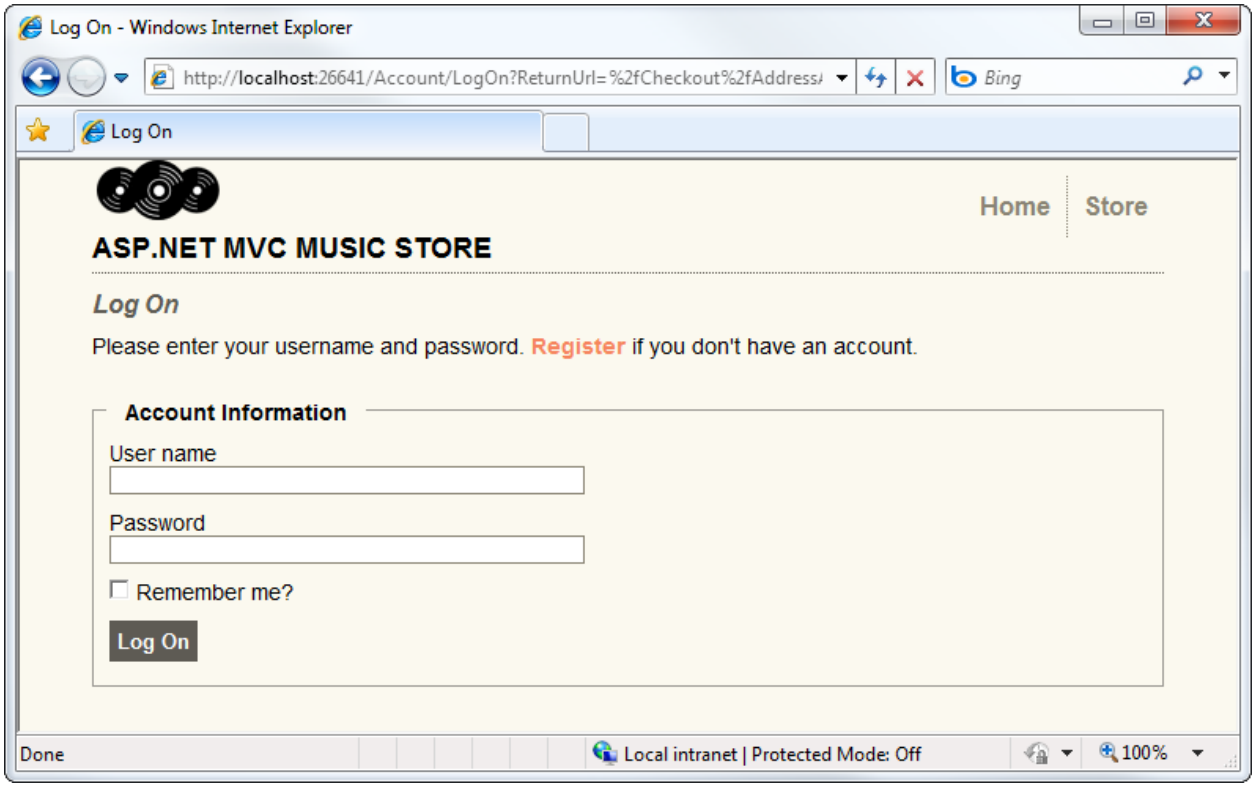
9. Registration and Checkout

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.

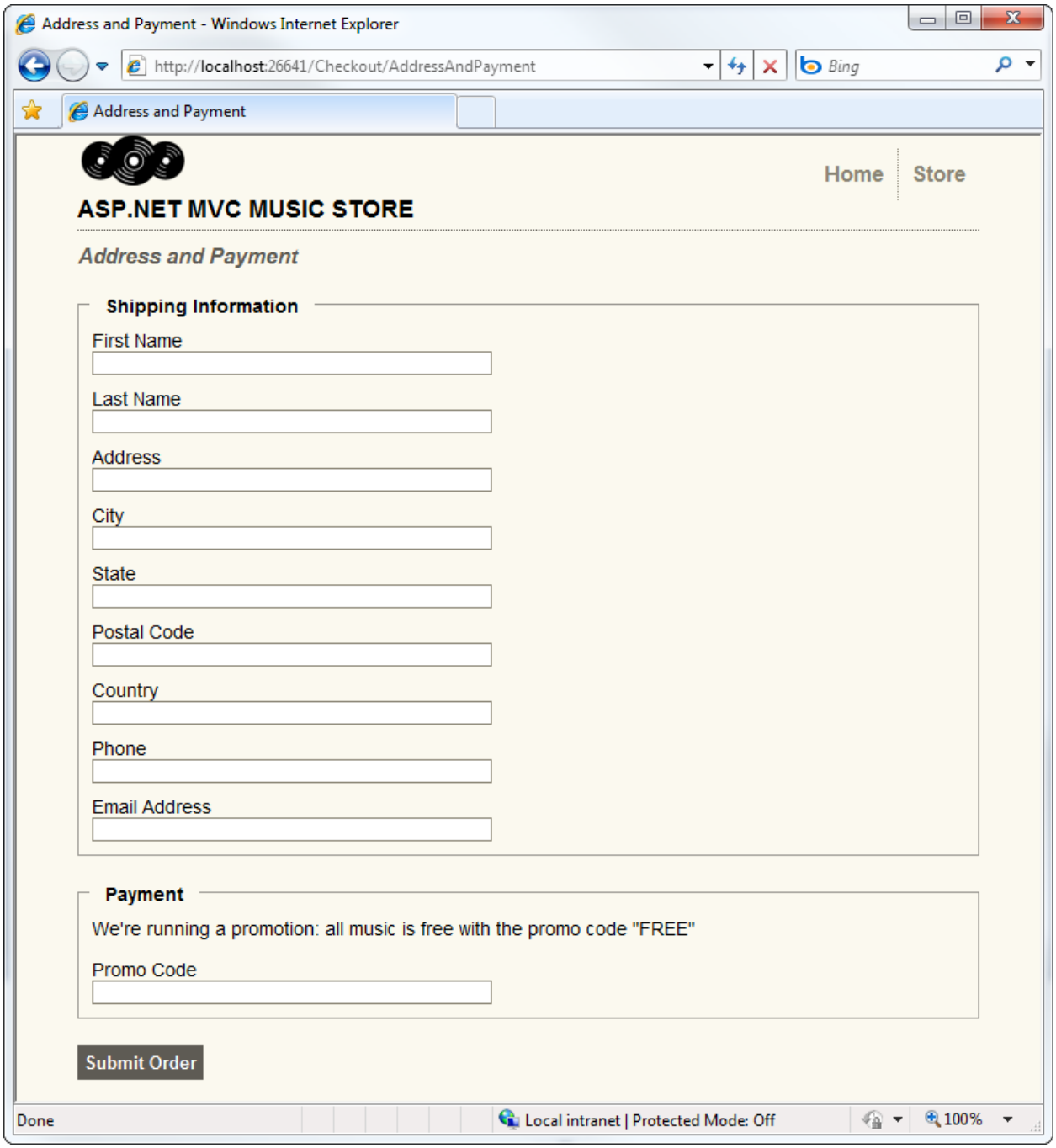
Users will navigate to the checkout process from their shopping cart by clicking the "Checkout" button.



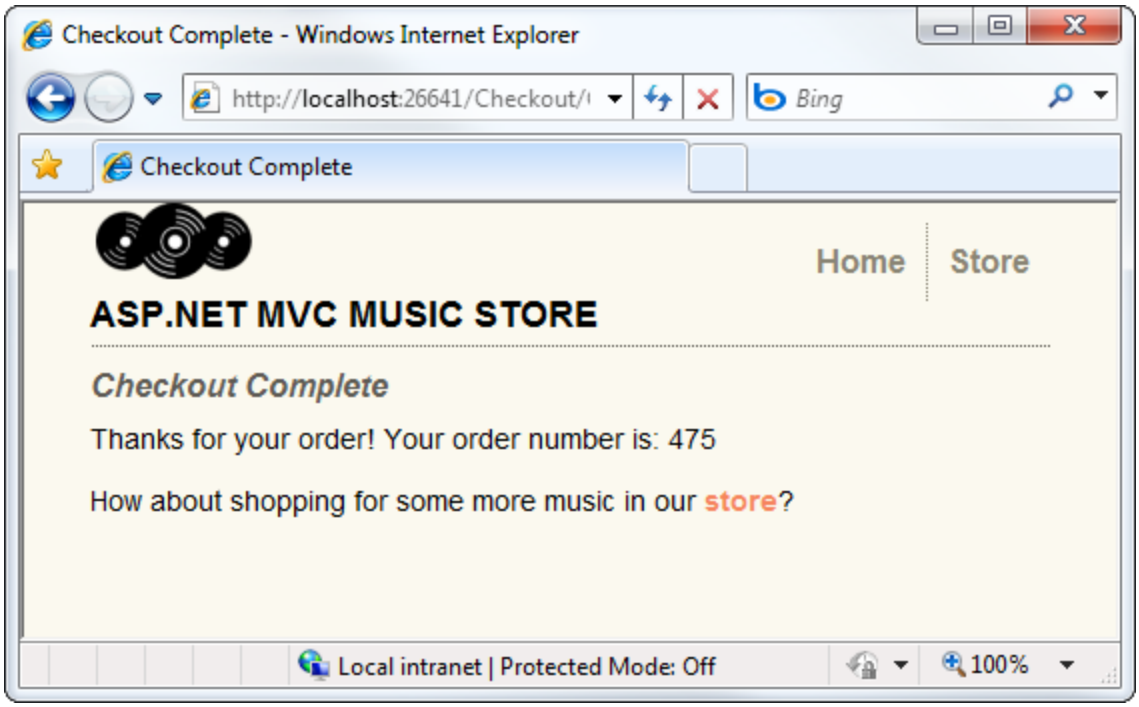
If the user is not logged in, they will be prompted to.



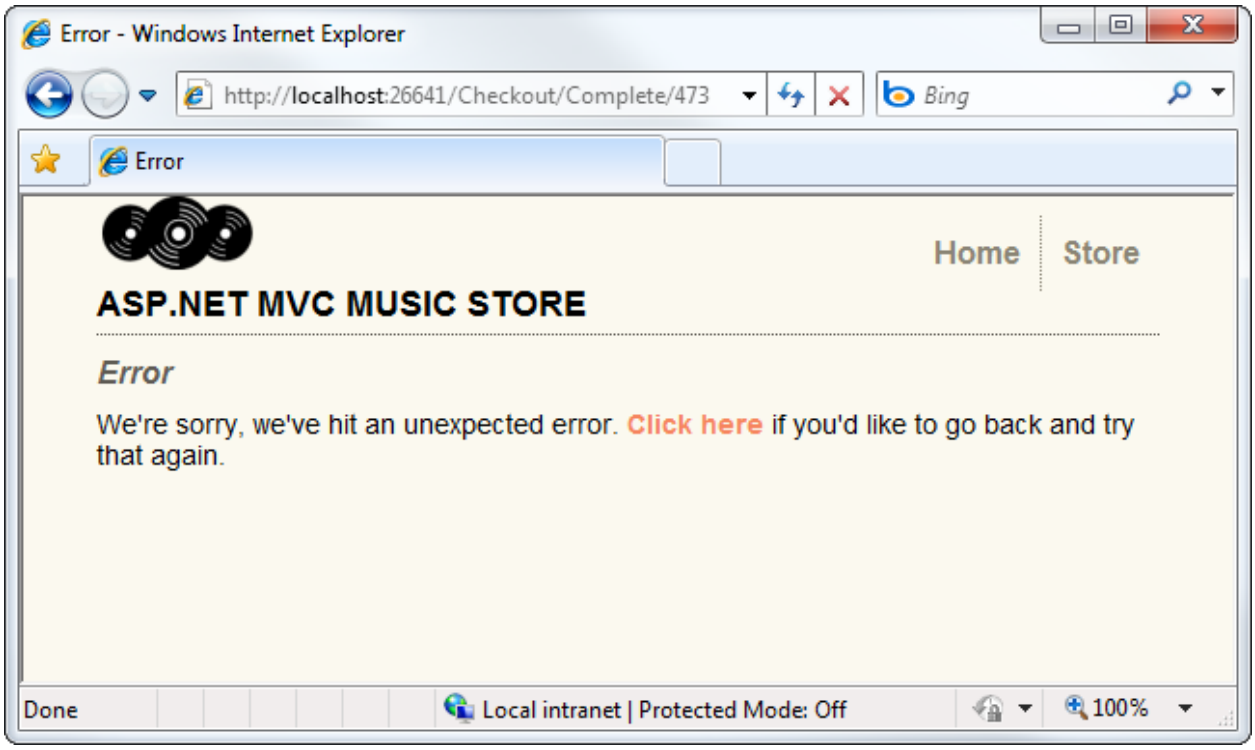
Upon successful login, the user is then shown the Address and Payment view.



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 a using statement referencing MvcMusicStore.Models, then 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:

```
//
// POST: /Account/LogOn

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

            FormsAuthentication.SetAuthCookie(model.UserName, model.RememberMe);
            if (Url.IsLocalUrl(returnUrl) && returnUrl.Length > 1 &&
returnUrl.StartsWith("/")
                && !returnUrl.StartsWith("//") && !returnUrl.StartsWith("/\\"))
            {
                return Redirect(returnUrl);
            }
            else
            {
                return RedirectToAction("Index", "Home");
            }
        }
        else
        {
            ModelState.AddModelError("", "The user name or password provided is
incorrect.");
```


Tutorial under Creative Commons Attribution 3.0 License

```
        }
    }

    // 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:

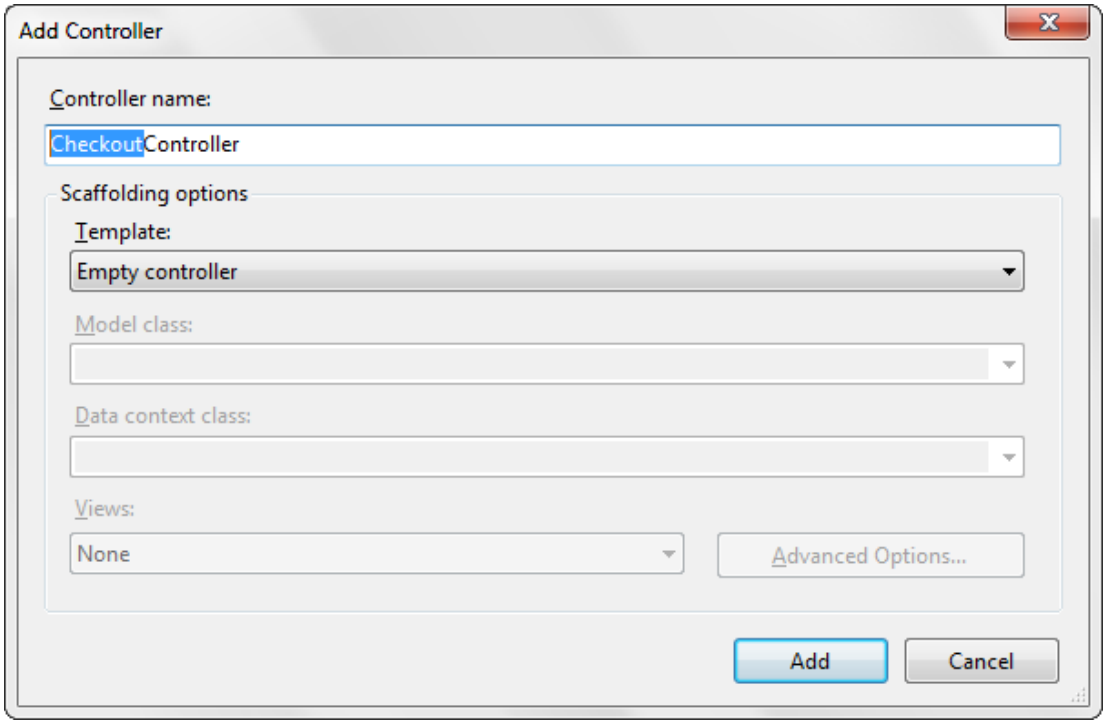
```
//
// POST: /Account/Register

[HttpPost]
public ActionResult Register(RegisterModel model)
{
    if (ModelState.IsValid)
    {
        // Attempt to register the user
        MembershipCreateStatus createStatus;
        Membership.CreateUser(model.UserName, model.Password, model.Email,
               "question", "answer", true, null, out createStatus);

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

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

    // If we got this far, something failed, redisplay form
    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 using the Empty controller template.

Tutorial under Creative Commons Attribution 3.0 License



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);
        }
```


Tutorial under Creative Commons Attribution 3.0 License

```
        else
        {
            order.Username = User.Identity.Name;
            order.OrderDate = DateTime.Now;

            //Save Order
            storeDB.Orders.Add(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:

Tutorial under Creative Commons Attribution 3.0 License

```
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.Orders.Add(order);
                    storeDB.SaveChanges();

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

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

            }
```

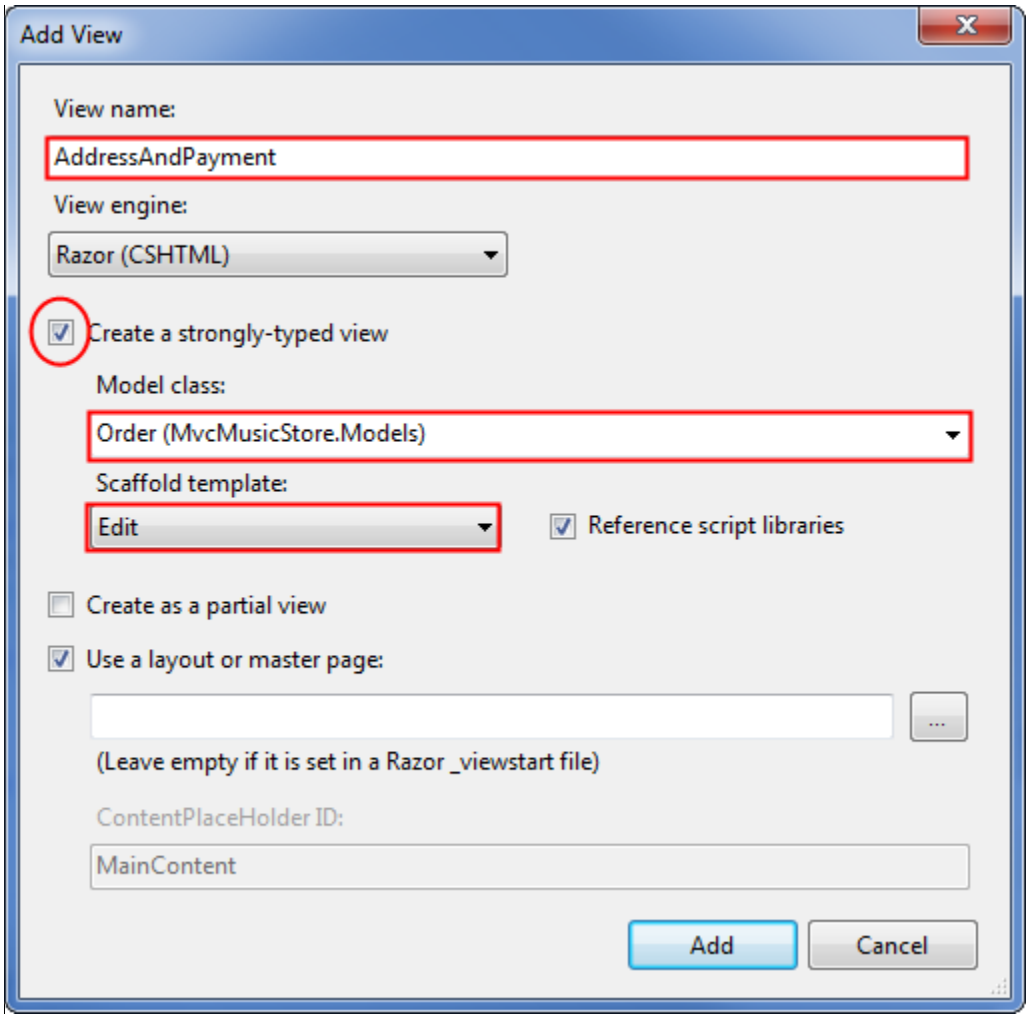

Tutorial under Creative Commons Attribution 3.0 License

```
            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.

Tutorial under Creative Commons Attribution 3.0 License



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.

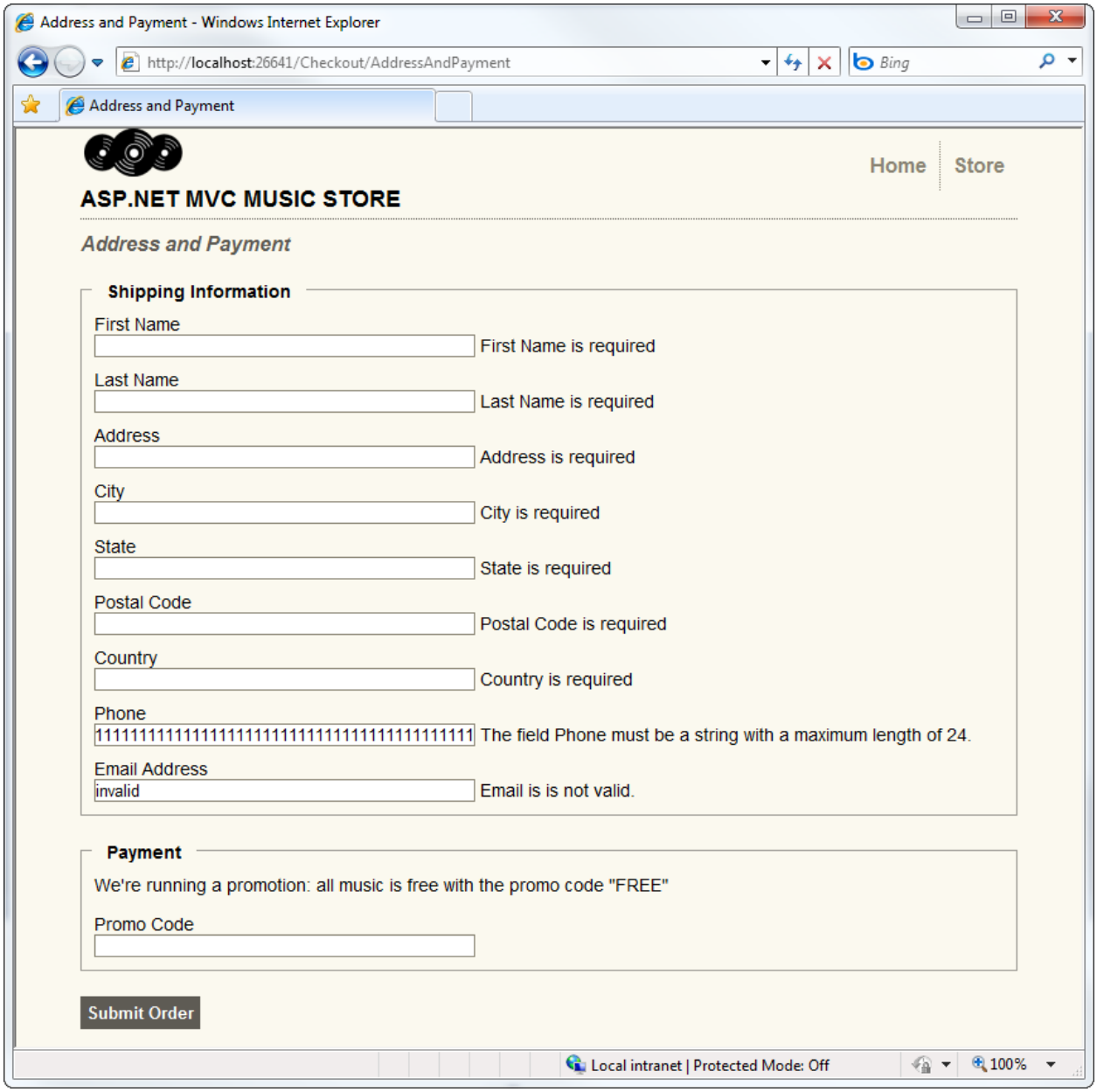
```
using System.Collections.Generic;
using System.ComponentModel;
using System.ComponentModel.DataAnnotations;
using System.Web.Mvc;

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; }
```


Tutorial under Creative Commons Attribution 3.0 License

```
        [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.

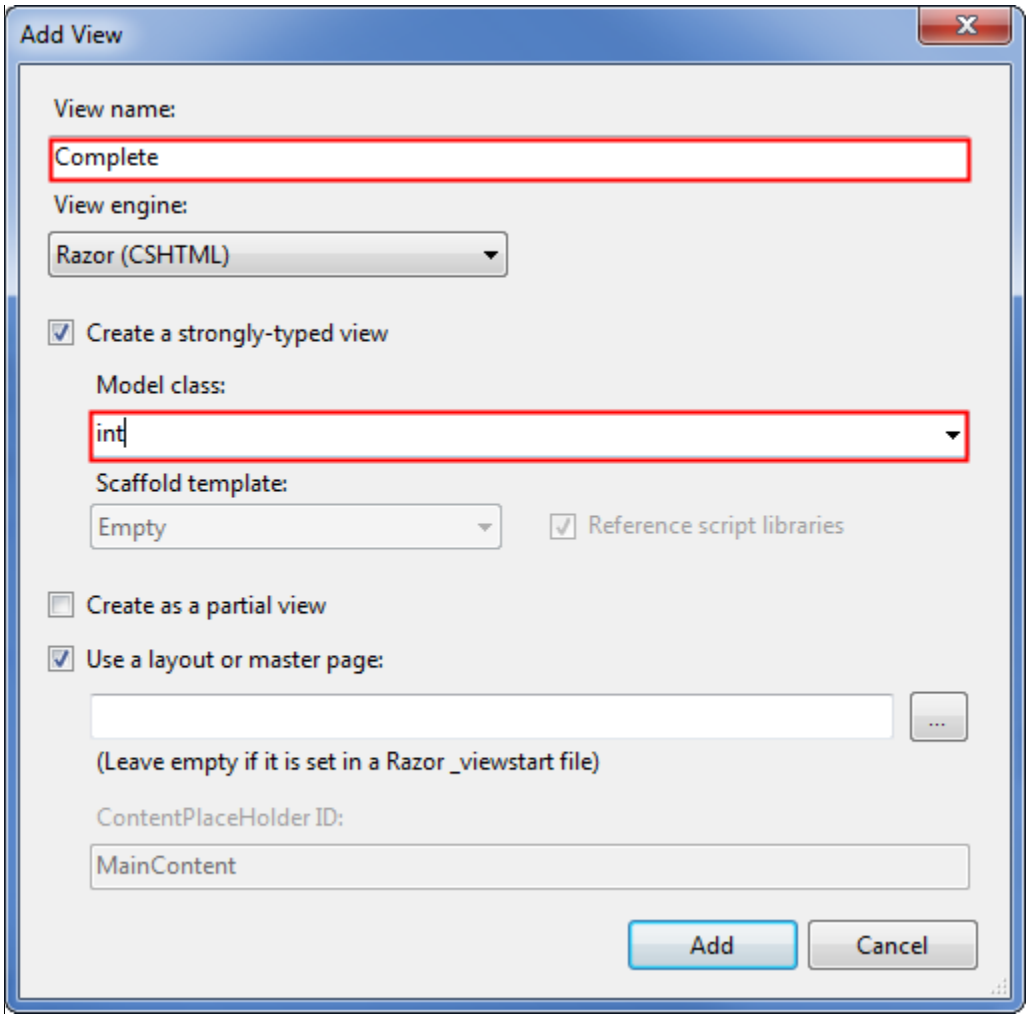
Tutorial under Creative Commons Attribution 3.0 License



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>
```

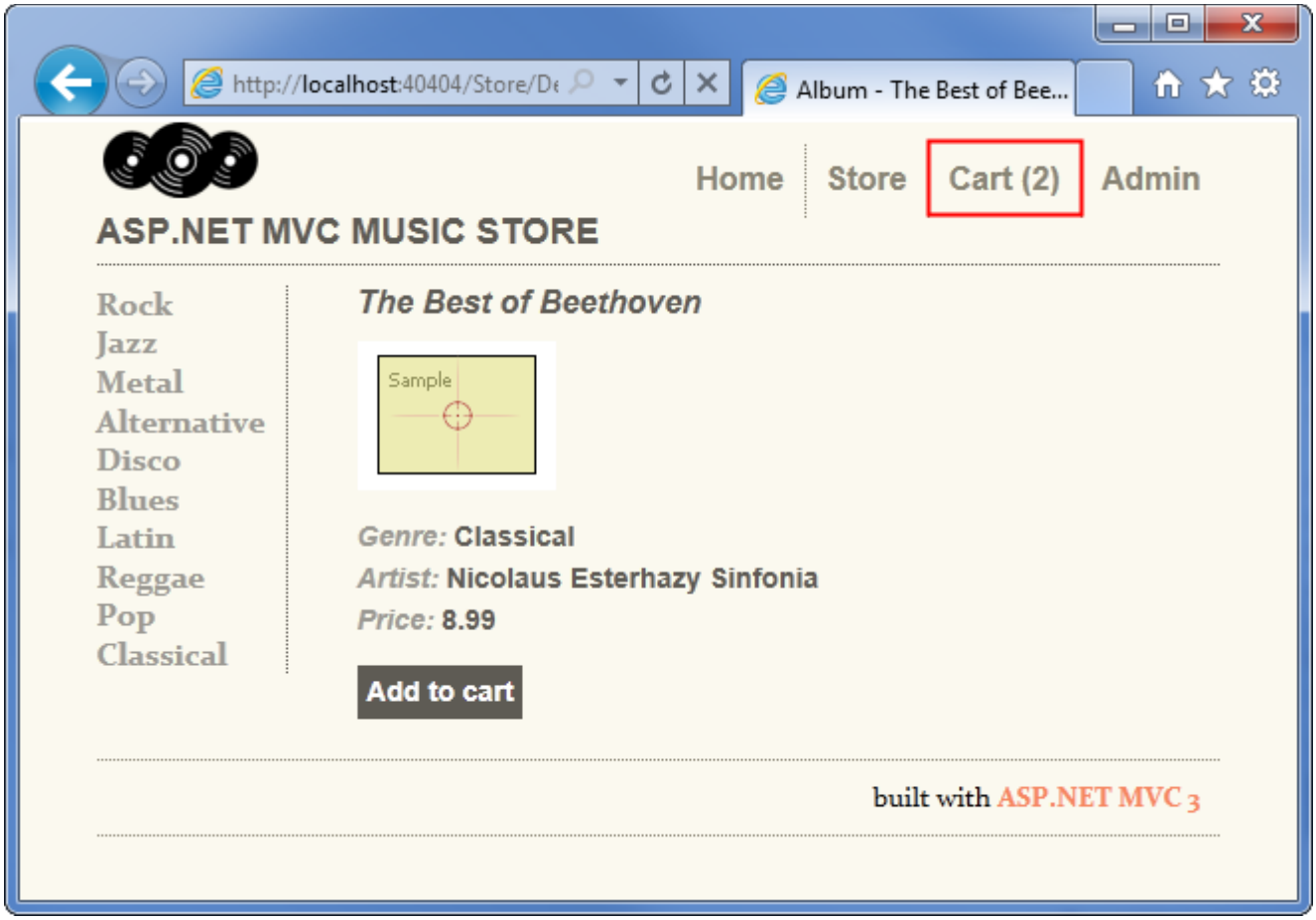

Tutorial under Creative Commons Attribution 3.0 License

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.



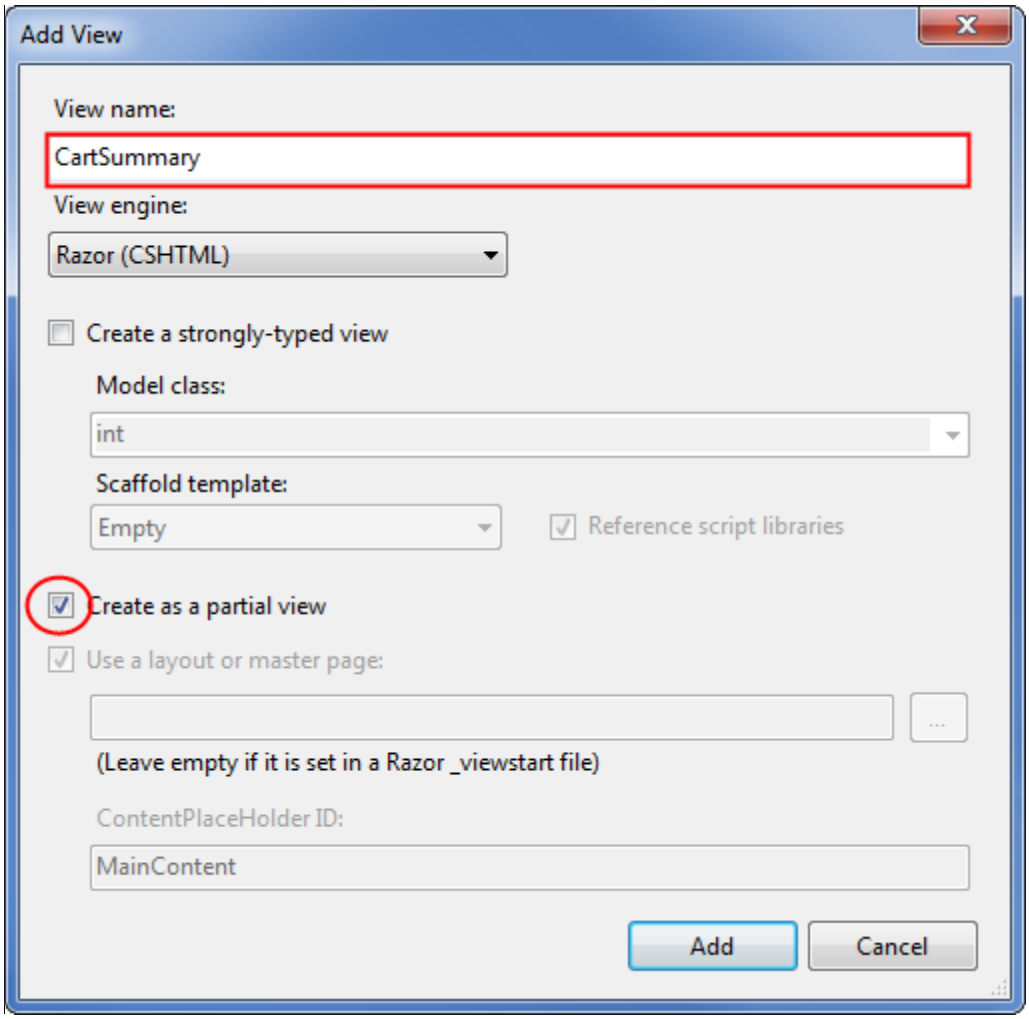
We can easily implement this by creating a partial view which is added to our Site.master.

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" })
```

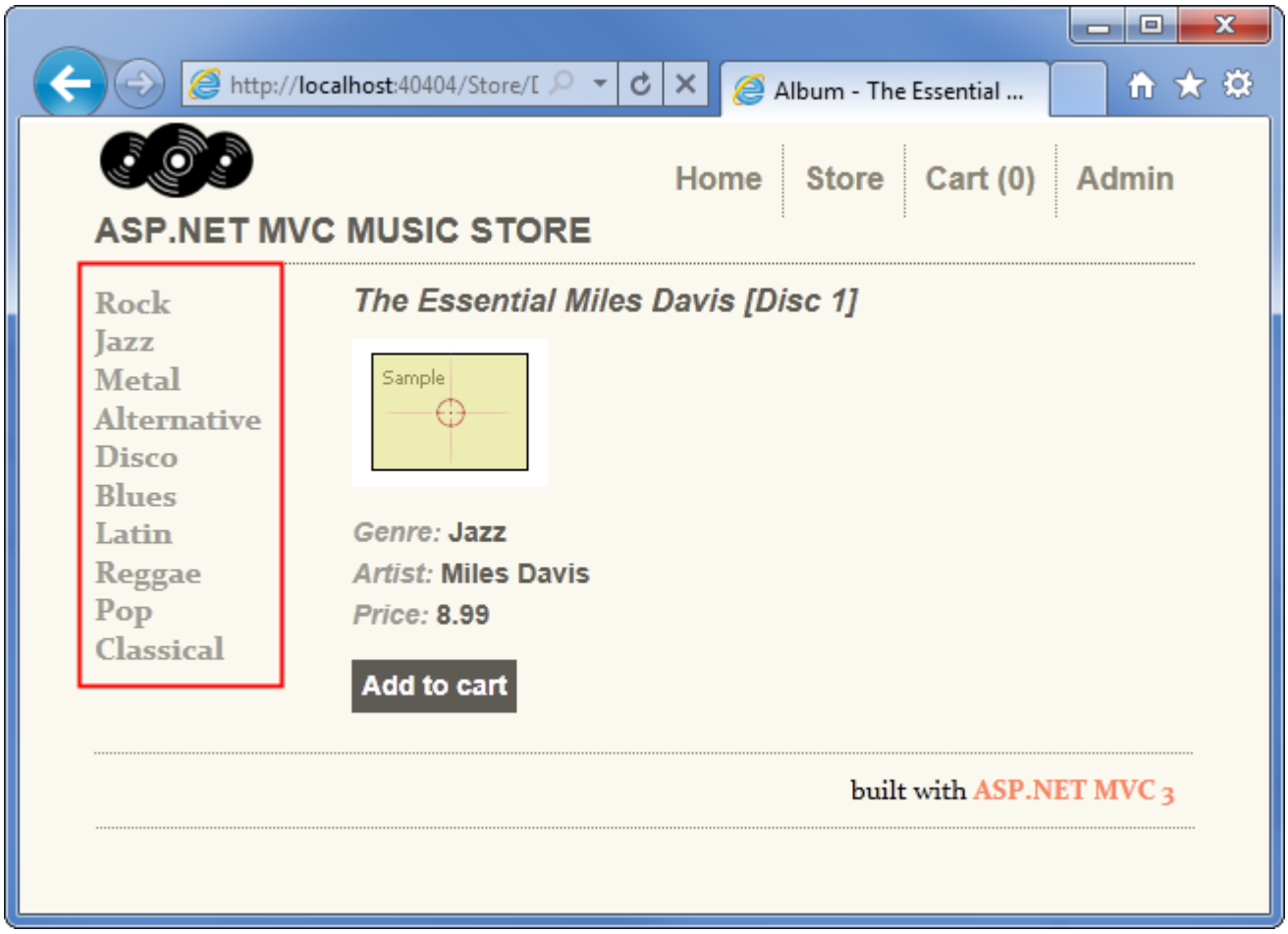
We can include a partial view in any page in the site, including the Site master, by using the Html.RenderAction method. RenderAction requires us to specify the Action Name ("CartSummary") and the Controller Name ("ShoppingCart") as below.

```
@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.



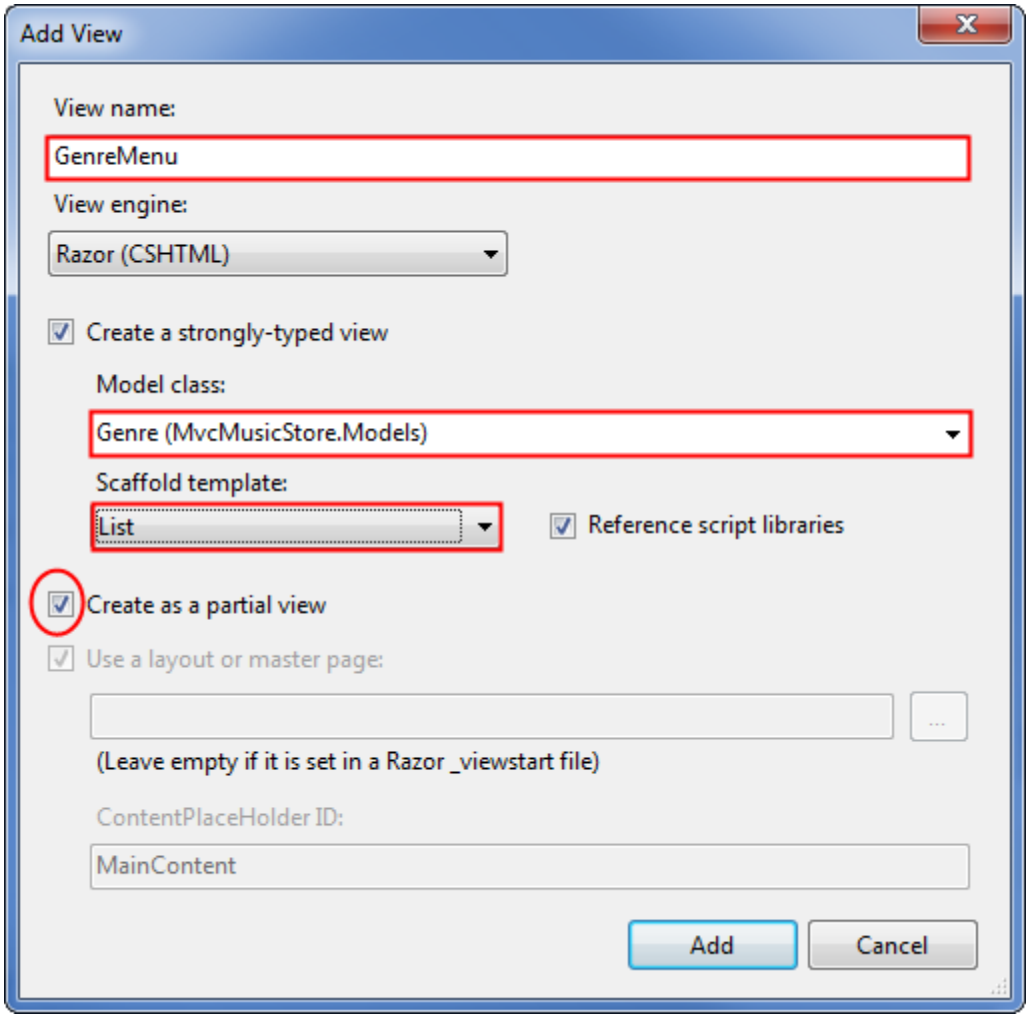
We will follow the same steps also create a GenreMenu partial view, and then we can add them both to the Site master. First, add the following GenreMenu controller action to the StoreController:

```
//
// 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 (/Views/Shared/_Layout.cshtml) 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:

Tutorial under Creative Commons Attribution 3.0 License

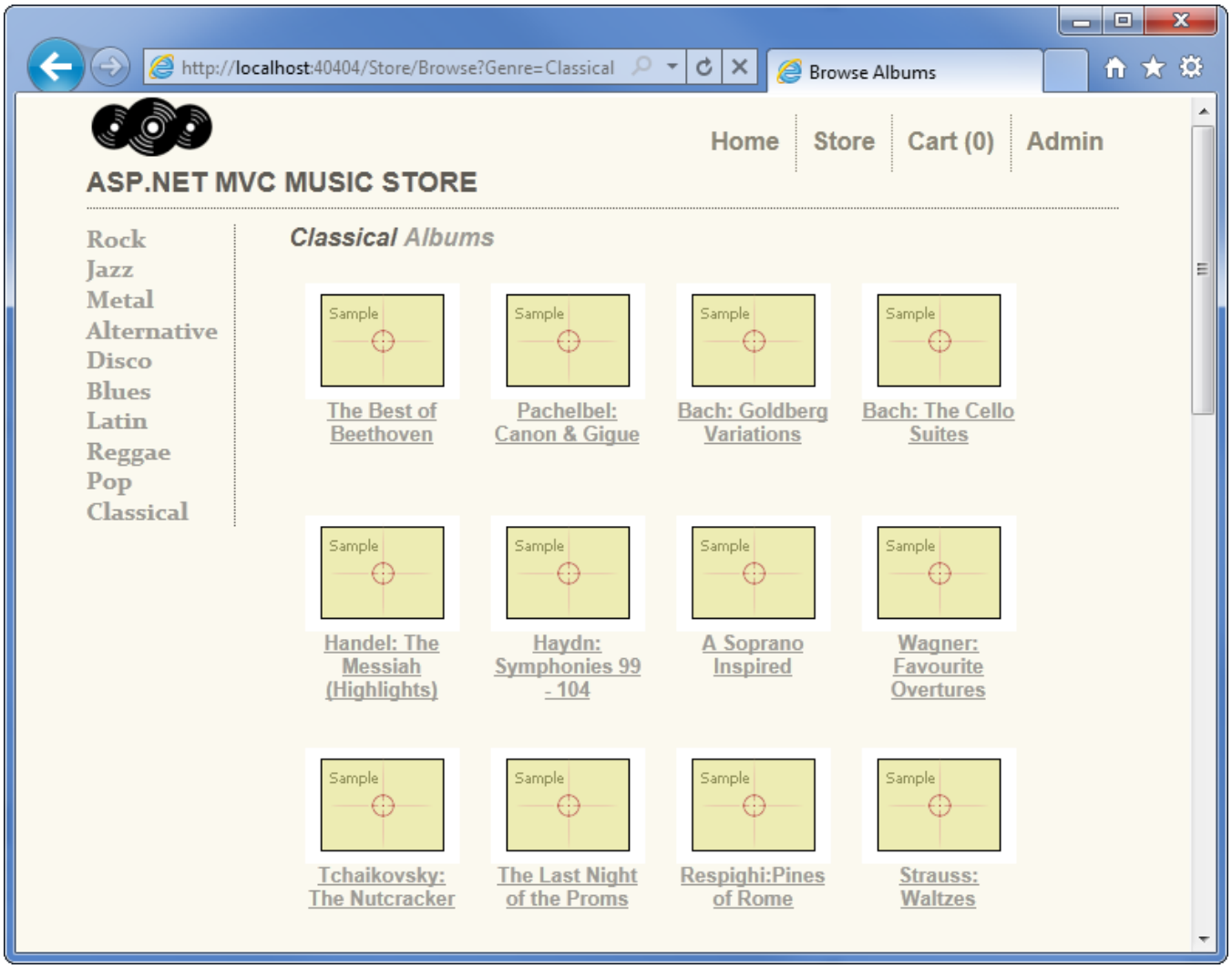
```
@model MvcMusicStore.Models.Genre

@{
    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.

Tutorial under Creative Commons Attribution 3.0 License



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; }
    }
}
```

Note: This will require adding a using statement to bring in the System.Collections.Generic namespace.

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);
```


Tutorial under Creative Commons Attribution 3.0 License

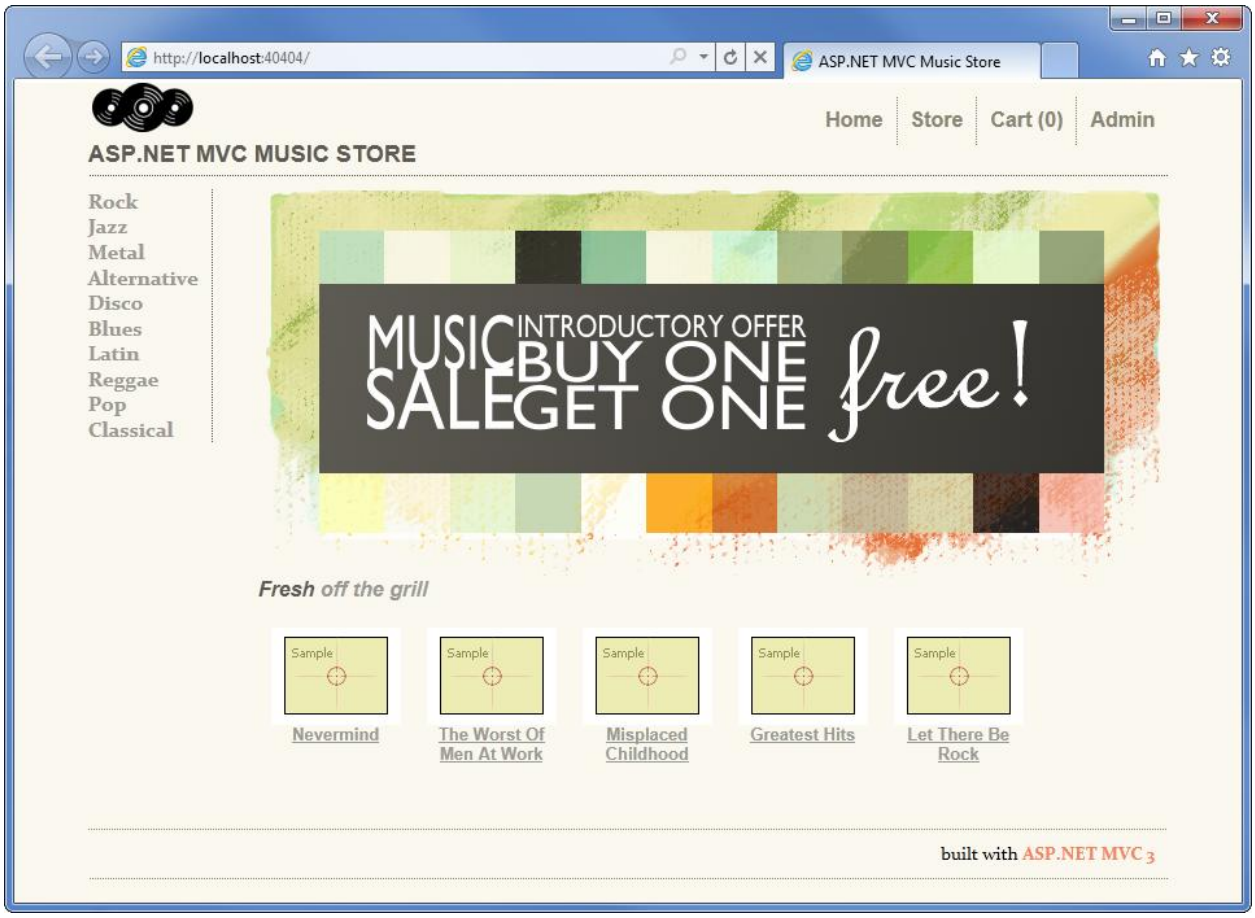
```
            return View(albums);
        }

        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.

Tutorial under Creative Commons Attribution 3.0 License



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!

Tutorial under Creative Commons Attribution 3.0 License