ASP.NET MVC Music Store Tutorial

Version 1.0

Jon Galloway - Microsoft

10/8/2010

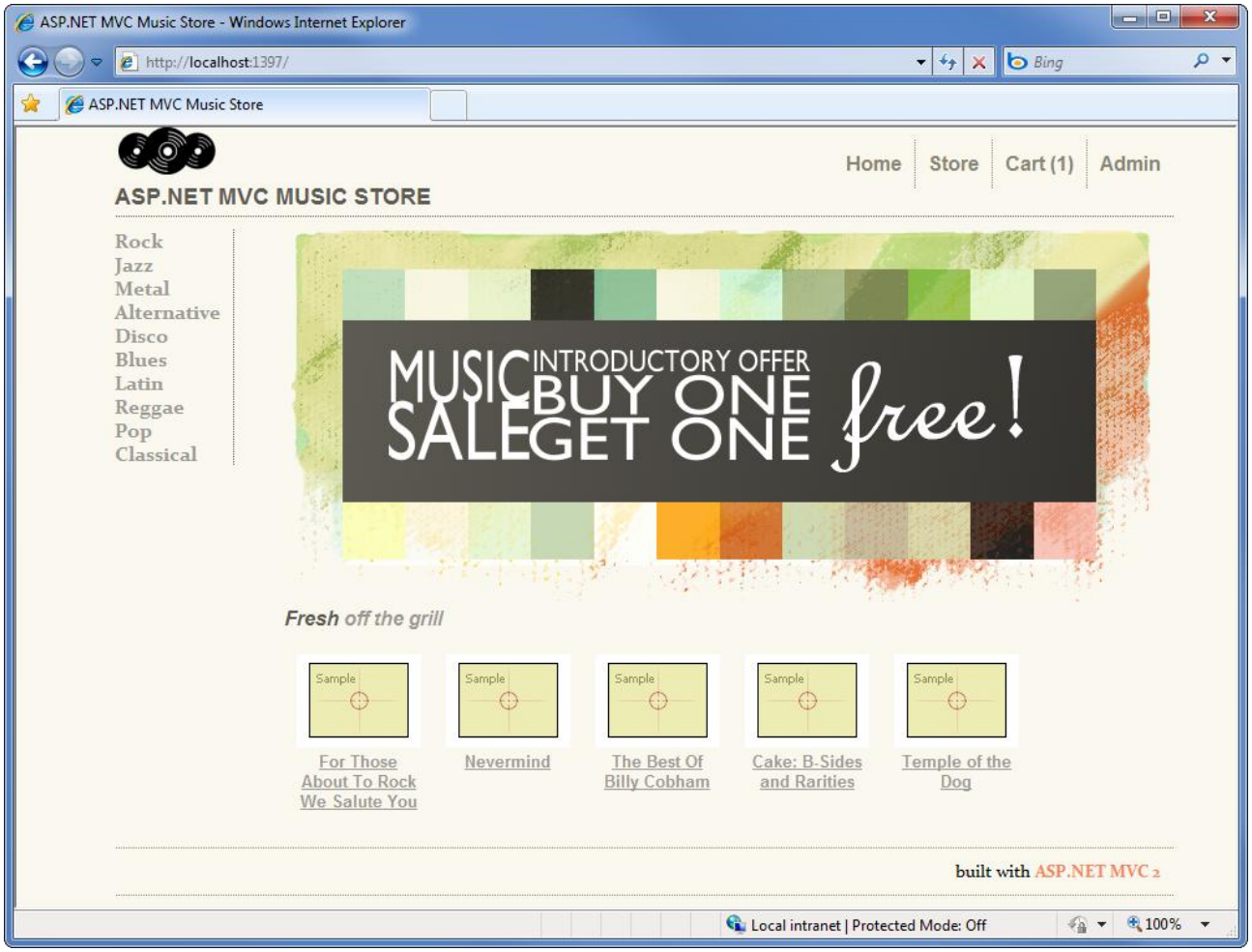
ASP.NET MVC Music Store Tutorial

Contents

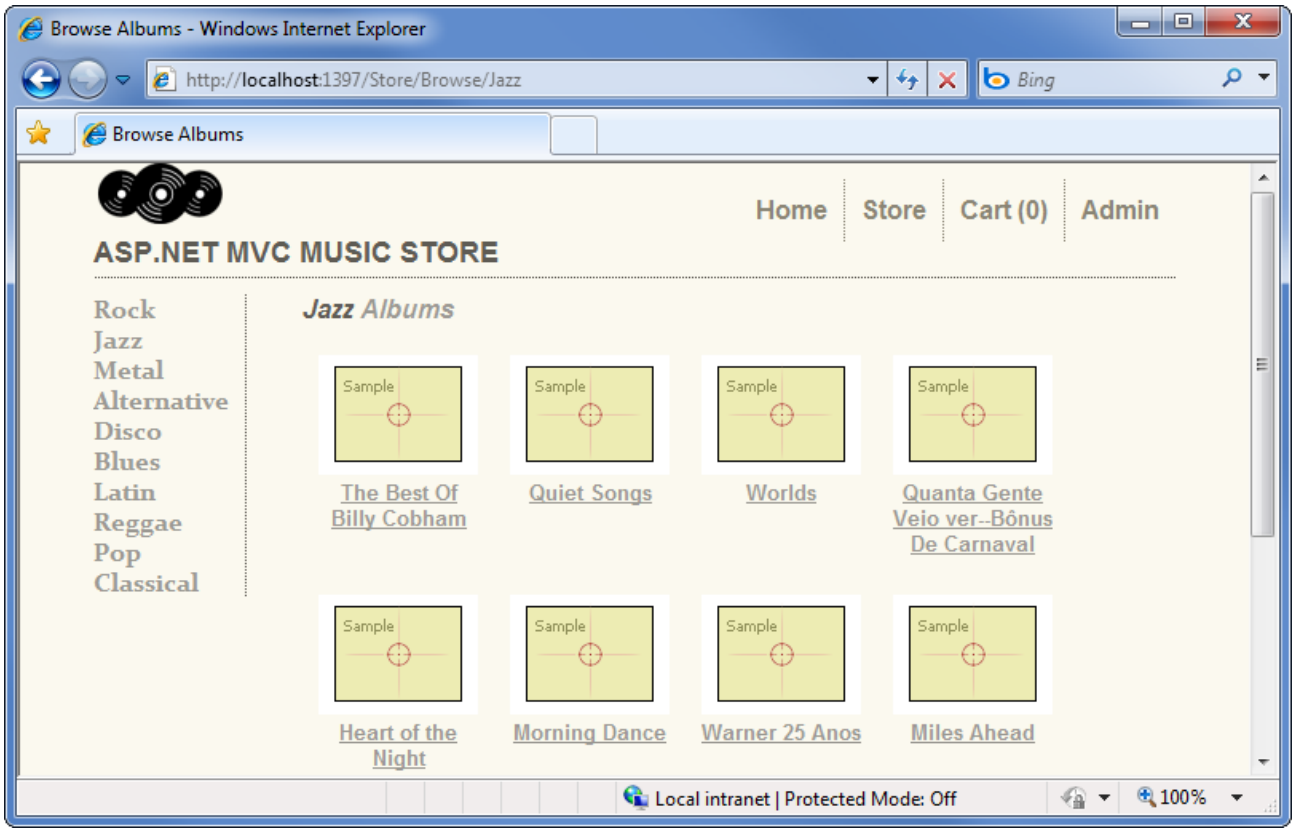
Overview

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

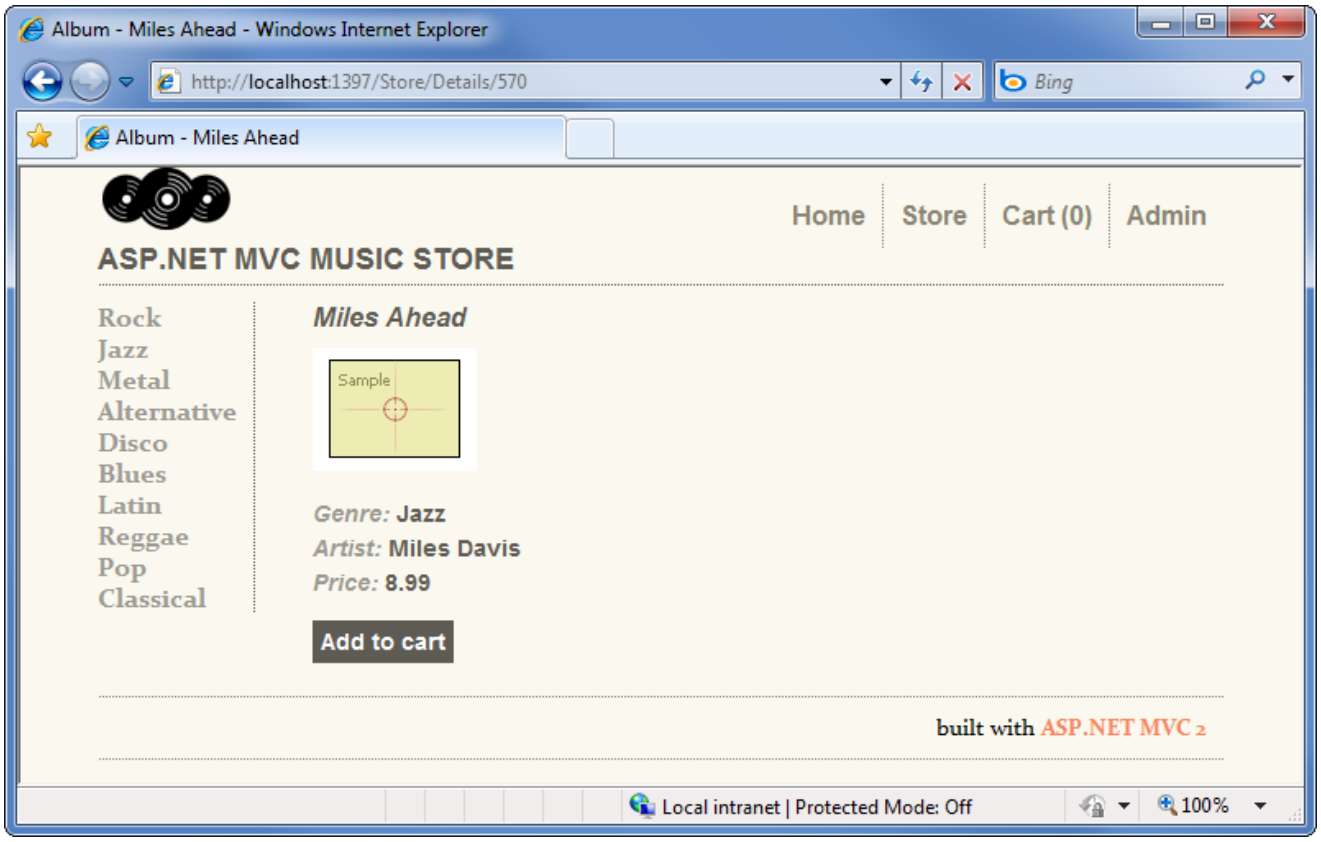
The application we'll be building is a simple music store. There are three main parts to the application: shopping, checkout, and administration.



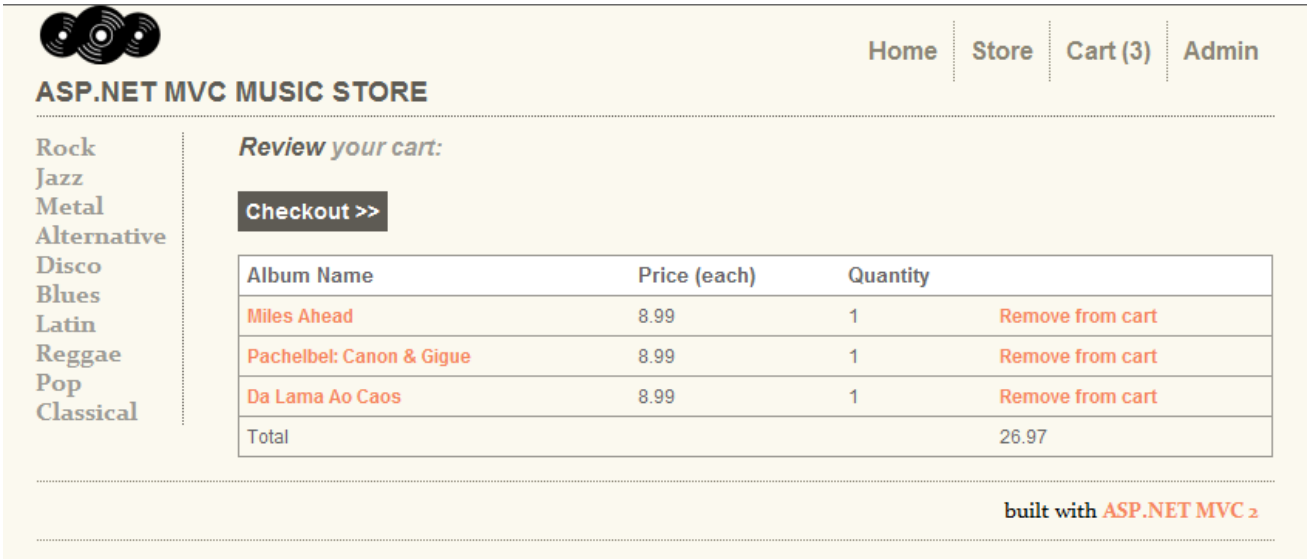
Visitors can browse Albums by Genre:



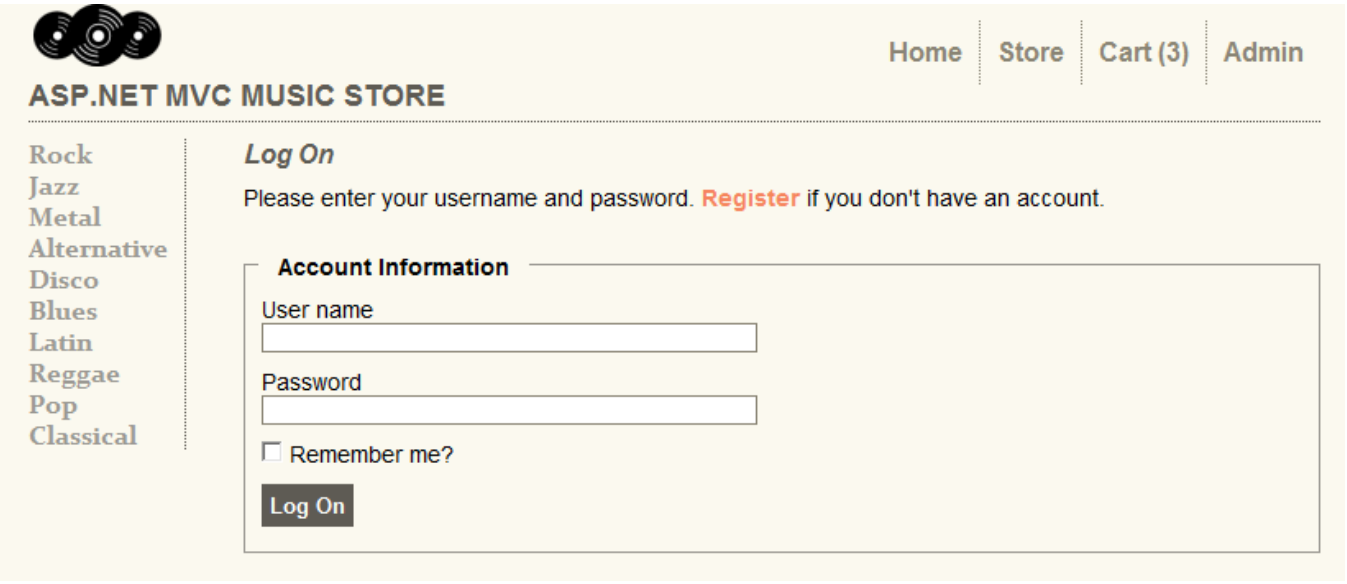
They can view a single album and add it to their cart:

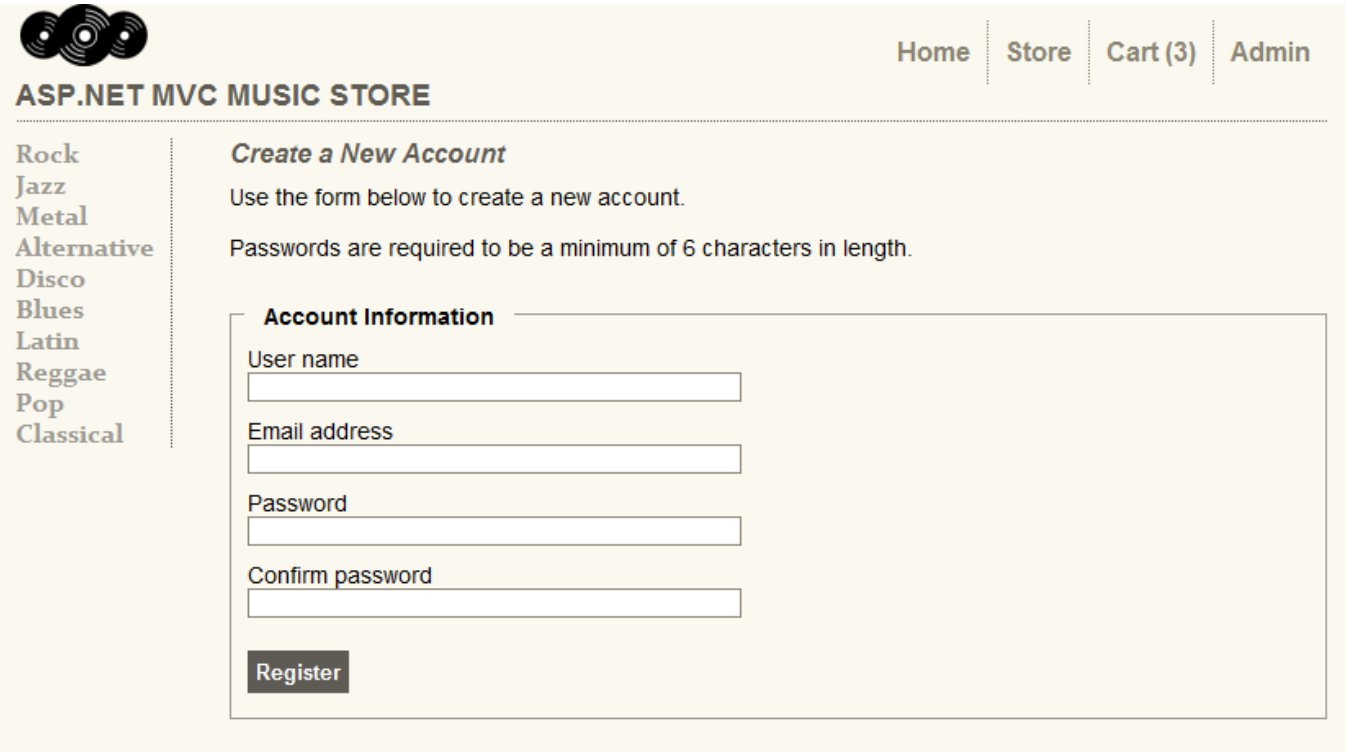


They can review their cart, removing any items they no longer want:

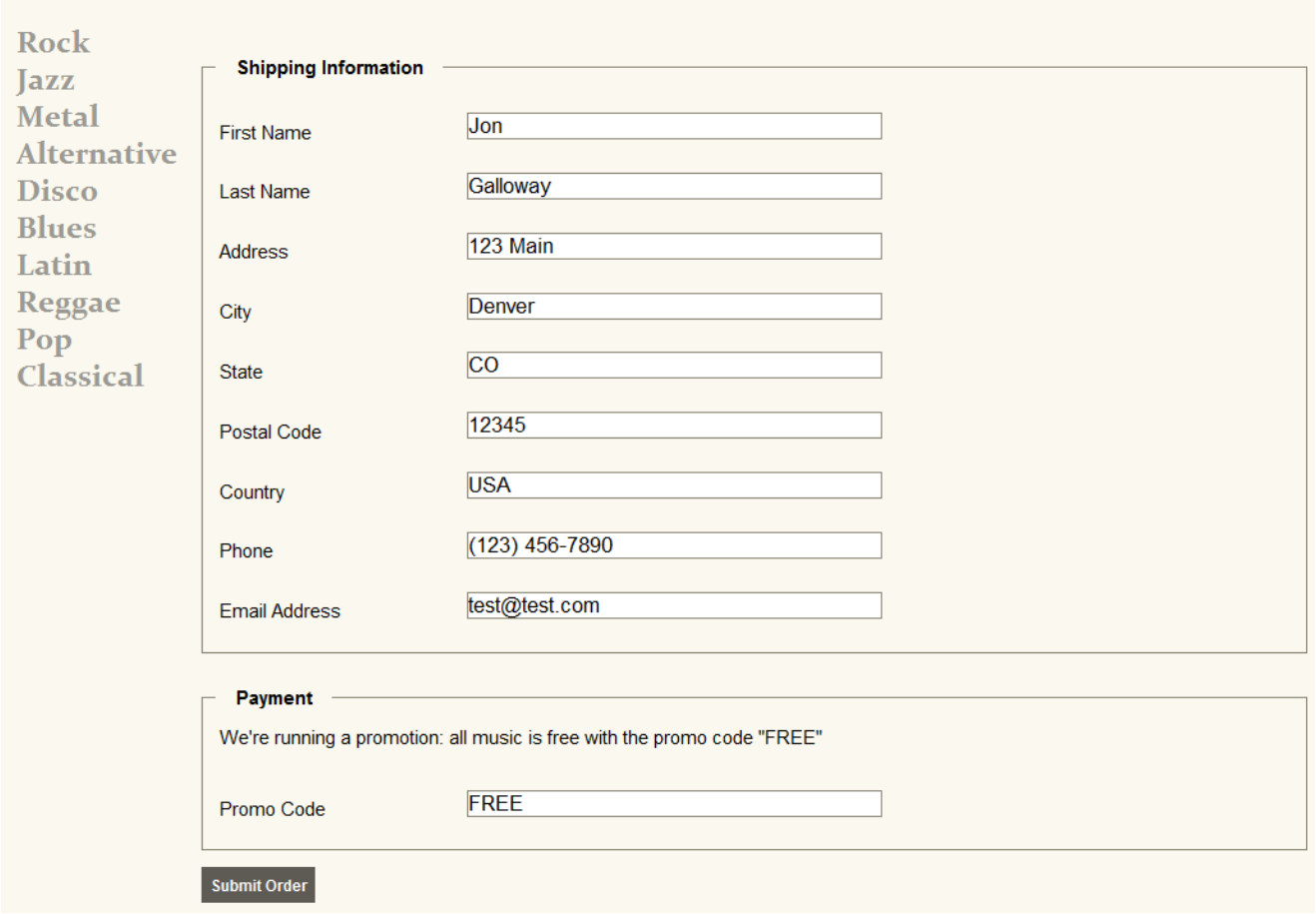


Proceeding to Checkout will prompt them to login or register for a user account.

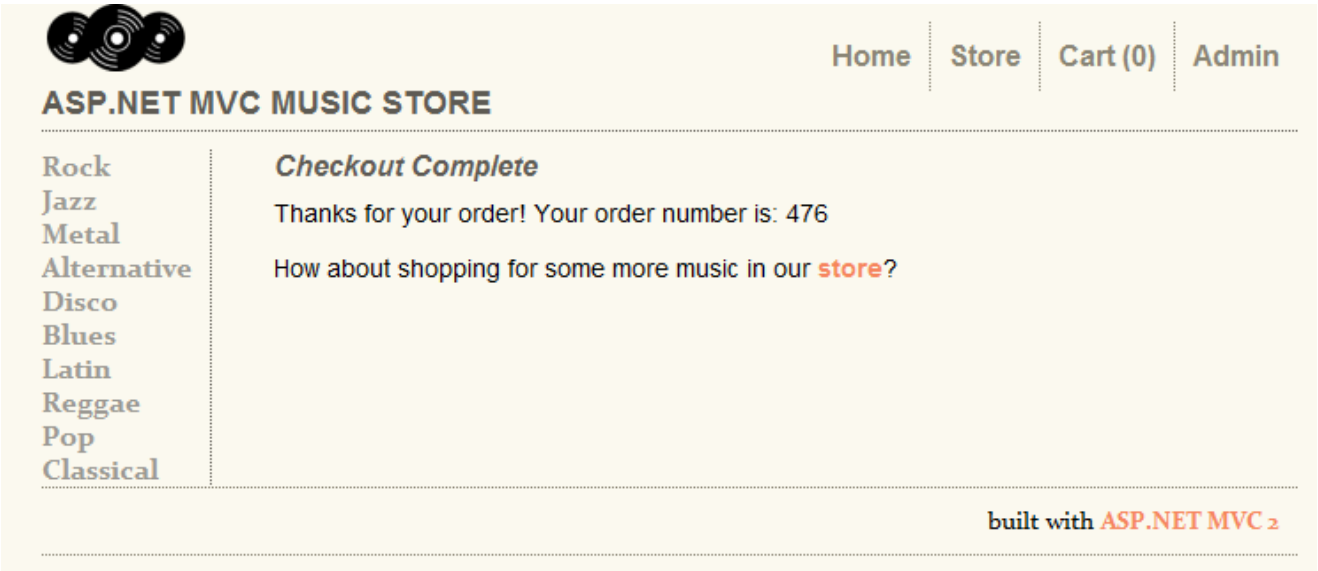




After creating an account, they can complete the order by filling out shipping and payment information. To keep things simple, we're running an amazing promotion: everything's free if they enter promotion code "FREE"!



After ordering, they see a simple confirmation screen:



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

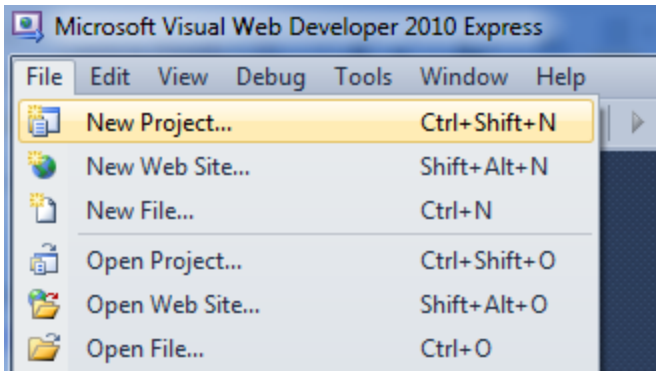
This tutorial will begin by creating a new ASP.NET MVC 2 project using the free Visual Web Developer 2010 Express (which is free), and then we'll incrementally add features to create a complete functioning application. Along the way, we'll cover database access, form posting scenarios,, data validation,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

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

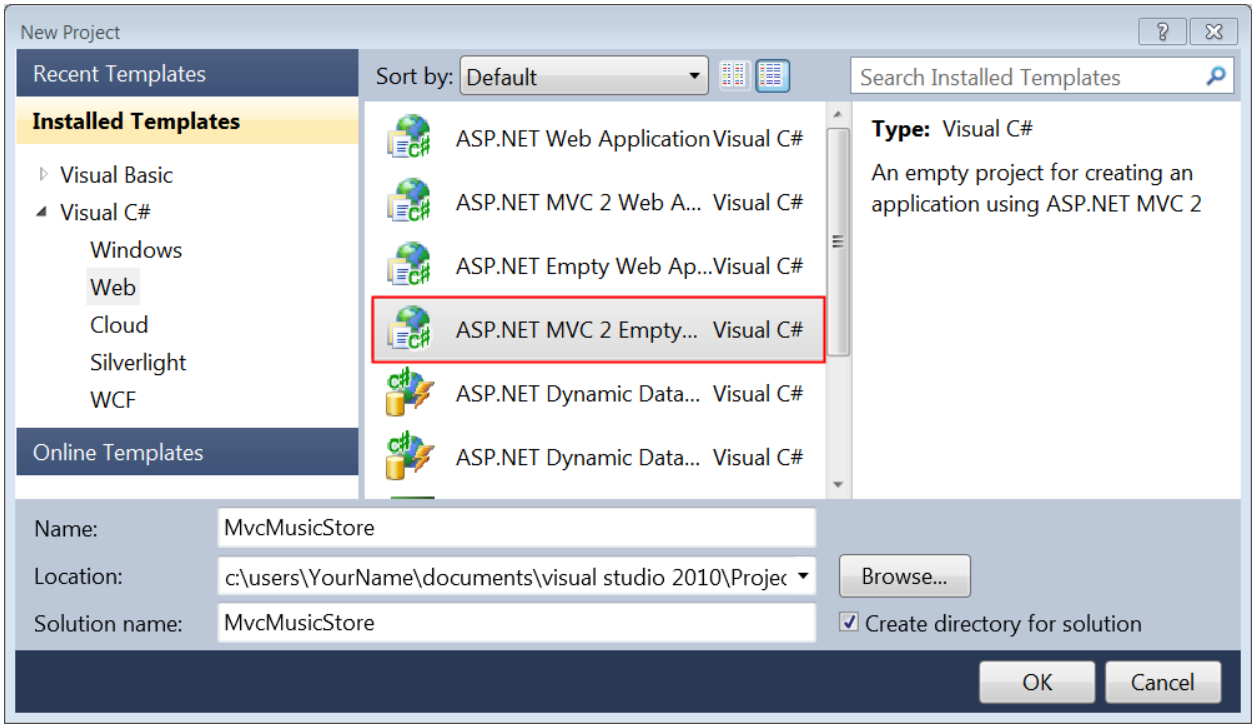
You can use either Visual Studio 2010 or the free Visual Web Developer 2010 Express to build the application. We'll be using the free SQL Server Express to host the database. You can install ASP.NET MVC, Visual Web Developer Express and SQL Server Express using a simple installer here: http://www.asp.net/downloads

1. File -> New Project

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

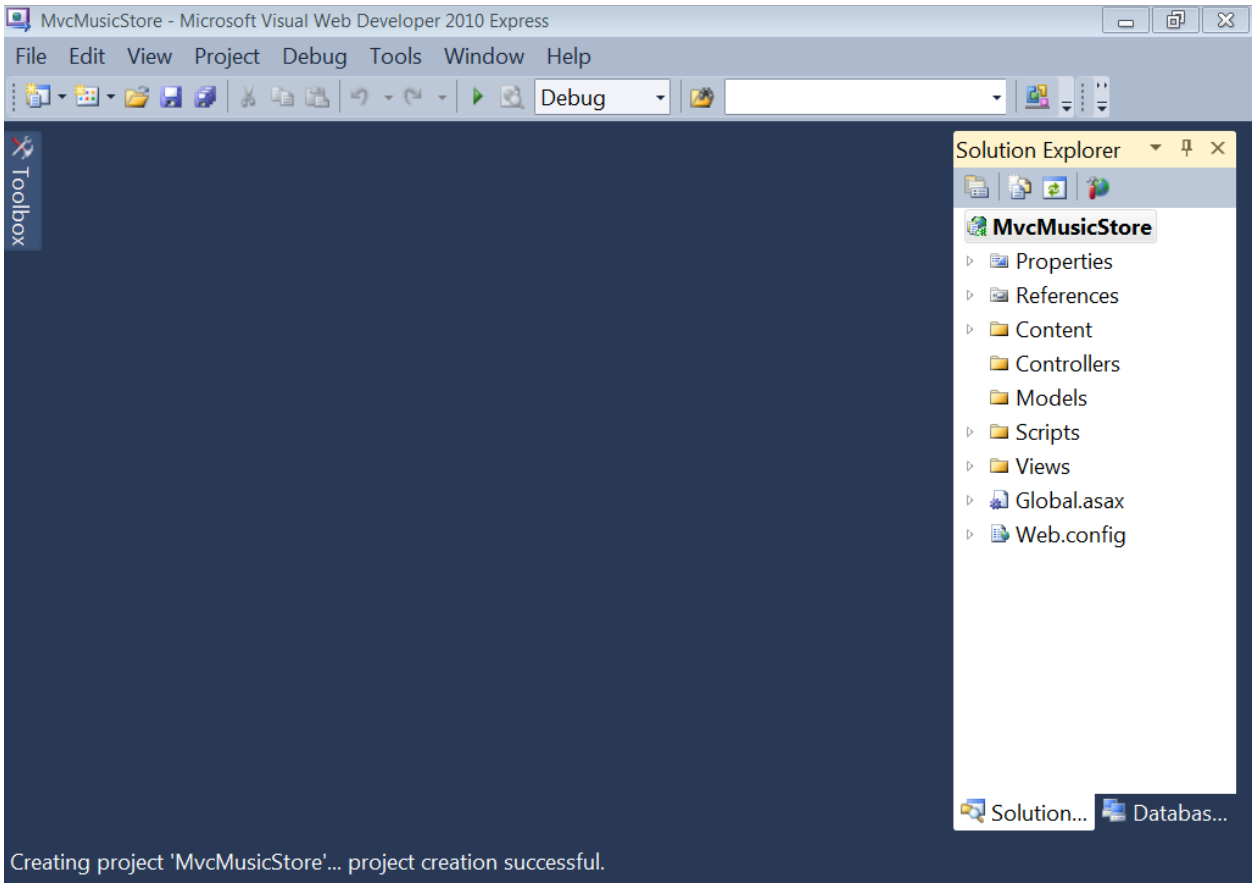


We'll select the Visual C# -> Web Templates group on the left, then choose the "ASP.NET MVC 2 Empty Web Application" template in the center column. Name your project MvcMusicStore and press the OK button.

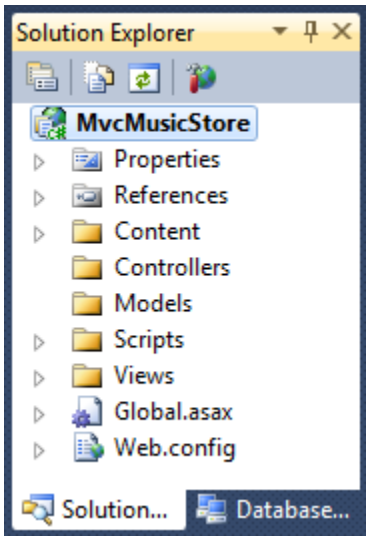


Note: The "New Project" dialog has both a "ASP.NET MVC 2 Web Application" project template and a "ASP.NET MVC 2 Empty Web Application" template. We'll be using the "empty" project template for this tutorial.

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 2 template isn't completely empty – it adds a basic folder structure:



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

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

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

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

2. Controllers

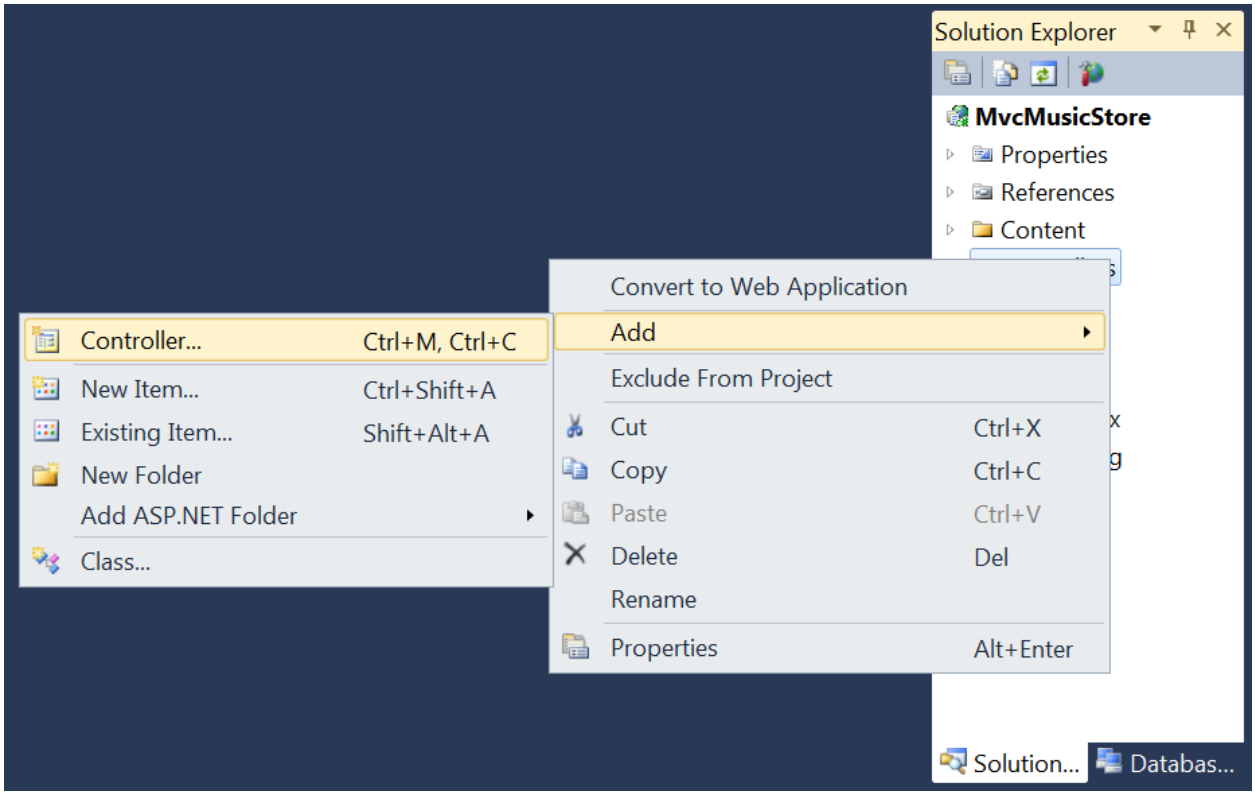
With traditional web frameworks, incoming URLs are typically mapped to files on disk. For example: a request for a URL like "/Products.aspx" or "/Products.php" might be processed by a "Products.aspx" or "Products.php" file.

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

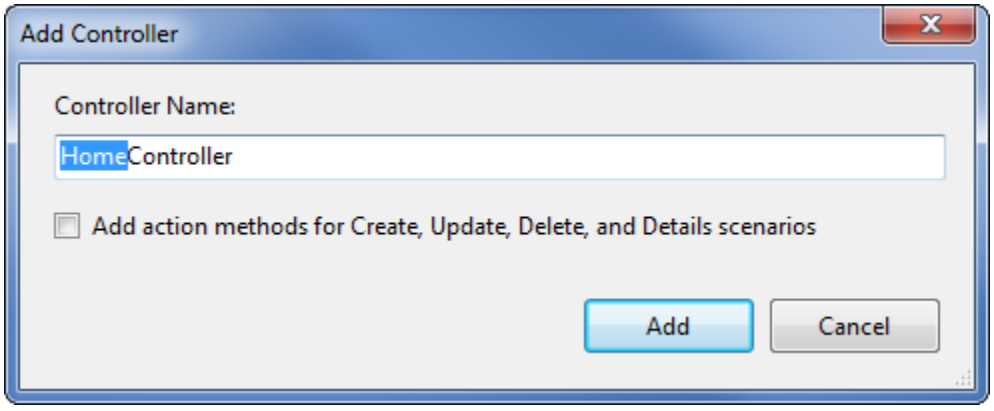
Adding a HomeController

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

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



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



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

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

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

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

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

    }
}
```

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

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

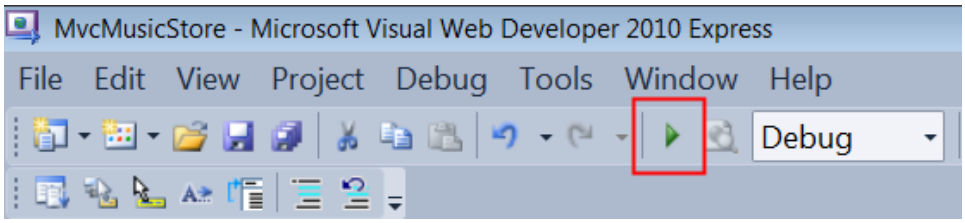
The method should now look like this:

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

Running the Application

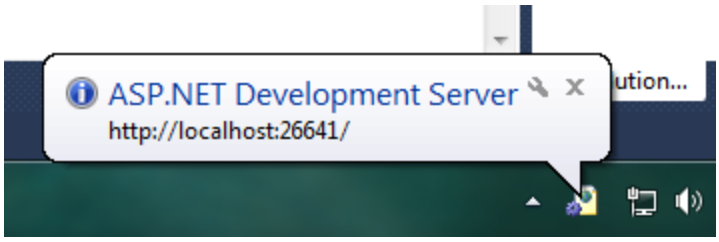
Now let's run the site. We can start our web-server and try out the site using any of the following::

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

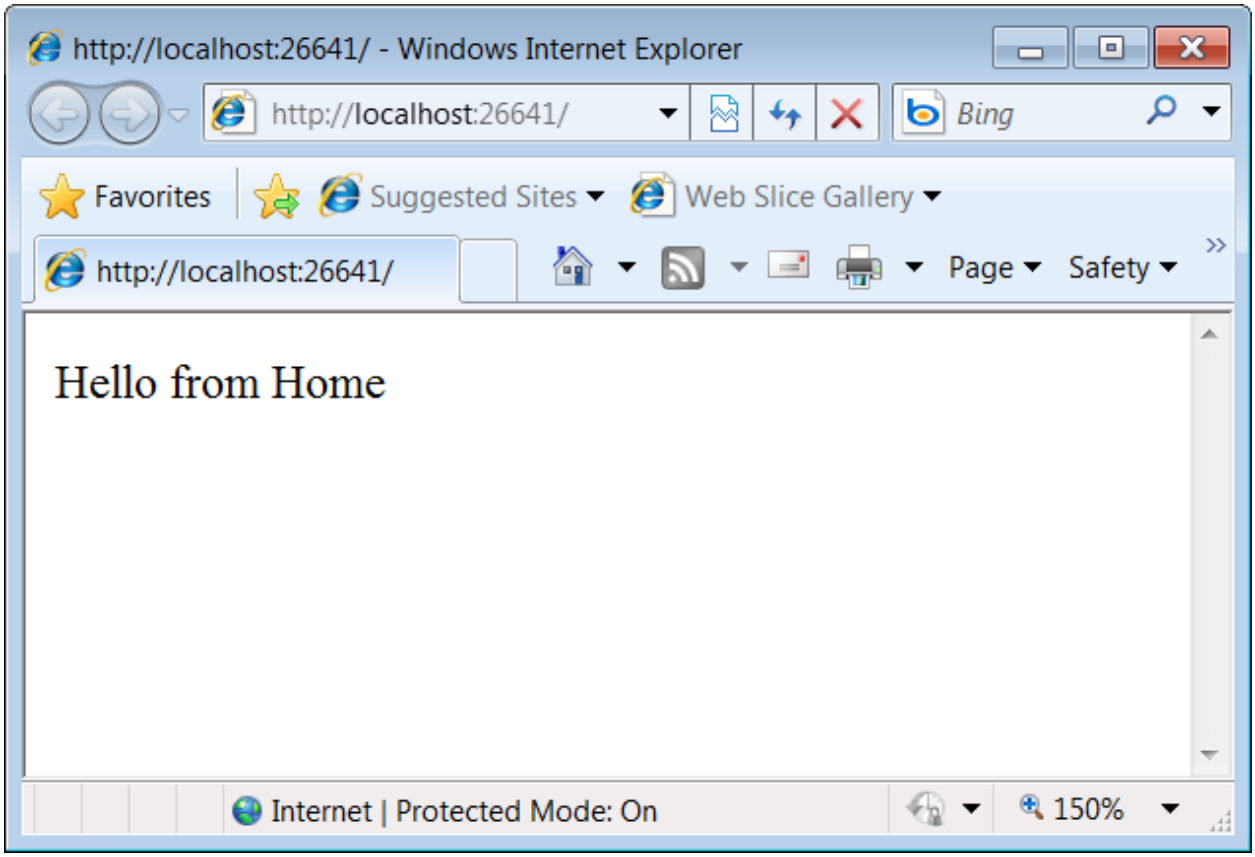


- Use the keyboard shortcut, F5.

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



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



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

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

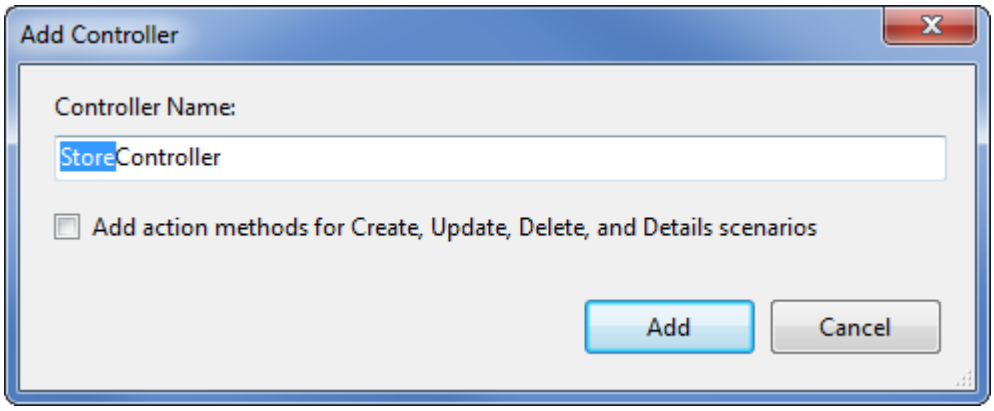
Adding a StoreController

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

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

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

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



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

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

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

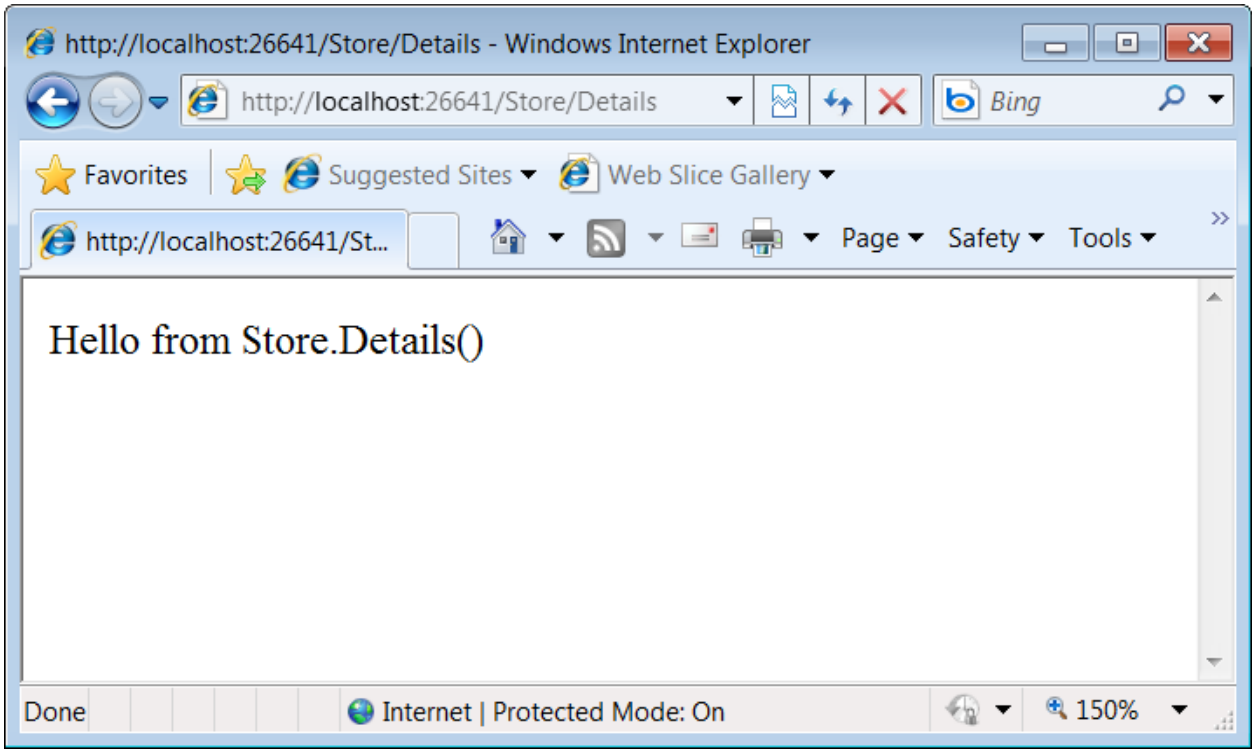
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
```

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

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

        //
        // GET: /Store/Browse

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

        //
        // GET: /Store/Details

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

Run the project again and browse the following URLs:

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

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



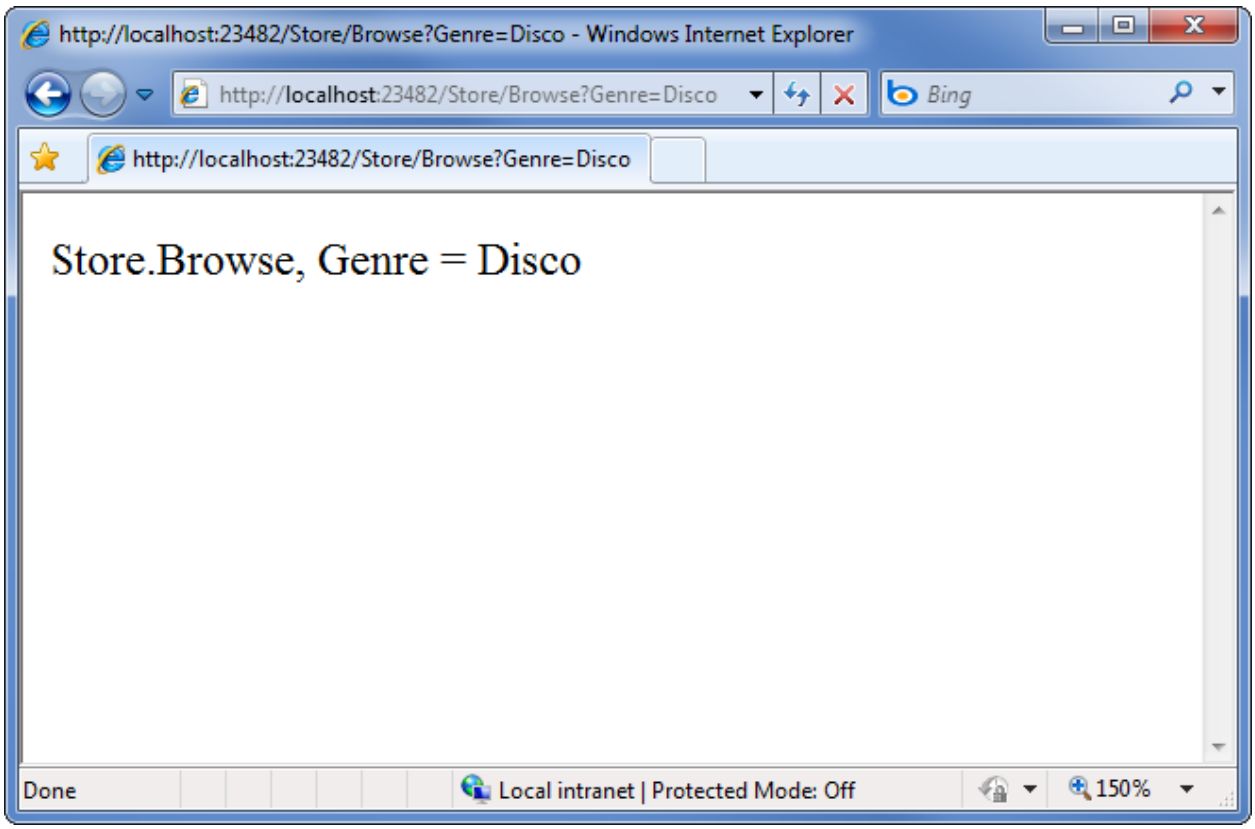
That's great, but these are just constant strings. Let's make them dynamic, so they take information from the URL and display it in the page output.

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

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

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

    return message;
}
```

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

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



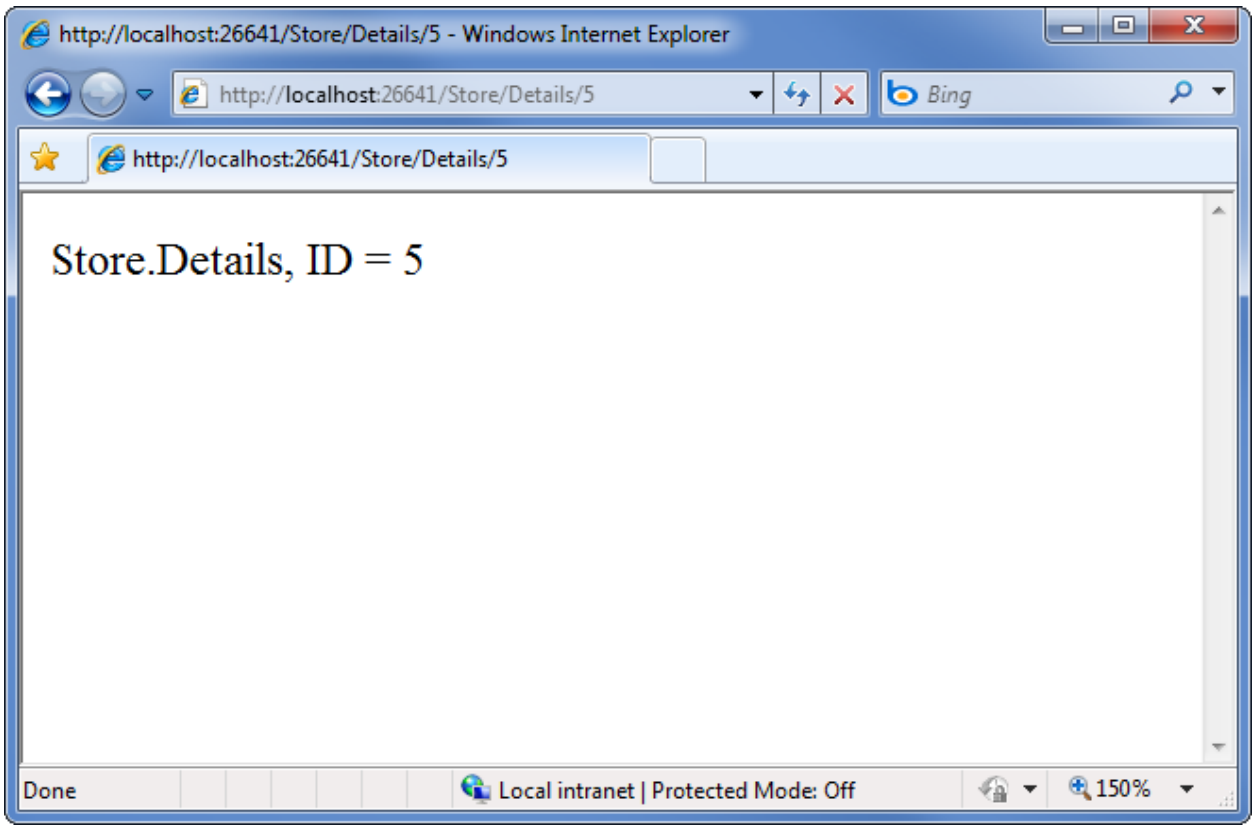
Note: We're using the Server.HtmlEncode utility method to sanitize the user input. This prevents users from injecting Javascript into our View with a link like /Store/Browse?Genre=<script>window.location='http://hackersite.com'</script>.

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

ASP.NET MVC let's us easily do this without having to configure anything. ASP.NET MVC's default routing convention is to treat the segment of a URL after the action method name as a parameter named "ID". If your action method has a parameter named ID then ASP.NET MVC will automatically pass the URL segment to you as a parameter.

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

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

    return message;
}
```

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



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

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

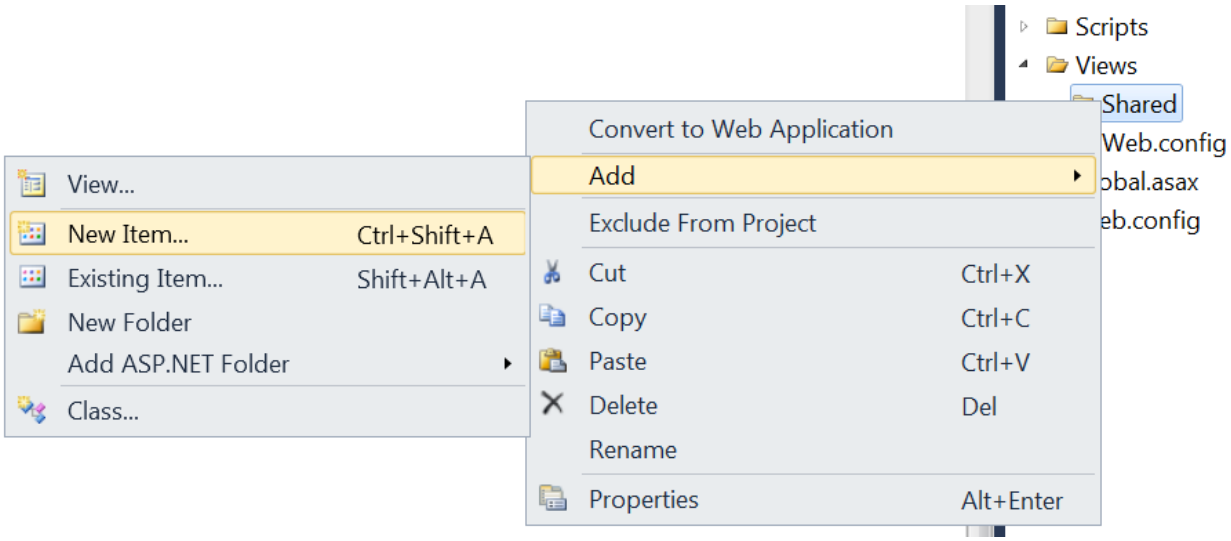
3. Views and ViewModels

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.

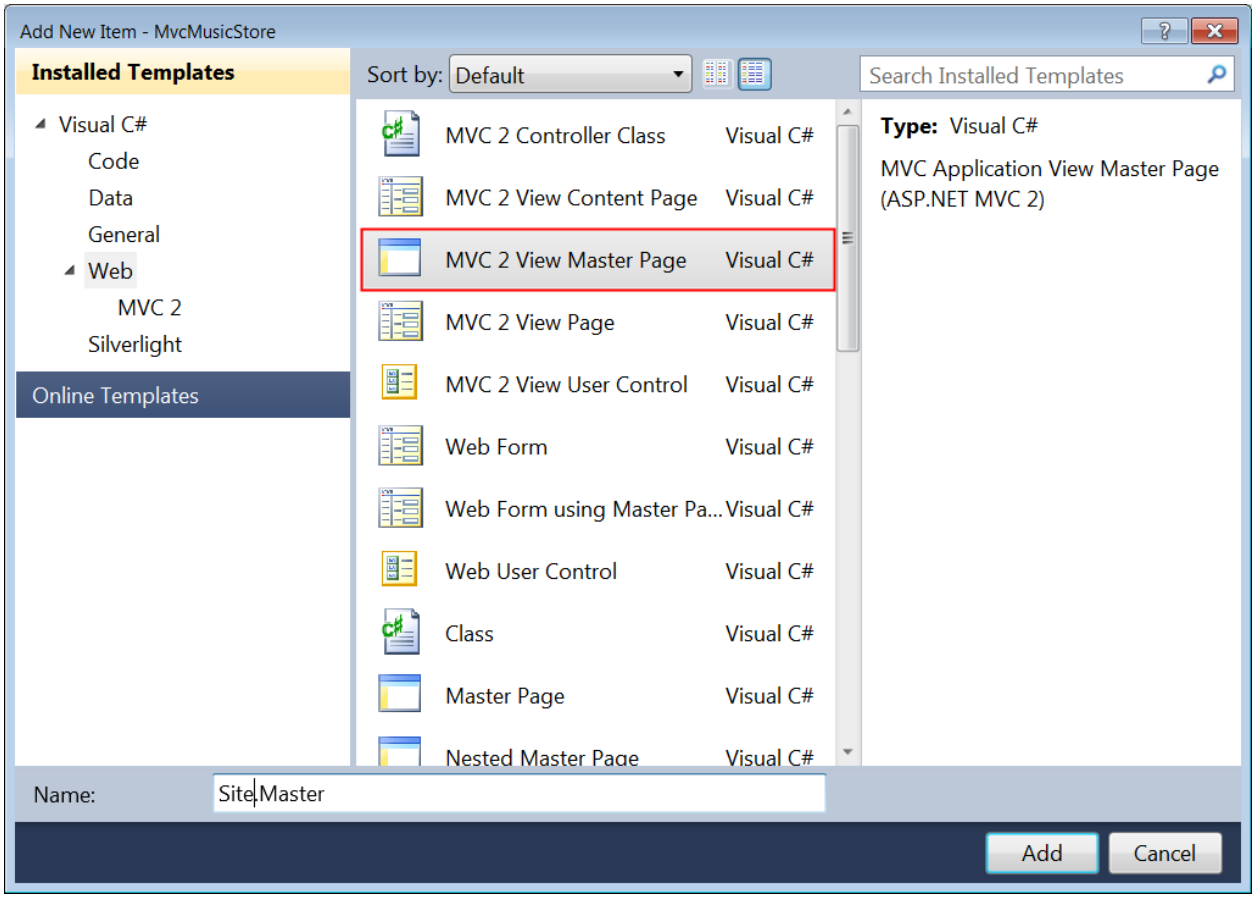
Using a MasterPage for common site elements

We aregoing to update our HomeController to use a view template to return HTML. Before we implement the view template, though, we'll first add a CSS stylesheet and a MasterPage to our site. ASP.NET MasterPages allow us to setup a template for common HTML that we will use across the entire website. These are similar to include files, but a lot more powerful.

A MasterPage template is a layout file that is typically shared by all controllers in a site. Because it is a shared resource we'll create it under the /Views/Shared folder. Expand the Views folder and right-click the Shared folder, then select Add ⇨ New Item… ⇨ MVC 2 View Master Page.



Name it Site.Master and click the Add button.



Our Site.master template will contain the HTML container layout for all pages on our site. It contains the <html> element for our HTML response, as well as the <head> and <body> elements. We'll be able to add <asp:ContentPlaceholder/> tags within the HTML content that identify regions our view templates can "fill in" with dynamic content.

We'll use a CSS stylesheet to define the styles of our site. We'll add a reference to this by adding a <link> element into the <head> tag of our Site.Master file:

```
<head runat="server">
    <link href="/Content/Site.css" rel="Stylesheet" type="text/css" />
    <title><asp:ContentPlaceHolder ID="TitleContent" runat="server" /></title>
</head>
```

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 Site.master template directly below the opening element.

Our Site.Master now looks like this:

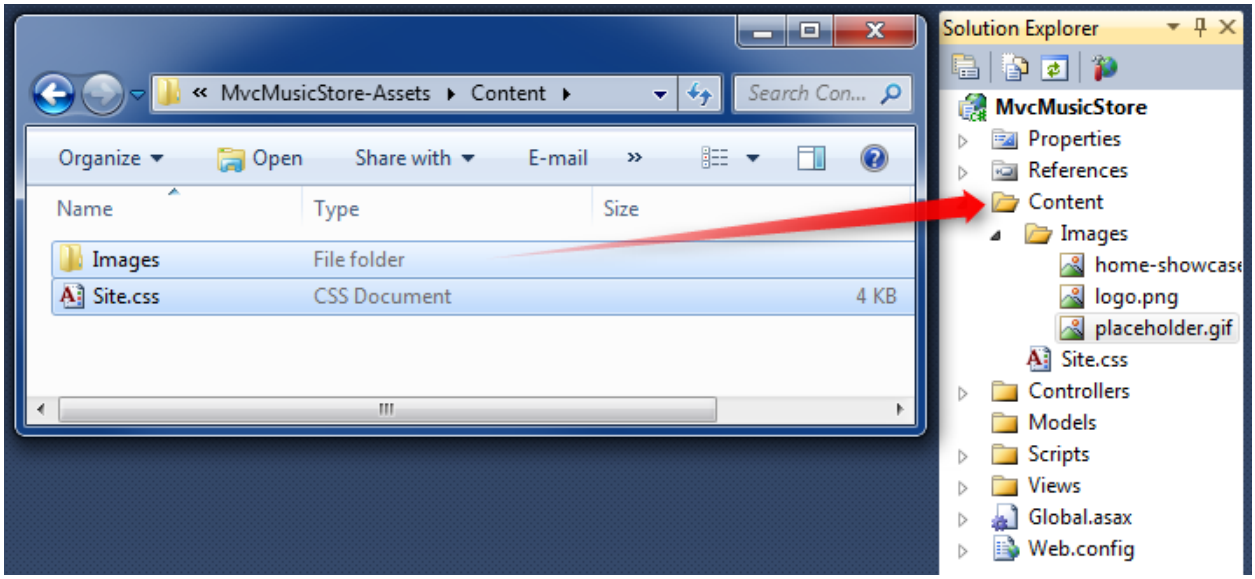
```
<%@ Master Language="C#" Inherits="System.Web.Mvc.ViewMasterPage" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

```
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <link href="/Content/Site.css" rel="Stylesheet" type="text/css" />
    <title><asp:ContentPlaceHolder ID="TitleContent" runat="server" /></title>
</head>
<body>
    <div>
        <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>
        <asp:ContentPlaceHolder ID="MainContent" runat="server">

        </asp:ContentPlaceHolder>
    </div>
</body>
</html>
```

Adding a 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:



Adding a View template

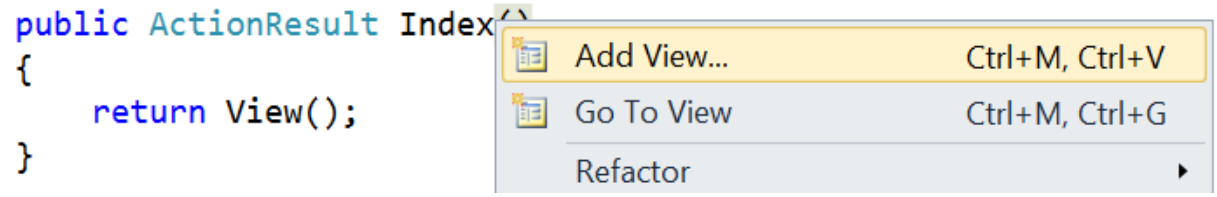
Let's now return to our HomeController and have it take advantage of a View template to generate HTML responses to visitors.

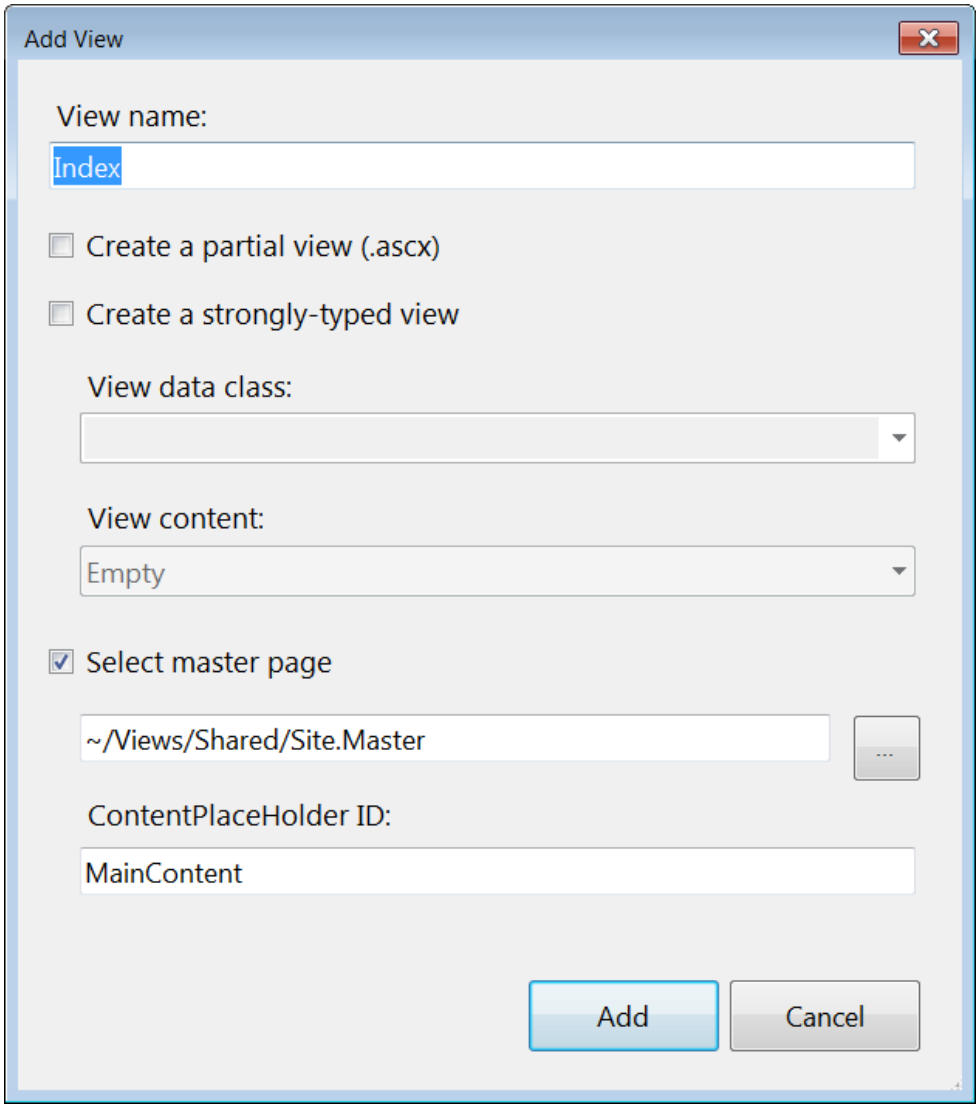
To use a view-template, we'll change the HomeController Index method to return an ActionResult, and have it return View(), like below:

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

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

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

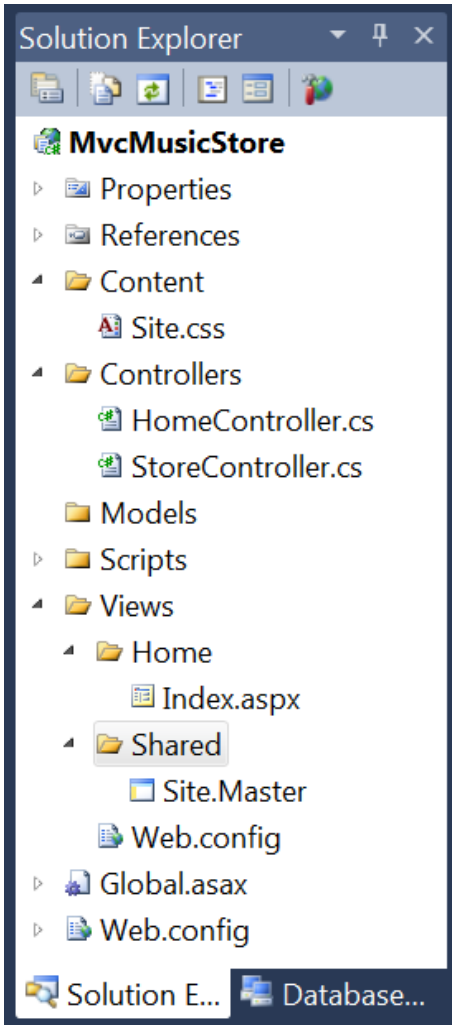
We'll now add an appropriate View template to our project. To do this we'll position the text cursor within the Index action method, then right-click and select "Add View". This will bring up the Add View dialog:





The "Add View" dialog allows us to quickly and easily generate View template files. By default the "Add View" dialog pre-populates the name of the View template to create so that it matches the action method that will use it. Because we used the "Add View" context menu within the Index() action method of our HomeController, the "Add View" dialog above has "Index" as the view name pre-populated by default. Because we have a Site.Master MasterPage template within our project, it also pre-filled that name as the master page template our view should be based on.

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



The name and folder location of the "Index.aspx" 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.aspx view template when we write code like below within our HomeController:

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

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

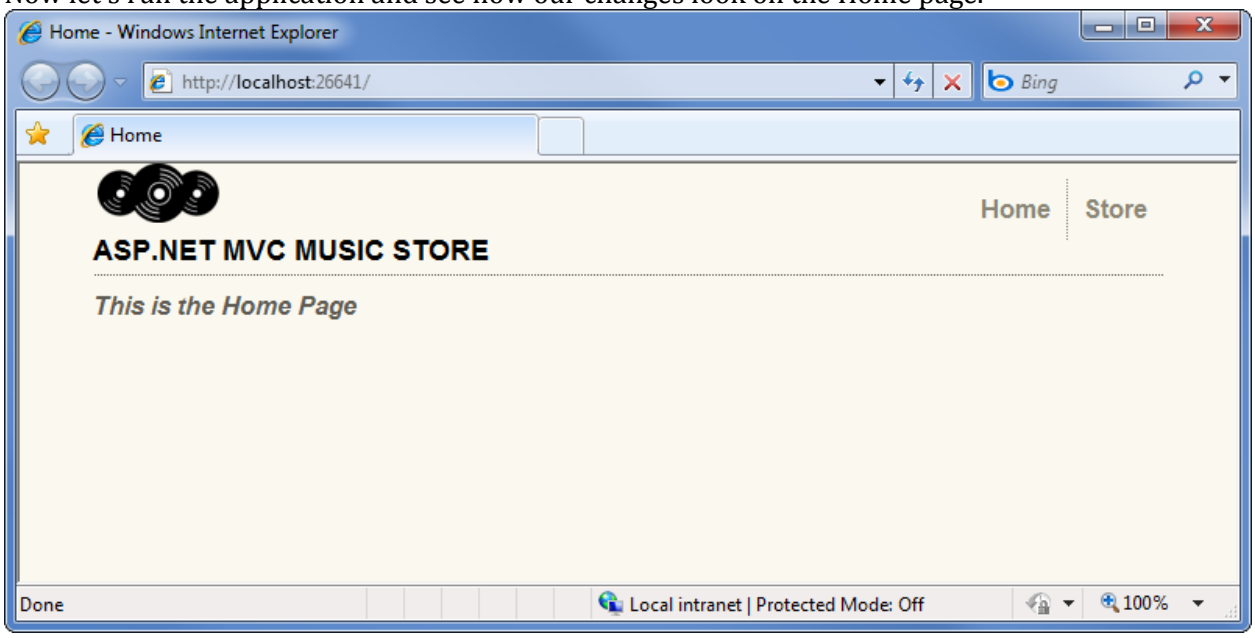
```
}
```

Visual Studio created and opened the "Index.aspx" view template after we clicked the "Add" button within the "Add View" dialog. The view template is based on the Site.master template we defined earlier, and contains two <asp:content> sections that enables us to override/fill-in our page content.

Let's update the Title to "Home", and change the main content to say "This is the Home Page", as shown in the code below:

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage" %>
<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
       Home
</asp:Content>
<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

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

</asp:Content>
```

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



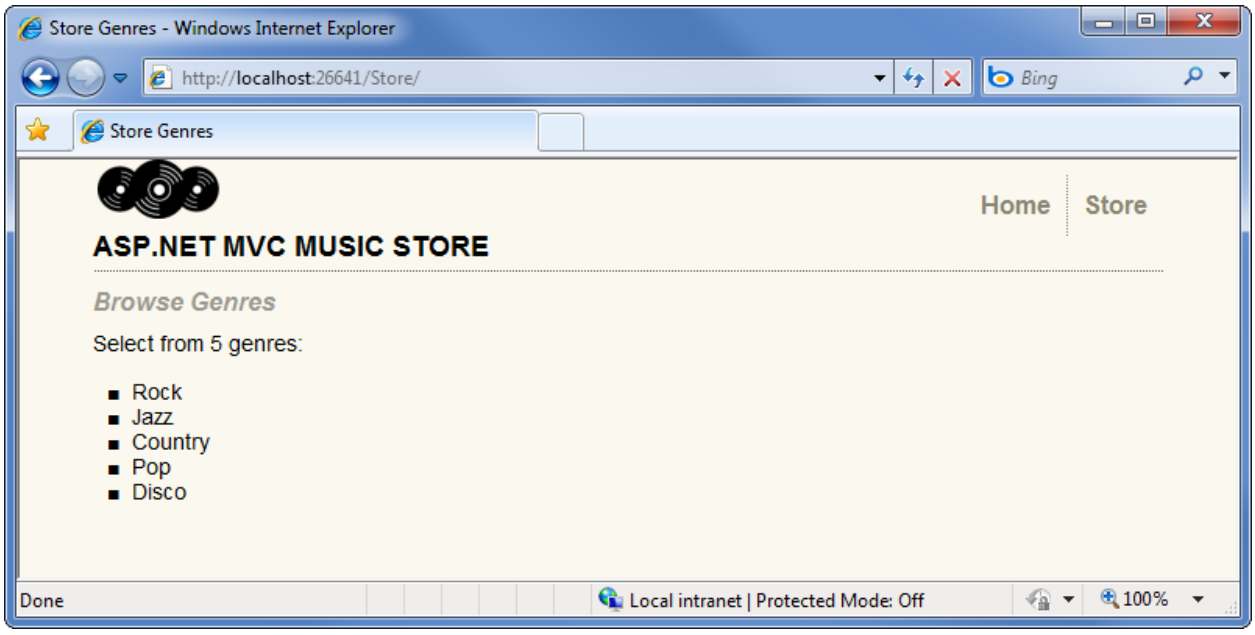
- Let's review what's changed:The HomeController's Index action method found and displayed the \Views\Home\Index.aspx View template, even though our code called "return View()", because our View template followed the standard naming convention.
- The Home Page is displaying a simple welcome message that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

Using a ViewModel 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.

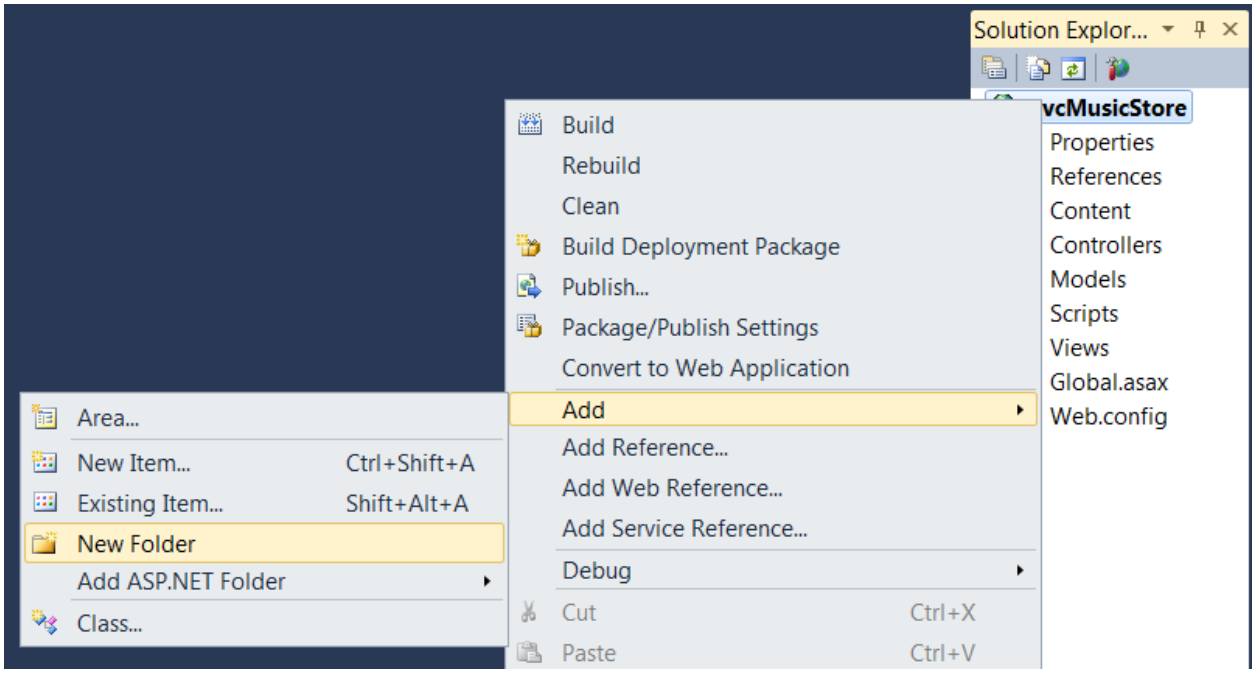
One common technique we can use to enable this is the "ViewModel" pattern, which allows a Controller to cleanly package up all the information needed to generate a response, and then pass this information off to a View template to use to generate the appropriate HTML response. We will use this ViewModel pattern within our MVC Music Store.

Let's begin by changing our Store Index page to list the music genres our store carries, so it looks like this:

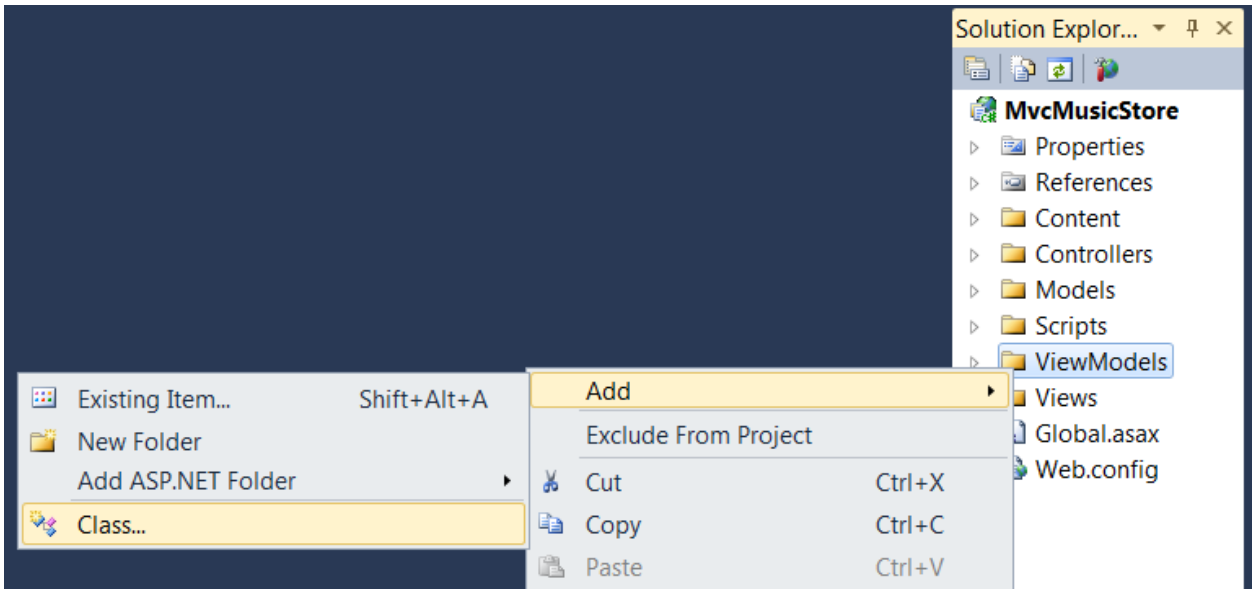


To implement the above UI we need to pass two bits of data from our StoreController to the View template that generates the response: 1) the number of genres in the store, 2) a list of those genres. We'll create a ViewModel class to help encapsulate this information.

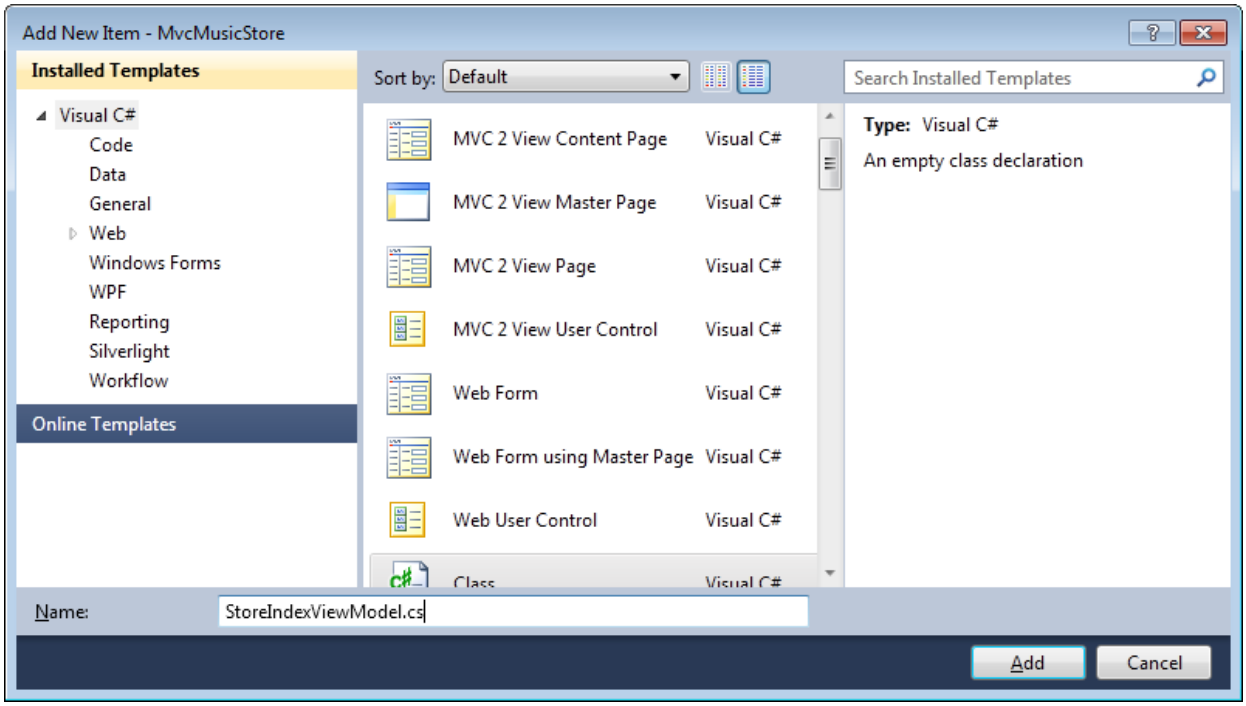
We'll begin by creating a ViewModel directory to hold our ViewModel. We'll do this by right-clicking our top-level MvcMusicStore project, select Add⇨New Folder, and name the new folder "ViewModels":



Next, we'll create a ViewModel class that we'll use to implement our Store genre listing scenario scenario. Right-click on the ViewModels folder and select Add⇨Class…



Name the class StoreIndexViewModel and press the Add button:



If you recall, we said we needed to pass two bits of information from our StoreController to a View template to generate the HTML response we wanted: 1) the number of genres in the store, 2) a list of those genres. We'll do this by adding two properties to our StoreIndexViewModel class:

- A property named "NumberOfGenres" which is an integer
- A property named "Genres" which is a List of strings

The code to do this looks like below:

```
using System;
using System.Collections.Generic;

namespace MvcMusicStore.ViewModels
{
    public class StoreIndexViewModel
    {
        public int NumberOfGenres { get; set; }
        public List<string> Genres { 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.*

We now have a class that encapsulates the information we need to pass from our StoreController's Index() method to a View template which generates a response. Let's now update the StoreController to use it.

In order to use the StoreIndexViewModel class from our StoreController, we first need to add the following namespace using statement to the top of the StoreController code:

```
using MvcMusicStore.ViewModels;
```

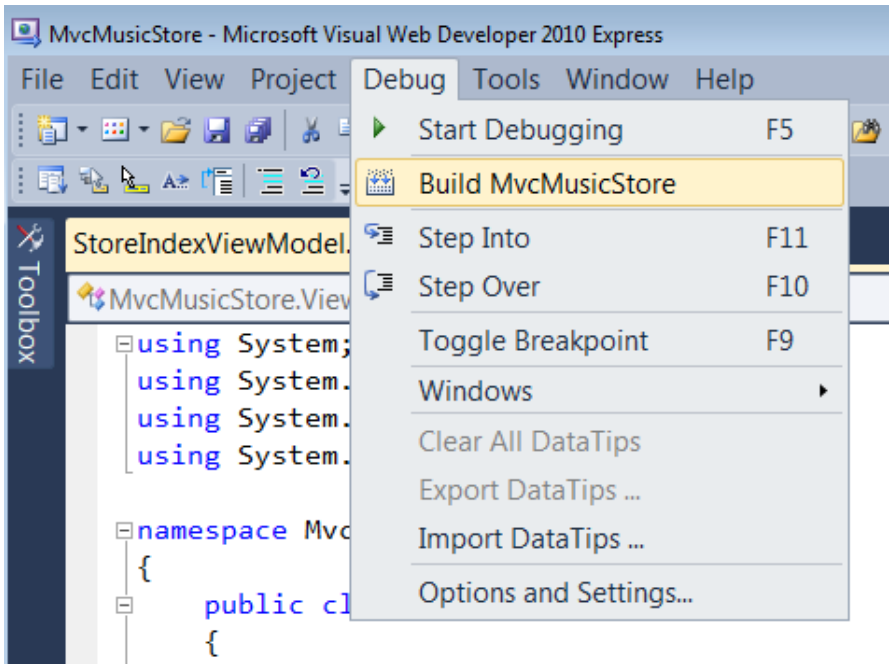
Now we'll change the StoreController's Index action method so that it creates and populates a StoreIndexViewModel object and then passes it off to a View template to generate an HTML response with it.

Later in this tutorial we will write code that retrieves the list of store genres from a database. To begin with, though, we'll just use the following C# code to create some "dummy data genres" that we will populate our StoreIndexViewModel with:. Notice that after creating and setting up our StoreIndexViewModel object, we are passing it as an argument to the View() method. This indicates that we want to pass it to our View template to use:

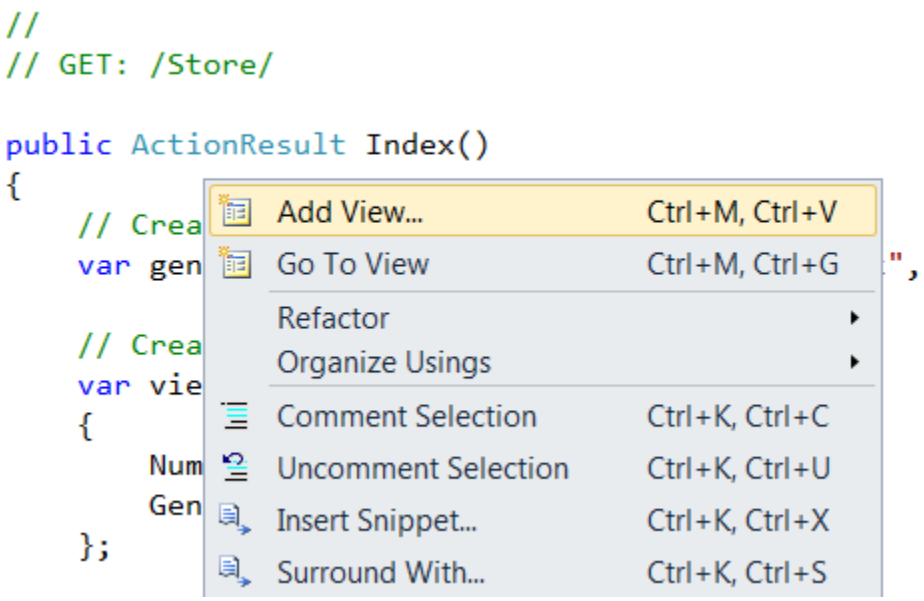
```
//
// GET: /Store/

public ActionResult Index()
{
    // Create a list of genres
    var genres = new List<string> {"Rock", "Jazz", "Country", "Pop", "Disco" };

    // Create our view model
    var viewModel = new StoreIndexViewModel {
        NumberOfGenres = genres.Count(),
        Genres = genres
    };

    return View(viewModel);
}
```

Note: If you're unfamiliar with C#, you may assume that using var means that our viewModel 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 viewModel is of type StoreIndexViewModel and compiling the local viewModel variable as a StoreIndexViewModel type, so we get compile-time checking and Visual Studio code-editor support.

Let's now create a View template that uses our StoreIndexViewModel 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 StoreIndexViewModel class. You can build the project by selecting the Debug⇨Build MvcMusicStore menu item.



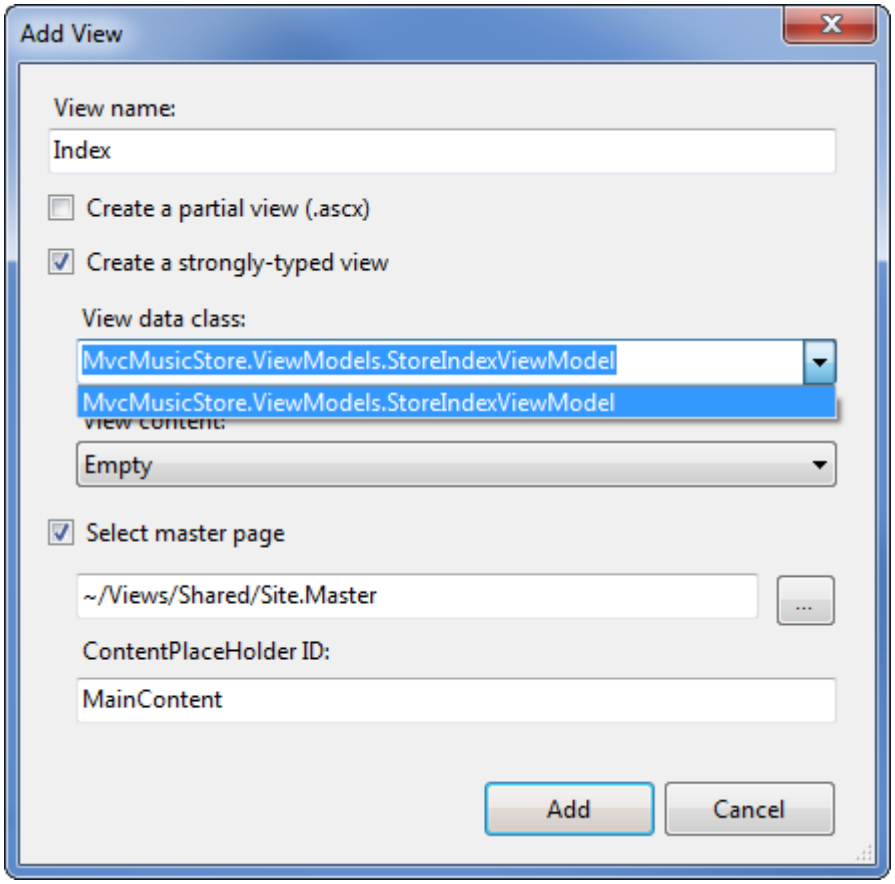
Right-click Store.Index() and click "Add View".



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.aspx file. Like before it will also be based on our Site.master MasterPage template

Unlike before, we are going to check the "Create a strongly-typed" view checkbox. We are then going to select our "StoreIndexViewModel" class within the "View data-class" drop-downlist. This will cause the

"Add View" dialog to create a View template that expects that a StoreIndexViewModel object will be passed to it to use.



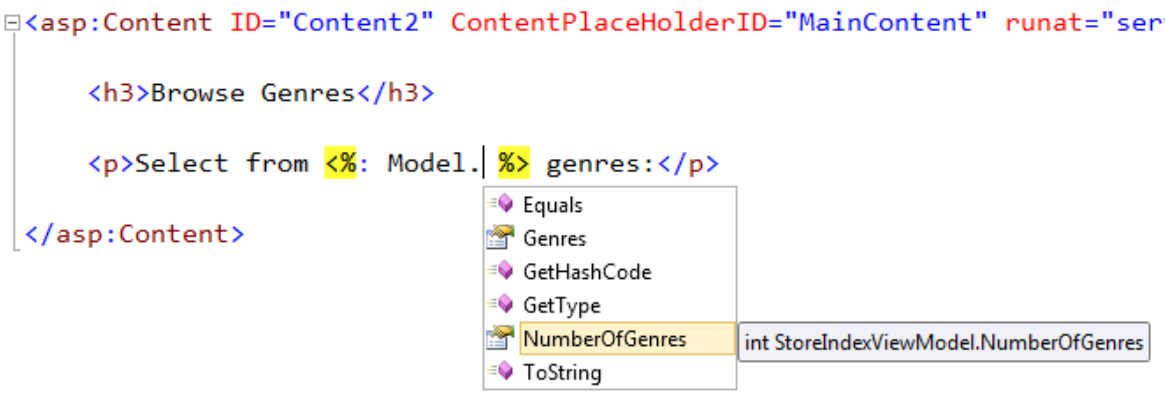
When we click the "Add" button our \Views\Store\Index.aspx View template will be created.

We'll now update the View template to output the number of genres within a message at the top of the page. We'll be using <%: %> syntax (often referred to as "code nuggets") to execute code within our View template. There are two main ways you'll see this used:

- Code enclosed within <% %> will be executed
- Code enclosed within <%: %> will be executed, and the result will be output to the page

Note: Prior to ASP.NET 4, the <%= %> syntax was used to execute code and write it out to the page. Starting with ASP.NET 4, you should use the <%: %> syntax instead, since it will automatically HTML Encode the results, which helps protect against cross-site scripting and HTML injection attacks..

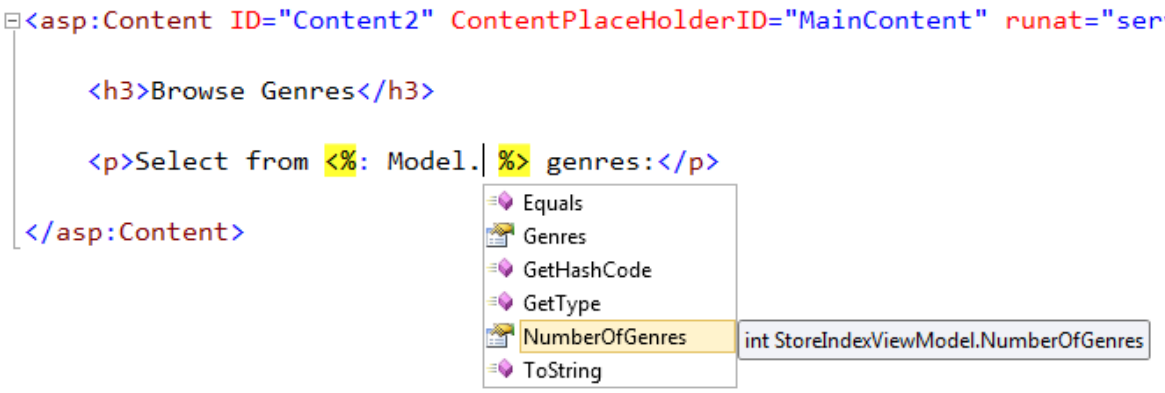
Start typing the code below within a <%: %> code nugget block:



Note that as soon as you finish typing the period after the word "Model", Visual Studio's IntelliSense feature kicks in and supplies you with a list of possible properties and methods to choose from.

The Model property references the StoreIndexViewModel object that our controller passed to the View template. This means that we can access all of the data passed by our controller to our View template via the Model property, and format it into an appropriate HTML response within the View template..

You can just select the "Model.NumberOfGenres" property from the Intellisense list rather than typing it in.



When you click the tab key it will auto-complete it. Our View template will now look like:

```
<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
    <h3>Browse Genres</h3>
    <p>Select from <%: Model.NumberOfGenres %> genres:</p>
</asp:Content>
```

Next we'll loop over the list of Genre string in our StoreIndexViewModel and create an HTML <ul> list using a foreach loop, like this:

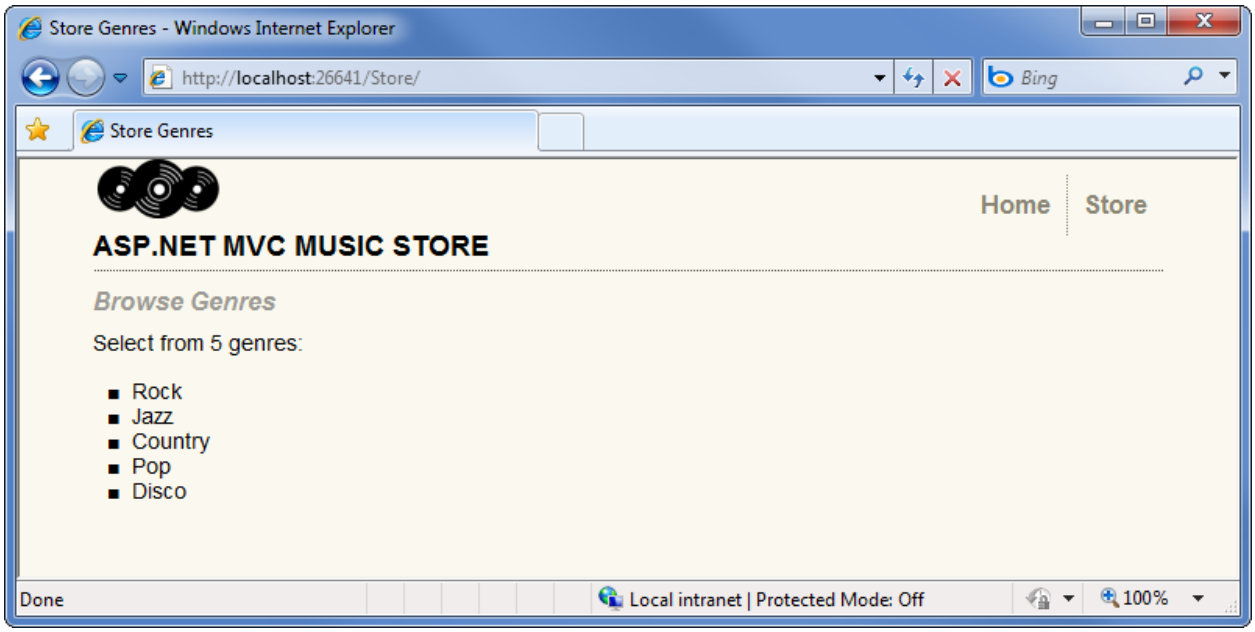
```
<ul>
    <% foreach (string genreName in Model.Genres) { %>
```

```
        <li>
            <%: genreName %>
        </li>
    <% } %>
</ul>
```

Our completed View template is shown below. If you look at the top of the Index page View template, you'll notice that the template inherits a ViewPage<StoreIndexViewModel>. It is this declaration that causes the "Model" property on our View template to be strongly-typed as a "StoreIndexViewModel" object.:

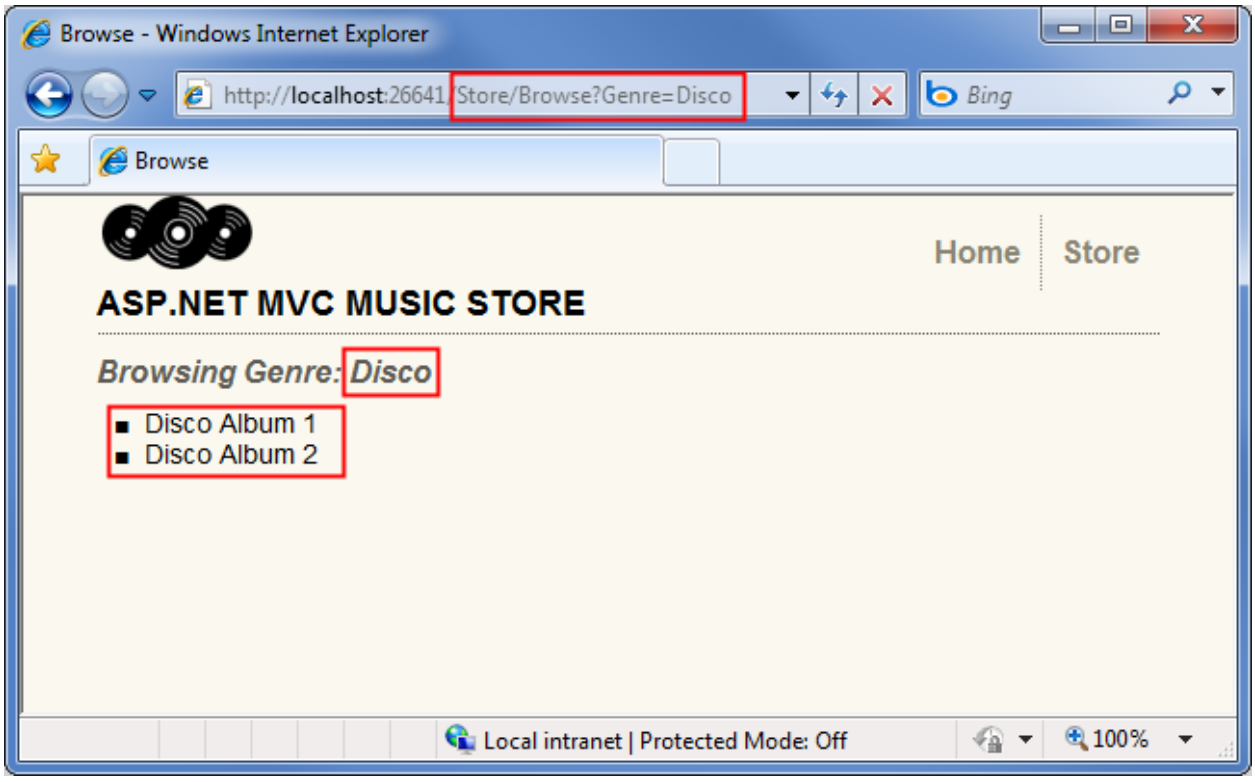
```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage<MvcMusicStore.ViewModels.StoreIndexViewModel>" %>

<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
      Store Genres
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

    <h3>Browse Genres</h3>

    <p>Select from <%: Model.NumberOfGenres %> genres:</p>

    <ul>
        <% foreach (string genreName in Model.Genres) { %>
            <li>
             <%: genreName %>
            </li>
        <% } %>
    </ul>

</asp:Content>
```

Run and visit the "/Store" URL. Our page now looks like below:



More complex ViewModels for Store Browse and Index

Now let's take a look at a slightly more complex example with the /Store/Browse URL. This page reads the Genre name from the URL and displays the Genre name and lists the Albums within it, as shown below.



First we'll create some Model classes to represent Genres and Albums. Unlike ViewModels, which are created just to pass information from our Controller to our View, **Model classes are built to contain and manage our data and domain logic.**

We'll be using the concept of a "Genre" and "Album" repeatedly, so we'll create classes that represent them. Later on, we'll be hooking our Genre and Album classes to a database, and they'll map directly to database tables.

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; }
}
```

Now let's create an Album class that has a Title and a Genre property:

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

Our /Store/Browse page will show one Genre and all the Albums in that Genre. A ViewModel is a great way to expose that information to the view. Let's create a StoreBrowseViewModel class (again, right-clicking the ViewModels folder and selecting Add⇨Class).

We'll add a using namespace statement to the top of the new StoreBrowseViewModel.cs file to reference our Models folder, then add a Genre property and a List of Albums. Here's how the class looks after our changes:

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

namespace MvcMusicStore.ViewModels
{
    public class StoreBrowseViewModel
    {
        public Genre Genre { get; set; }
        public List<Album> Albums { get; set; }
    }
}
```

Now we're ready to change the StoreController's Browse and Details action methods to use our new ViewModel. We'll start by adding a using MvcMusicStore.Models statement to the using statements list at the top of the StoreController class, like this:

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

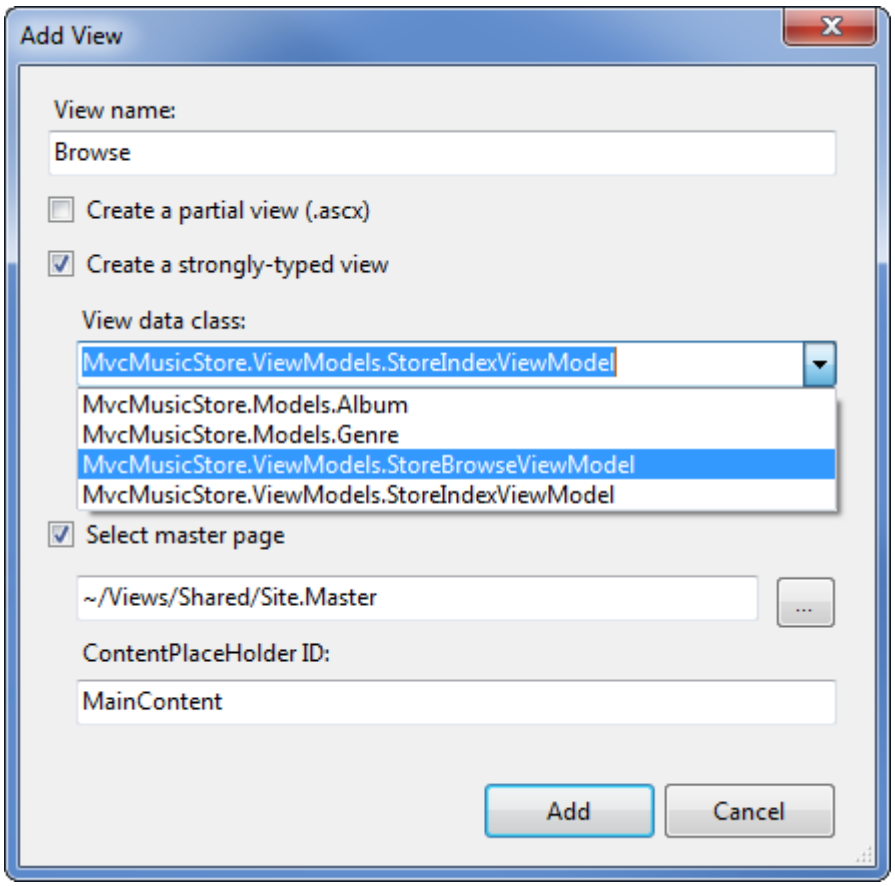
```
using System.Web.Mvc;
using MvcMusicStore.ViewModels;
using MvcMusicStore.Models;
```

We'll then then modify the Browse and Details methods to appear as follows. 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:

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

public ActionResult Browse(string genre)
{

    var genreModel = new Genre {
        Name = genre
    };

    var albums = new List<Album>(){
       new Album { Title = genre + " Album 1" },
       new Album { Title = genre + " Album 2" }
    };

    var viewModel = new StoreBrowseViewModel
    {
        Genre = genreModel,
        Albums = albums
    };

    return View(viewModel);
}
```

Build the project by selecting Debug⇨Build MvcMusicStore menu item, as before.

Now that we've set up our supporting classes, we're ready to build our View template. Right-click on Browse and add a strongly typed Browse view. Set the View Data Class to be "StoreBrowseViewModel":

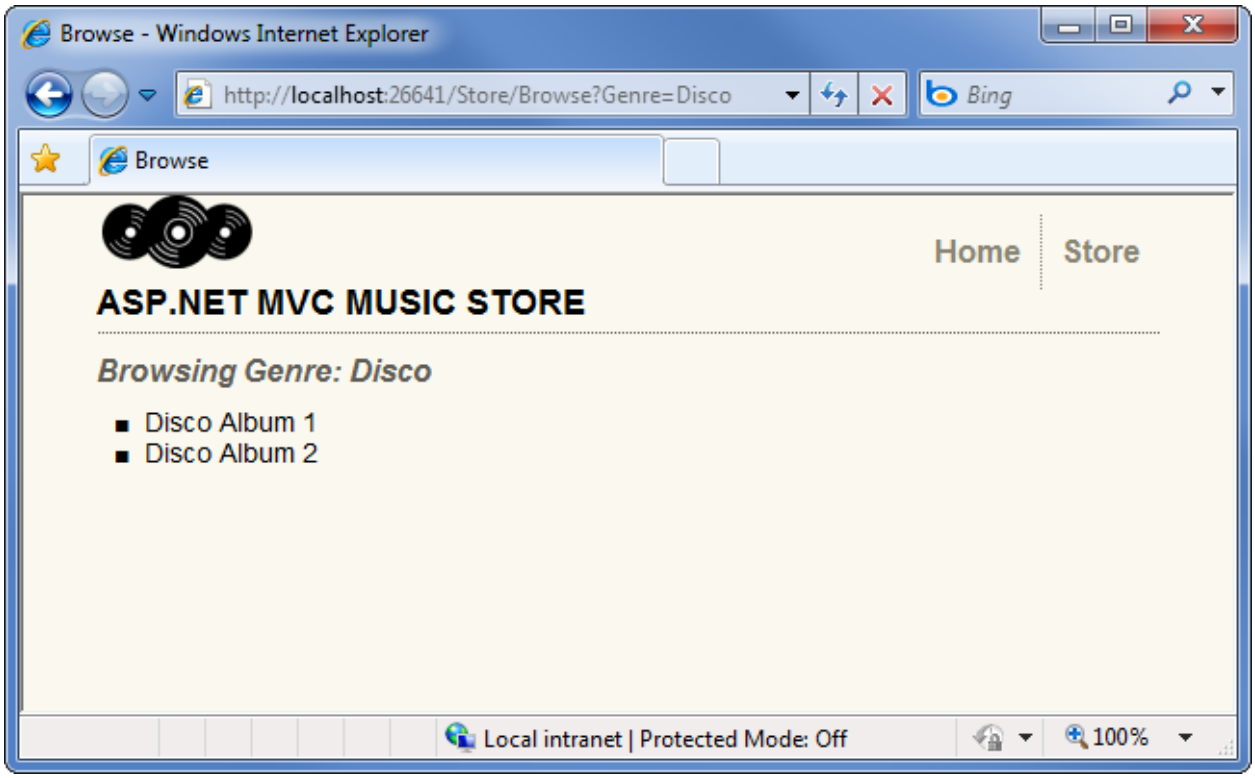


When we click the "Add" button the \Views\Store\Browse.aspx View template will be created.

We can modify the Browse.aspx View template to display the Genre information, accessing the StoreBrowseViewModel object that we passed to the View templateusing the "Model" property like we did before:

```
<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

    <h2>Browsing Genre: <%: Model.Genre.Name %></h2>

    <ul>

    <% foreach (var album in Model.Albums) { %>
        <li><%: album.Title %></li>
    <% } %>

    </ul>
</asp:Content>
```

Let's now re-run our project and visit the /Store/Browse?genre=Disco URL. We'll see details of our genre like below:

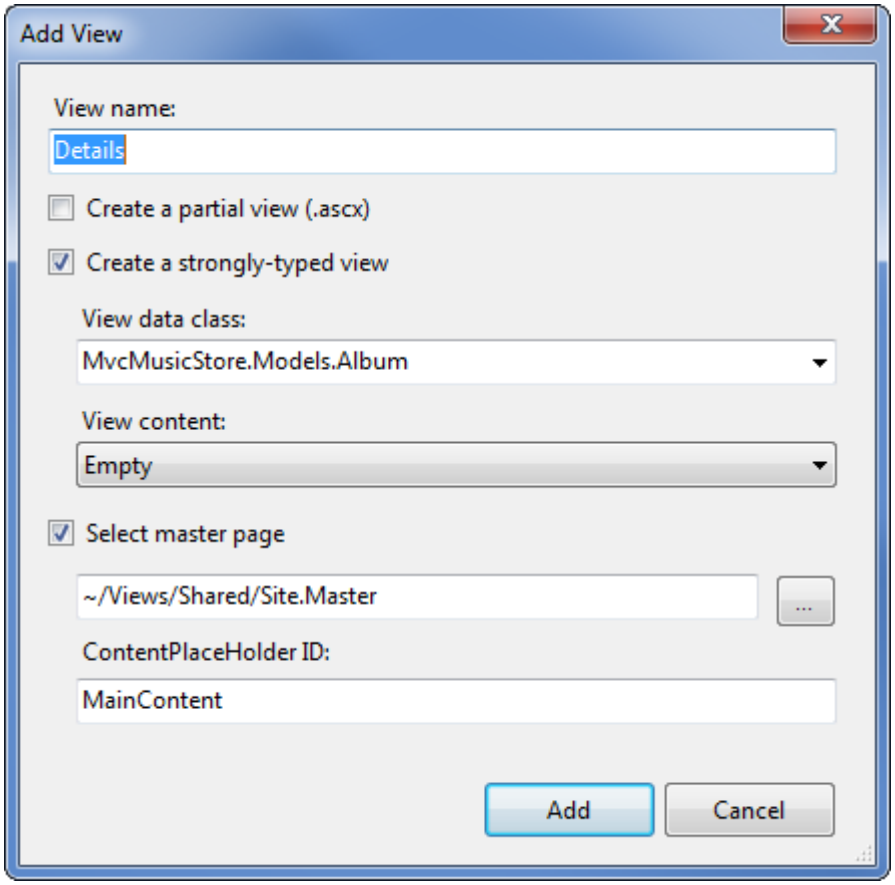


Let's now implement the /Store/Details URLto display information about a specific album. We could create a StoreDetailsViewModel to pass to our View template – but in this case that is unnecessary since the Album class contains everything the View template will need to render a response. So instead of creating a ViewModel we'll just pass the Album class to the View template.

We'll update the Details() action method within our StoreController with the below code to do this:

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

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

    return View(album);
}
```

Now right-click in the Details action method and create a new Details view. We'll indicate that we are passing an "Album" class to the View

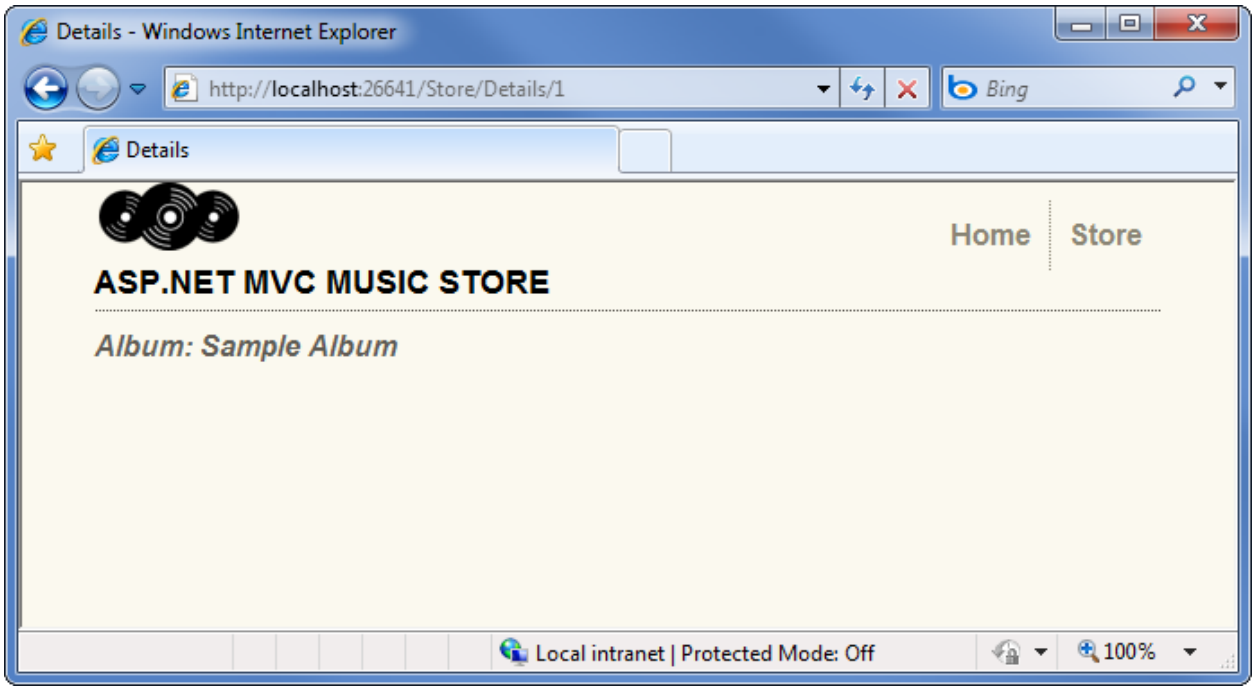


When we click the "Add" button it will create a \Views\Store\Details.aspx file.

We can then update this file to display some Album details:

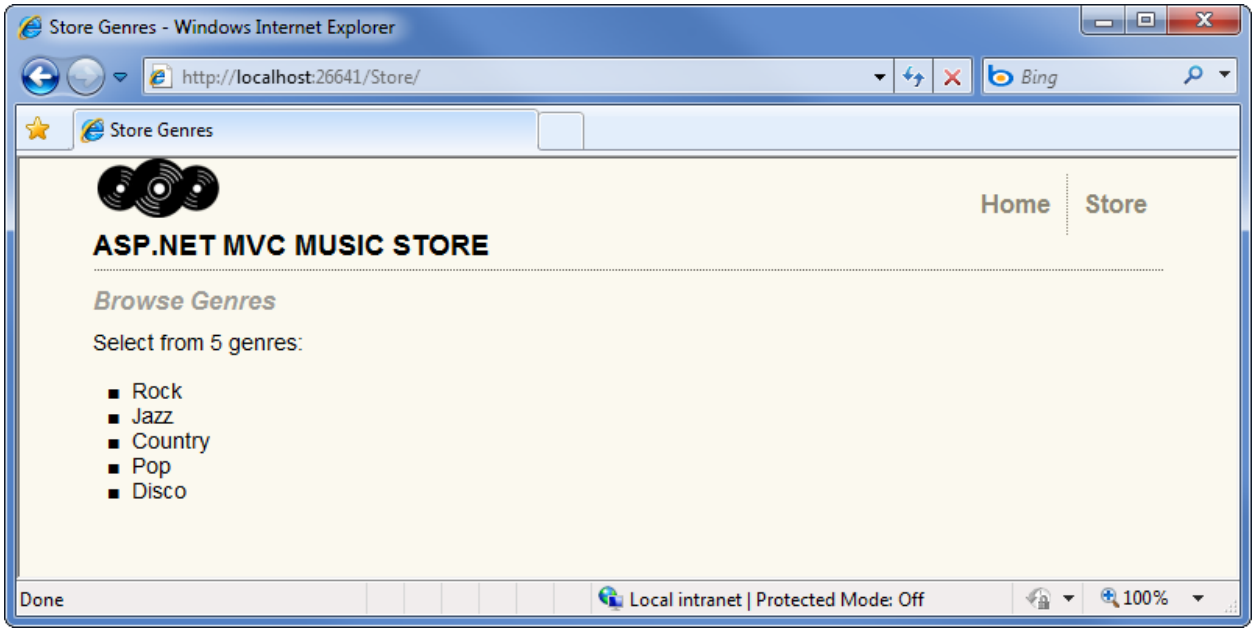
```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage<MvcMusicStore.Models.Album>" %>

<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
      Details
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

    <h2>Album: <%: Model.Title %></h2>

</asp:Content>
```

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



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.aspx View template tooutput these links using code like below:

```
<ul>
    <% foreach (string genreName in Model.Genres) { %>
```

```
        <li>
        <a href="/Store/Browse?genre=<%: genreName %>"><%: genreName %></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 (string genreName in Model.Genres) { %>
        <li>
        <%: Html.ActionLink(genreName, "Browse", new { genre = genreName })%>
        </li>
    <% } %>
</ul>
```

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

The HTML for the genre list looks like this:

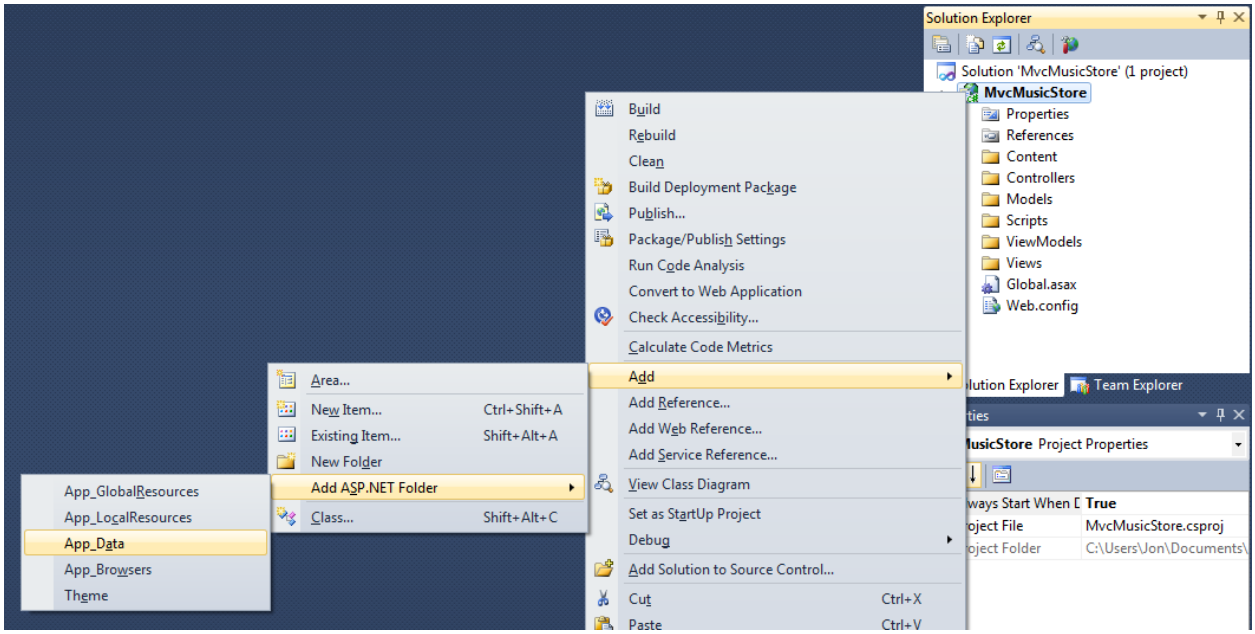
```
<ul>
    <li><a href="/Store/Browse?genre=Rock">Rock</a> </li>
    <li><a href="/Store/Browse?genre=Jazz">Jazz</a> </li>
    <li><a href="/Store/Browse?genre=Country">Country</a> </li>
    <li><a href="/Store/Browse?genre=Pop">Pop</a> </li>
    <li><a href="/Store/Browse?genre=Disco">Disco</a> </li>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

4. Models and Data Access

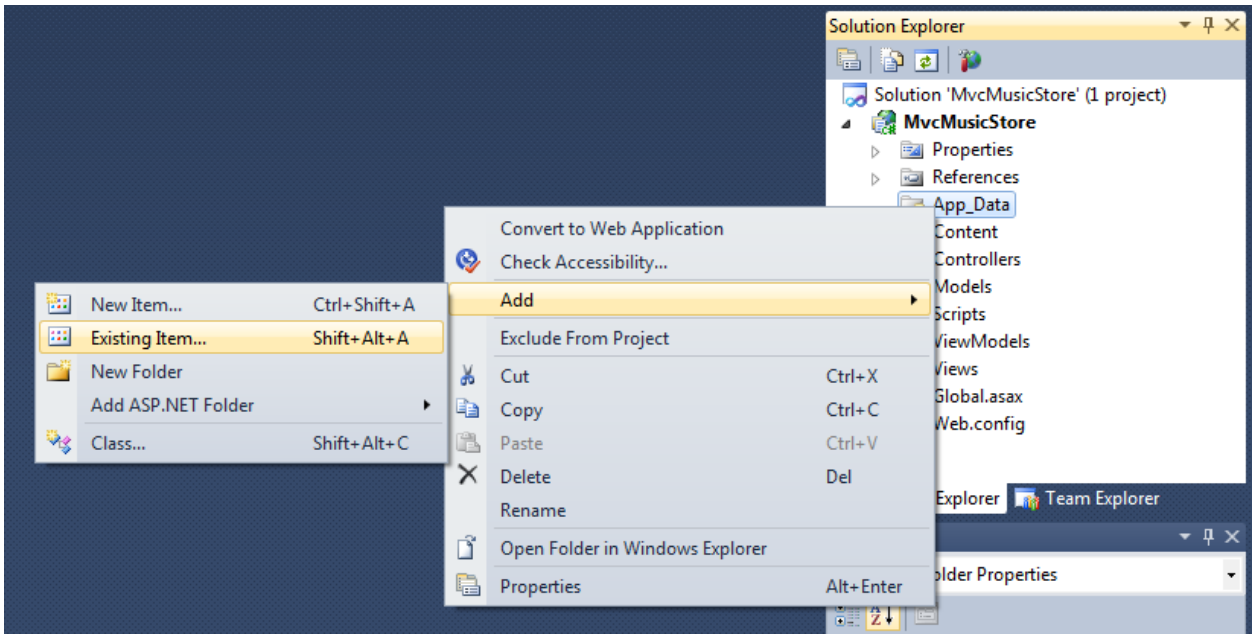
So far, we've just been passing "dummy data" from our Controllers to our View templates. Now we're ready to hook up a real database. In this tutorial we'll be covering how to use the free SQL Server Express as our database engine. The code will also work with the full SQL Server.
We'll start by adding an App_Data directory to our project to hold our SQL Server Express database files. App_Data is a special directory in ASP.NET which already has the correct security access permissions for database access.

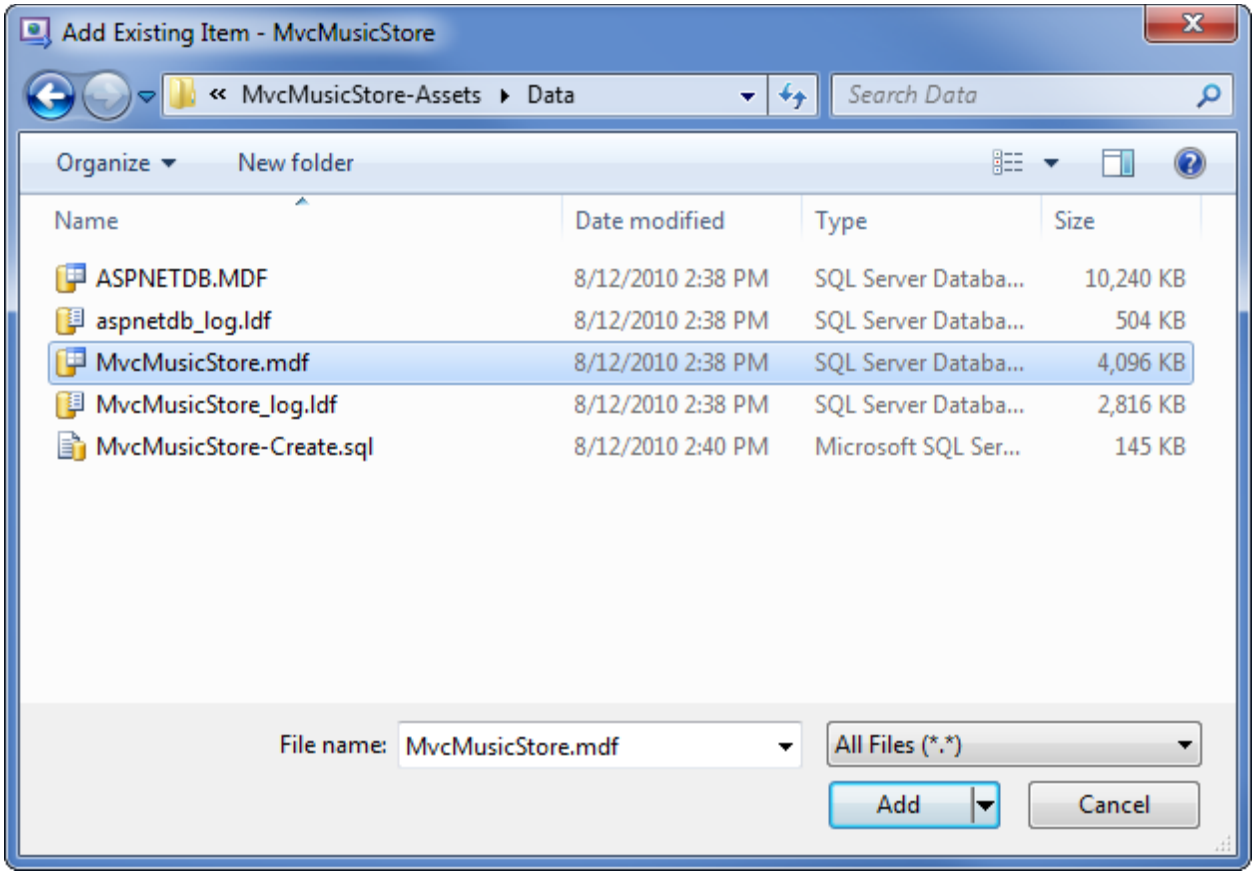
Adding a Database

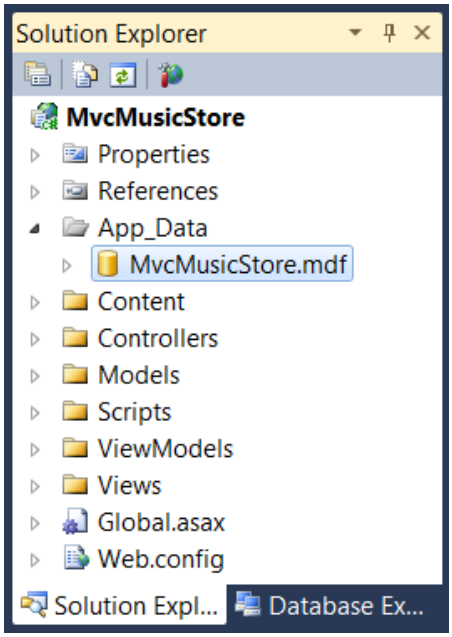
Right-click the project and select Add⇨Add ASP.NET Folder⇨App_Data.



Now we'll add a database file. For this tutorial we'll be using a database that we've already created called MvcMusicStore.mdf – it is included in the project downloads available at http://mvcmusicstore.codeplex.com. To add it to our project, we can right-click the new App_Data folder, select Add⇨Existing Item… and browse to select the MvcMusicStore.mdf file we've downloaded to our local computer.


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

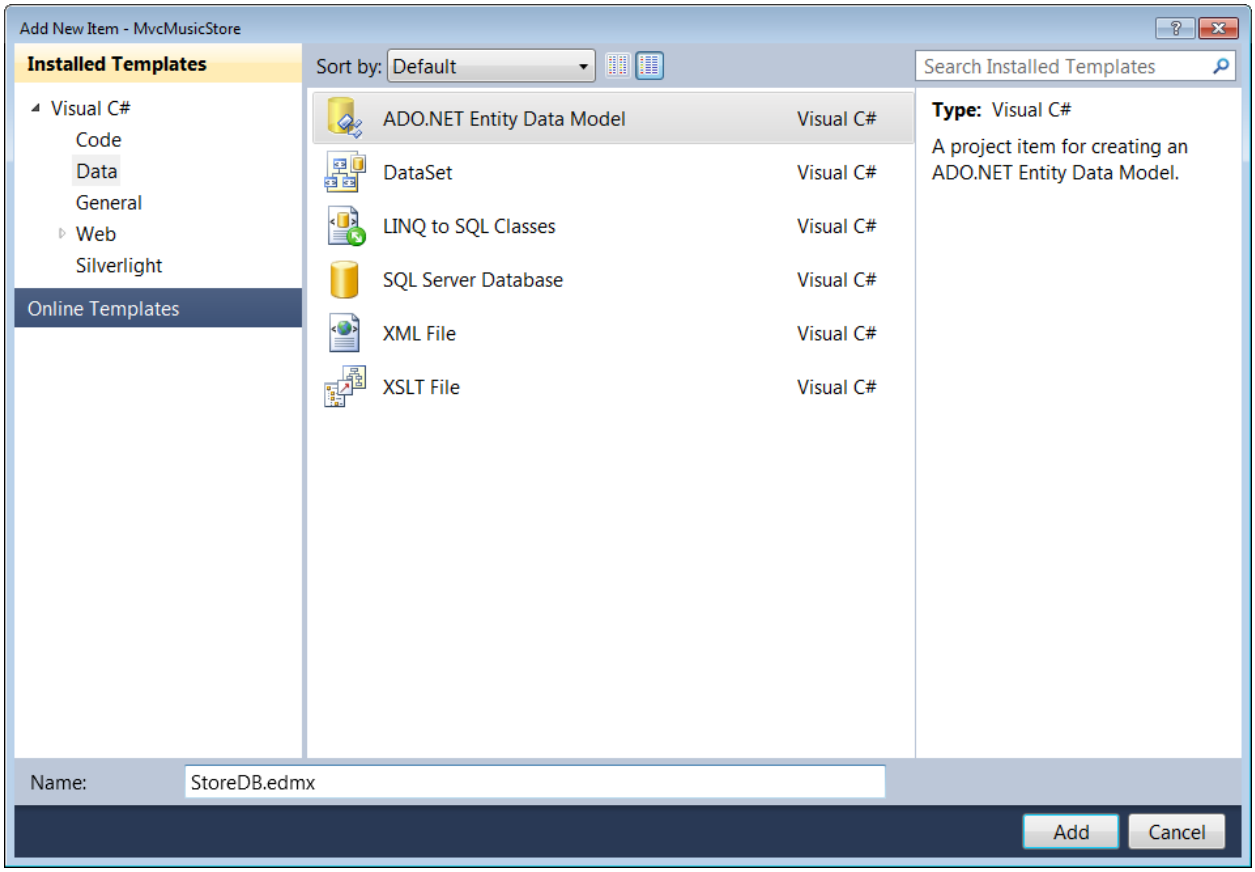

Add Existing Item - MvcMusicStore
« MvcMusicStore-Assets ▸ Data
Search Data
Organize
New folder
Name
Date modified
Type
Size
ASPNETDB.MDF 8/12/2010 2:38 PM SQL Server Databa... 10,240 KB
aspnetdb_log.ldf 8/12/2010 2:38 PM SQL Server Databa... 504 KB
MvcMusicStore.mdf 8/12/2010 2:38 PM SQL Server Databa...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



Creating an Entity Data Model with Entity Framework

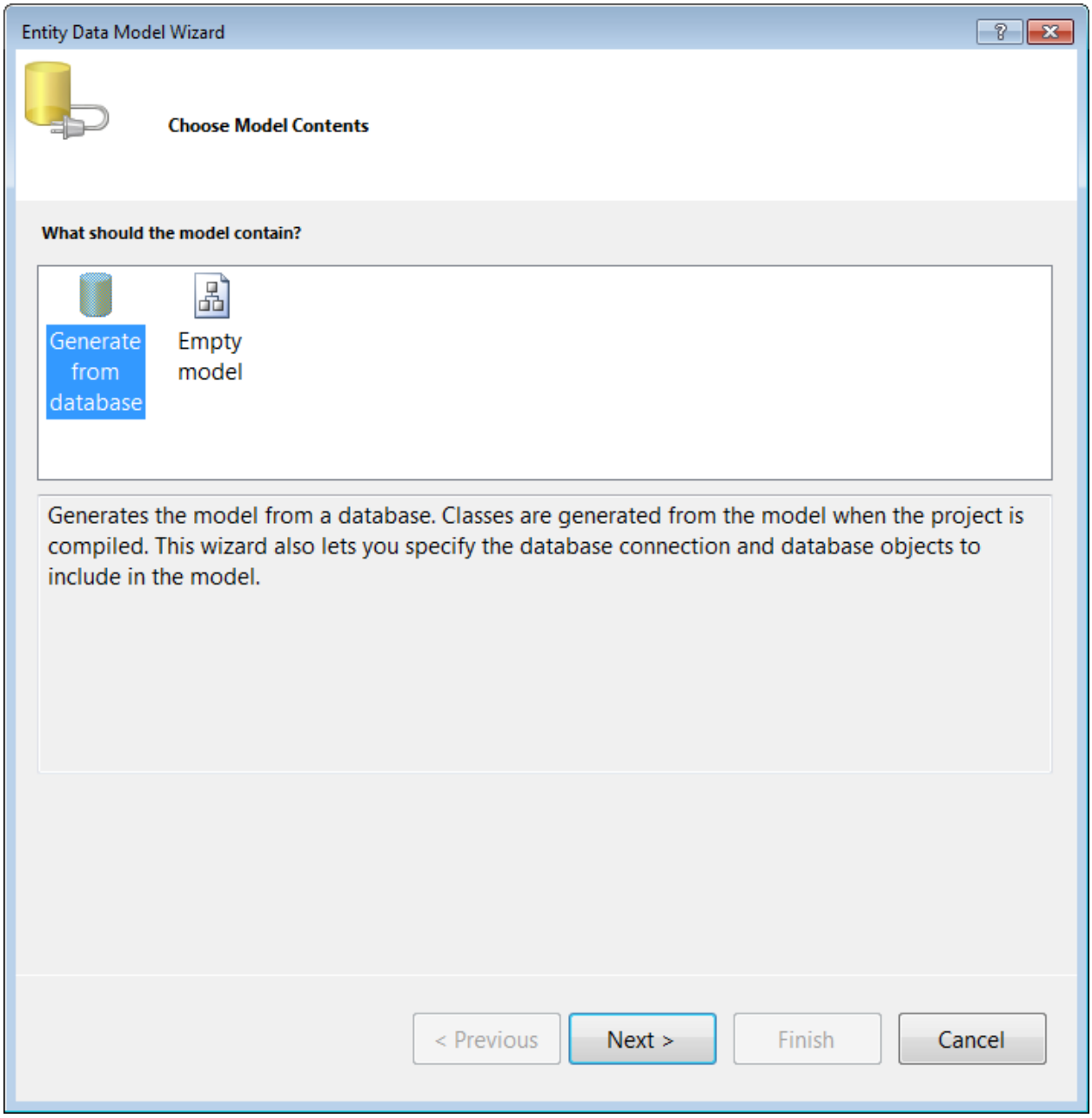
Now that the database has been added to the project, we can write code to query and update the database. We'll use the Entity Framework (EF) that is built into .NET 4 to do this. EF is a flexible object relational mapping (ORM) data API that enables developers to query and update data stored in a database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

We'll start by creating an Entity Framework data model that represents our database. We'll do this by right-clicking the Models folder in our project, and then select the Add⇨New Item menu. This will bring up the "Add New Item" dialog – within it we'll filter by "Data" and select the "ADO.NET Entity Data Model" item:

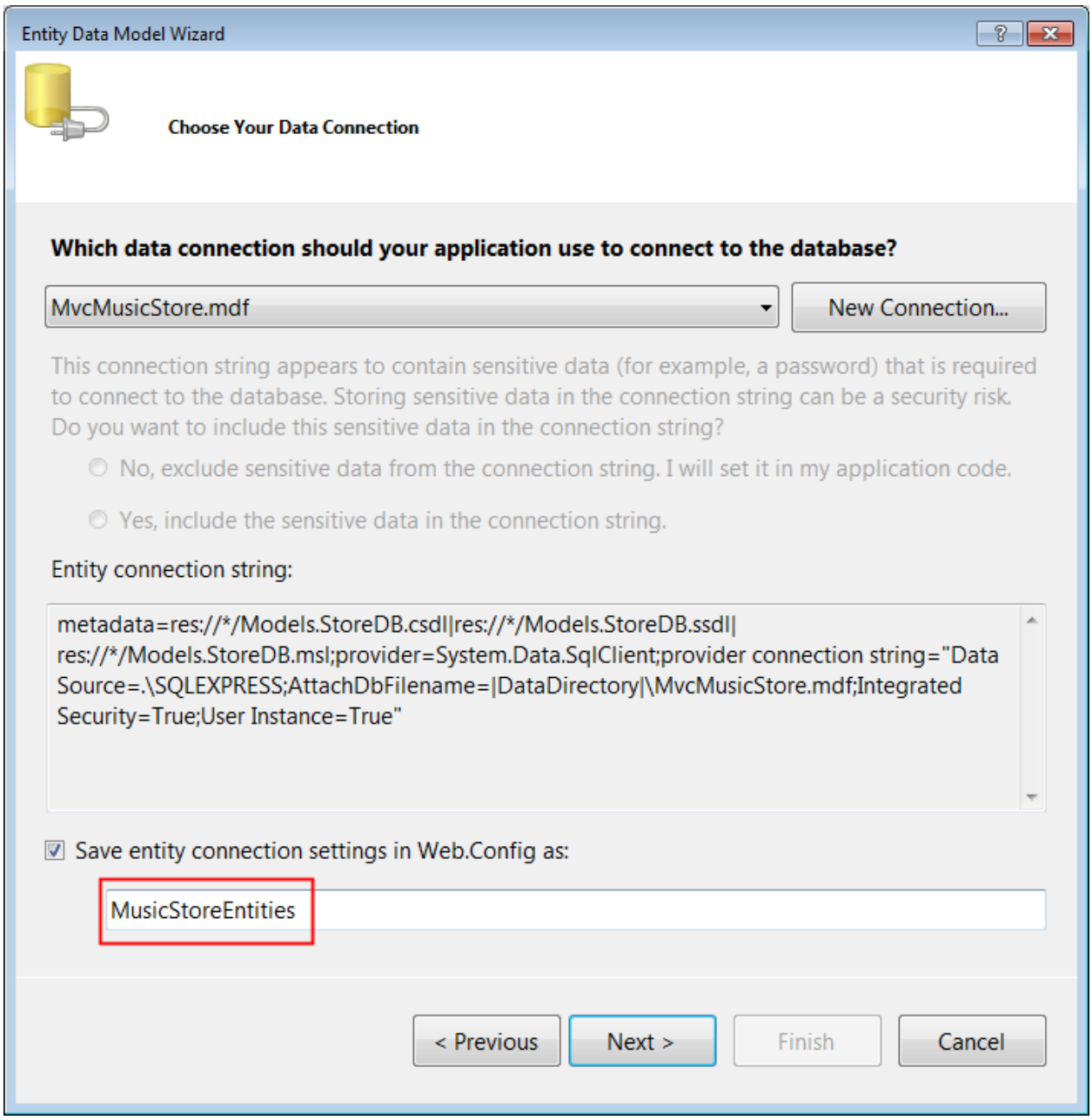


Name the data model StoreDB.edmx and press the Add button. When you do this Visual Studio will bring up a wizard that helps you create your data model layer.

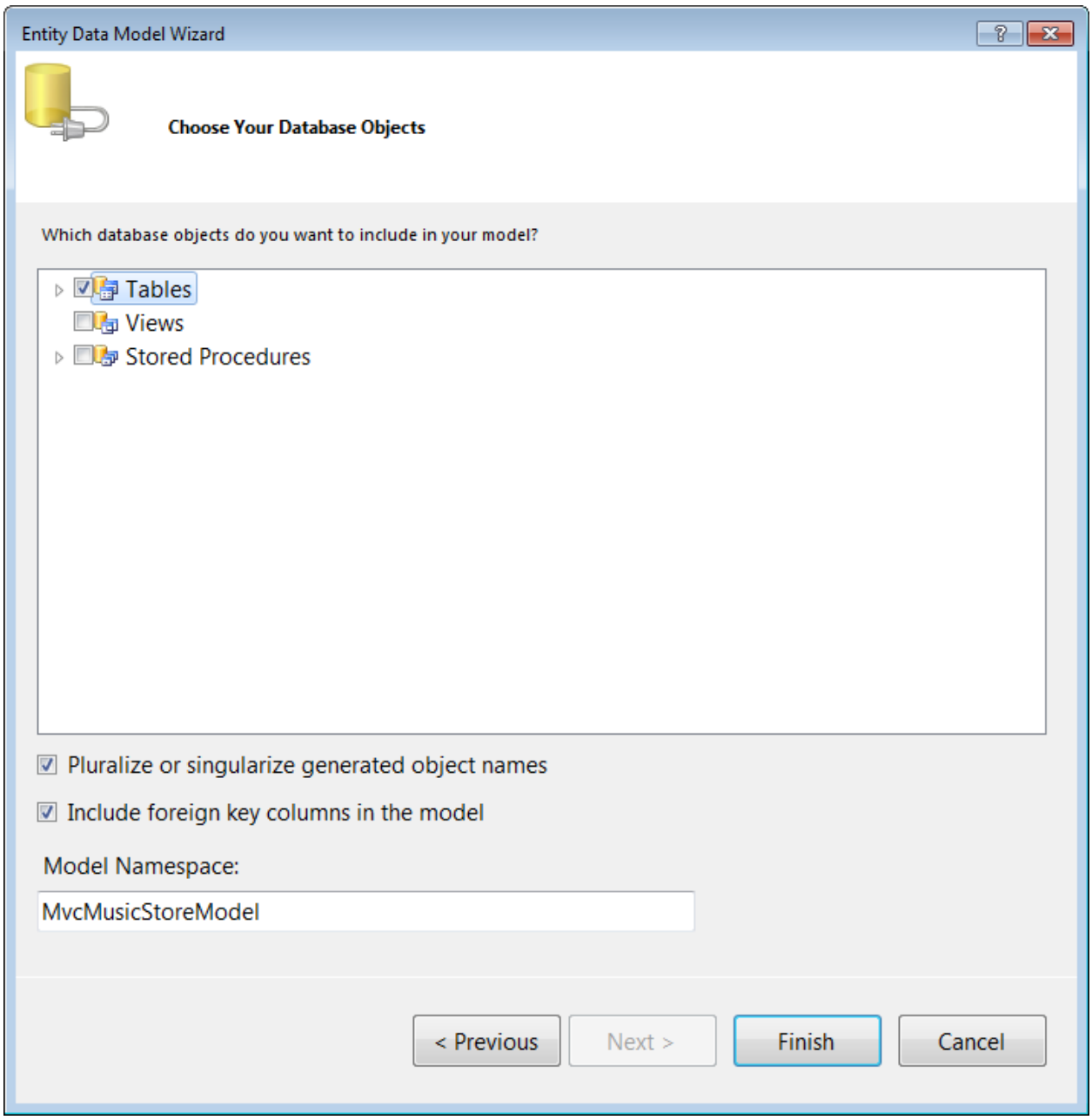
The Entity Data Model Wizard first asks if you want to generate a model from a database or create an empty model. We'll select "Generate from database" and click the Next button.



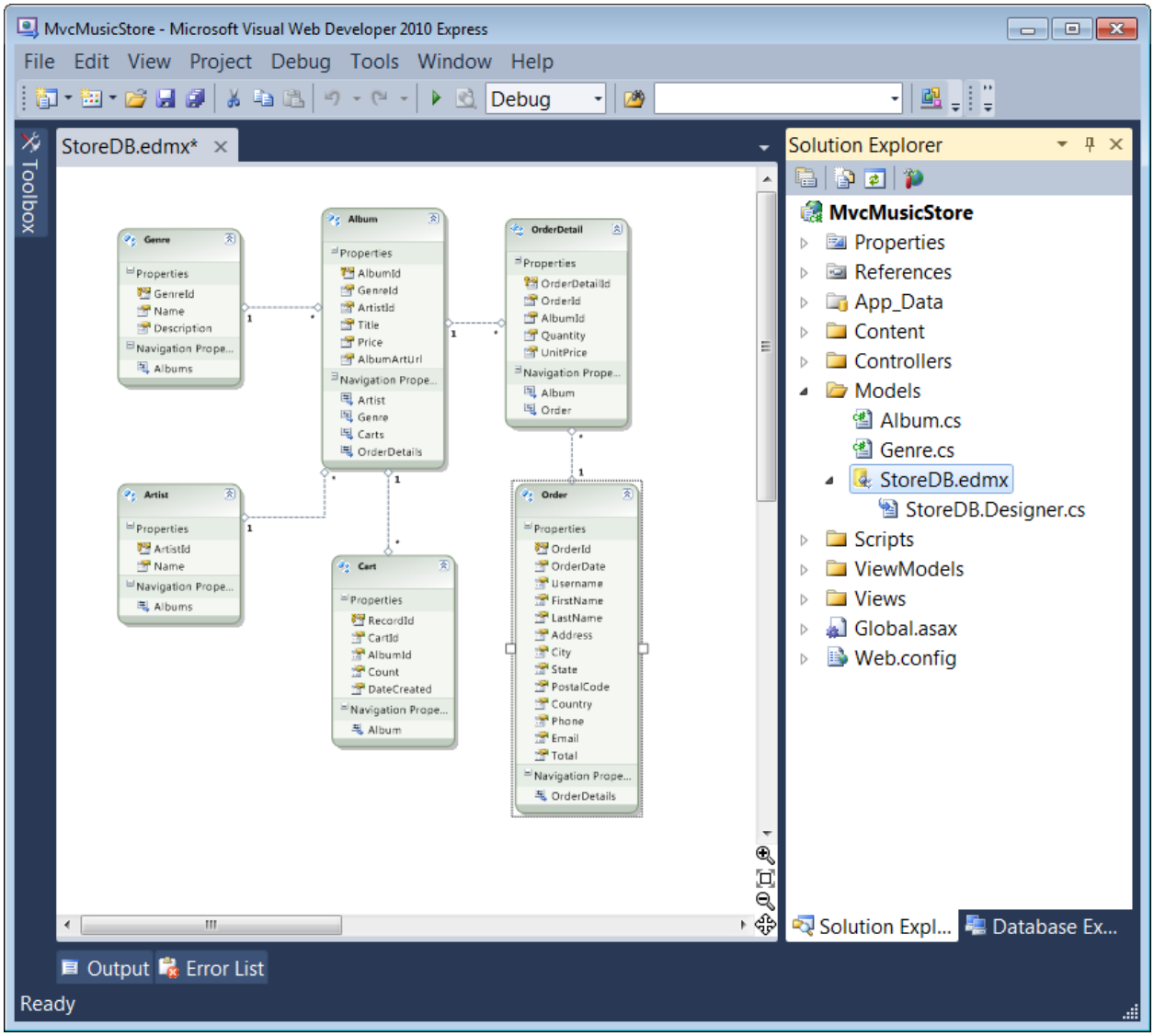
Since we're generating our model from a database, we'll need to specify which database we want to use. The wizard's smart enough to see that we've got a database in our App_Data folder, so it fills in the correct connection information for that database for us. The generated class will have the same name as the entity connection string, so let's change it to MusicStoreEntities, as shown below:

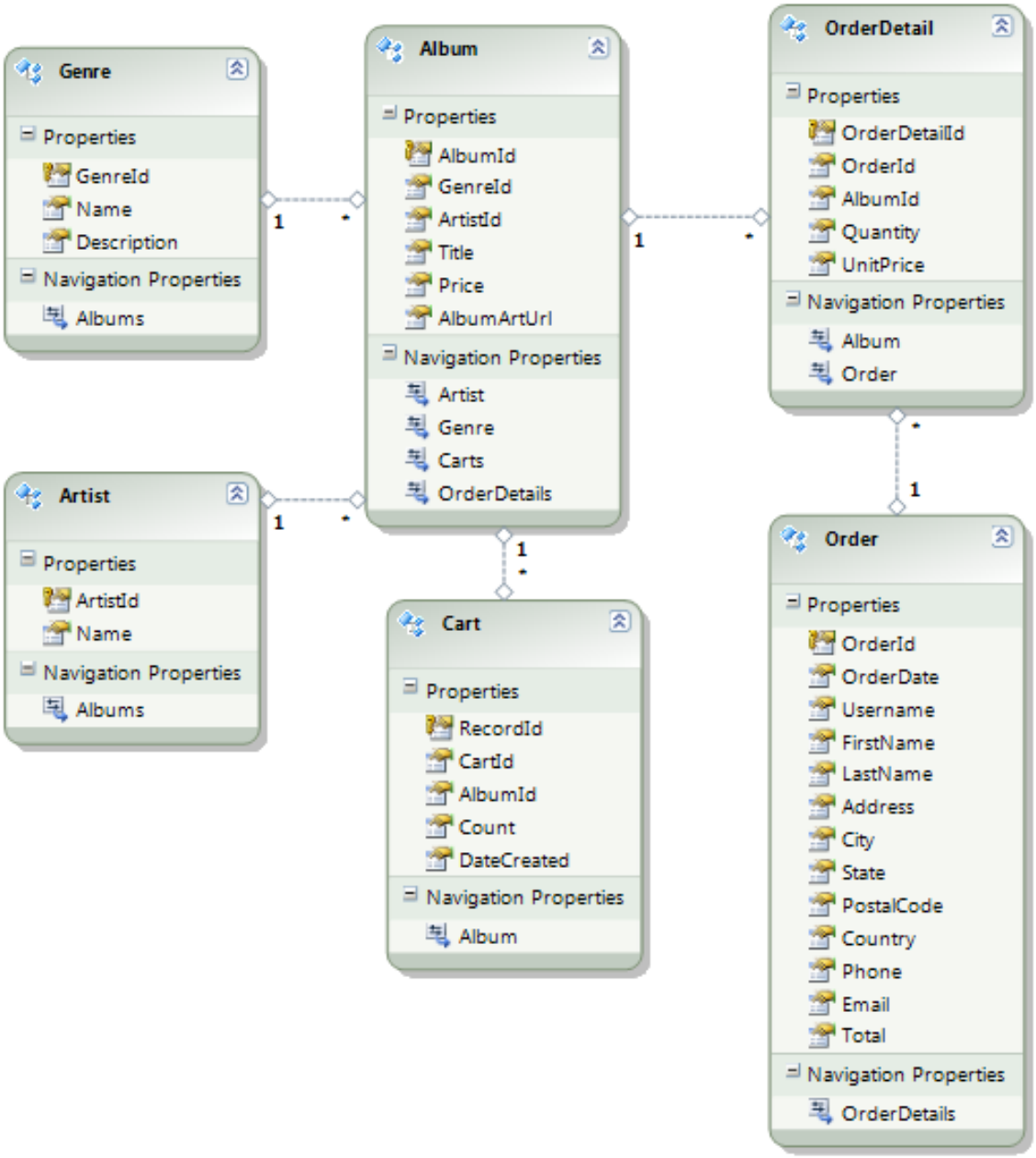


Check the Tables button and ensure that the "Include foreign key columns in the model" checkbox is checked. Change the Model Namespace to MvcMusicStore and press the Finish button.



This brings up an entity diagram for our database. The Entity Framework by default will create a separate class that maps to each table within our database. For example, the "Albums" table would cause an "Album" class to be created – with each column in the table mapping to a property on the class.


MvcMusicStore - Microsoft Visual Web Developer 2010 Express
File Edit View Project Debug Tools Window Help
Debug
Toolbox
StoreDB.edmx*
Genre
Properties
GenreId
Name
Description
Navigation Prope...
Albums
Album
Properties
AlbumId
GenreId
ArtistId
Title
Price
AlbumArtUrl
Navigation Prope...
Artist
Carts
OrderDetails
OrderDetail
Properties
OrderDetailId
OrderId
AlbumId
Quantity
UnitPrice
Navigation Prope...
Album
Order
Artist
Properties
ArtistId
Name
Navigation Prope...
Albums
Cart
Properties
RecordId
CartId
AlbumId
Count
DateCreated
Navigation Prope...
Album
Order
Properties
OrderId
OrderDate
Username
FirstName
LastName
Address
City
State
PostalCode
Country
Phone
Email
Total
Navigation Prope...
OrderDetails
Solution Explorer
MvcMusicStore
Properties
References
App_Data
Content
Controllers
Models
Album.cs
Genre.cs
StoreDB.edmx
StoreDB.Designer.cs
Scripts
ViewModels
Views
Global.asax
Web.config
Solution Expl...
Database Ex...
Output
Error List
Ready



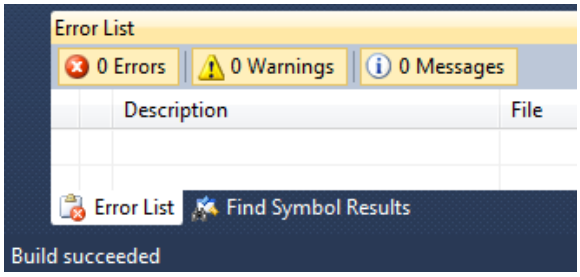
Above you can see that we have Album, Genre, and Artist classes to track our music. Since we're running a store, we also have tables for Cart, Order, and OrderDetails.

The Entity Framework will automatically create and add all of the above classes to our project – allowing us to easily query and work with objects that represent rows within our database.

Now that the Entity Framework is maintaining an Album and Genre class for us based on our database, we can delete the placeholder Album and Genre classes that we earlier manually created ourselves . To-do this,

just expand the Models folder within the Solution Explorer of Visual Studio, select Album.cs, and press the Delete key. Then select the Genre.cs file and delete it as well.

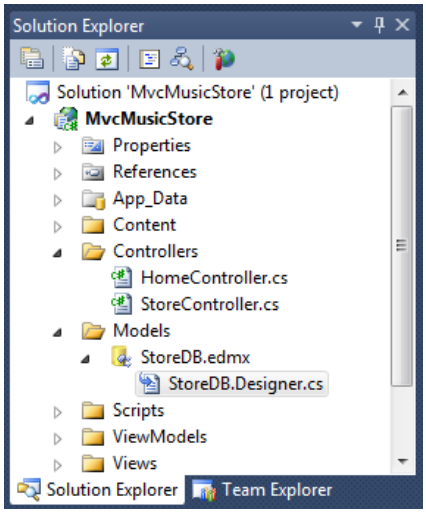
Despite having deleted our previous Albums and Genre class, the project still builds and the pages still work.



Why does this still work?

It works because our database tables have fields which include the properties we were using in the earlier Album and Genre classes that we manually created, and so our Entity Framework data model classes are a drop-in replacement.

While the Entity Framework designer displays the entities in a diagram format, they're really just C# classes. Expand the StoreDB.edmx node in the Solution Explorer, and you'll see a file called StoreDB.Designer.cs.



Querying the Database

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

```
public class StoreController : Controller
{
```

```
    MusicStoreEntities storeDB = new MusicStoreEntities();
```

Store Index using a LINQ Query Expression

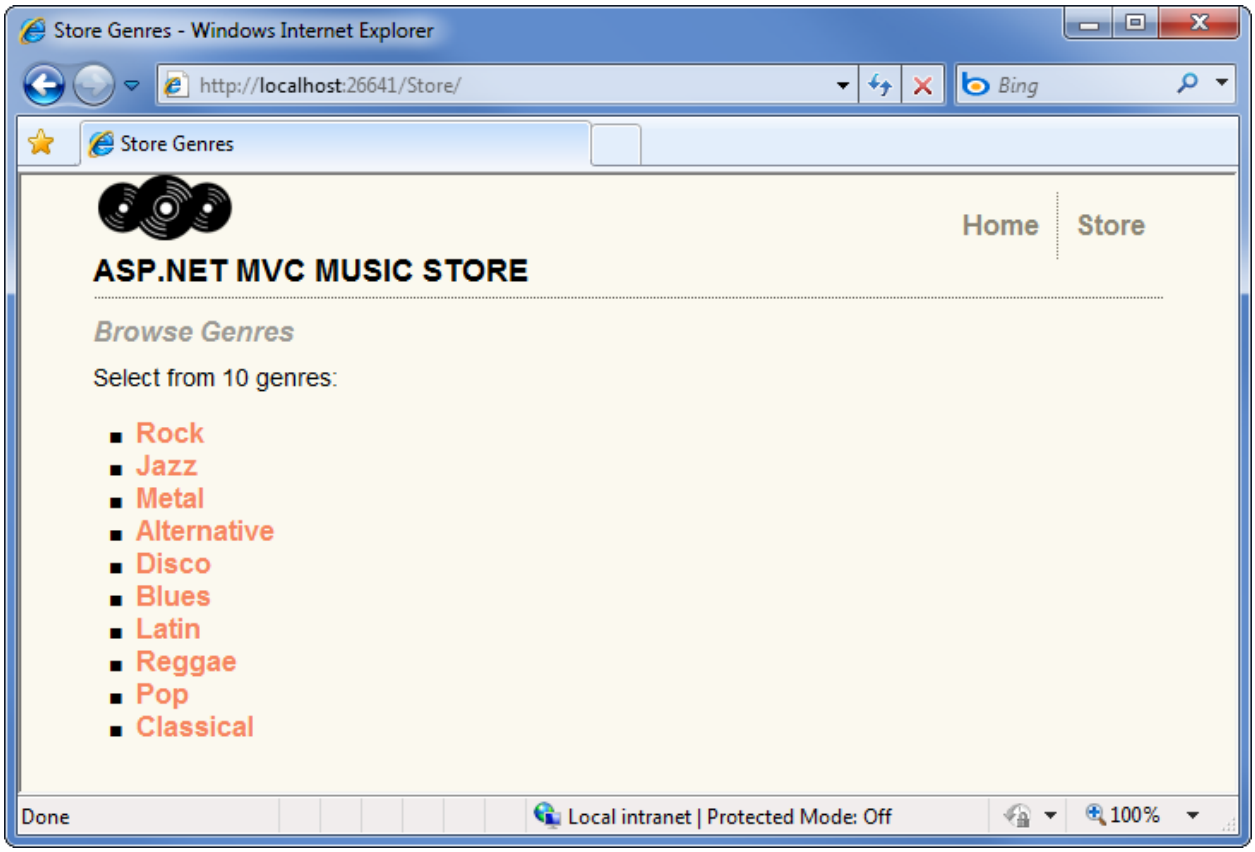
The MusicStoreEntities class is maintained by the Entity Framework and exposes a collection property for each table in our database. We can use a cool capability of .NET called LINQ (language integrated query) to write strongly-typed query expressions against these collections – which will execute code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

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

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

public ActionResult Index()
{
    // Retrieve list of Genres from database
    var genres = from genre in storeDB.Genres
                 select genre.Name;

    // Set up our ViewModel
    var viewModel = new StoreIndexViewModel()
    {
        Genres = genres.ToList(),
        NumberOfGenres = genres.Count()
    };

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

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

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



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

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

```
var genre = 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.

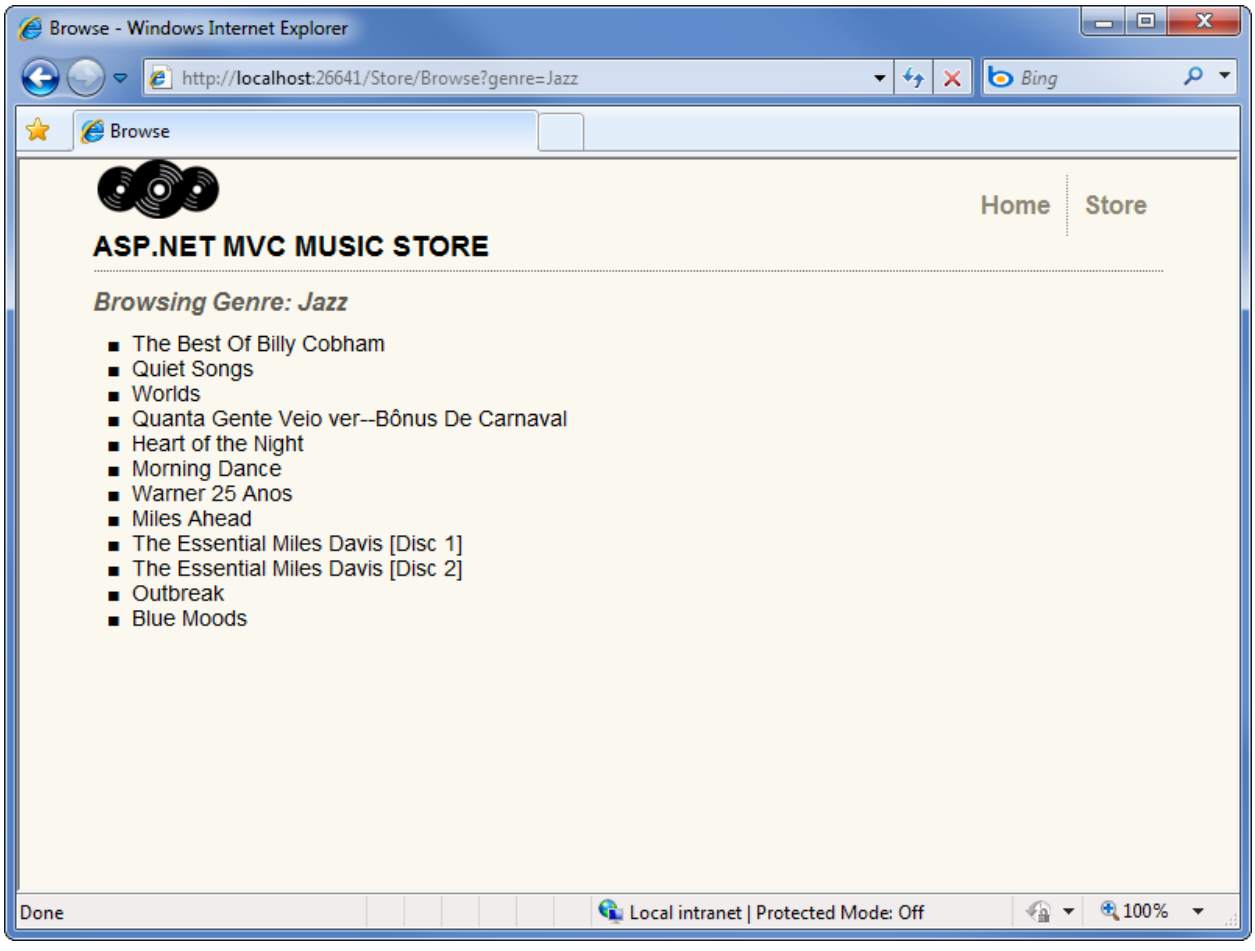
We'll take advantage of an Entity Framework feature that allows us to indicate other related entities we want loaded as well when the Genre object is retrieved. This feature is called Query Result Shaping, and enables us to reduce the number of times we need to access the database to retrieve all of 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:

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

public ActionResult Browse(string genre)
```

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

    var viewModel = new StoreBrowseViewModel()
    {
        Genre = genreModel,
        Albums = genreModel.Albums.ToList()
    };

    return View(viewModel);
}
```

Running our application and browsing to /Store/Browse?genre=Jazz shows that our results are now being pulled from the database.

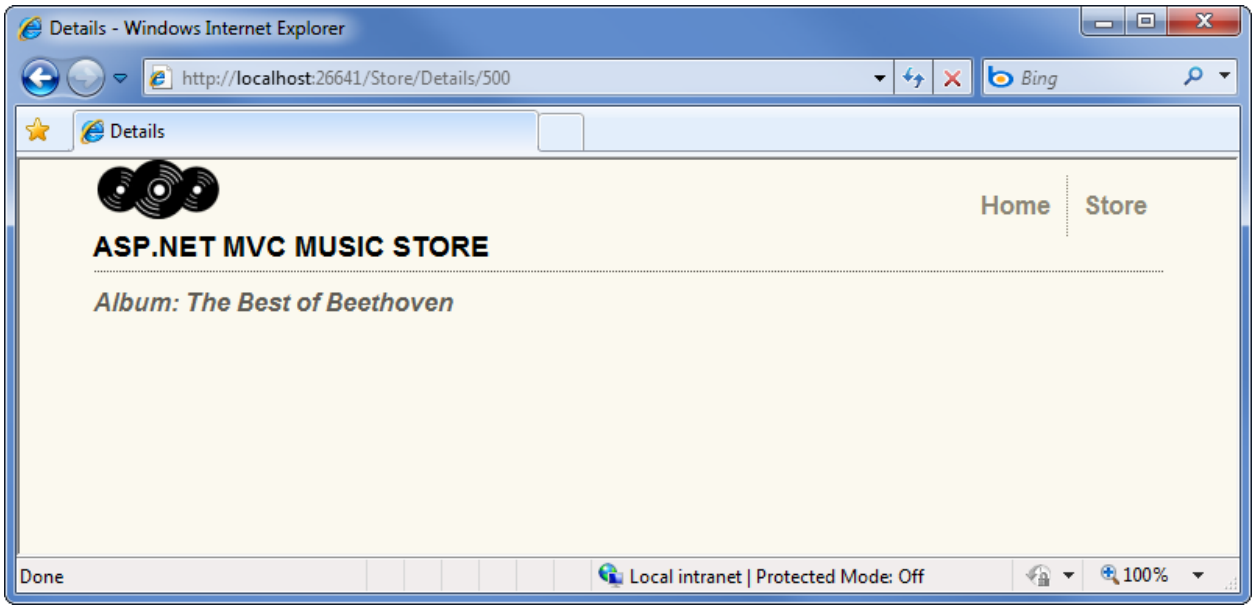


We'll make the same change to our /Store/Details/[id] URL, and replace our dummy data with a database query which loads an Album whose ID matches the parameter value.

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

```
public ActionResult Details(int id)
{
    var album = storeDB.Albums.Single(a => a.AlbumId == id);

    return View(album);
}
```

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



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

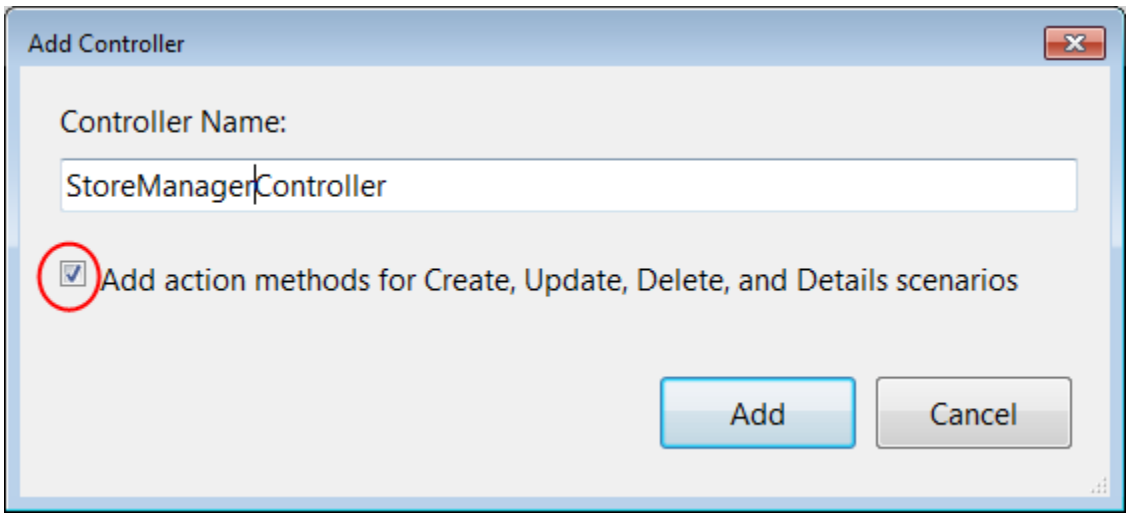
```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
    Inherits="System.Web.Mvc.ViewPage<MvcMusicStore.ViewModels.StoreBrowseViewModel>" %>

<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
      Browse
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

    <h2>Browsing Genre: <%: Model.Genre.Name %></h2>

    <ul>

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

```
    </ul>
</asp:Content>
```

5. Edit Forms and Templating

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

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



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

```
public class StoreManagerController : Controller
{
    //
    // GET: /StoreManager/

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

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

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

    //
    // GET: /StoreManager/Create

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

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

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

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

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

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

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

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

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

We don't need the Details controller actions, so we can delete both Delete methods. Then we'll get to work on building out the other controller actions and views.

Customizing the Store Manager Index

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

```
using MvcMusicStore.Models;
Now add a the storeDB field, as shown below.
public class StoreManagerController : Controller
{
    MusicStoreEntities storeDB = new MusicStoreEntities();
```

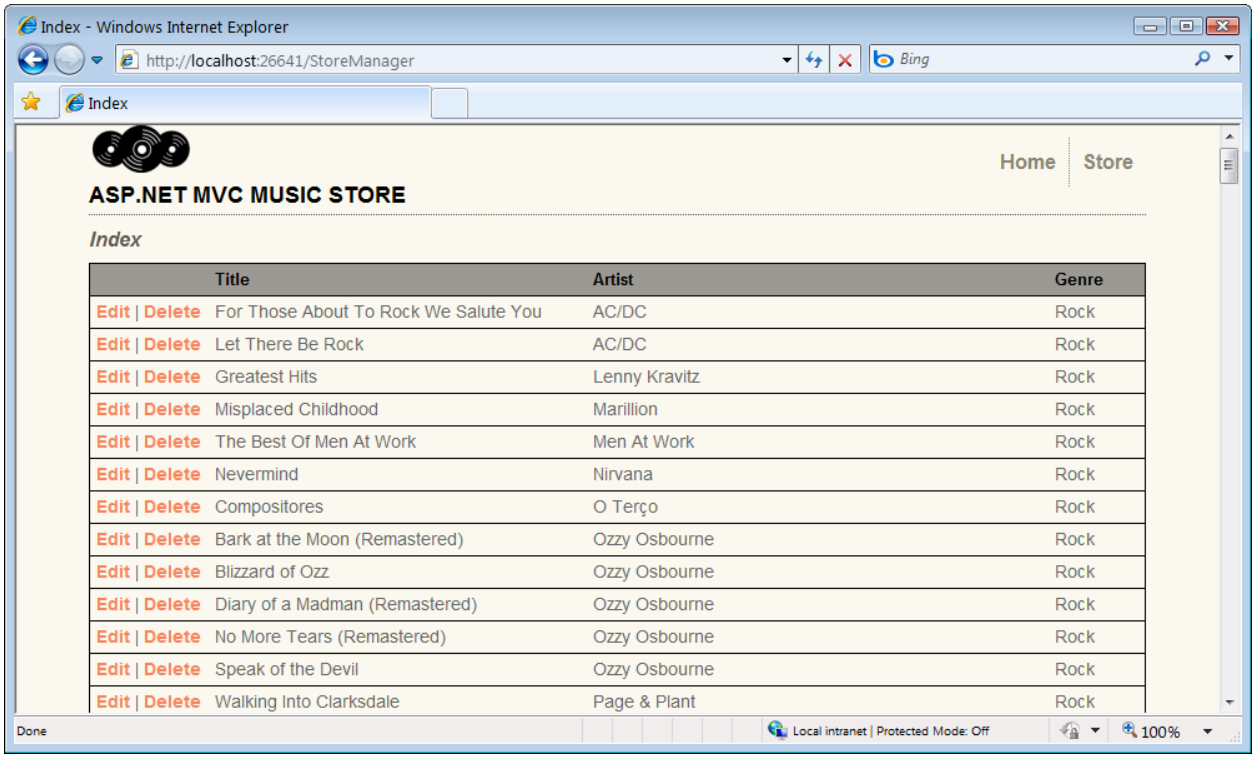
Now we will implement the Store Manager Index action. This action will display a list of albums, so the controller action logic will be pretty similar to the Store Controller's Index action we wrote earlier. We'll use LINQ to retrieve all albums, including Genre and Artist information for display.

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

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

    return View(albums);
}
```

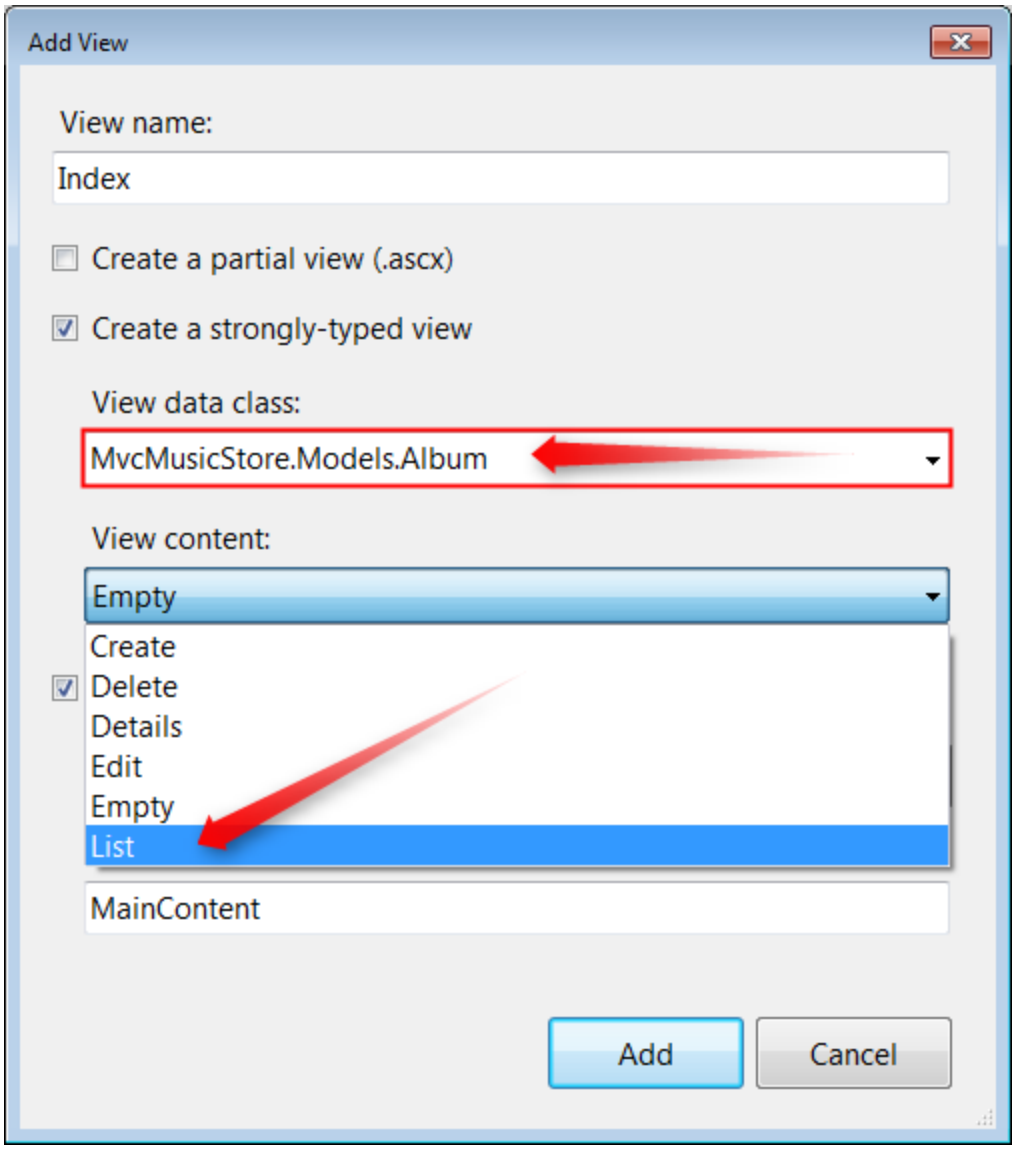
Scaffold View templates

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

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



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



Let's look at the generated list of fields:

```
<table>
    <tr>
        <th></th>
        <th>AlbumId</th>
        <th>GenreId</th>
        <th>ArtistId</th>
        <th>Title</th>
        <th>Price</th>
        <th>AlbumArtUrl</th>
    </tr>
<% foreach (var item in Model) { %>
    <tr>
        <td>
```

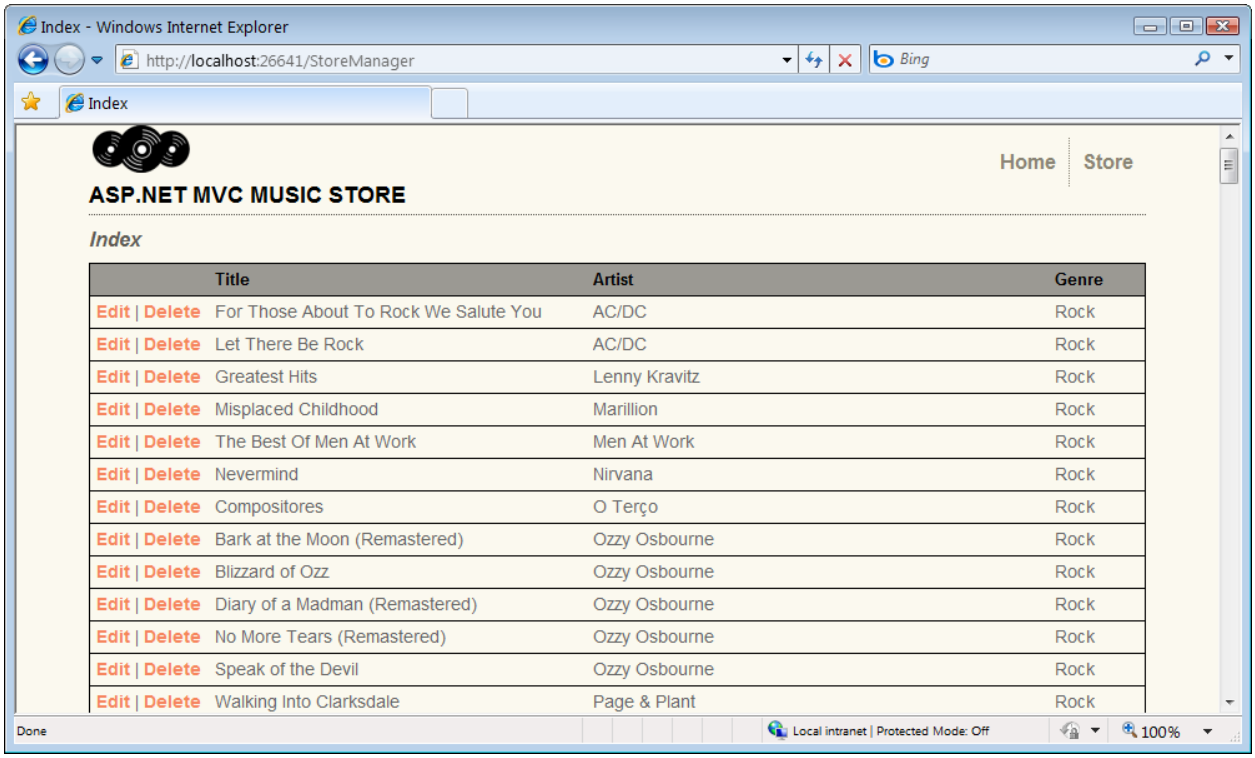
```
            <%: Html.ActionLink("Edit", "Edit", new { id=item.AlbumId }) %> |
            <%: Html.ActionLink("Details", "Details", new { id=item.AlbumId })%> |
            <%: Html.ActionLink("Delete", "Delete", new { id=item.AlbumId })%>
        </td>
        <td><%: item.AlbumId %></td>
     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

Note: That this template is following the same coding practices we've been learning so far – using <%: to HTML encode our values, and using Html.ActionLink to generate links to other controller actions.

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

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

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

<% } %>

</table>
```

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



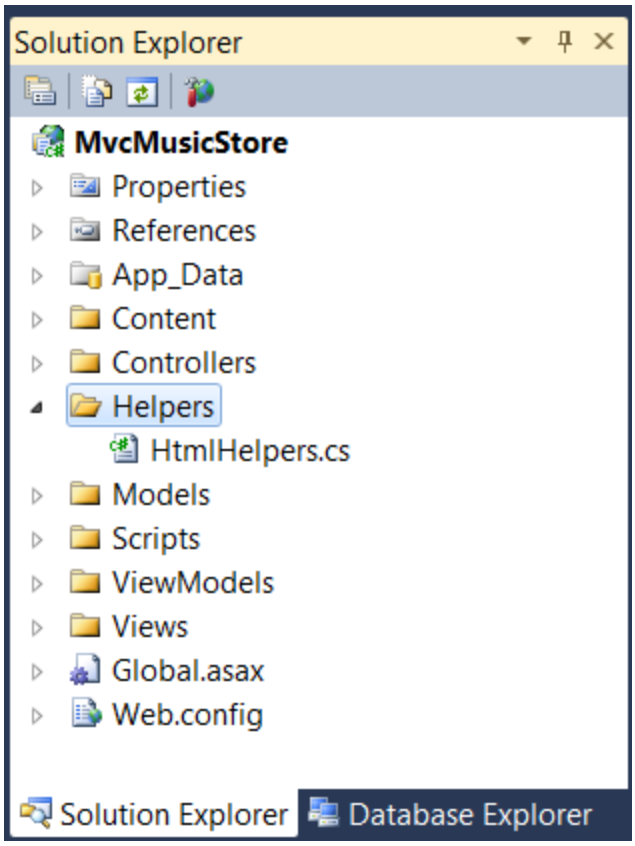
Using a custom HTML Helper to truncate text

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

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

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

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



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

```
using System.Web.Mvc;

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

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

To use our custom HTML helper we need to register it with our application. To do this, we'll go into our Web.config file and indicate that we want to import the namespace it lives in. Double-click on the Web.config file in the Solution Explorer and add the highlighted line to the <pages> section, as shown below.

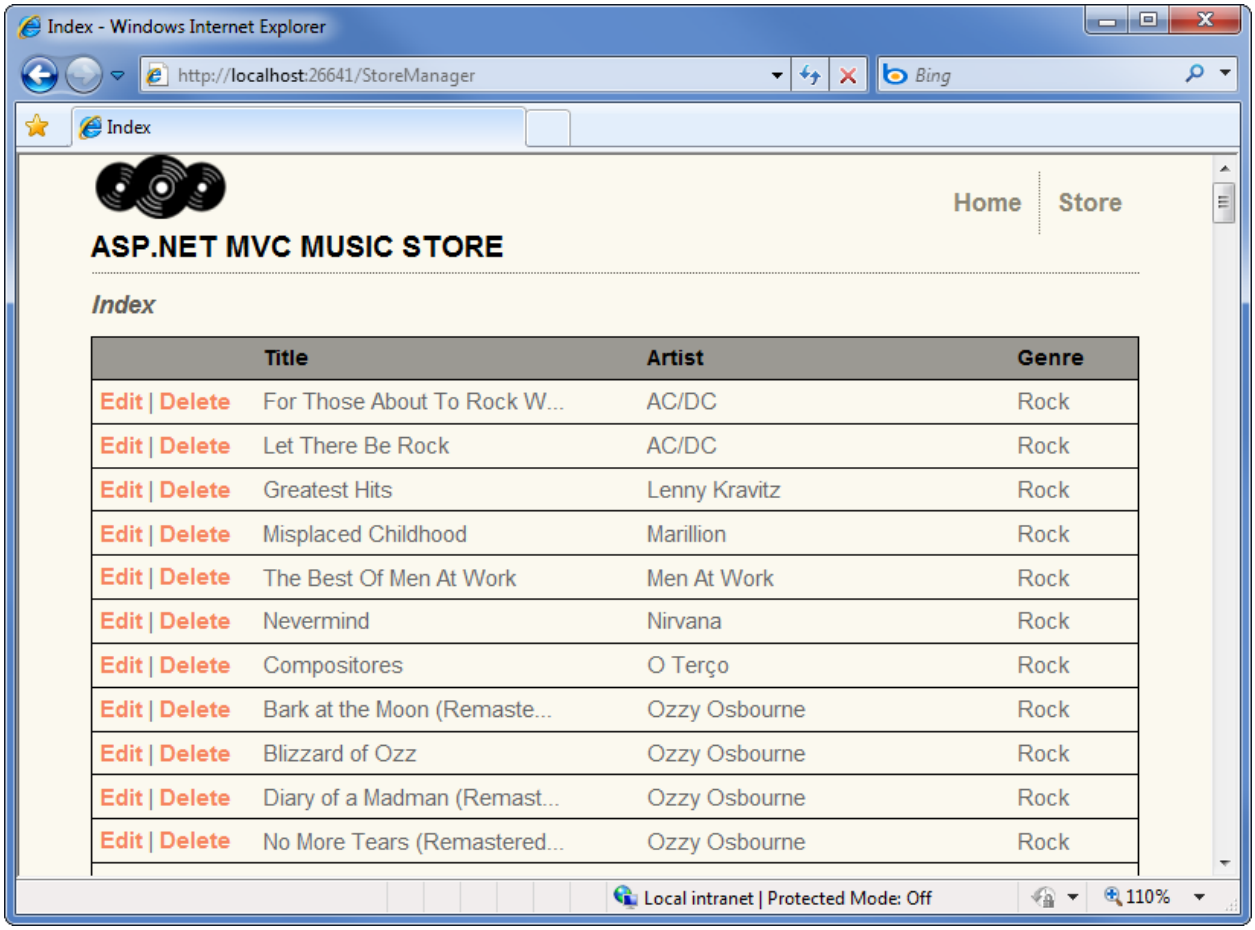
```
<pages>
    <namespaces>
    <add namespace="System.Web.Mvc" />
    <add namespace="System.Web.Mvc.Ajax" />
    <add namespace="System.Web.Mvc.Html" />
    <add namespace="System.Web.Routing" />
    <add namespace="MvcMusicStore.Helpers"/>
    </namespaces>
</pages>
```

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

```
<%@ Page Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage<IEnumerable<MvcMusicStore.Models.Album>>" %>

<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
       Store Manager - All Albums
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

    <h2>Albums</h2>

    <p>
        <%: Html.ActionLink("Create New Album", "Create") %>
    </p>

    <table>
        <tr>
            <th></th>
            <th>Title</th>
            <th>Artist</th>
            <th>Genre</th>
        </tr>

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

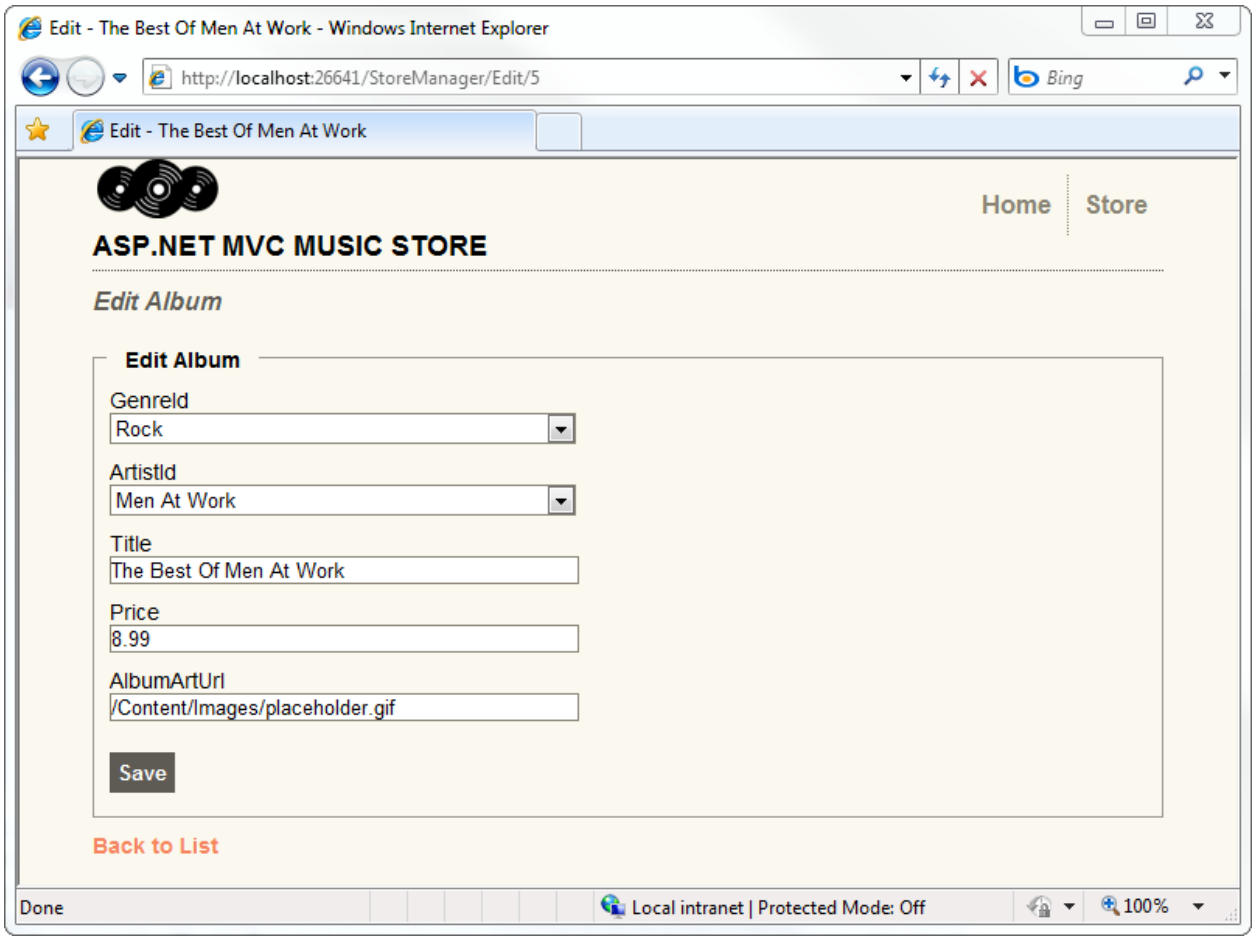
```
            <td><%: Html.Truncate(item.Artist.Name, 25) %></td>
            <td><%: item.Genre.Name %></td>
        </tr>

    <% } %>

    </table>

</asp:Content>
```

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



Creating the Edit View

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



Implementing the Edit Action Methods

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

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

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

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

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

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

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

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

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

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

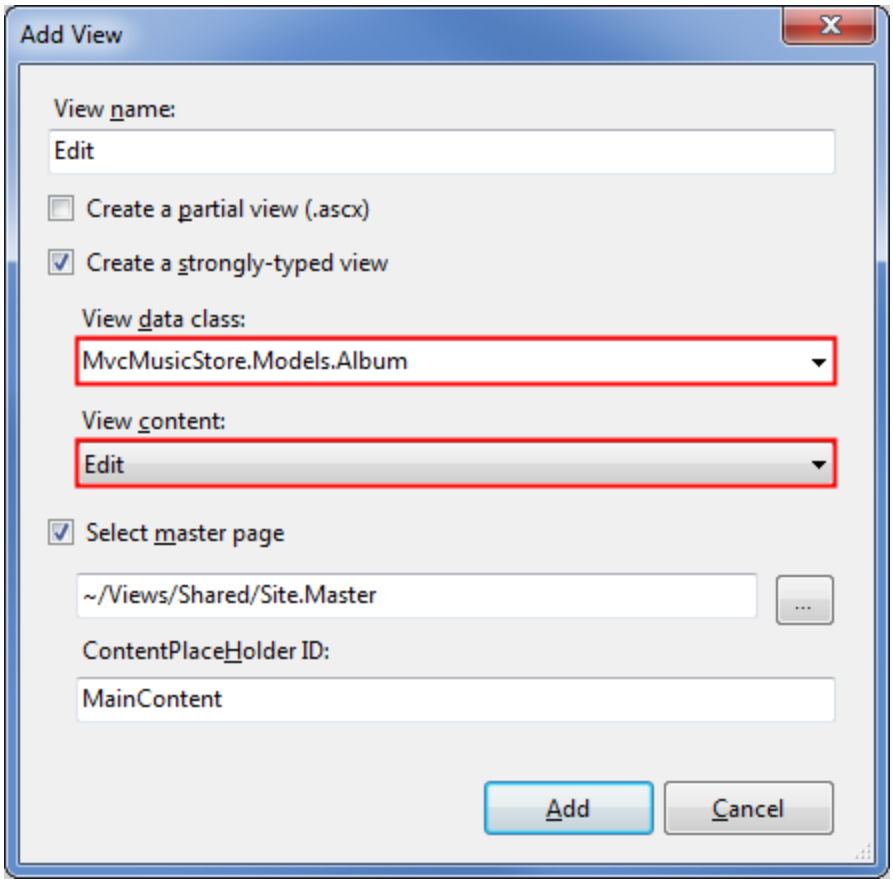
Writing the HTTP-GET Edit Controller Action

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

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

public ActionResult Edit(int id)
{
    Album album = storeDB.Albums.Single(a => a.AlbumId == id);

    return View(album);
}
```

Creating the Edit View

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



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

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage<MvcMusicStore.Models.Album>" %>

<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
       Edit
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

    <h2>Edit</h2>

    <% using (Html.BeginForm()) {%>
        <%: Html.ValidationSummary(true) %>

        <fieldset>
            <legend>Fields</legend>

            <div class="editor-label">
                <%: Html.LabelFor(model => model.AlbumId) %>
            </div>
            <div class="editor-field">
                <%: Html.TextBoxFor(model => model.AlbumId) %>
                <%: Html.ValidationMessageFor(model => model.AlbumId) %>
            </div>
```

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

            <div class="editor-label">
                <%: Html.LabelFor(model => model.ArtistId) %>
            </div>
            <div class="editor-field">
                <%: Html.TextBoxFor(model => model.ArtistId) %>
                <%: Html.ValidationMessageFor(model => model.ArtistId) %>
            </div>

            <div class="editor-label">
                <%: Html.LabelFor(model => model.Title) %>
            </div>
            <div class="editor-field">
                <%: Html.TextBoxFor(model => model.Title) %>
                <%: Html.ValidationMessageFor(model => model.Title) %>
            </div>

            <div class="editor-label">
                <%: Html.LabelFor(model => model.Price) %>
            </div>
            <div class="editor-field">
                <%: Html.TextBoxFor(model => model.Price, String.Format("{0:F}",
                    Model.Price)) %>
                <%: Html.ValidationMessageFor(model => model.Price) %>
            </div>

            <div class="editor-label">
                <%: Html.LabelFor(model => model.AlbumArtUrl) %>
            </div>
            <div class="editor-field">
                <%: Html.TextBoxFor(model => model.AlbumArtUrl) %>
                <%: Html.ValidationMessageFor(model => model.AlbumArtUrl) %>
            </div>

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

    <% } %>

    <div>
        <%: Html.ActionLink("Back to List", "Index") %>
    </div>

</asp:Content>
```

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

Using an Editor Template

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

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

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage<MvcMusicStore.Models.Album>" %>

<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
      Edit - <%: Model.Title %>
```

```
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

    <h2>Edit Album</h2>

    <% using (Html.BeginForm()) {%>
        <%: Html.ValidationSummary(true) %>

    <fieldset>
        <legend>Edit Album</legend>
        <%: Html.EditorForModel() %>

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

    <% } %>

    <div>
        <%: Html.ActionLink("Back to List", "Index") %>
    </div>

</asp:Content>
```

Above we are indicating that we want to create an Editing view for our model. Run the application and visit the /StoreManager/Edit/5 URL again. The exact same form is displayed, shown below.

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

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

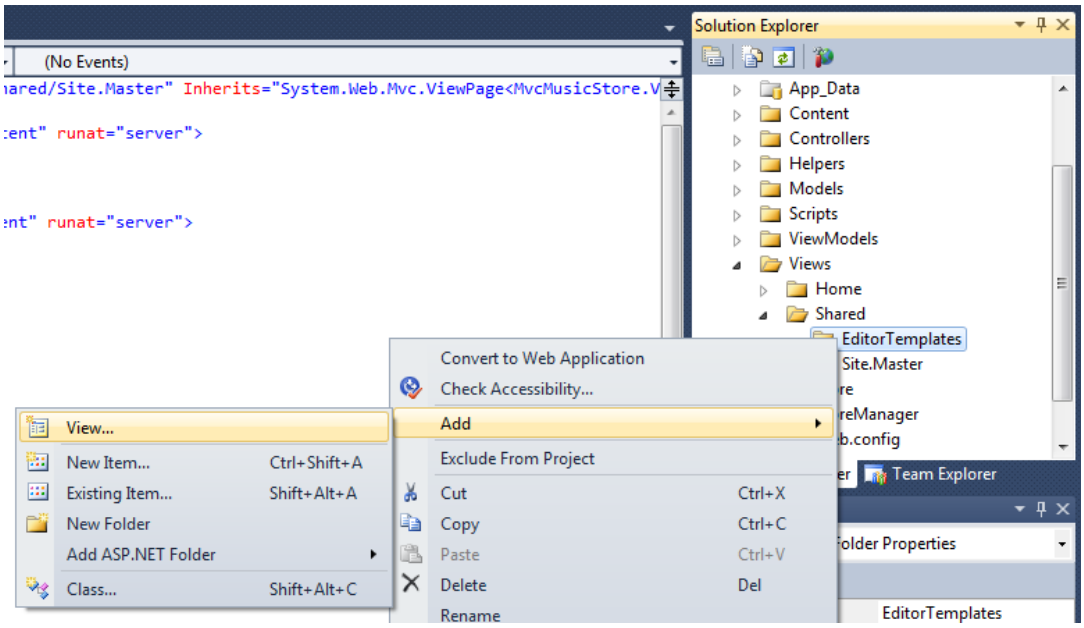
Both of these options are useful in different scenarios. In this case, we'll continue to use the Html.EditorForModel() templating feature to allow for reuse other Views within the site.

Creating a Shared Album Editor Template

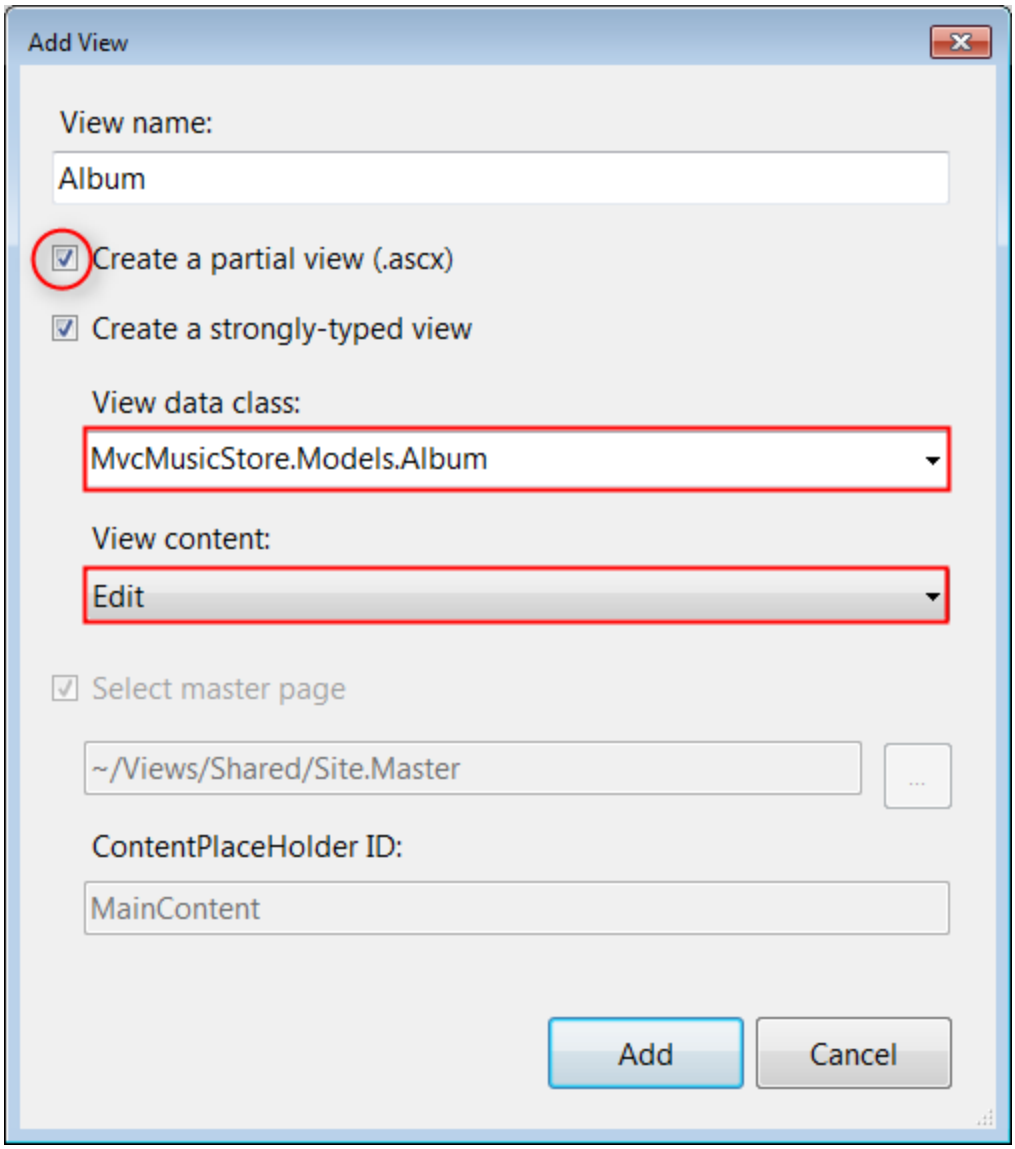
Our Album Edit view and Album Create view will have the same form fields, so we can make use of a shared Album Editor Template. Reusing an Edito rTemplate throughout our site gives our users a consistent user interface. It also makes it easier to manage and maintain our code by eliminating duplicated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

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

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



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



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

```
<%@ Control Language="C#"
Inherits="System.Web.Mvc.ViewUserControl<MvcMusicStore.Models.Album>" %>

<div class="editor-label">
    <%: Html.LabelFor(model => model.AlbumId) %>
</div>
<div class="editor-field">
    <%: Html.TextBoxFor(model => model.AlbumId) %>
    <%: Html.ValidationMessageFor(model => model.AlbumId) %>
</div>

<div class="editor-label">
    <%: Html.LabelFor(model => model.GenreId) %>
</div>
```

```
<div class="editor-field">
    <%: Html.TextBoxFor(model => model.GenreId) %>
    <%: Html.ValidationMessageFor(model => model.GenreId) %>
</div>

<div class="editor-label">
    <%: Html.LabelFor(model => model.ArtistId) %>
</div>
<div class="editor-field">
    <%: Html.TextBoxFor(model => model.ArtistId) %>
    <%: Html.ValidationMessageFor(model => model.ArtistId) %>
</div>

<div class="editor-label">
    <%: Html.LabelFor(model => model.Title) %>
</div>
<div class="editor-field">
    <%: Html.TextBoxFor(model => model.Title) %>
    <%: Html.ValidationMessageFor(model => model.Title) %>
</div>

<div class="editor-label">
    <%: Html.LabelFor(model => model.Price) %>
</div>
<div class="editor-field">
    <%: Html.TextBoxFor(model => model.Price, String.Format("{0:F}", Model.Price)) %>
    <%: Html.ValidationMessageFor(model => model.Price) %>
</div>

<div class="editor-label">
    <%: Html.LabelFor(model => model.AlbumArtUrl) %>
</div>
<div class="editor-field">
    <%: Html.TextBoxFor(model => model.AlbumArtUrl) %>
    <%: Html.ValidationMessageFor(model => model.AlbumArtUrl) %>
</div>
```

Running the application and browsing to /StoreManager/Edit/5 will show that our Album edit form hasn't changed. Now it's time to customize the Album editor template.

Creating the StoreManagerViewModel

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

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

We'll create a new "StoreManagerViewModel" class to help manage all of the above data. We'll use this class within both our Edit and Create action methods.

Create the StoreManagerViewModel class in the ViewModels folder and define it like so:

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

namespace MvcMusicStore.ViewModels
{
    public class StoreManagerViewModel
    {
        public Album         Album   { get; set; }
        public List<Artist>  Artists { get; set; }
        public List<Genre>   Genres  { get; set; }
    }
}
```

Next, we need to modify our Edit action so that it will return a StoreManagerViewModel rather than an Album. To do that, we will change the StoreManagerController's Edit action as follows:

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

public ActionResult Edit(int id)
{
    var viewModel = new StoreManagerViewModel
    {
        Album = storeDB.Albums.Single(a => a.AlbumId == id),
        Genres = storeDB.Genres.ToList(),
        Artists = storeDB.Artists.ToList()
    };

    return View(viewModel);
}
```

The StoreManagerViewModel is now populated with the album matching the ID which was passed in, as well as a list of Genres and Artists.

Updating the Edit View to display the StoreManagerViewModel

Since we have changed the type of the object we are returning to the View, we need to make three small changes to our Edit view.

First, we need to update the type declaration. We do this by changing the Inherits attribute at the top of /Views/StoreManager/Edit.aspx to reference our new StoreManagerViewModel as shown below.

```
<%@ Page Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage<MvcMusicStore.ViewModels.StoreManagerViewModel>" %>
```

Next, we will need to change any code that refers directly to properties of the Model so that they are up to date with the new Model type for the page. Our view originally displayed an Album, which had a Title property. That Album is now contained inside our StoreMangagerViewModel, so we need to update the Title of the page to display Model.Album.Title.

```
<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
       Edit - <%: Model.Album.Title %>
</asp:Content>
```

Finally, we need to change our call to Html.EditorForModel so that it will display the Album template. Rather than using Html.EditorForModel, we will use Html.EditorFor, which allows us to specify a specific Editor type. In this case, we want an editor for an Album, so we will pass that in as the first parameter.

We also want to pass additional information from our ViewModel to the Editor template. We can make use an overload that allows us to pass in our Artist and Genre lists for the dropdowns. The completed Edit view code appears below.

```
<%@ Page Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
   Inherits="System.Web.Mvc.ViewPage<MvcMusicStore.ViewModels.StoreManagerViewModel>" %>

<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
       Edit - <%: Model.Album.Title %>
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

    <h2>Edit Album</h2>

    <% using (Html.BeginForm()) {%>
        <%: Html.ValidationSummary(true) %>

    <fieldset>
        <legend>Edit Album</legend>
        <%: Html.EditorFor(model => model.Album,
            new { Artists = Model.Artists, Genres = Model.Genres}) %>

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

    <% } %>

    <div>
        <%: Html.ActionLink("Back to List", "Index") %>
    </div>

</asp:Content>
```

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

Implementing Dropdowns on the Album Editor Template

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

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

Here's what our call looks like:

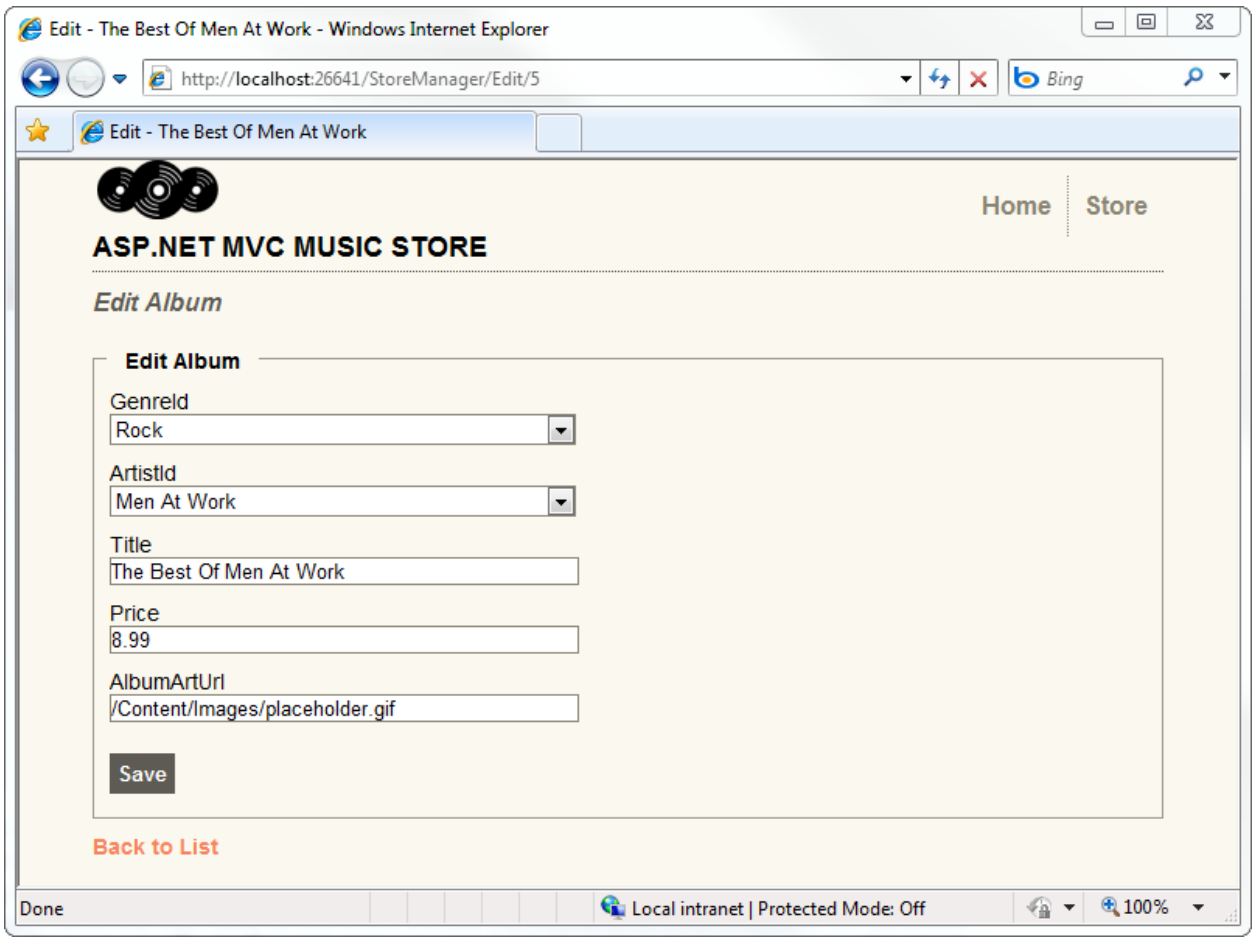
```
<%: Html.DropDownList("ArtistId", new SelectList(ViewData["Artists"] as IEnumerable,
"ArtistId", "Name", Model.ArtistId))%>
```

Note: The only one on that list that may be a little confusing is the SelectList. Remember how we used the Html.EditorFor call earlier, passing in the select items as the second and third parameters? Those are then passed to the EditorTemplate as ViewData properties.

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

```
<%@ Control Language="C#"
Inherits="System.Web.Mvc.ViewUserControl<MvcMusicStore.Models.Album>" %>

<p>
    <%: Html.LabelFor(model => model.Title)%>
    <%: Html.TextBoxFor(model => model.Title)%>
    <%: Html.ValidationMessageFor(model => model.Title)%>
</p>
<p>
    <%: Html.LabelFor(model => model.Price)%>
    <%: Html.TextBoxFor(model => model.Price)%>
    <%: Html.ValidationMessageFor(model => model.Price)%>
</p>
<p>
    <%: Html.LabelFor(model => model.AlbumArtUrl)%>
    <%: Html.TextBoxFor(model => model.AlbumArtUrl)%>
    <%: Html.ValidationMessageFor(model => model.AlbumArtUrl)%>
</p>
<p>
    <%: Html.LabelFor(model => model.Artist)%>
    <%: Html.DropDownList("ArtistId", new SelectList(ViewData["Artists"] as IEnumerable,
"ArtistId", "Name", Model.ArtistId))%>
</p>
<p>
    <%: Html.LabelFor(model => model.Genre)%>
    <%: Html.DropDownList("GenreId", new SelectList(ViewData["Genres"] as IEnumerable,
"GenreId", "Name", Model.GenreId))%>
</p>
```

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



Implementing the HTTP-POST Edit Action Method

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

This controller action method will perform three steps:

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

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

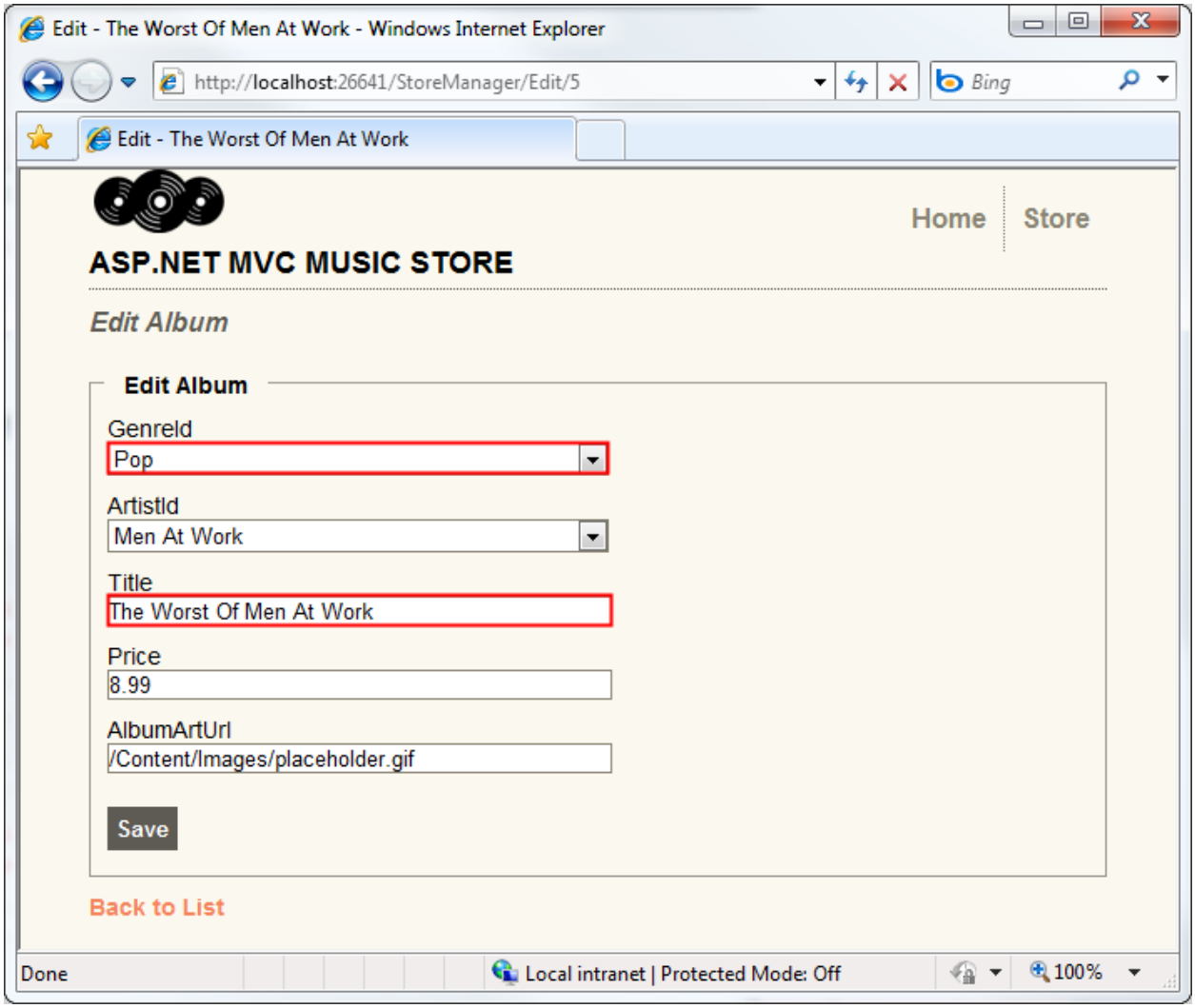
```
//
// POST: /StoreManager/Edit/5
```

```
[HttpPost]
public ActionResult Edit(int id, FormCollection formValues)
{
    var album = storeDB.Albums.Single(a => a.AlbumId == id);

    try
    {
        //Save Album

        UpdateModel(album, "Album");
        storeDB.SaveChanges();

        return RedirectToAction("Index");
    }
    catch
    {
        //Error occurred – so redisplay the form

        var viewModel = new StoreManagerViewModel
        {
            Album = album,
            Genres = storeDB.Genres.ToList(),
            Artists = storeDB.Artists.ToList()
        };

        return View(viewModel);
    }
}
```

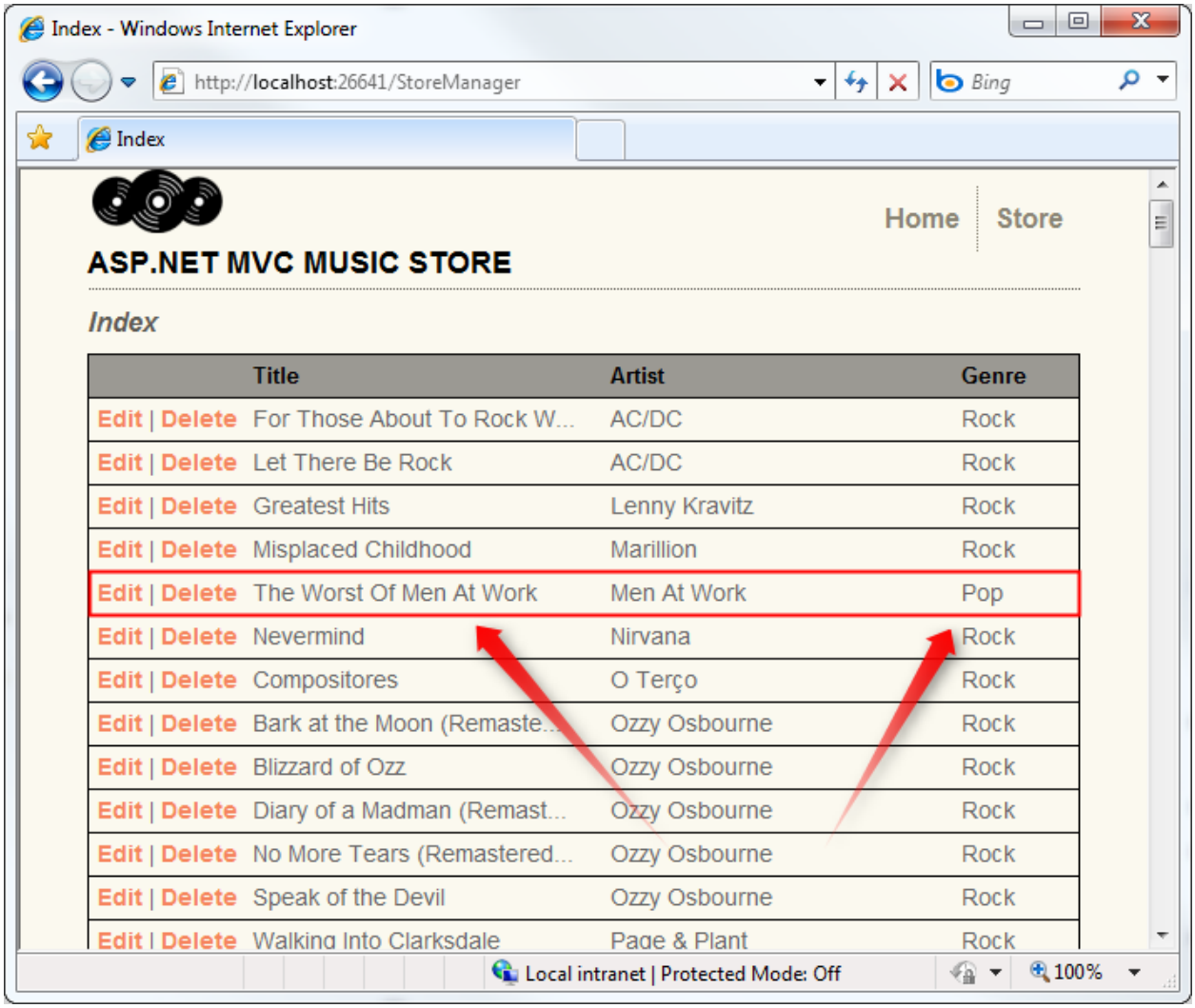
We use the [HttpPost] attribute to indicate that this form will only respond to the HTTP-POST method, so it is only displayed when a user submits the form.

Note: If we were calling UpdateModel from a form which edited a simple type, we would just call UpdateModel(album). However, this edit form is being shown for a complex type - a StoreManagerViewModel. For that reason, it is necessary to specify a prefix so that the model binder will know how to map the form values to the model type that it is updating.

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



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



Implementing the Create Action

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

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

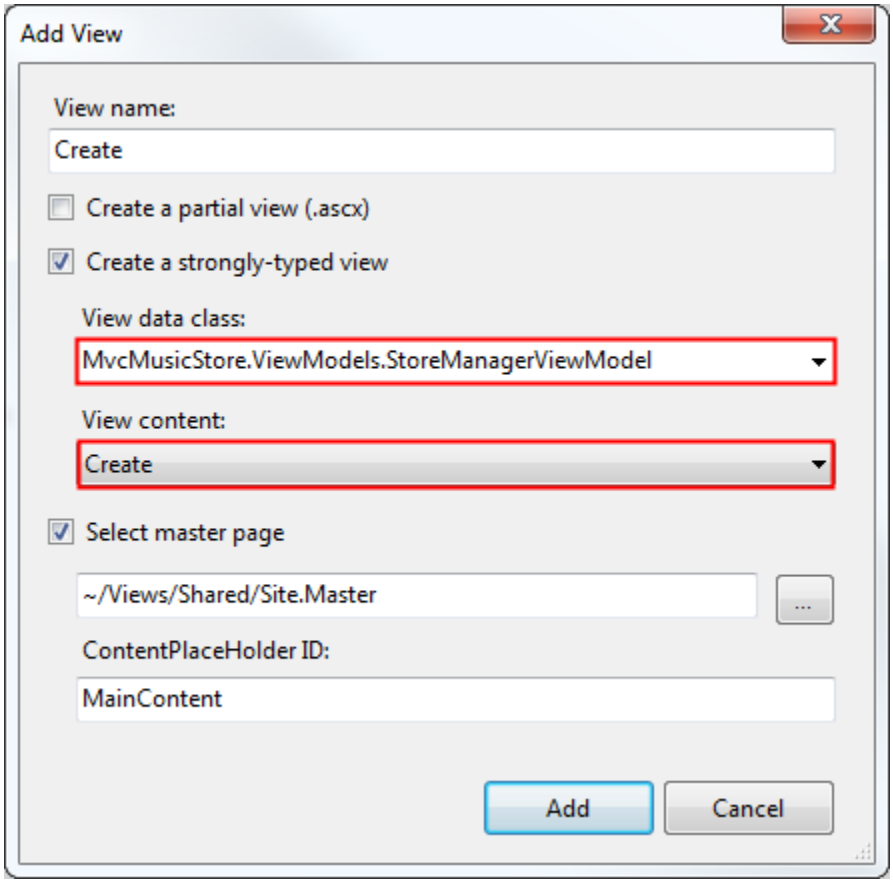
Implementing the HTTP-GET Create Action Method

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

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

public ActionResult Create()
```

```
{
    var viewModel = new StoreManagerViewModel
    {
        Album = new Album(),
        Genres = storeDB.Genres.ToList(),
        Artists = storeDB.Artists.ToList()
    };

    return View(viewModel);
}
```

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

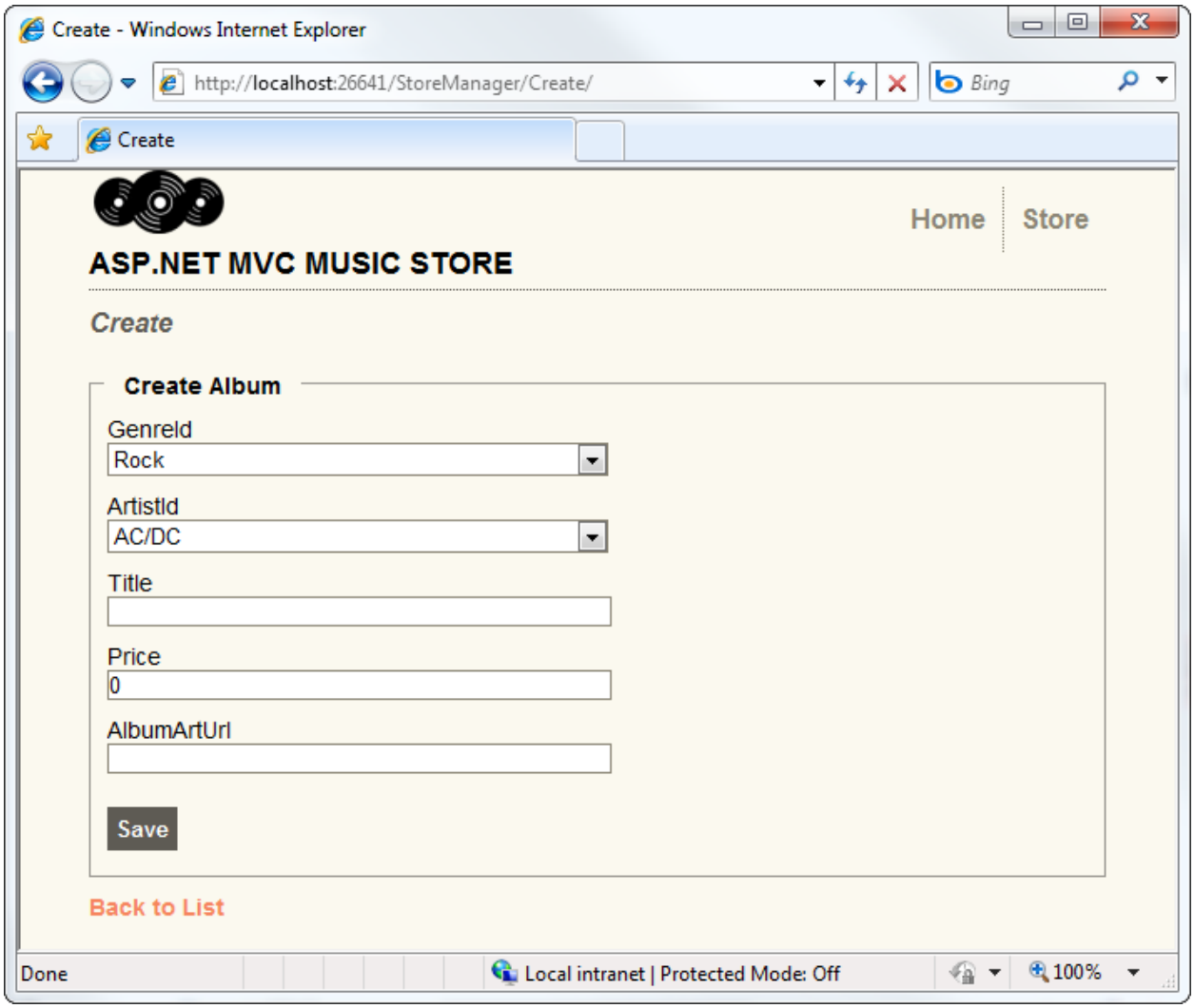


We will update the generated Create.aspx view template to invoke the Html.EditorFor() helper method, just as we did in the Edit form before. As planned, this View will be able to leverage our Album editor template.

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage<MvcMusicStore.ViewModels.StoreManagerViewModel>" %>

<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
      Create
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
```

```
    <h2>Create</h2>

    <% using (Html.BeginForm()) {%>
        <%: Html.ValidationSummary(true) %>

    <fieldset>
        <legend>Create Album</legend>
        <%: Html.EditorFor(model => model.Album,
            new { Artists = Model.Artists, Genres = Model.Genres })%>
        <p>
            <input type="submit" value="Save" />
        </p>
    </fieldset>


    <% } %>

    <div>
        <%: Html.ActionLink("Back to List", "Index") %>
    </div>

</asp:Content>
```

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



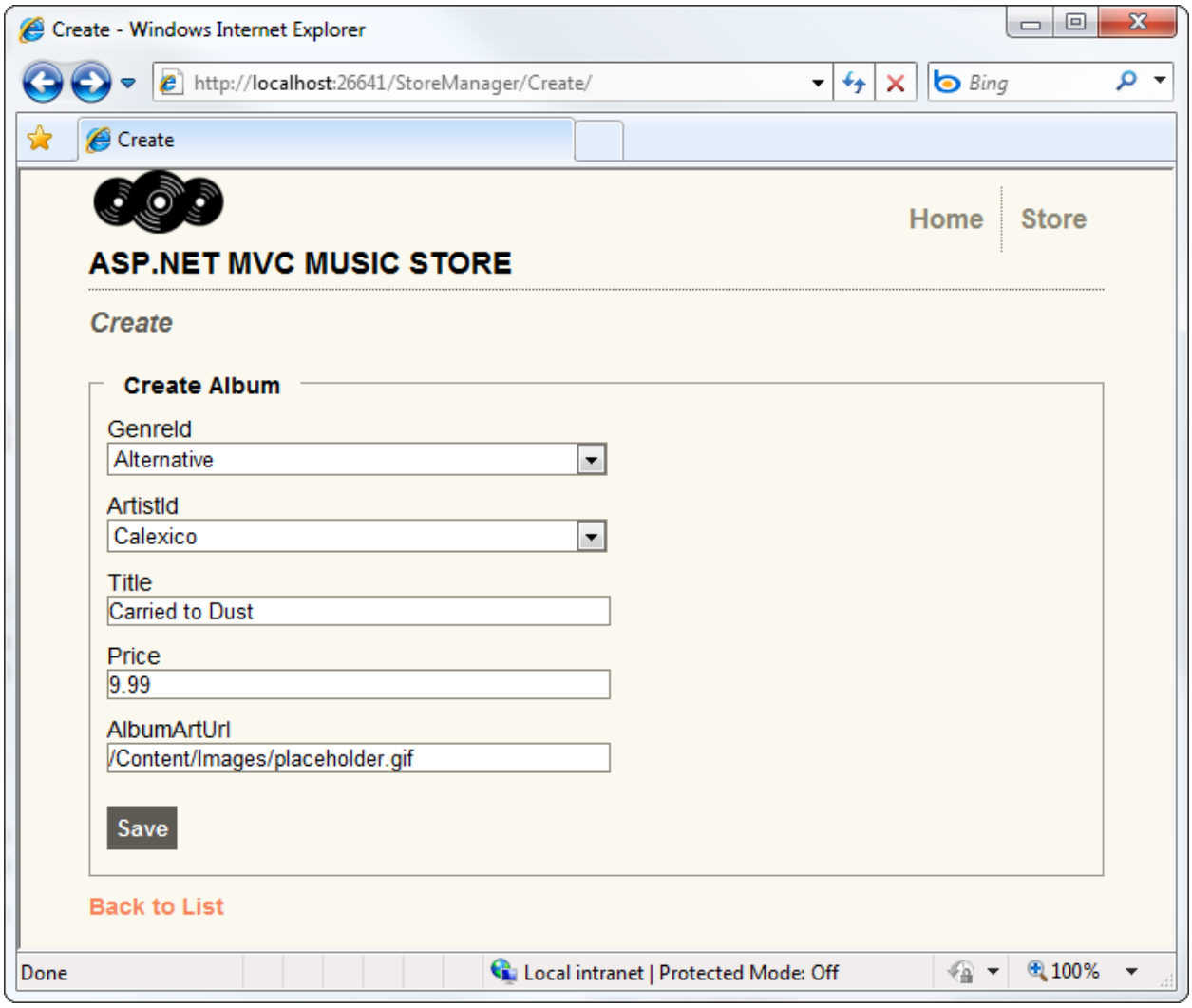
Lastly, we'll implement our HTTP-POST Create controller action method, which will be invoked when a store manager posts a new album back to the server. We'll implement it using the code below:

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

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

        //Save Album
        storeDB.AddToAlbums(album);
        storeDB.SaveChanges();

        return RedirectToAction("Index");

    }
```

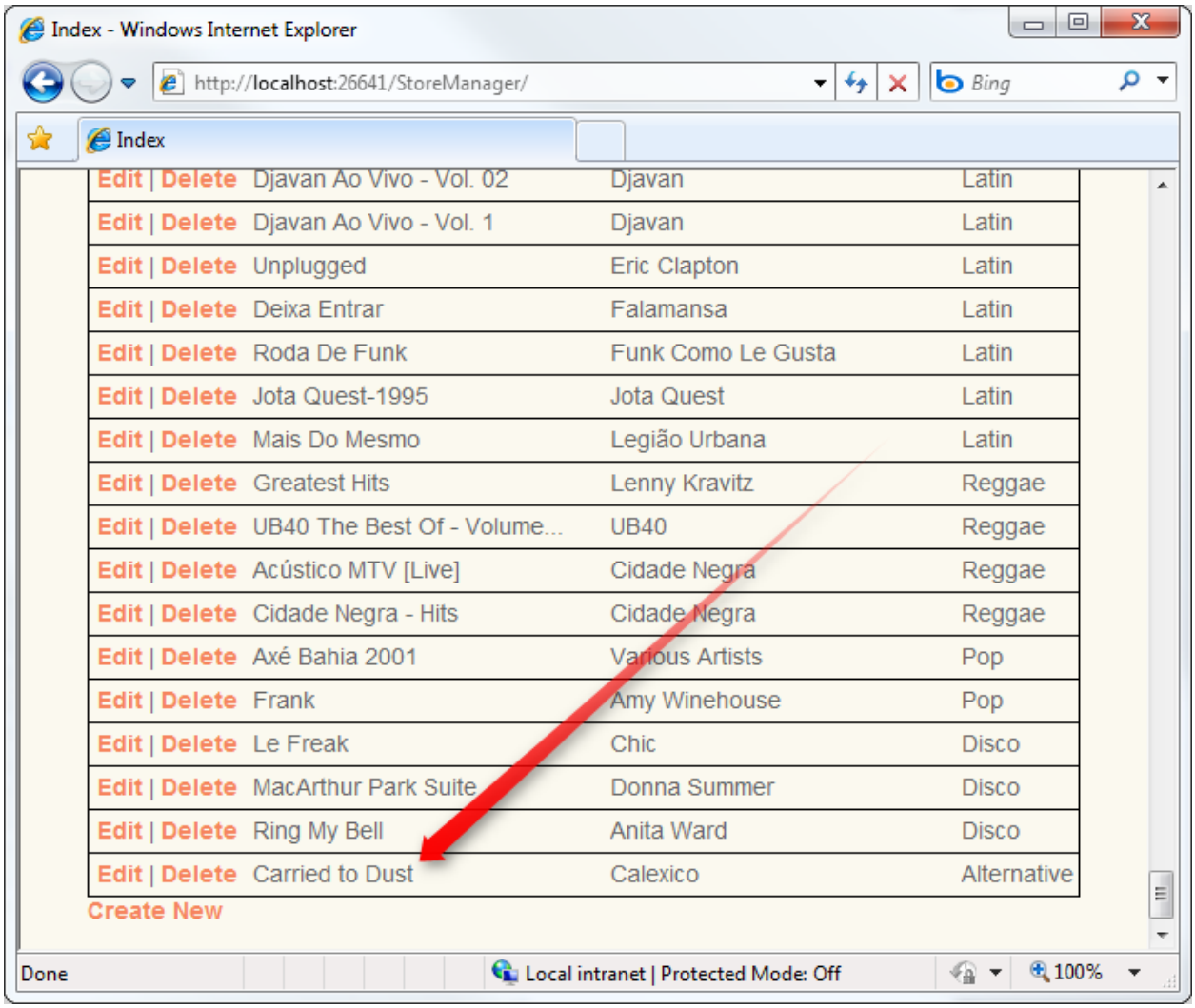
```
    // Invalid - redisplay with errors
    var viewModel = new StoreManagerViewModel
    {
        Album = album,
        Genres = storeDB.Genres.ToList(),
        Artists = storeDB.Artists.ToList()
    };

    return View(viewModel);
}
```

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

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



Pressing the Save button adds this album to the list.



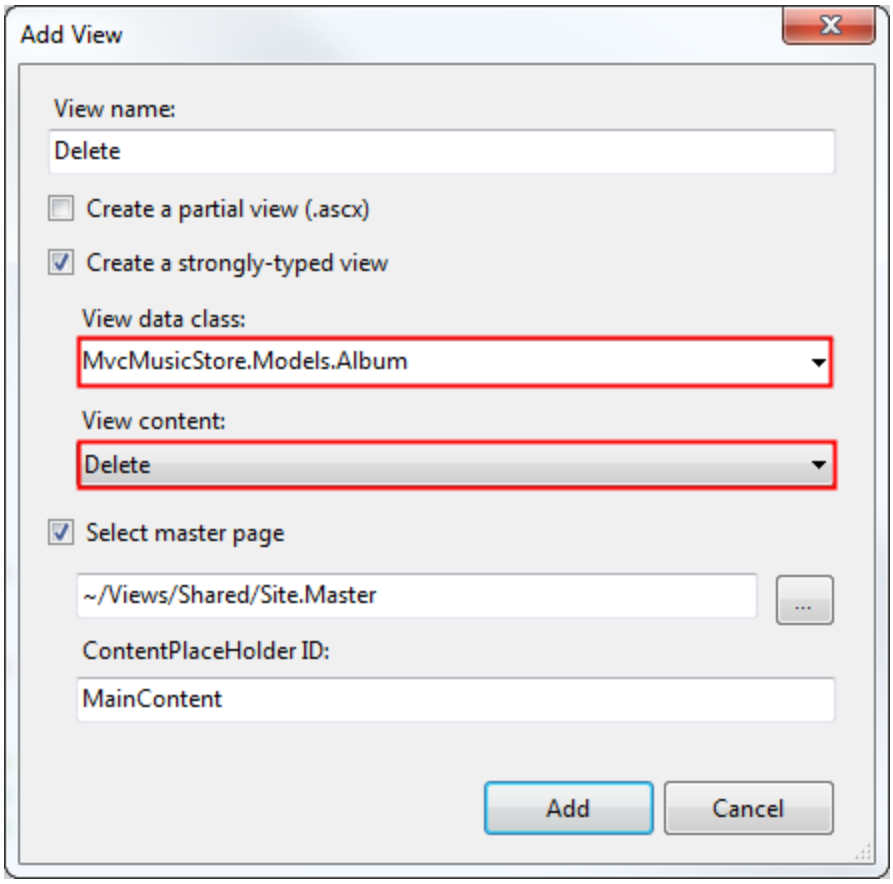
Handling Deletion

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

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

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

public ActionResult Delete(int id)
{
    var album = storeDB.Albums.Single(a => a.AlbumId == 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.

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage<MvcMusicStore.Models.Album>" %>

<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
      Delete
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

    <h2>
        Delete Confirmation
    </h2>

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

    <div>
        <% using (Html.BeginForm()) {%>
            <input type="submit" value="Delete" />
        <% } %>
    </div>
```

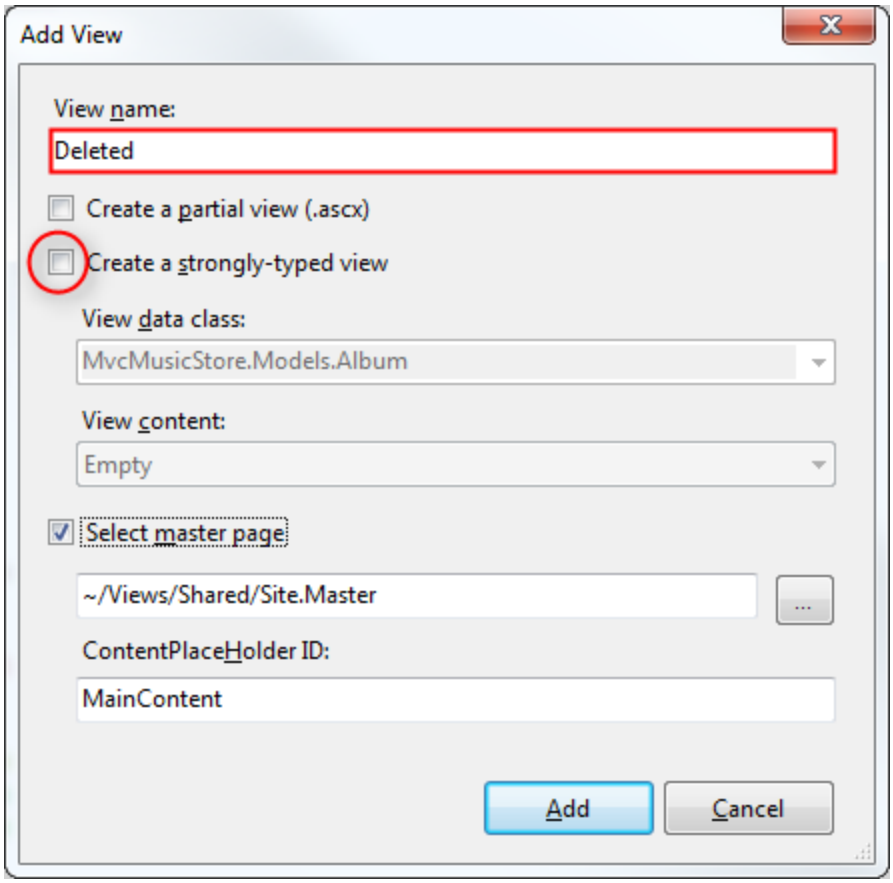
```
</asp:Content>
```

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

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

```
[HttpPost]
public ActionResult Delete(int id, string confirmButton)
{
    var album = storeDB.Albums
        .Include("OrderDetails").Include("Carts")
        .Single(a => a.AlbumId == id);

    // For simplicity, we're allowing deleting of albums
    // with existing orders We've set up OnDelete = Cascade
    // on the Album->OrderDetails and Album->Carts relationships

    storeDB.DeleteObject(album);
    storeDB.SaveChanges();

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

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

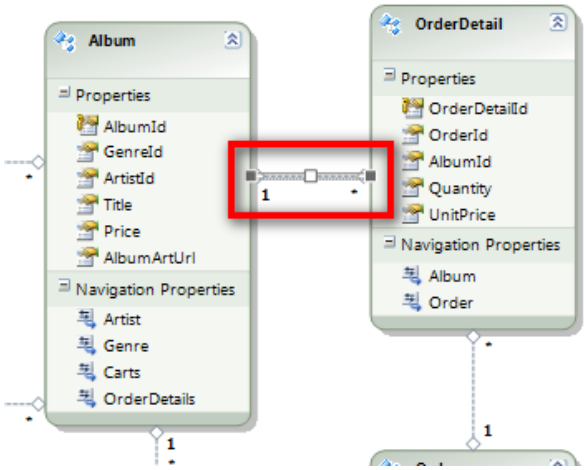


The Deleted view just shows a message that the album was deleted and shows a link back to the Store Manager Index.

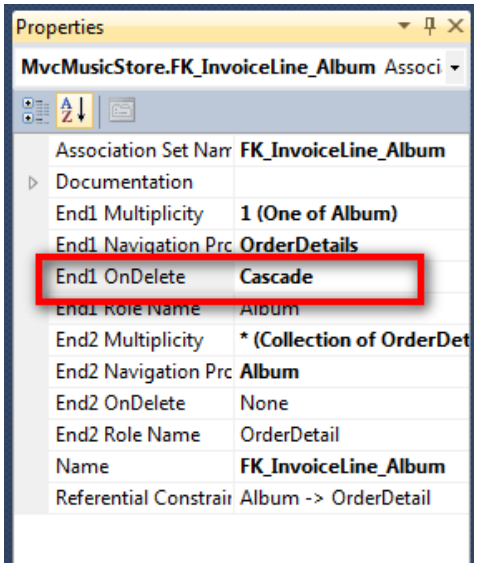
```
<%@ Page Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage" %>

<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
       Album Deleted
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

    <h2>Album Deleted</h2>

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

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

</asp:Content>
```

We're using cascading deletes so that deleting an Album will also delete associated OrderDetails and Cart entries. If we didn't do this, attempting to delete an Album which had previously been ordered would fail due to the foreign key relationship in the database. You would need to assess this for your specific business scenario, since in many cases you wouldn't want to allow deleting products which had previously been ordered.

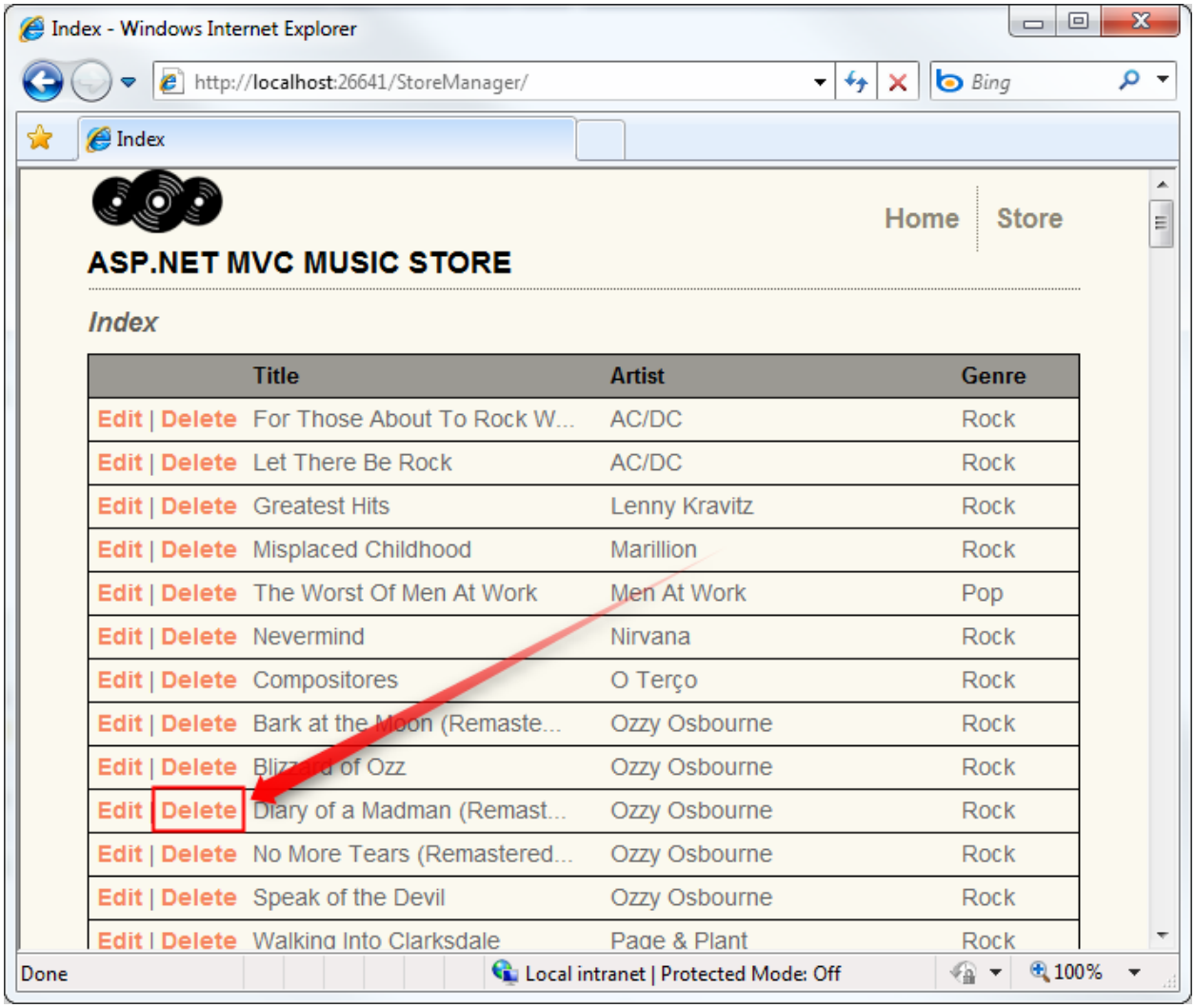
To enable cascade deletes, first select the relationship in the Entity Data Model designer.



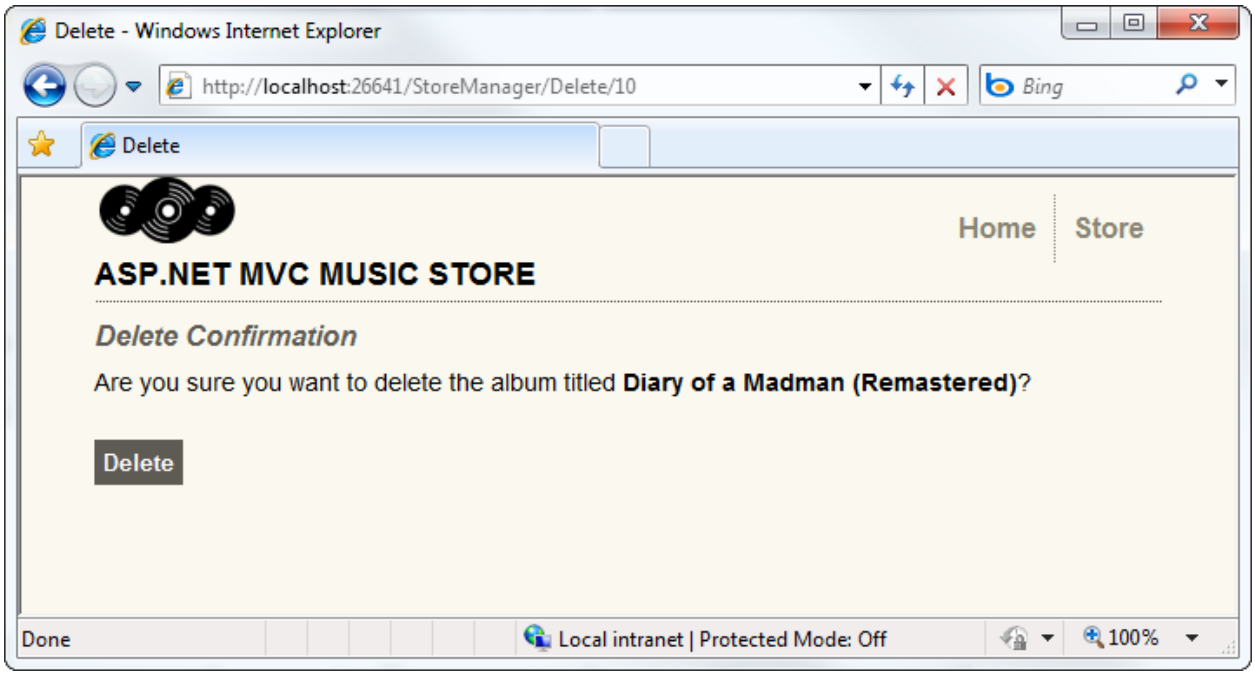
If the Properties window is not displayed, you can show it by clicking on the arrow between Album and OrderDetails and pressing the F4 key. Now in the properties window, set the OnDelete value for the associated table to Cascade. In this case, Album is shown as "End1 Role Name", so we will set the "End1 OnDelete" action to Cascade.



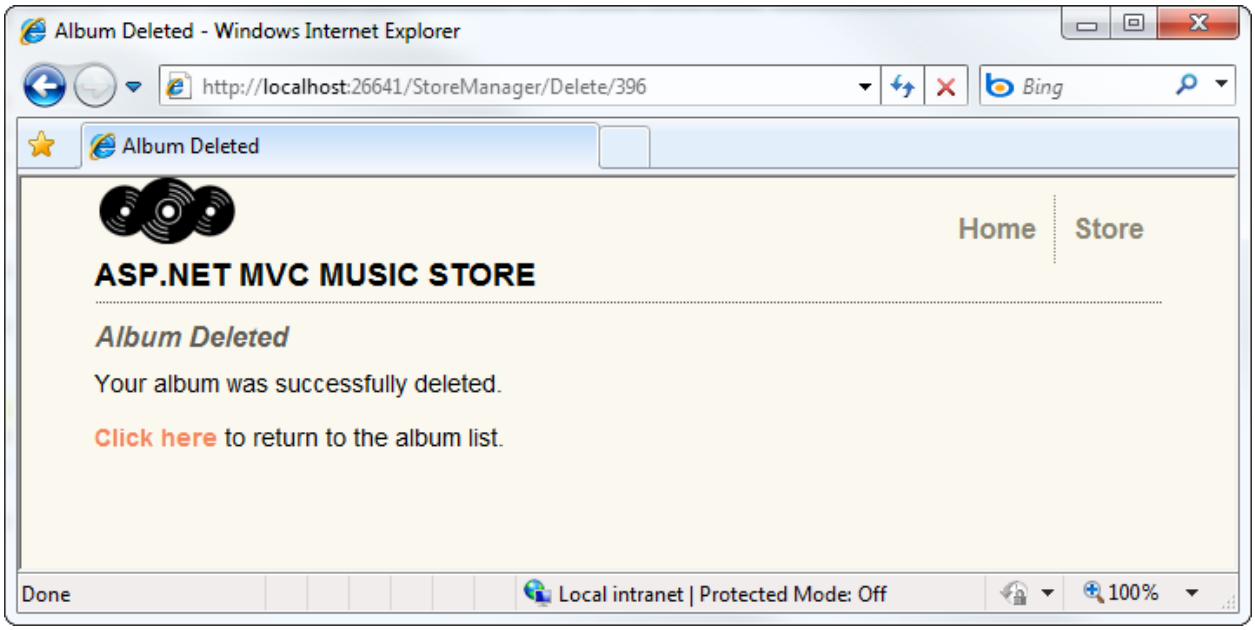
To test this, run the application and browse to /StoreManager. Select an album from the list and click the Delete link.



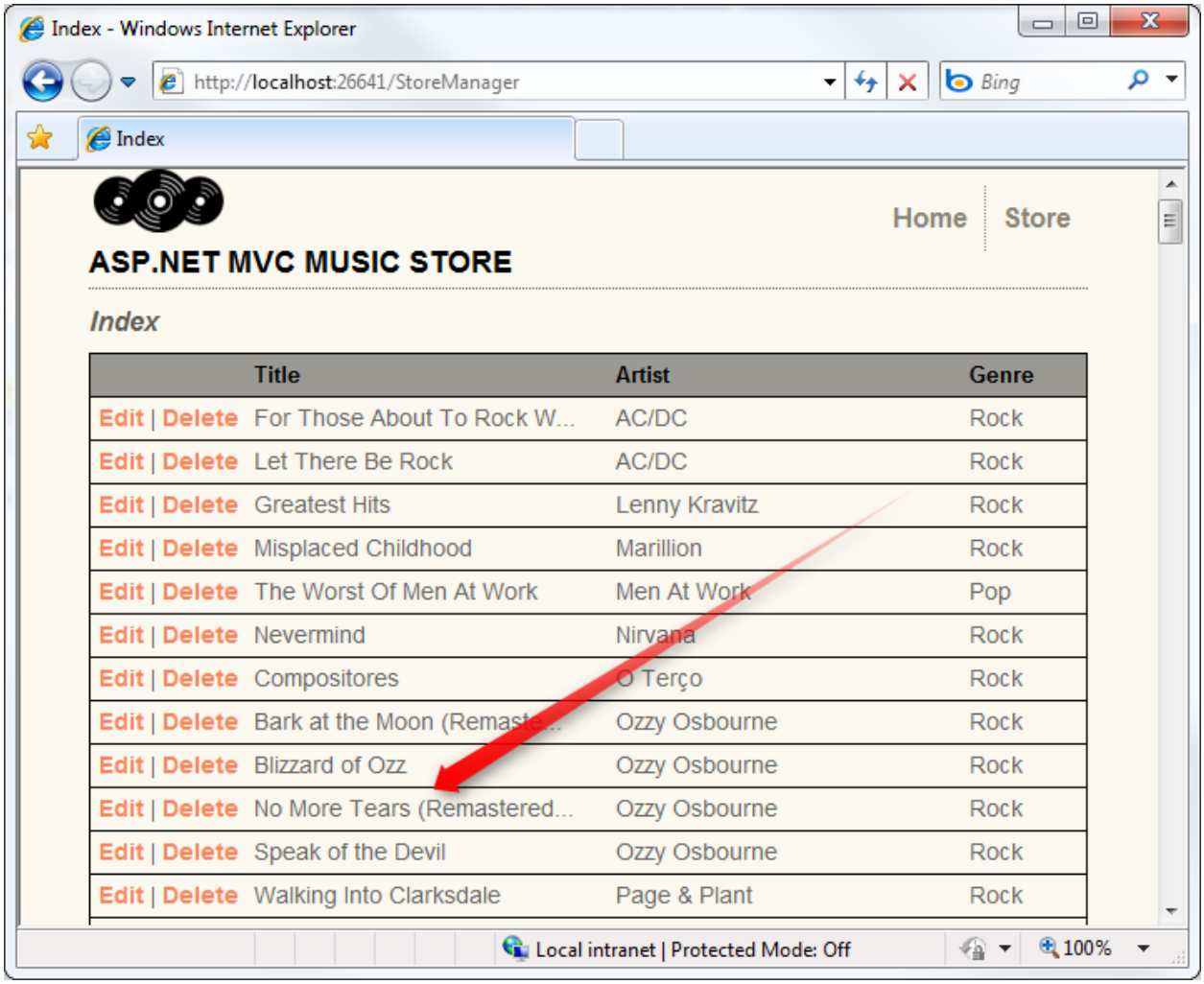
This displays our Delete confirmation screen.



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



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


Index - Windows Internet Explorer
http://localhost:26641/StoreManager
Bing
Index
Home
Store
ASP.NET MVC MUSIC STORE
Index
Title
Artist
Genre
Edit | Delete For Those About To Rock W... AC/DC Rock
Edit | Delete Let There Be Rock AC/DC Rock
Edit | Delete Greatest Hits Lenny Kravitz Rock
Edit | Delete Misplaced Childhood Marillion Rock
Edit | Delete The Worst Of Men At Work Men At Work Pop
Edit | Delete Nevermind Nirvana Rock
Edit | Delete Compositores O Terço Rock
Edit | Delete Bark at the Moon (Remaste... Ozzy Osbourne Rock
Edit | Delete Blizzard of Ozz Ozzy Osbourne Rock
Edit | Delete No More Tears (Remastered... Ozzy Osbourne Rock
Edit | Delete Speak of the Devil Ozzy Osbourne Rock
Edit | Delete Walking Into Clarksdale Page & Plant Rock
Local intranet | Protected Mode: Off
100%

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.

For a simple Model class, adding a Data Annotation is just handled by adding a using statement for System.ComponentModel.DataAnnotation, then placing attributes on your properties, like this:

```
using System.ComponentModel.DataAnnotations;

namespace SuperheroSample.Models
{
    public class Superhero
    {
        [Required]
        public string Name { get; set; }

        public bool WearsCape { get; set; }
    }
}
```

The above example makes the Name value a required field.

Using MetaData Partial Classes with Entity Framework

This is a little more complex in the case of Entity Framework models, since the model classes are generated. If we added Data Annotations directly to our model classes, they would be overwritten if we update our model from the database.

Instead, we can make use of metadata partial classes which exist to hold our annotations and are associated with our model classes using the [MetadataType] attribute.

For our Album model, we would declare our metadata class as follows.

```
using System.ComponentModel.DataAnnotations;

namespace MvcMusicStore.Models
{
    [MetadataType(typeof(AlbumMetaData))]
    public partial class Album
    {
        // Validation rules for the Album class

        public class AlbumMetaData
        {
        // Rules go here
        }
    }
```

```
}
```

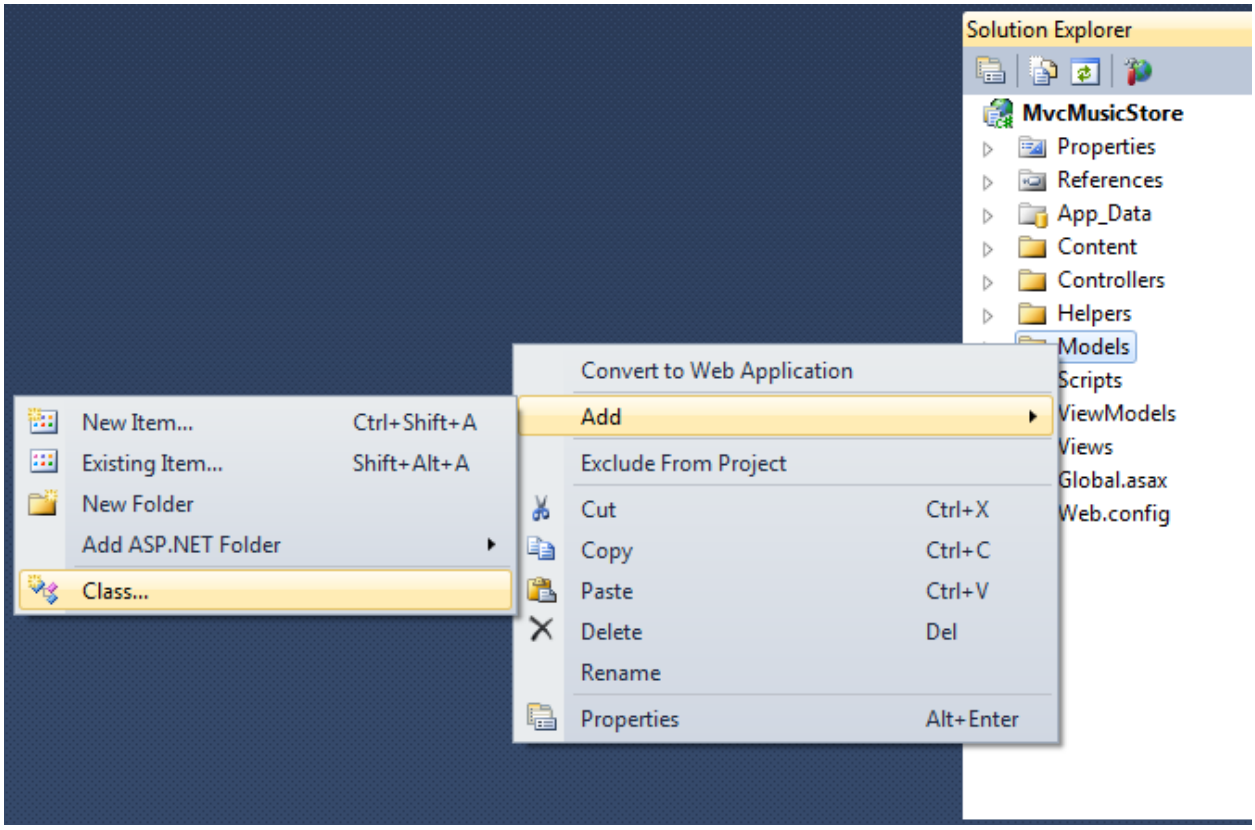
Adding Validation to our Album Forms

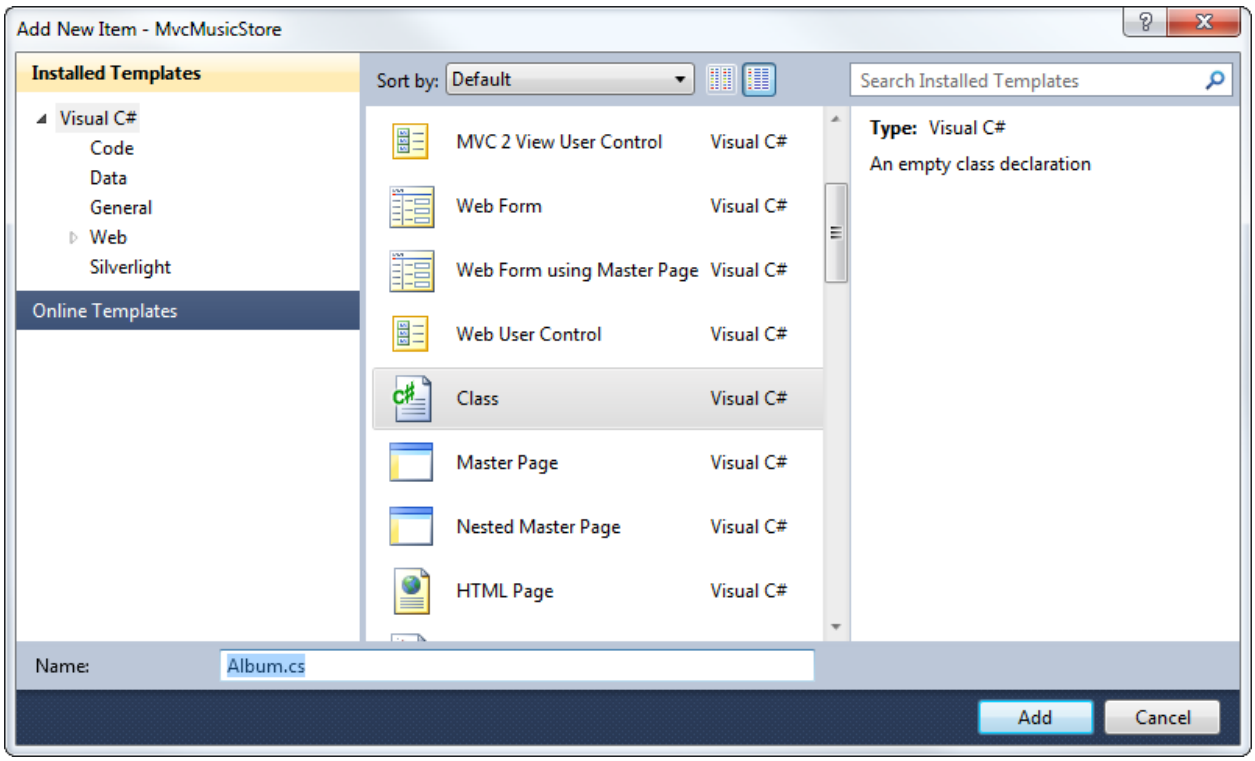
This Album partial class has a MetadataType attribute which points to the AlbumMetaData class for our Data Annotations. We can then annotate our Album model. We'll use the following Data Annotation attributes:

- 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

We can handle our validation by adding an Album class to the Models folder to hold our validation rules.



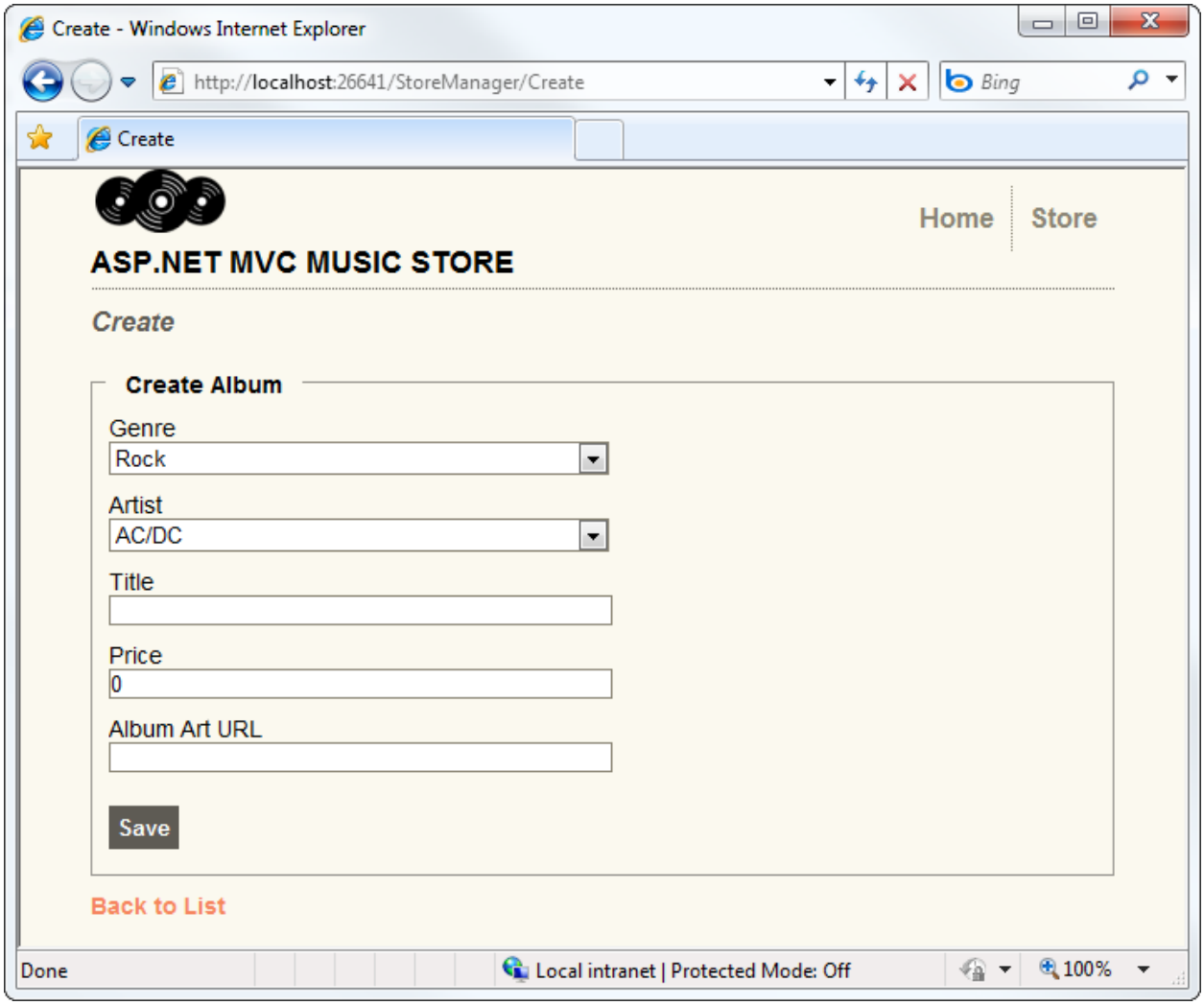


Add the following code to this class:

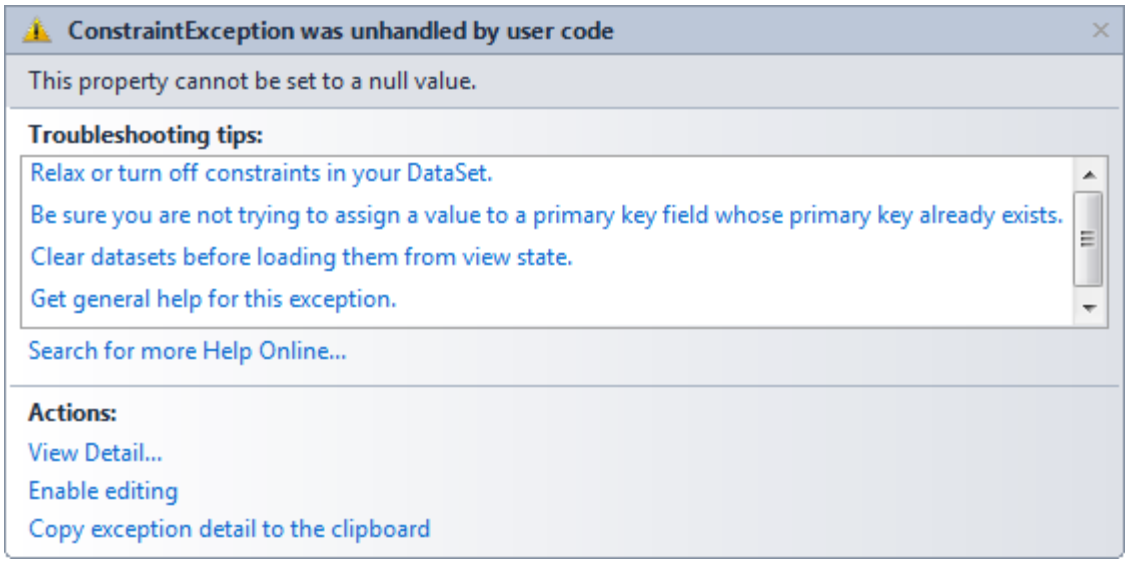
```
using System.ComponentModel;
using System.ComponentModel.DataAnnotations;
using System.Web.Mvc;

namespace MvcMusicStore.Models
{
    [MetadataType(typeof(AlbumMetaData))]
    public partial class Album
    {
        // Validation rules for the Album class

        [Bind(Exclude = "AlbumId")]
        public class AlbumMetaData
        {
            [ScaffoldColumn(false)]
            public object AlbumId { get; set; }

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

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

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

            [DisplayName("Album Art URL")]
```

```
            [StringLength(1024)]
            public object AlbumArtUrl { get; set; }

            [Required(ErrorMessage = "Price is required")]
            [Range(0.01, 100.00,
                ErrorMessage = "Price must be between 0.01 and 100.00")]
            public object Price { get; set; }
        }
    }
}
```

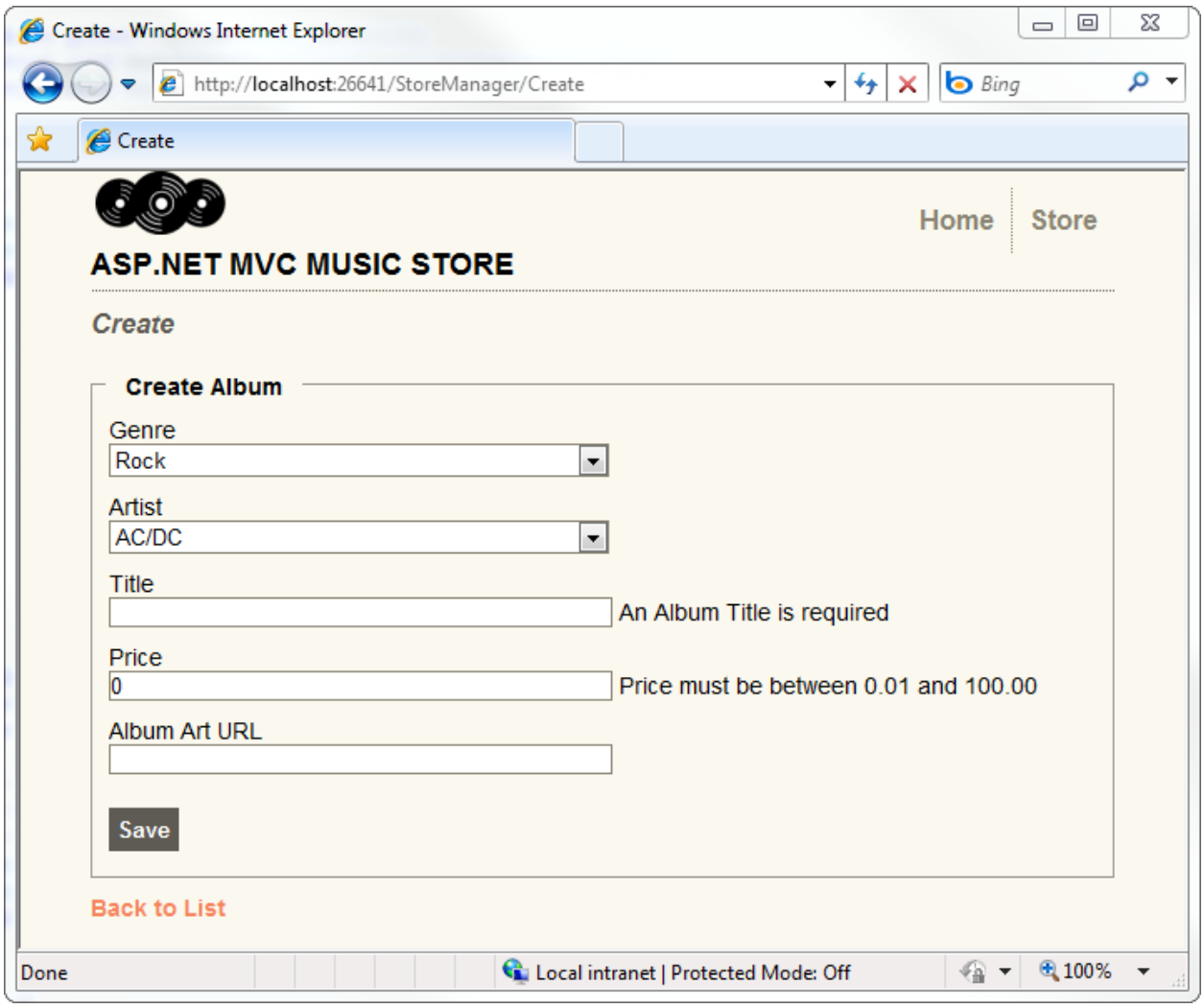
After having added this class, our Create and Edit screen immediately begin validating fields and using the Display Names we've chosen (e.g. Genre instead of GenreID). Run the application and browse to /StoreManager/Create.



When we click on the Save button, when running in Debug mode, Visual Web Developer will break in the StoreDB.Designer.cs code showing a ConstraintException. This is an expected exception, as this is how Entity Framework notifies the ASP.NET MVC framework that a constraint was not met.



This is expected, and will only be shown when running in Debug mode. Don't worry, this is expected, and will not affect your application when it is running in Release mode. Press F5 to continue, and you will see the form displayed with validation error messages showing which fields did not meet the validation rules we have defined.



Using Client-Side Validation

The above validation works, but there are three problems with it.

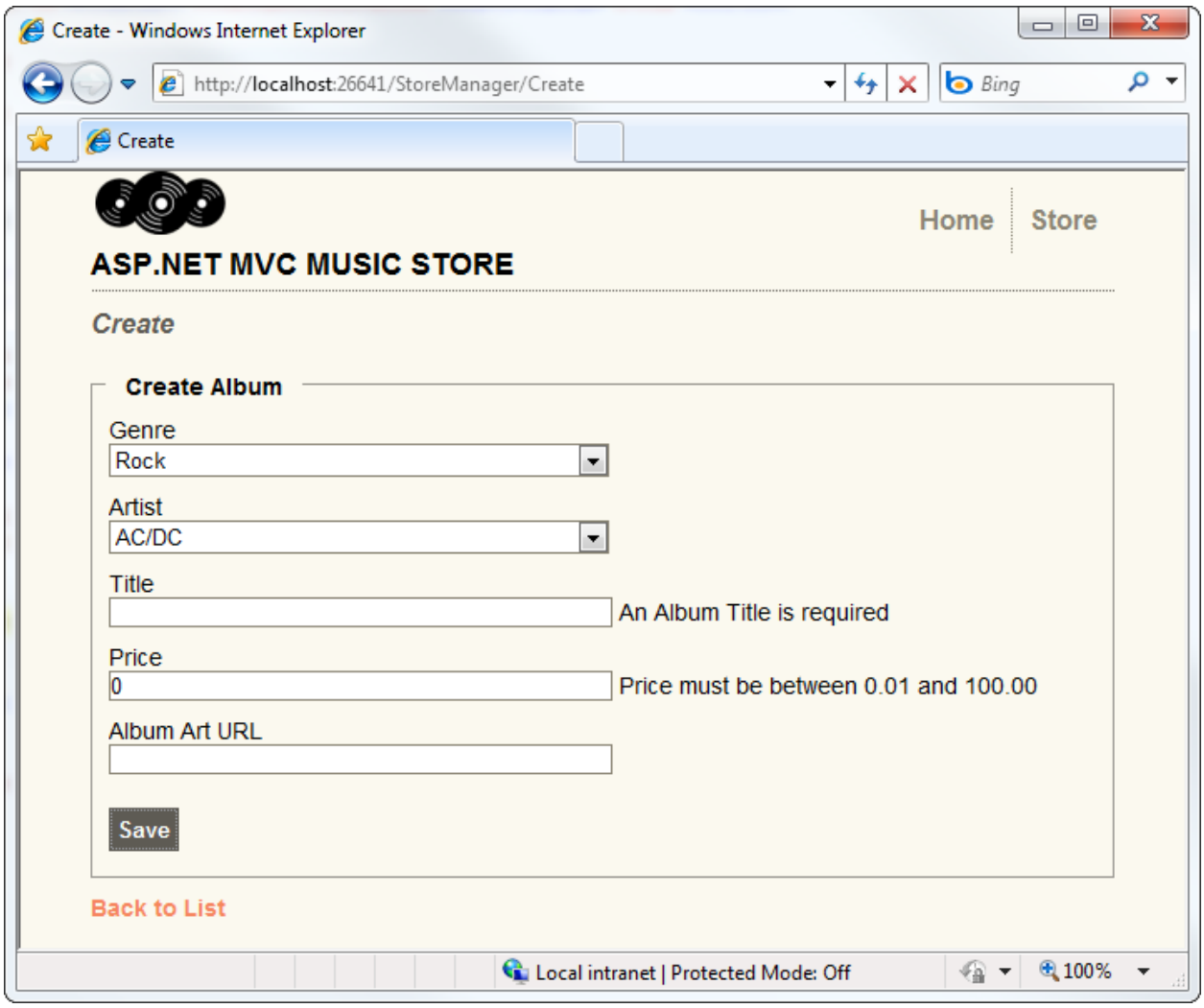
1. The user has to wait for the form to be posted, validated on the server, and for the response to be sent to their browser.
2. The user doesn't get immediate feedback when they correct a field so that it now passes the validation rules.
3. We are wasting server resources to perform validation logic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

We can add client-side (AJAX) validation to our forms with just a few lines of code. First we need to include some JavaScript references, either in our site MasterPage or to the specific forms that will be using client-side validation.

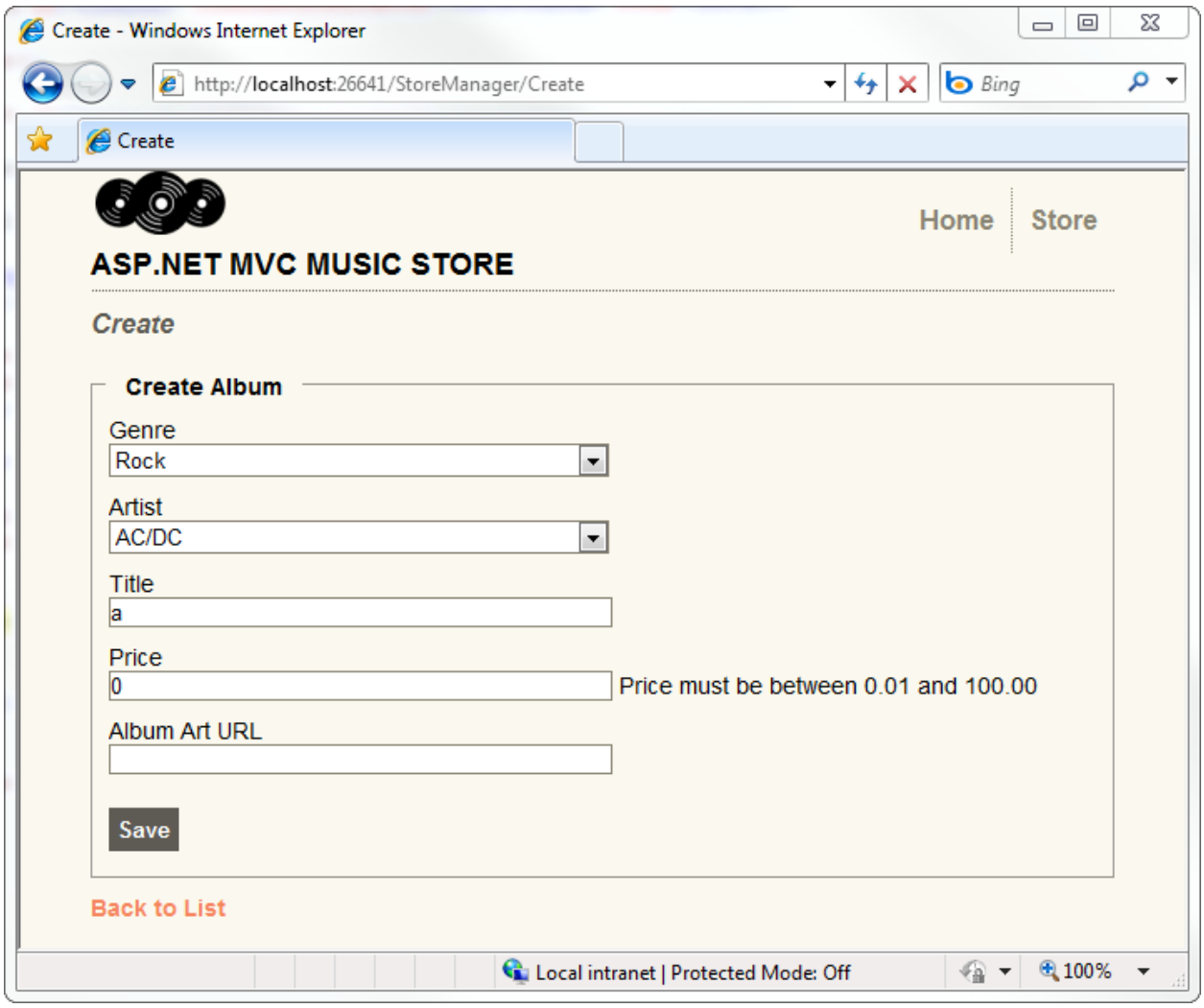
These scripts are already included in our project, so we just need to add the script references to our Album editor template, like this:

```
<%@ Import Namespace="MvcMusicStore"%>

<%@ Control Language="C#"
Inherits="System.Web.Mvc.ViewUserControl<MvcMusicStore.Models.Album>" %>

<script src="/Scripts/MicrosoftAjax.js" type="text/javascript"></script>
<script src="/Scripts/MicrosoftMvcAjax.js" type="text/javascript"></script>
<script src="/Scripts/MicrosoftMvcValidation.js" type="text/javascript"></script>

Now we use the Html.EnableClientValidation helper method to “turn on” client-side
validation. For the both the Create and Edit view templates, add that call directly above
the Html.BeginForm call, like so:
<h2>Create</h2>

<% Html.EnableClientValidation(); %>

<% using (Html.BeginForm()) {%>

<fieldset>
    <legend>Create Album</legend>
    <%: Html.EditorFor(model => model.Album, new { Artists = Model.Artists, Genres =
Model.Genres })%>
    <p>
        <input type="submit" value="Save" />
    </p>
</fieldset>

<% } %>
```

That’s all that’s required to set up client-side validation. Now all fields are checked in the browser before being submitted to our controller actions. Now when we run the application and browse to /StoreManager/Create we will see the validation messages right when we click the Save button, without having to wait for a postback.



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


Create - Windows Internet Explorer
http://localhost:26641/StoreManager/Create
Bing
Create
Home
Store
ASP.NET MVC MUSIC STORE
Create
Create Album
Genre
Rock
Artist
AC/DC
Title
a
Price
0
Price must be between 0.01 and 100.00
Album Art URL
Save
Back to List
Local intranet | Protected Mode: Off
100%

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 2 Web Application template and the ASP.NET MVC 2 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 2 Web Application template.

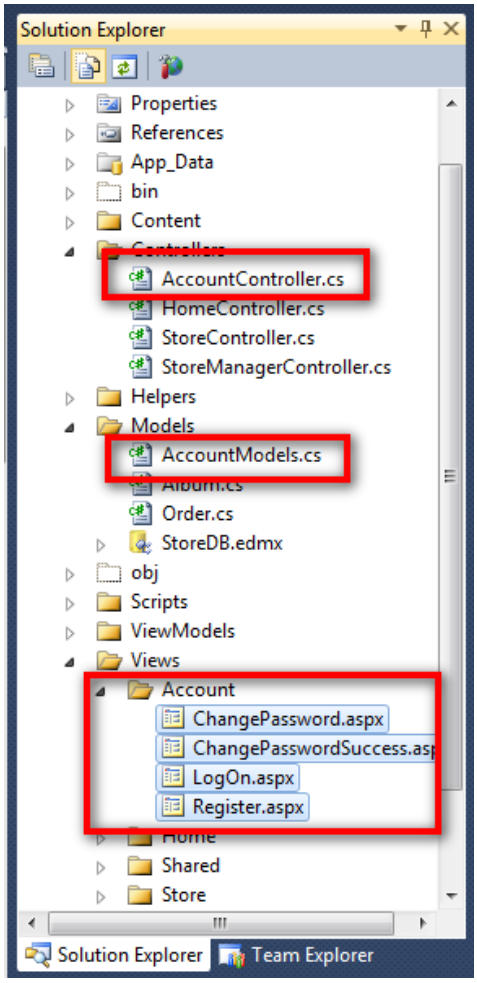
Create a new ASP.NET MVC application using the full ASP.NET MVC 2 Web Application template and copy the following files into the same directories in our project:

1. Copy AccountController.cs in the Controllers directory
2. Copy AccountModels 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: These files are also available in the MvcMusicStore-Assets.zip download from which we copied our site design files at the beginning of the tutorial. The Membership files are located in the Code directory.

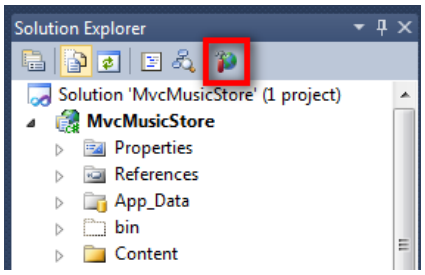
The updated solution should look like the following:



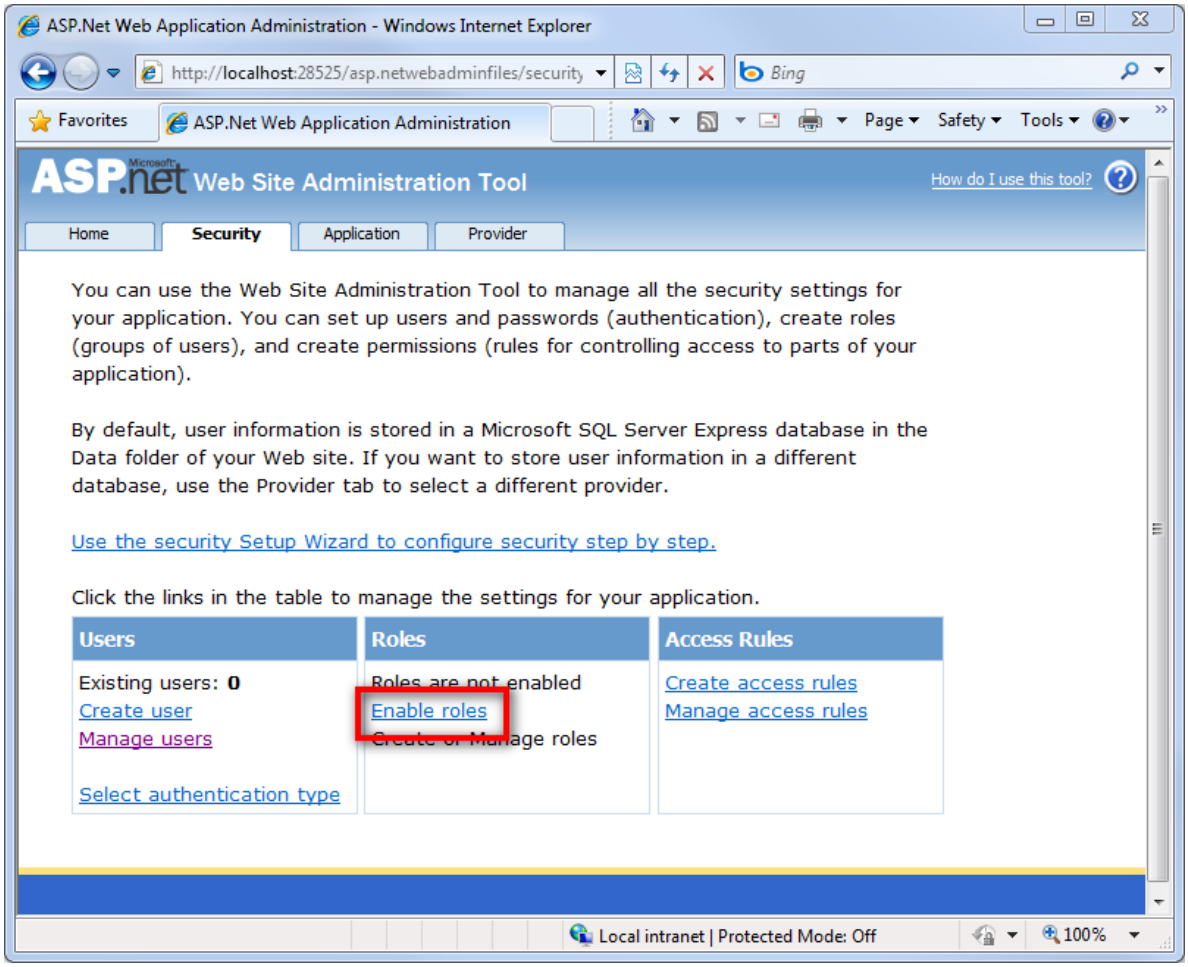
Adding an Administrative User with the ASP.NET Configuration site

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

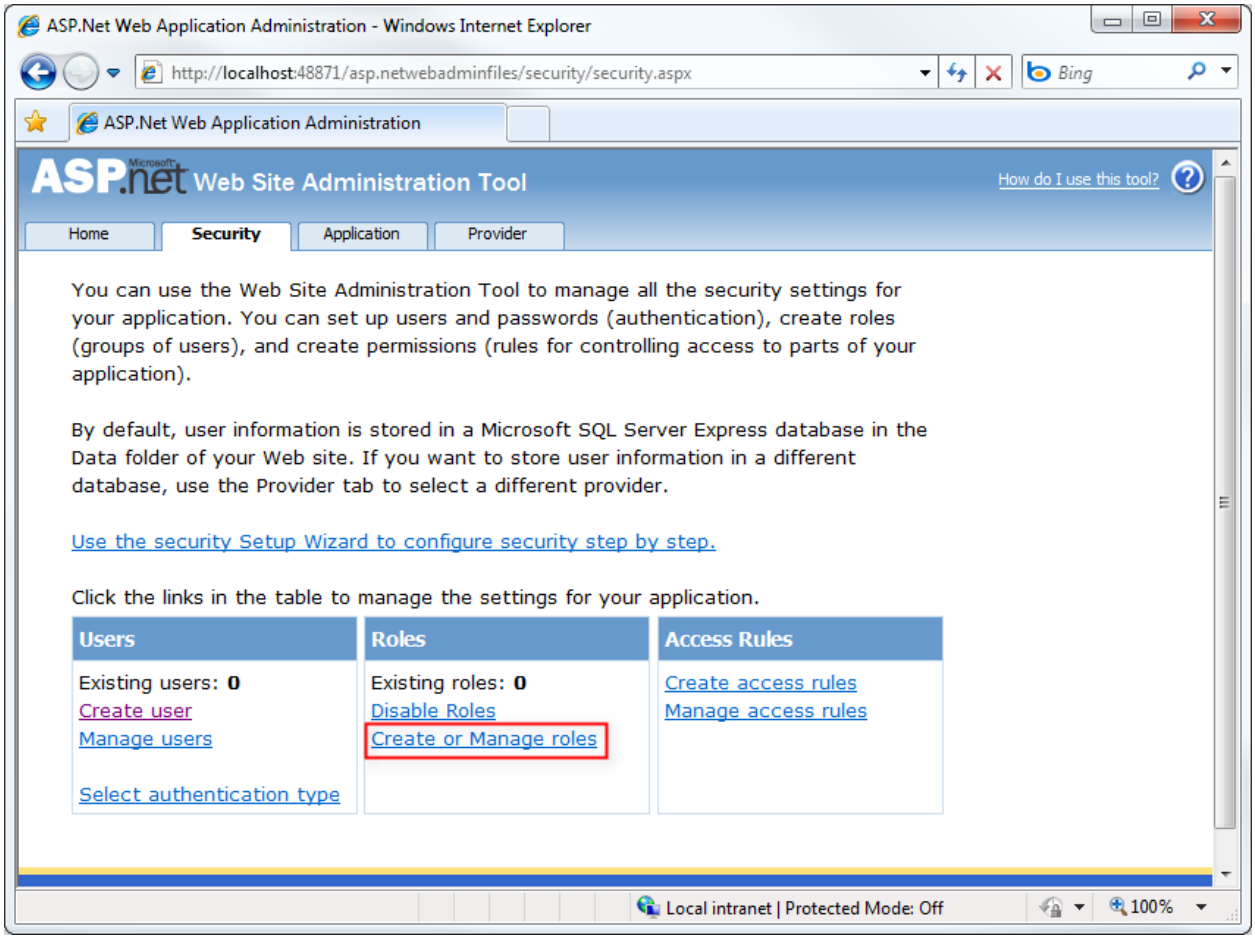
Launch the ASP.NET Configuration website by clicking following the icon in the Solution Explorer.



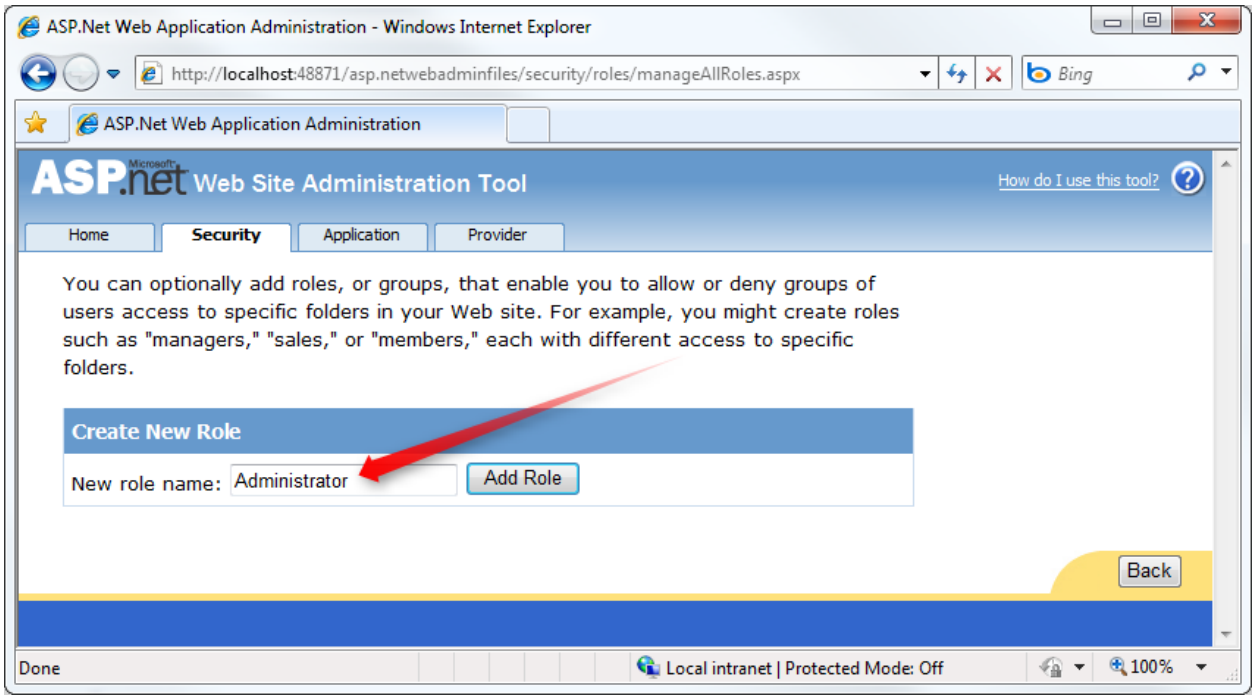
This launches a configuration website. Click on the Security tab on the home screen, then click the "Enable roles" link in the center of the screen.



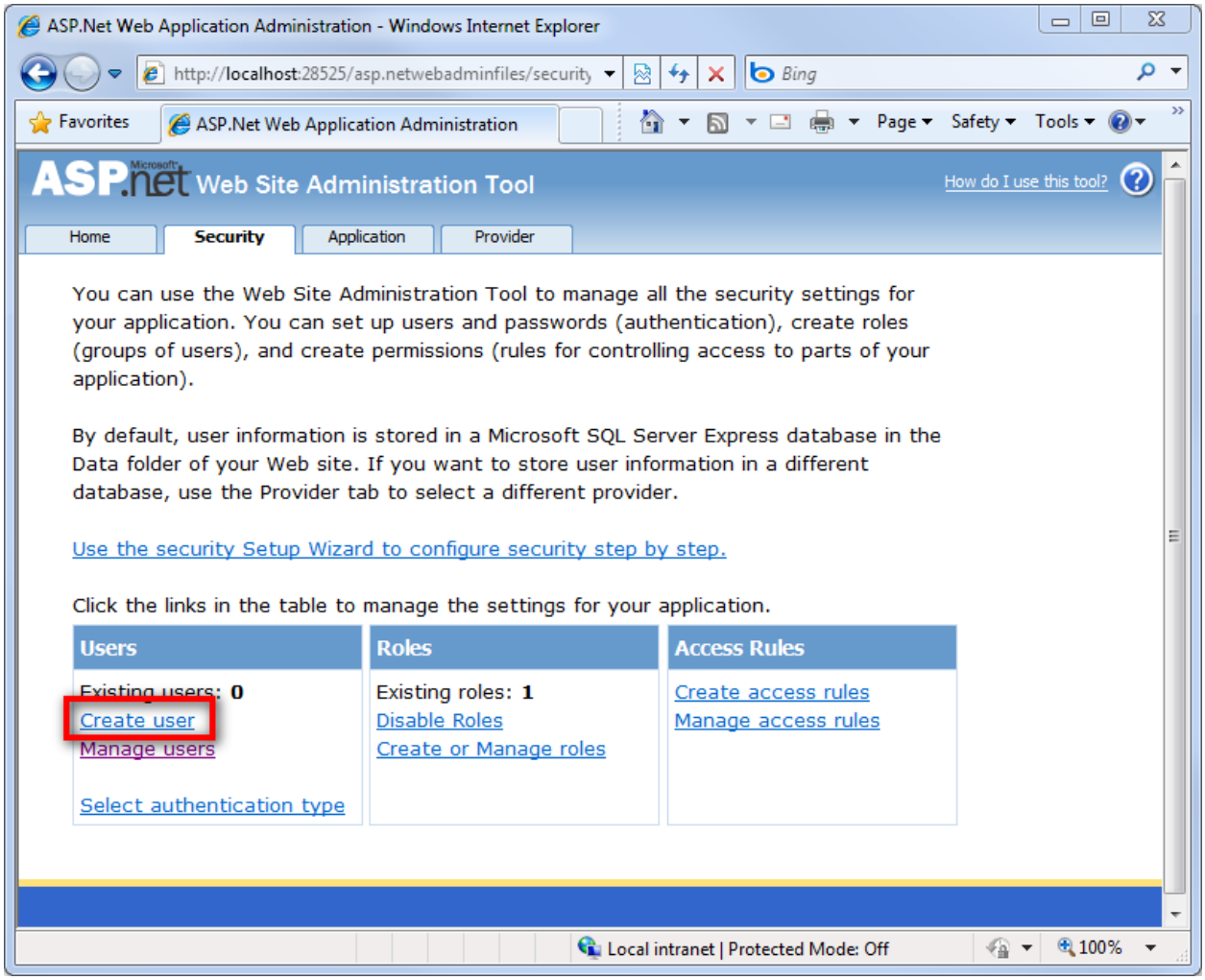
Click the “Create or Manage roles” link.



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



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

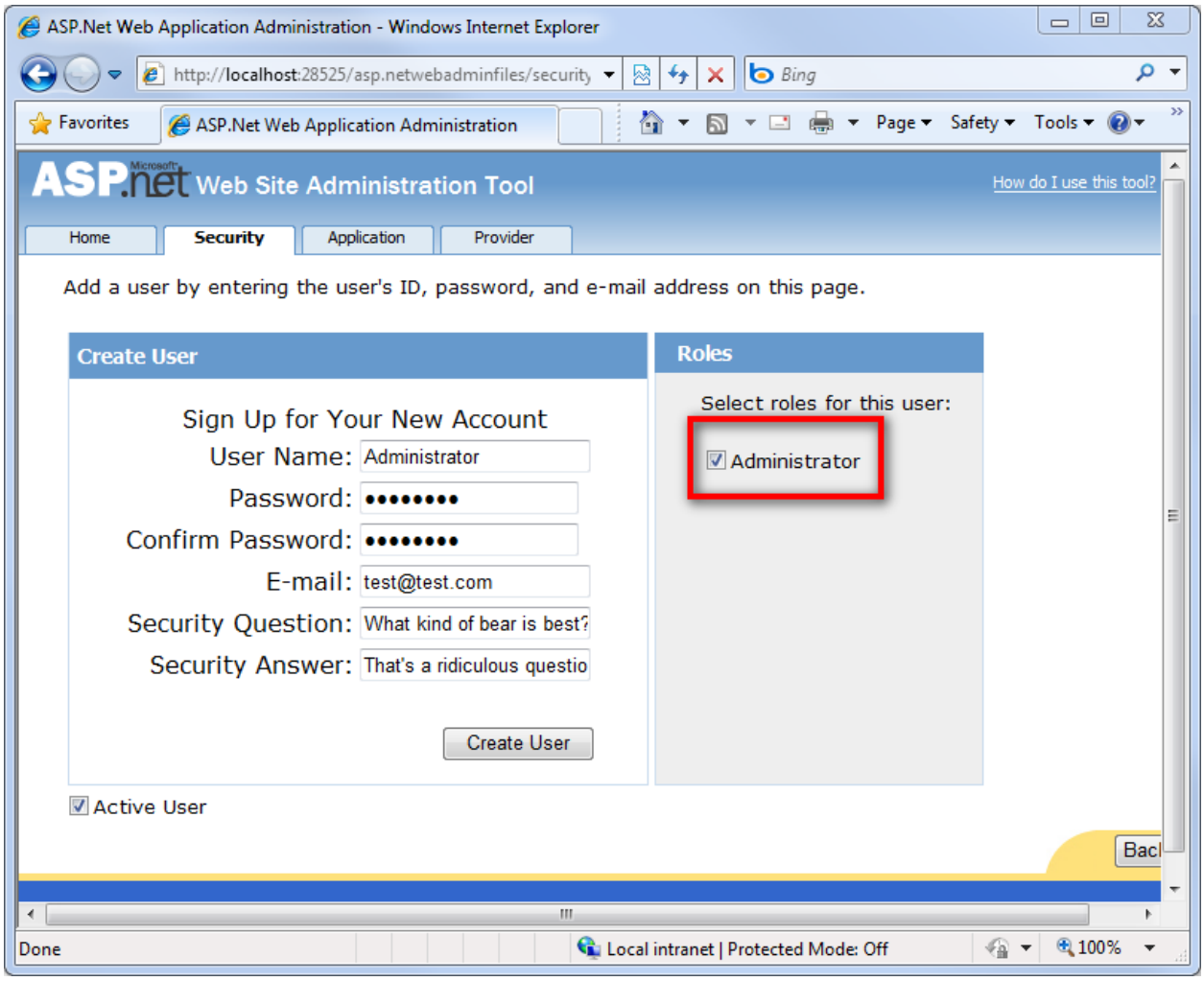


Fill in the user information fields on the left using the following information:

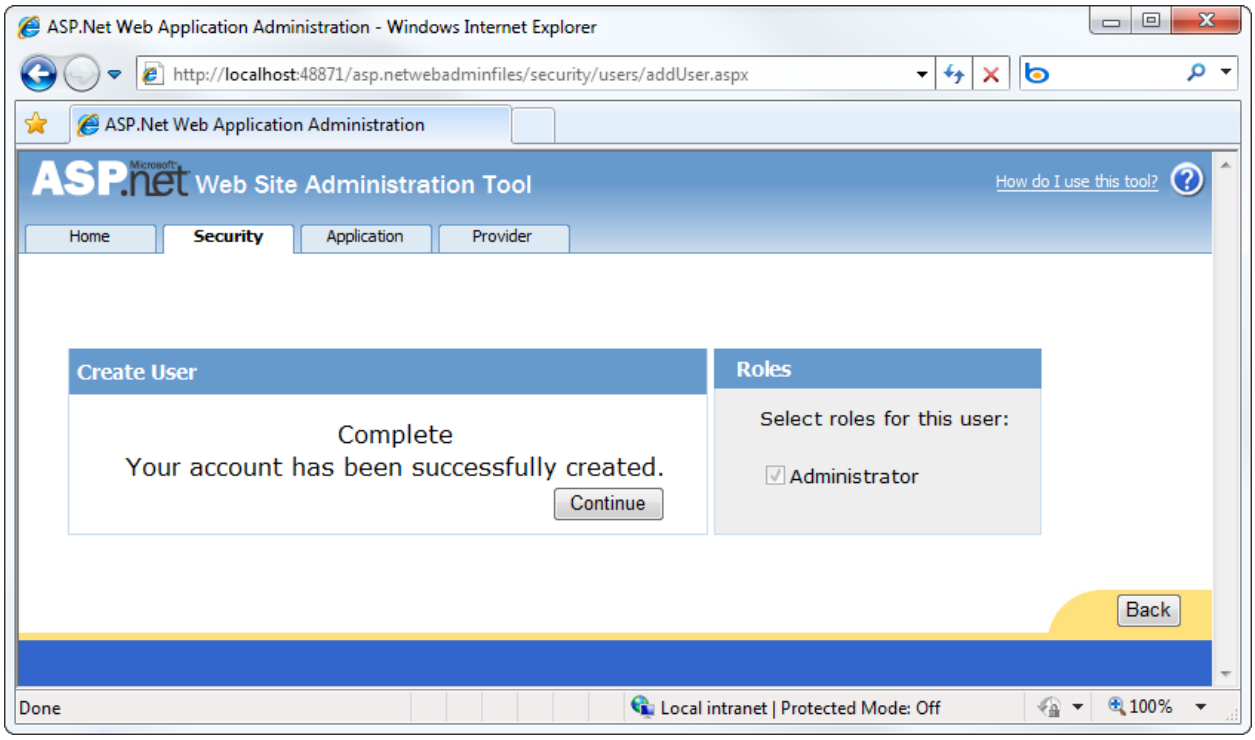
Field	Value
User Name	Administrator
Password	password123!
Confirm Password	password123!
E-mail	(any e-mail address will work)
Security Question	(whatever you like)
Security Answer	(whatever you like)

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

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



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



You can now close the browser window.

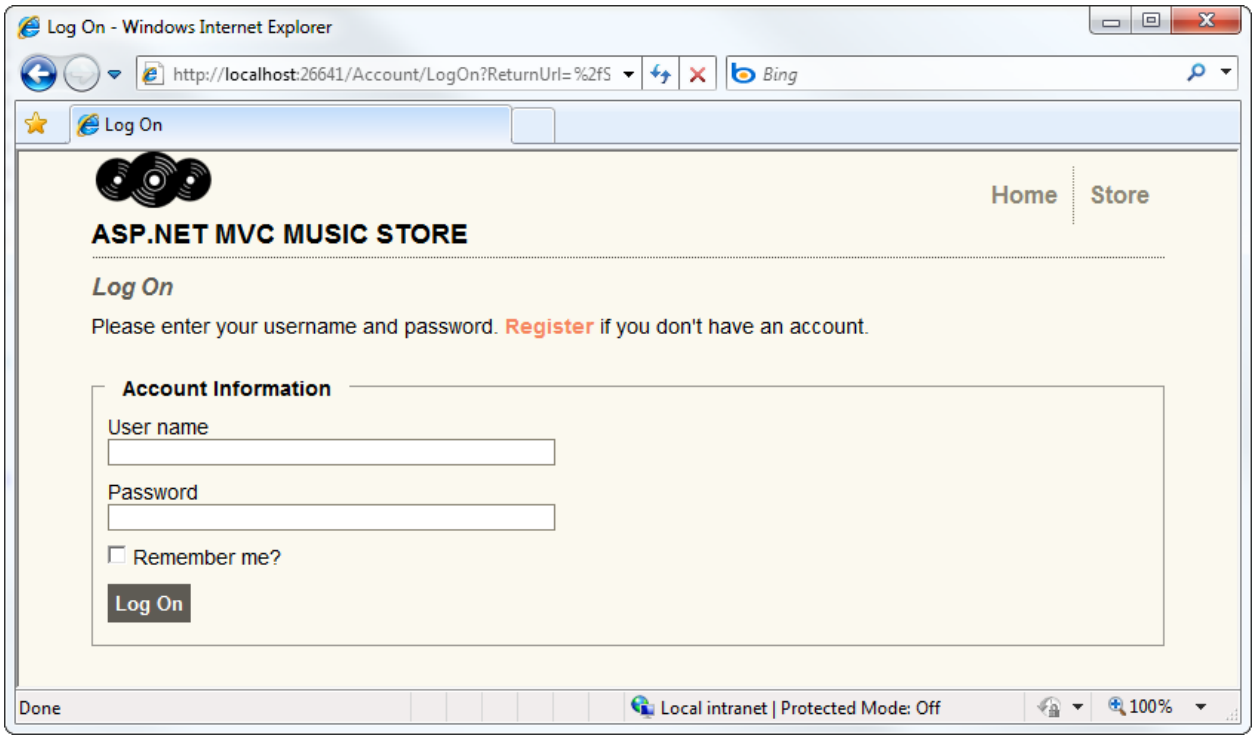
Role-based Authorization

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

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

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

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



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

8. Shopping Cart with Ajax Updates

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

First 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.

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; }
    }
}
```

Next, 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 DeleteId { get; set; }
    }
}
```

Managing the Shopping Cart business logic

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

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

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

The ShoppingCart class exposes the following methods:

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

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

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

GetCartItems retrieves a list of CartItems for display or processing.

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

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

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

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

Here's the complete ShoppingCart class:

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

namespace MvcMusicStore.Models
{
    public partial class ShoppingCart
    {
        MusicStoreEntities storeDB = new MusicStoreEntities();
        string shoppingCartId { get; set; }
        public const string CartSessionKey = "CartId";

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

        public void AddToCart(Album album)
        {
            var cartItem = storeDB.Carts.SingleOrDefault(
```

```
            c => c.CartId == shoppingCartId &&
            c.AlbumId == album.AlbumId);

        if (cartItem == null)
        {
            // Create a new cart item
            cartItem = new Cart
            {
                AlbumId = album.AlbumId,
                CartId = shoppingCartId,
                Count = 1,
                DateCreated = DateTime.Now
            };
            storeDB.AddToCarts(cartItem);
        }
        else
        {
            // Add one to the quantity
            cartItem.Count++;
        }

        // Save it
        storeDB.SaveChanges();
    }

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

        if (cartItem != null)
        {
            if (cartItem.Count > 1)
            {
                cartItem.Count--;
            }
            else
            {
                storeDB.Carts.DeleteObject(cartItem);
            }
            storeDB.SaveChanges();
        }
    }

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

        foreach (var cartItem in cartItems)
        {
            storeDB.DeleteObject(cartItem);
        }
```

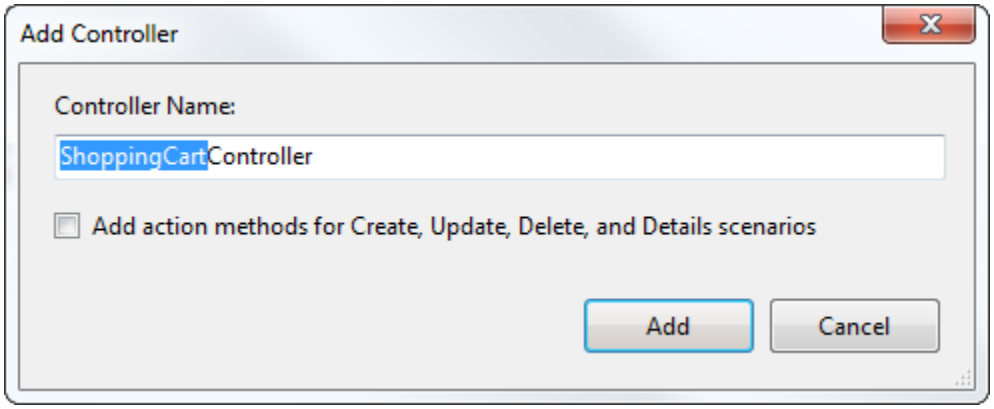
```
    storeDB.SaveChanges();
}

public List<Cart> GetCartItems()
{
    var cartItems = (from cart in storeDB.Carts
                     where cart.CartId == shoppingCartId
                     select cart).ToList();
    return cartItems;
}

public int GetCount()
{
    int? count = (from cartItems in storeDB.Carts
                  where cartItems.CartId == shoppingCartId
                  select (int?)cartItems.Count).Sum();

    return count ?? 0;
}

public decimal GetTotal()
{
    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 the items in the cart, adding Order Details for each
    foreach (var cartItem in cartItems)
    {
        var orderDetails = new OrderDetail
        {
            AlbumId = cartItem.AlbumId,
            OrderId = order.OrderId,
            UnitPrice = cartItem.Album.Price
        };

        storeDB.OrderDetails.AddObject(orderDetails);

        orderTotal += (cartItem.Count * cartItem.Album.Price);
    }

    //Save the order
    storeDB.SaveChanges();
```

```
            //Empty the shopping cart
            EmptyCart();

            //Return the OrderId as a 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))
                {
                    // User is logged in, associate the cart with there username
                    context.Session[CartSessionKey] = context.User.Identity.Name;
                }
                else
                {
                    // Generate a new random GUID using System.Guid Class
                    Guid tempCartId = Guid.NewGuid();

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

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

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

The Shopping Cart Controller

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

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



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

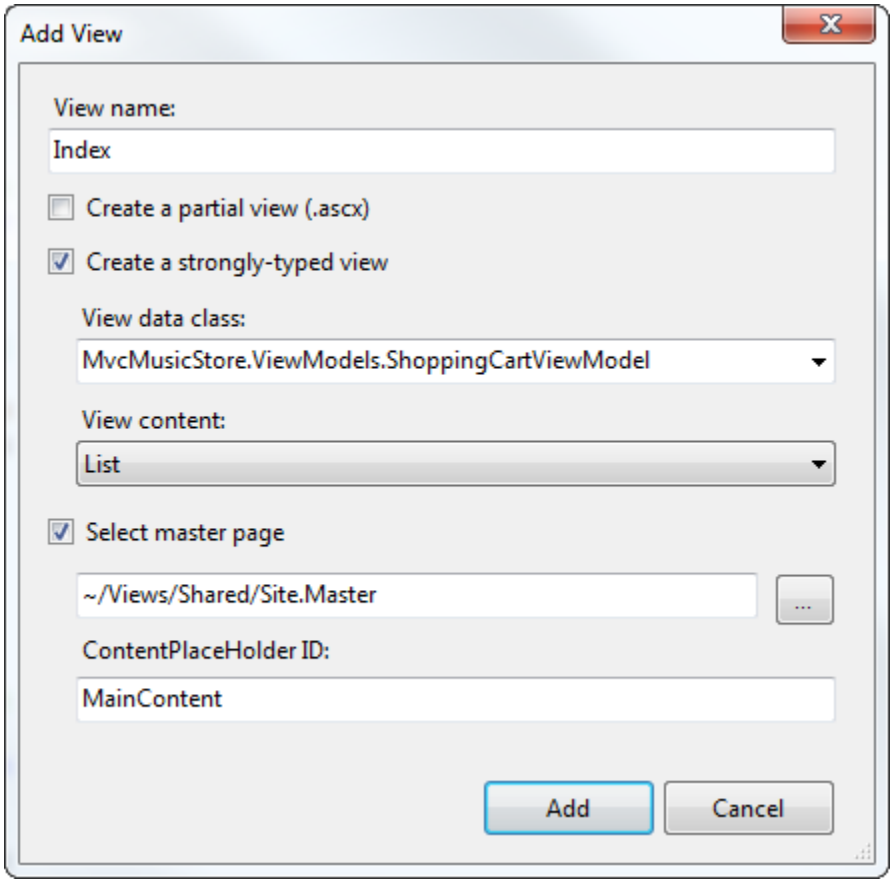
```
using System.Linq;
using System.Web.Mvc;
using MvcMusicStore.Models;
using MvcMusicStore.ViewModels;

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

        //
        // GET: /ShoppingCart/

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

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

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

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

        public ActionResult AddToCart(int id)
```

```
{

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

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

    cart.AddToCart(addedAlbum);

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

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

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

    // Get the name of the album to display confirmation
    string albumName = storeDB.Carts
        .Single(item => item.RecordId == id).Album.Title;

    // Remove from cart. Note that for simplicity, we're
    // removing all rather than decrementing the count.
    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(),
        DeleteId = id
    };

    return Json(results);
}

//
// GET: /ShoppingCart/CartSummary

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

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

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

Ajax Updates using Ajax.ActionLink

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



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

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

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

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

4. Updates the cart total price

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

```
<%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage<MvcMusicStore.ViewModels.ShoppingCartViewModel>" %>

<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
    Shopping Cart
</asp:Content>
<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

    <script src="/Scripts/MicrosoftAjax.js" type="text/javascript"></script>
    <script src="/Scripts/MicrosoftMvcAjax.js" type="text/javascript"></script>
    <script src="/Scripts/jquery-1.4.1.min.js" type="text/javascript"></script>

    <script type="text/javascript">
        function handleUpdate(context) {
            // Load and deserialize the returned JSON data
            var json = context.get_data();
            var data = Sys.Serialization.JavaScriptSerializer.deserialize(json);

            // Update the page elements
            $('#row-' + data.DeleteId).fadeOut('slow');
            $('#cart-status').text('Cart (' + data.CartCount + ')');
            $('#update-message').text(data.Message);
            $('#cart-total').text(data.CartTotal);
        }
    </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",
```

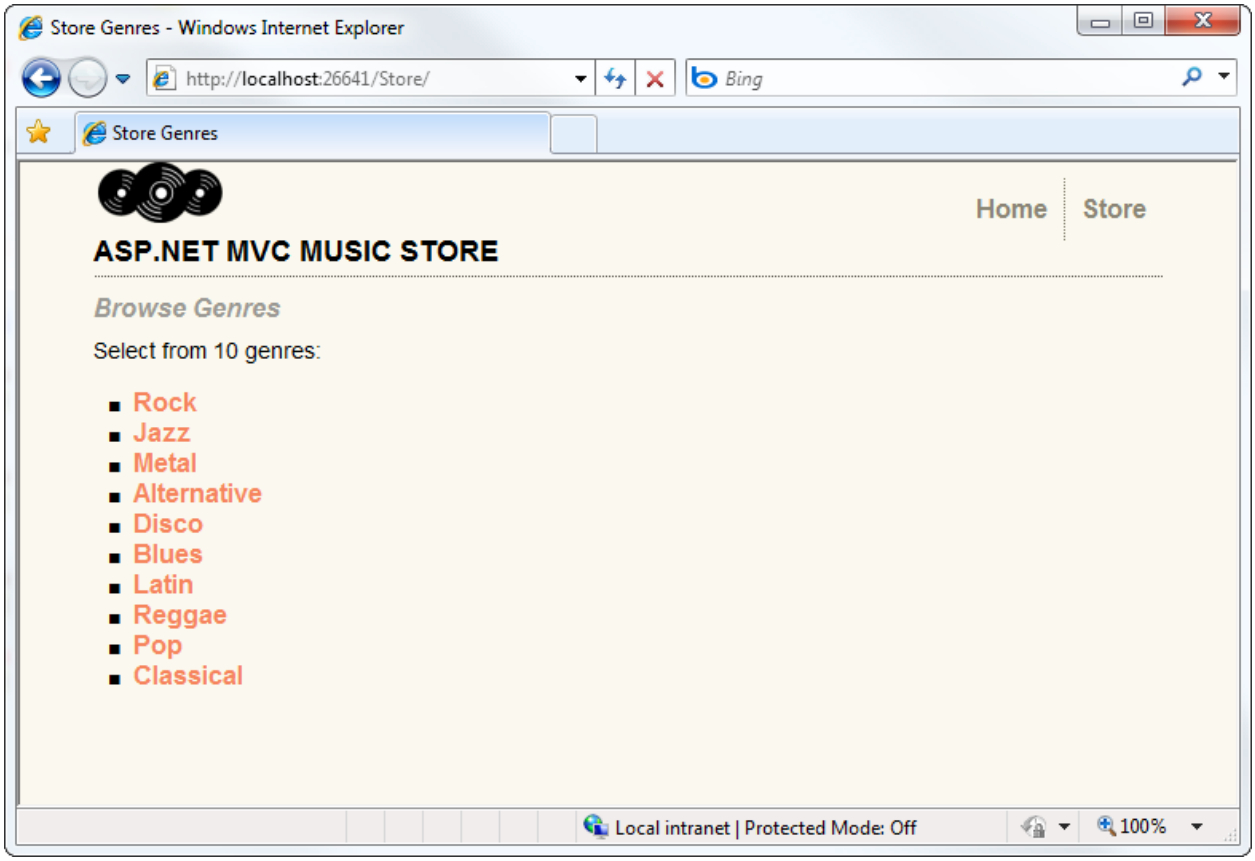
```
                    new { id = item.AlbumId }, null)%>
            </td>
            <td>
                <%: item.Album.Price %>
            </td>
            <td>
                <%: item.Count %>
            </td>
            <td>
                <%: Ajax.ActionLink("Remove from cart", "RemoveFromCart",
                    new { id = item.RecordId },
                    new AjaxOptions { OnSuccess = "handleUpdate" })%>
            </td>
        </tr>
        <% } %>

        <tr>
            <td>Total</td>
            <td></td>
            <td></td>
            <td id="cart-total">
                <%: Model.CartTotal %>
            </td>
        </tr>

    </table>

</asp:Content>
```

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

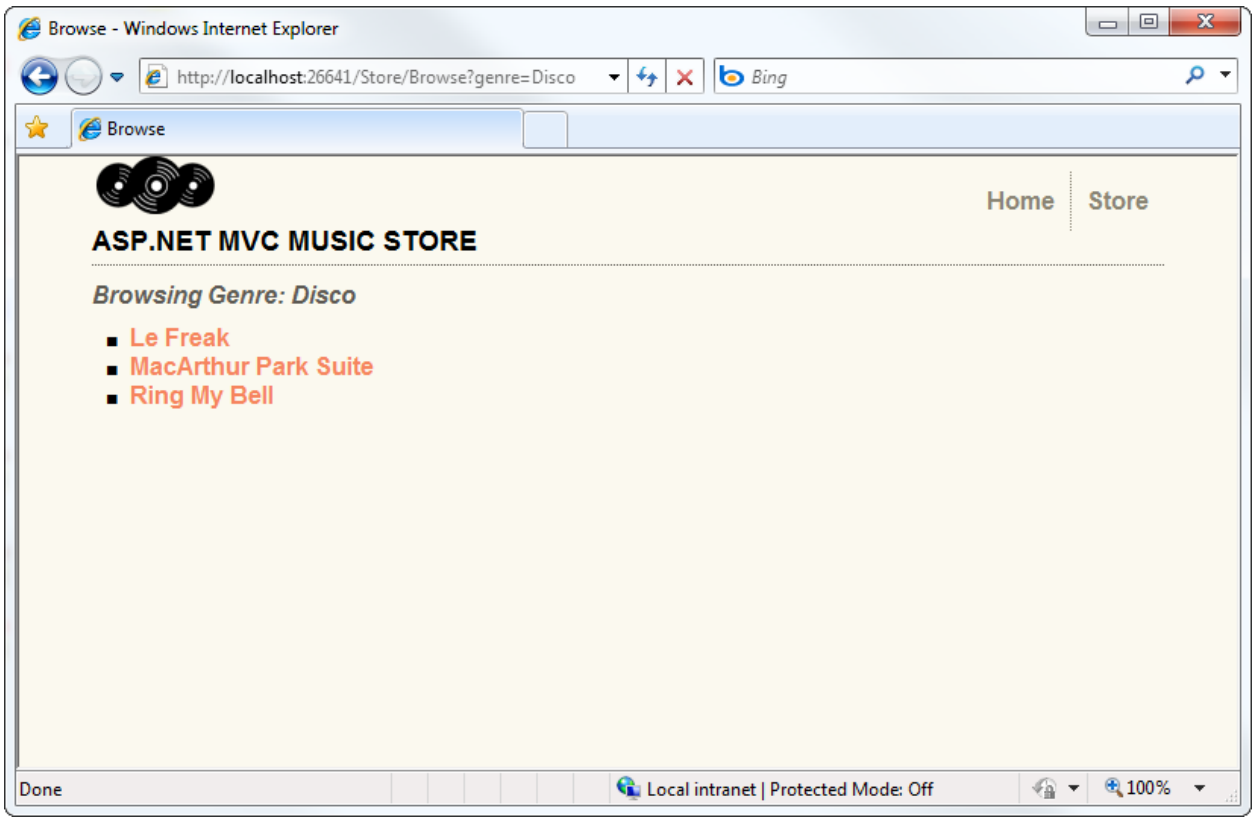
```
<%@ Page Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage<MvcMusicStore.Models.Album>" %>

<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
    Album - <%: Model.Title %>
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

    <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>

</asp:Content>
```

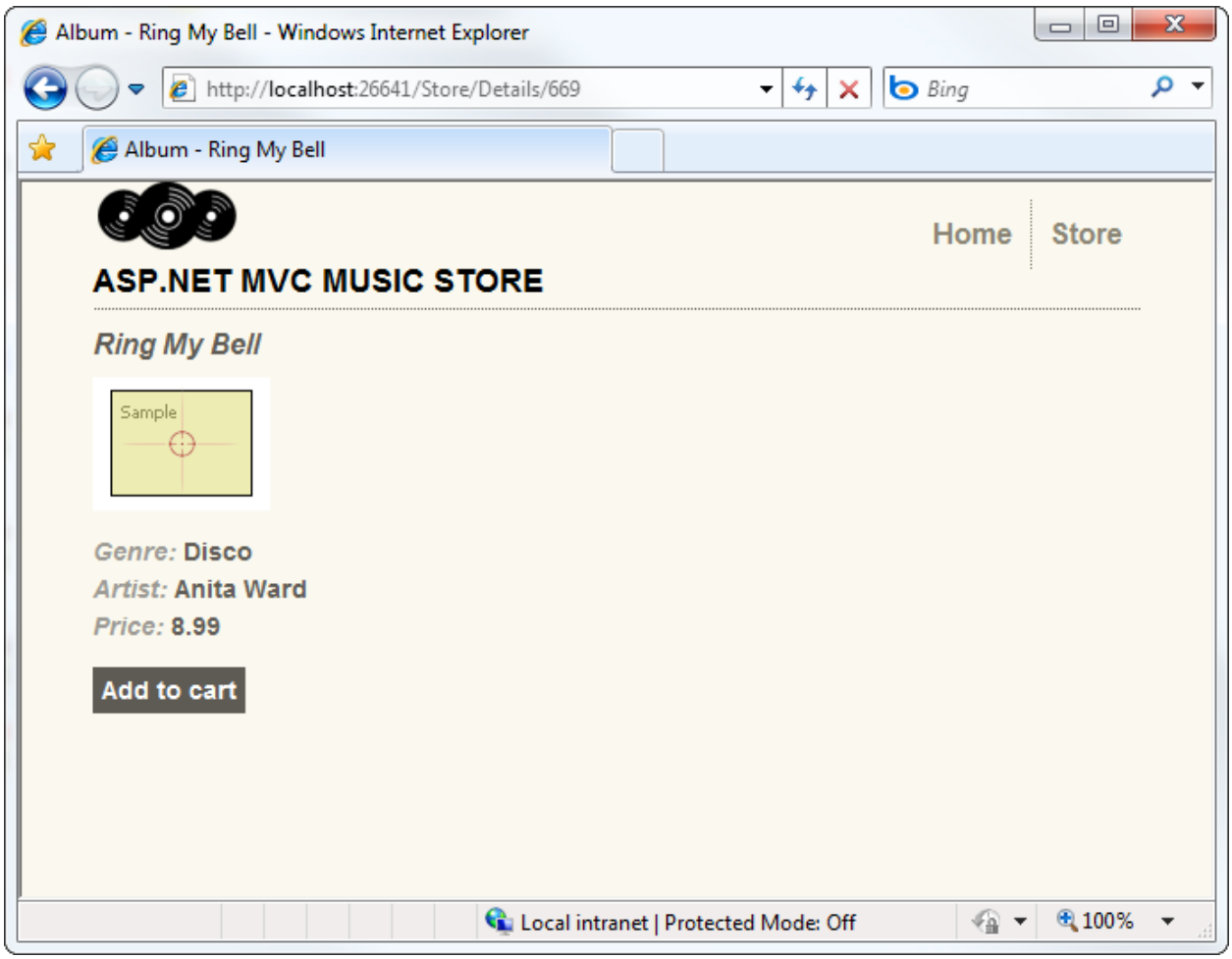
Now we can click through the store and test adding and removing Albums to and from our shopping cart. Run the application and browse to the Store Index.



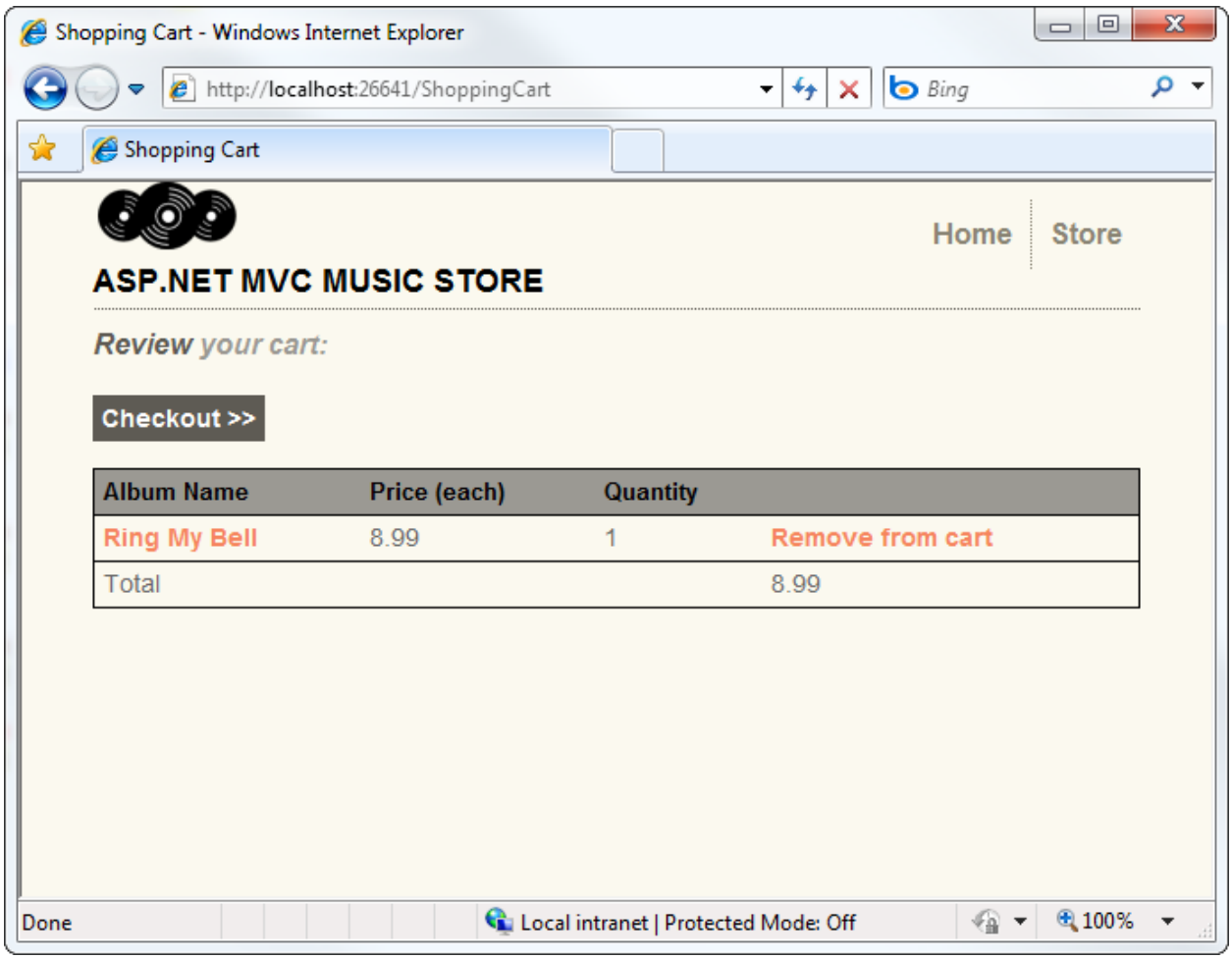
Next, click on a Genre to view a list of albums.



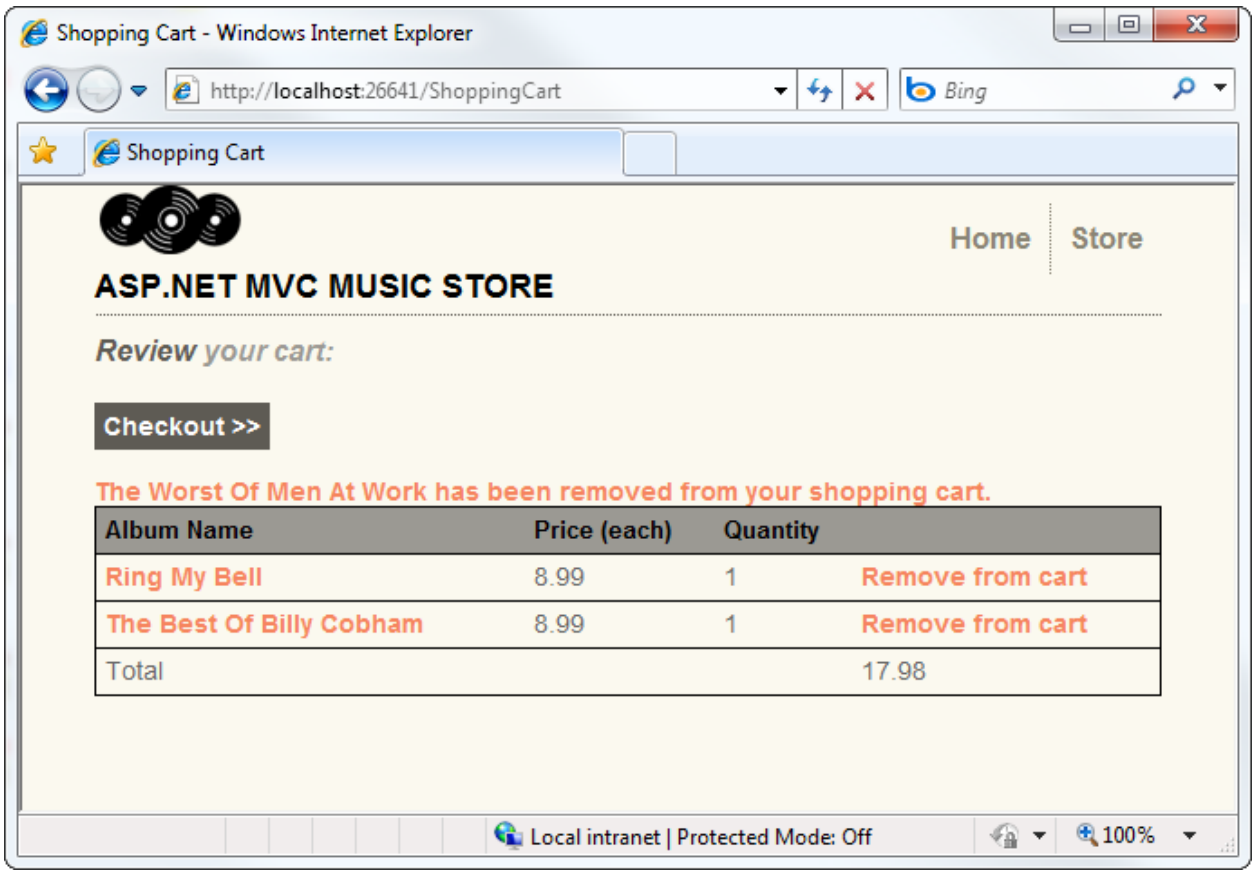
Clicking on an Album title now shows our updated Album Details view, including the "Add to cart" button.



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



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

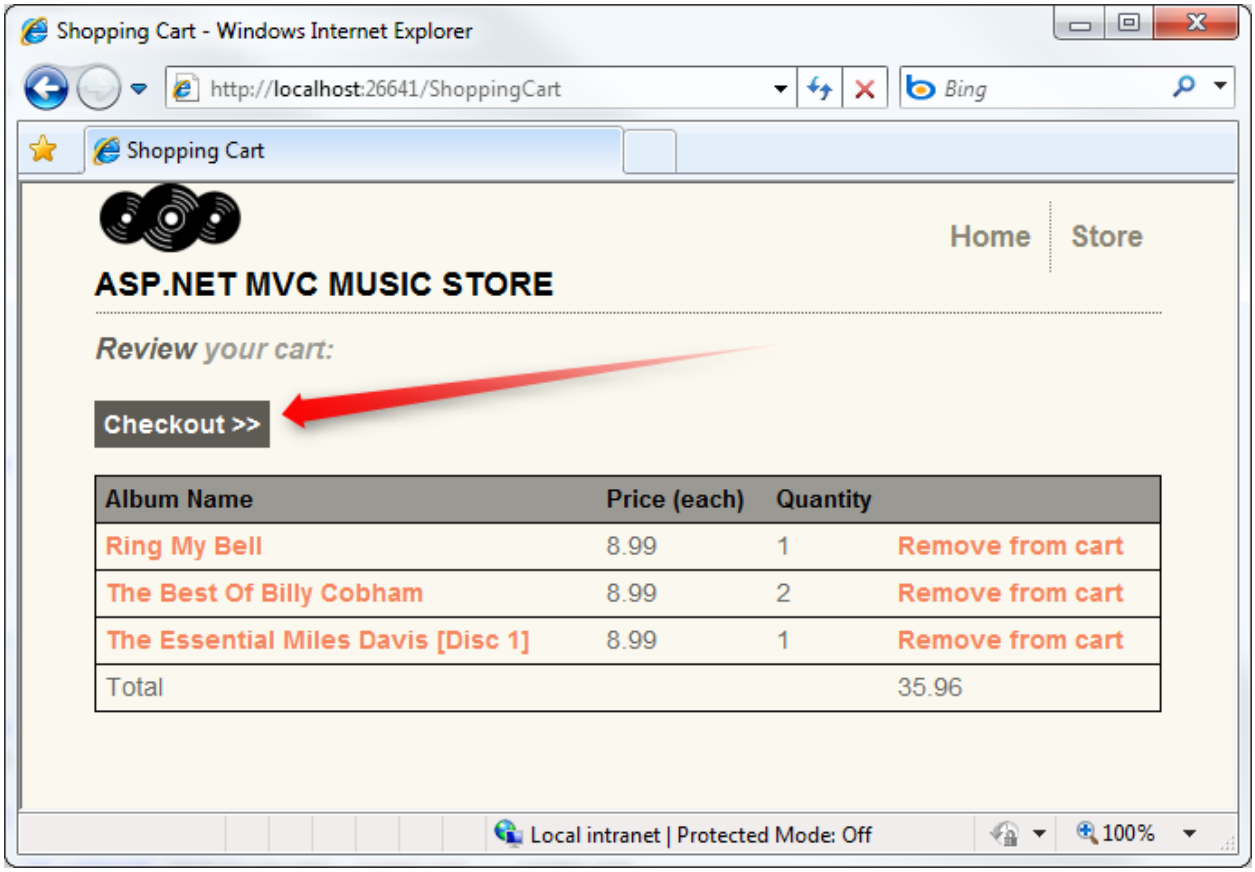


We've built out a working shopping cart which allows unregistered users to add items to their cart. In the following section, we'll allow them to register and complete the checkout process.

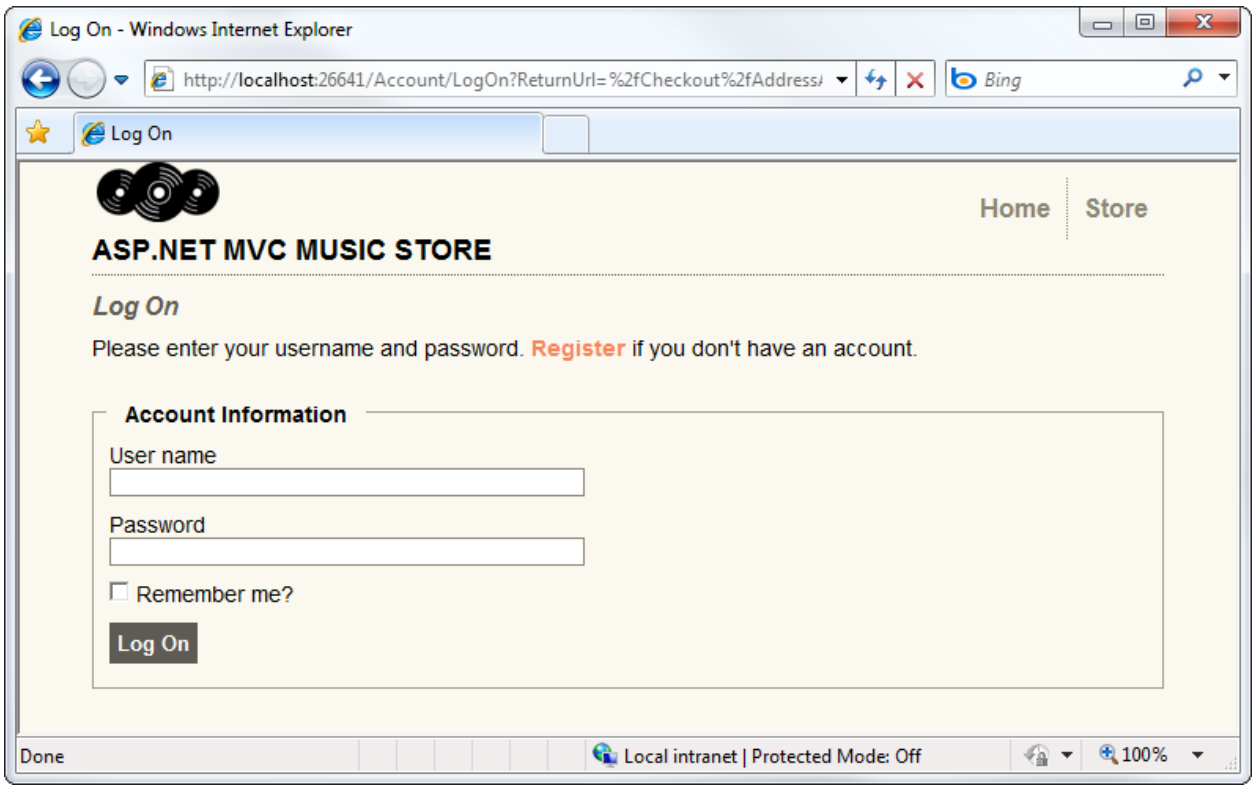
9. Registration and Checkout

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

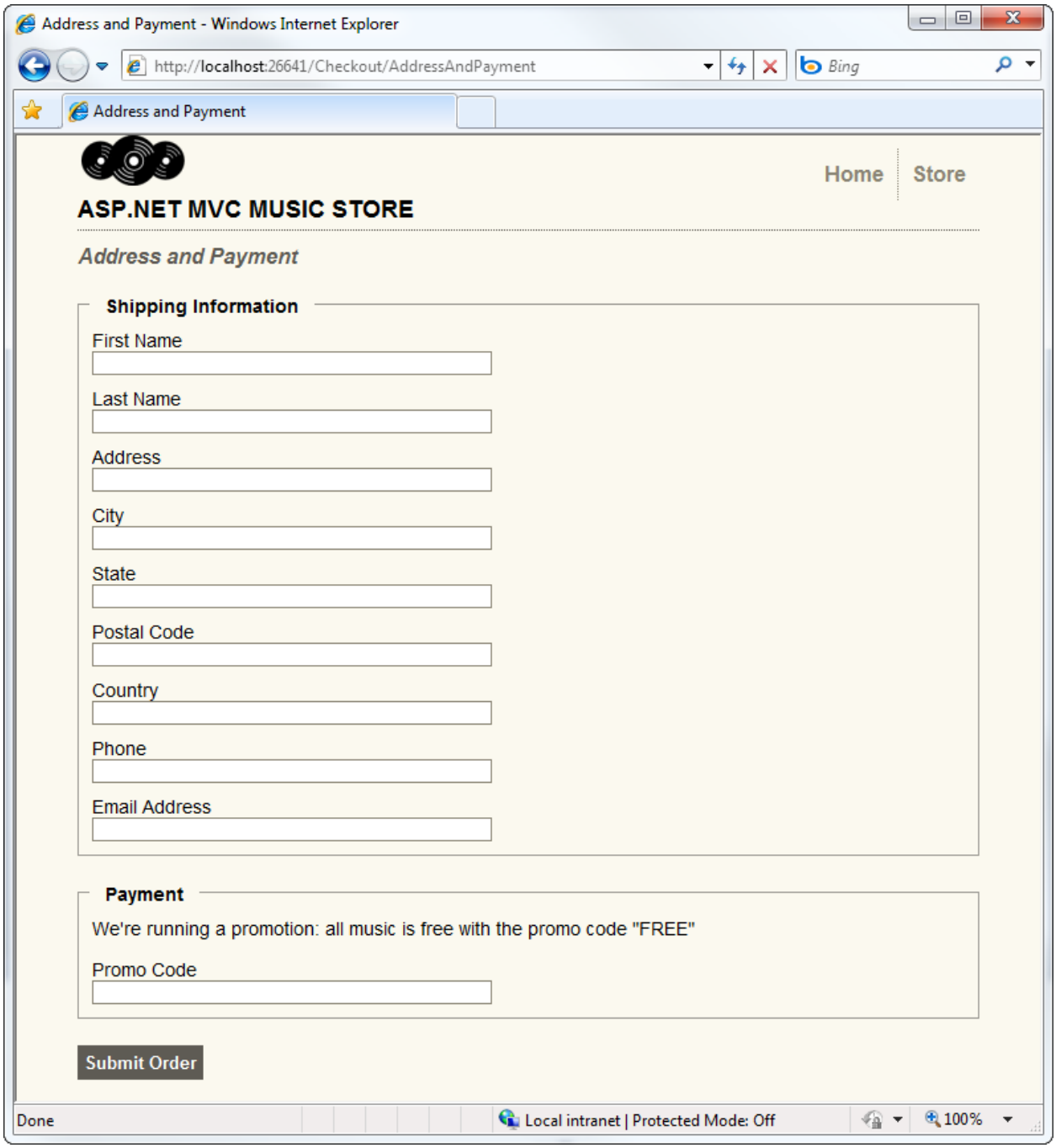
Users will navigate to the checkout process from their shopping cart by clicking the "Checkout" button.



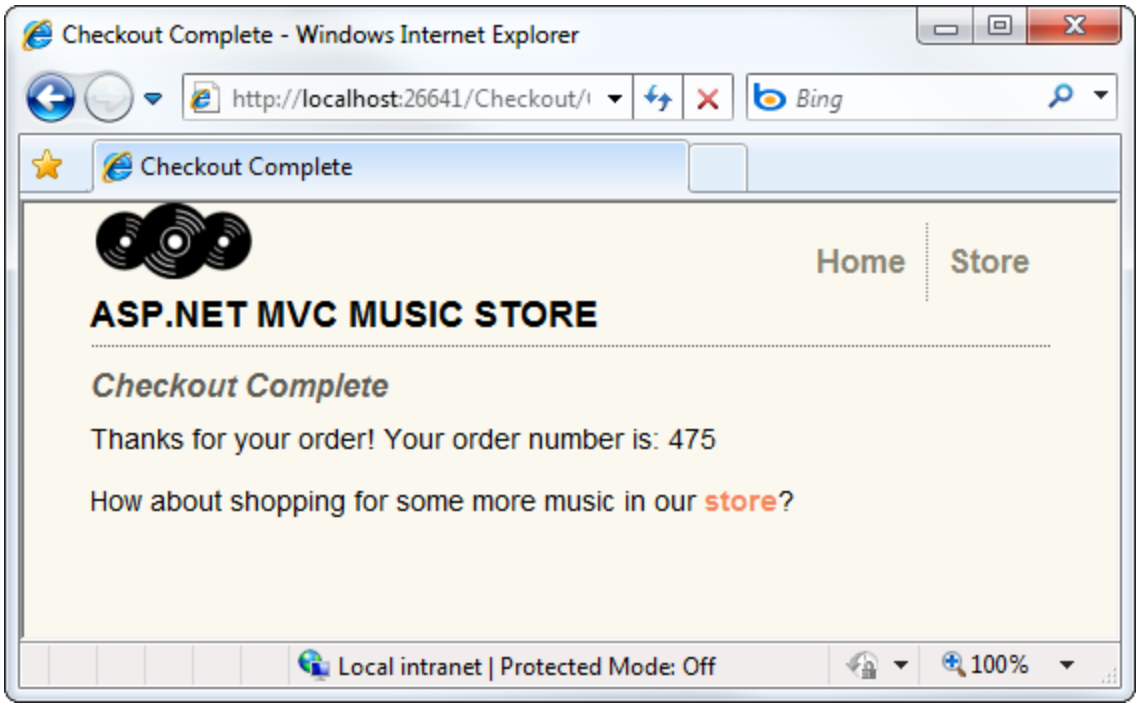
If the user is not logged in, they will be prompted to.



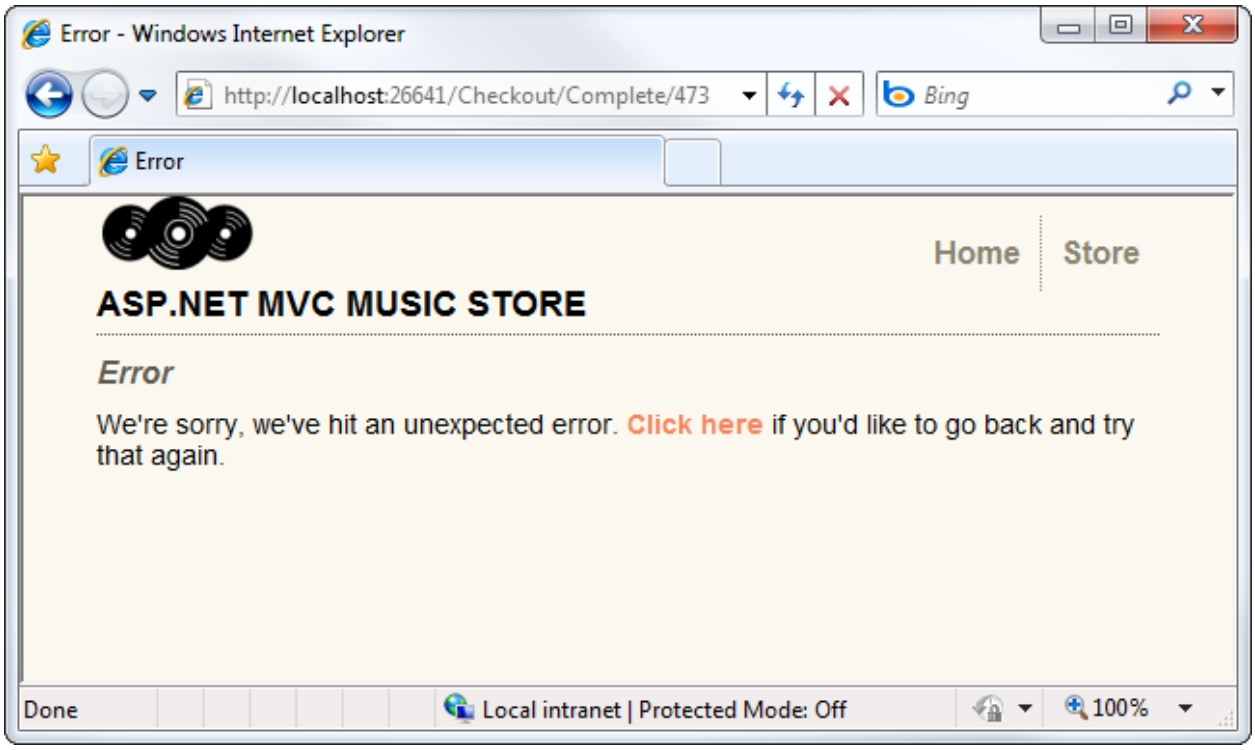
Upon successful login, the user is then shown the Address and Payment view.



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



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



Migrating the Shopping Cart

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

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

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

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

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

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

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

```
[HttpPost]
[SuppressMessage("Microsoft.Design", "CA1054:UriParametersShouldNotBeStrings",
    Justification = "Needs to take same parameter type as Controller.Redirect()")]
public ActionResult LogOn(LogOnModel model, string returnUrl)
{
    if (ModelState.IsValid)
    {
        if (MembershipService.ValidateUser(model.UserName, model.Password))
        {
            MigrateShoppingCart(model.UserName);

            FormsService.SignIn(model.UserName, model.RememberMe);

            if (!String.IsNullOrEmpty(returnUrl))
            {
                return Redirect(returnUrl);
            }
            else
            {
                return RedirectToAction("Index", "Home");
            }
        }
        else
        {
            ModelState.AddModelError("",
            "The user name or password provided is incorrect.");
        }
    }

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

```
}
```

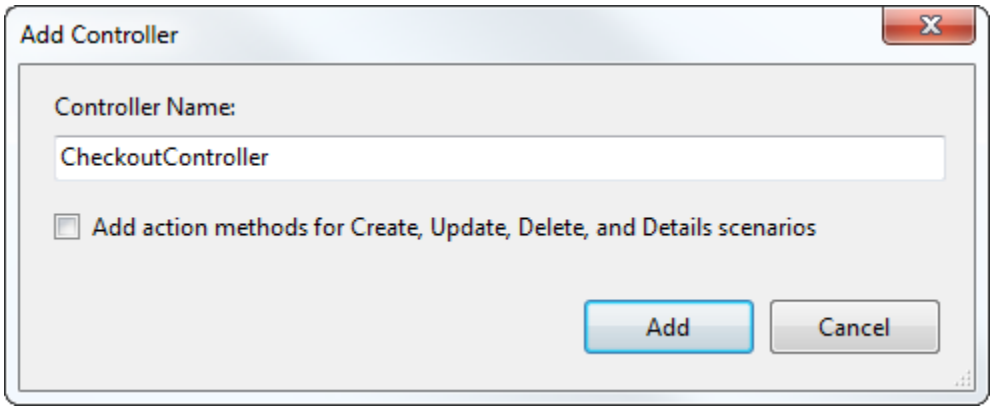
Make the same change to the Register post action, immediately after the user account is successfully created:

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

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

            FormsService.SignIn(model.UserName, false /* createPersistentCookie */);
            return RedirectToAction("Index", "Home");
        }
        else
        {
            ModelState.AddModelError("", 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. Do not select the checkbox to add Create, Update, Delete, or Details controller actions.



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

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

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

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

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

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

The CheckoutController will have the following controller actions:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Note: We haven't created the Error view yet. We will be adding that later in this section.

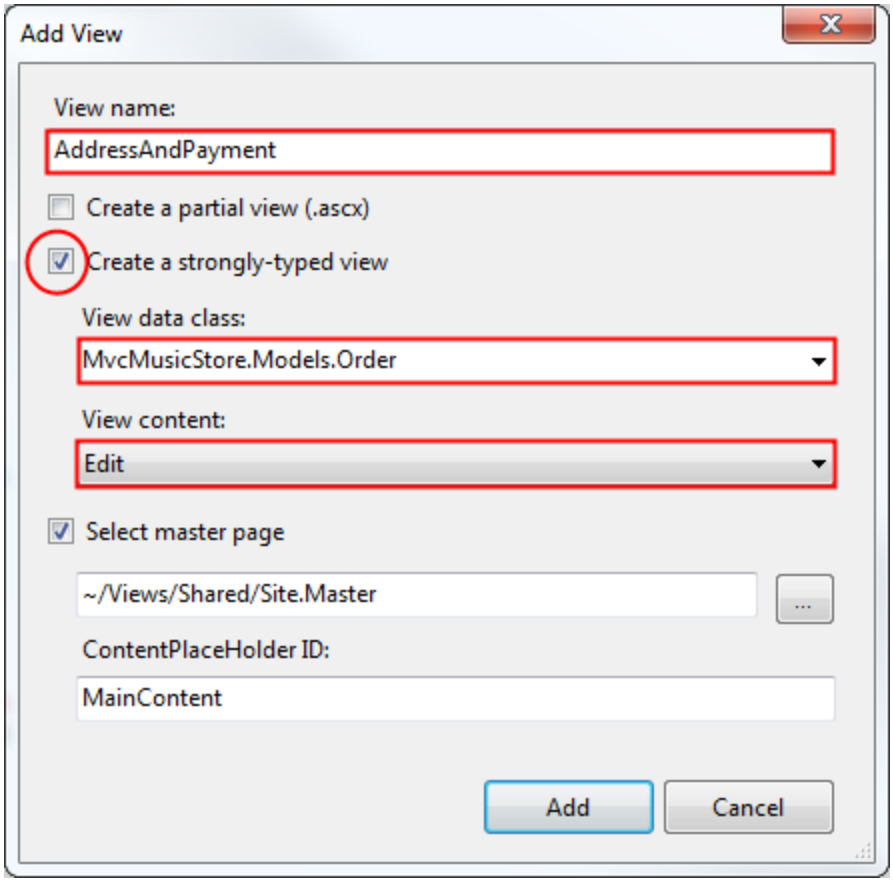
The complete CheckoutController code is as follows:

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

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

        //
        // GET: /Checkout/AddressAndPayment

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

        //
        // POST: /Checkout/AddressAndPayment

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

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

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

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

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

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

        //
        // GET: /Checkout/Complete

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

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

Adding the AddressAndPayment view

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



This view will make use of three 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 make use of client-side validation
- We will leverage validation rules using an Order class with validation attributes

First, we will add the client-side validation scripts and the call to Html.EnableClientValidation to the top, as shown:

```
<%@ Page Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage<MvcMusicStore.Models.Order>" %>

<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
    Address and Payment
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
```

```
    <script src="/Scripts/MicrosoftAjax.js" type="text/javascript"></script>
    <script src="/Scripts/MicrosoftMvcAjax.js" type="text/javascript"></script>
    <script src="/Scripts/MicrosoftMvcValidation.js" type="text/javascript"></script>
    <script src="/Scripts/jquery-1.4.1.min.js" type="text/javascript"></script>

    <% Html.EnableClientValidation(); %>
    <% using (Html.BeginForm()) {%>
```

Next, we'll update 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.

```
<%@ Page Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage<MvcMusicStore.Models.Order>" %>

<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
    Address and Payment
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

    <script src="/Scripts/MicrosoftAjax.js" type="text/javascript"></script>
    <script src="/Scripts/MicrosoftMvcAjax.js" type="text/javascript"></script>
    <script src="/Scripts/MicrosoftMvcValidation.js" type="text/javascript"></script>
    <script src="/Scripts/jquery-1.4.1.min.js" type="text/javascript"></script>

    <% Html.EnableClientValidation(); %>
    <% 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" />
    <% } %>
```

```
</asp:Content>
```

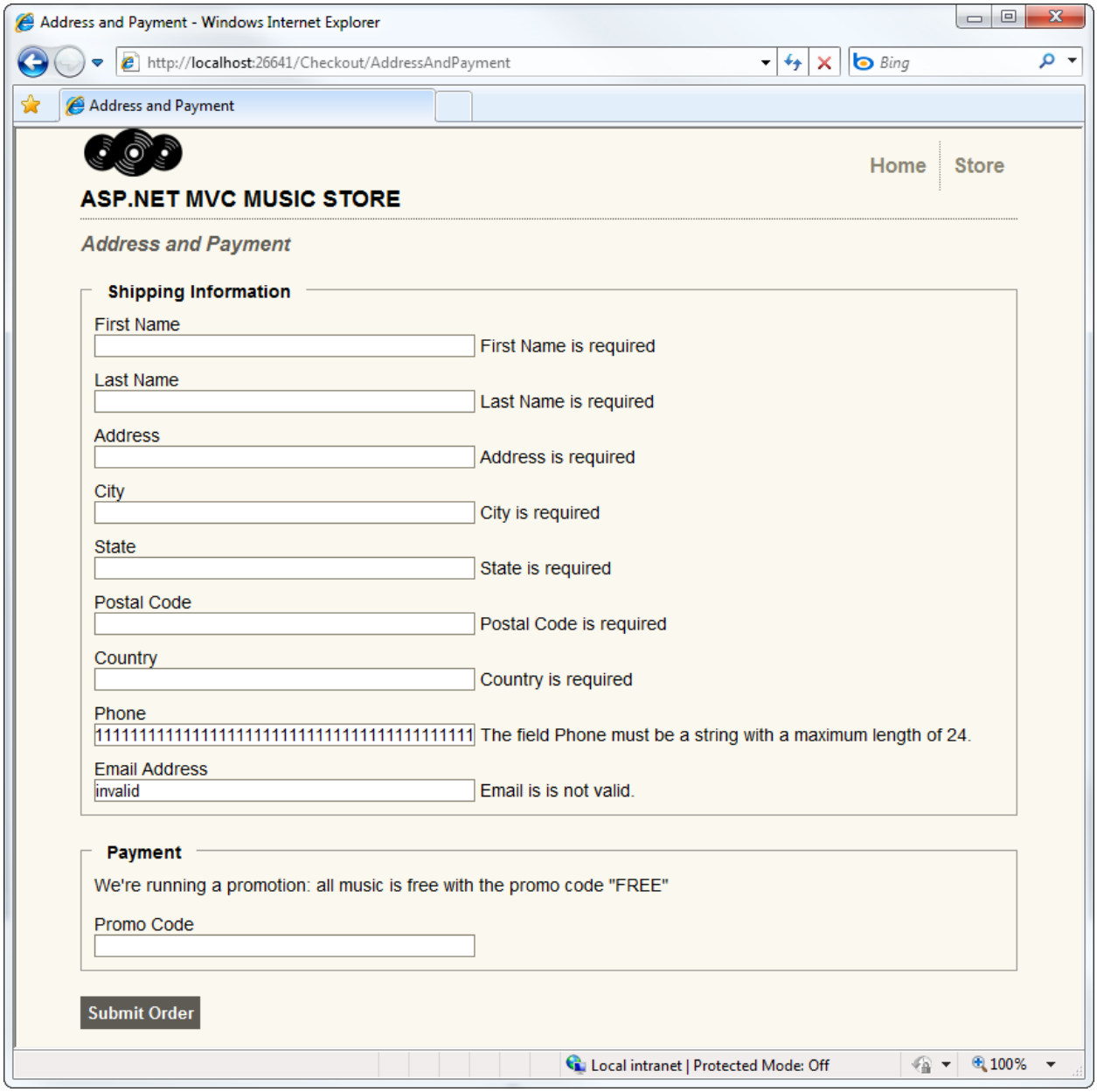
Defining validation rules for the Order

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

namespace MvcMusicStore.Models
{
    [MetadataType(typeof(OrderMetadata))]
    public partial class Order
    {
        // Validation rules for the Order class

        [Bind(Exclude = "OrderId")]
        public class OrderMetadata
        {
            [Required(ErrorMessage = "First Name is required")]
            [DisplayName("First Name")]
            [StringLength(160)]
            public object FirstName { get; set; }

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

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

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

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

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

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

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

            [ScaffoldColumn(false)]
            public object OrderId { get; set; }

            [ScaffoldColumn(false)]
            public object OrderDate { get; set; }

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

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

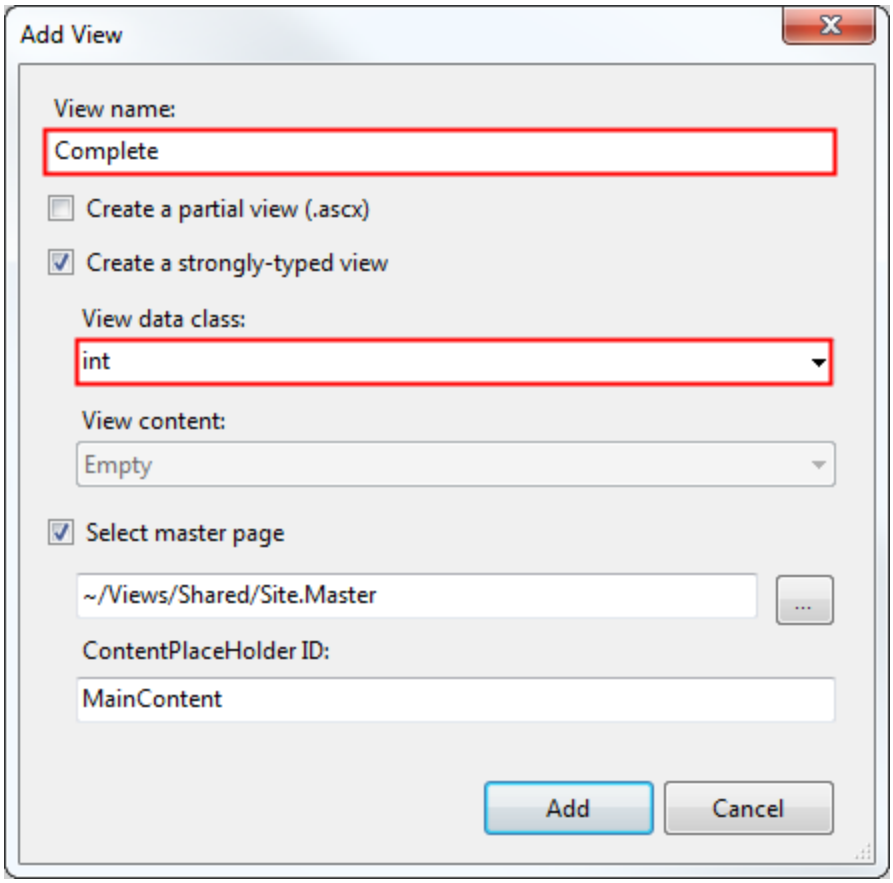
Attempting to submit the form with missing or invalid information will now show error message using client-side validation.



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

Adding the Checkout Complete view

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

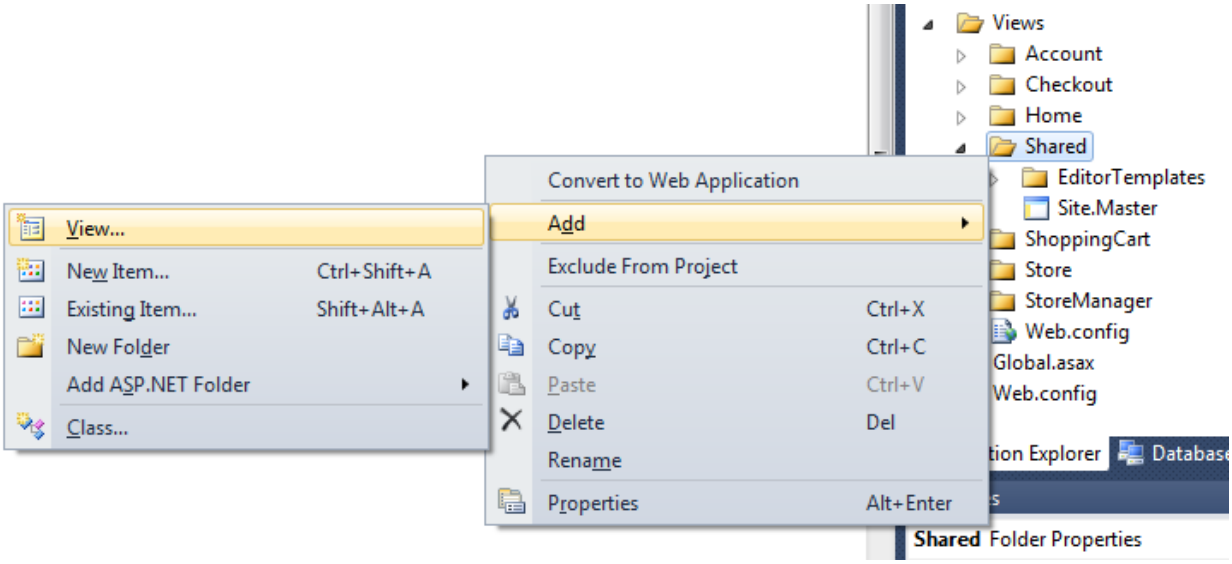


Now we will update the view code to display the Order ID, as shown below.

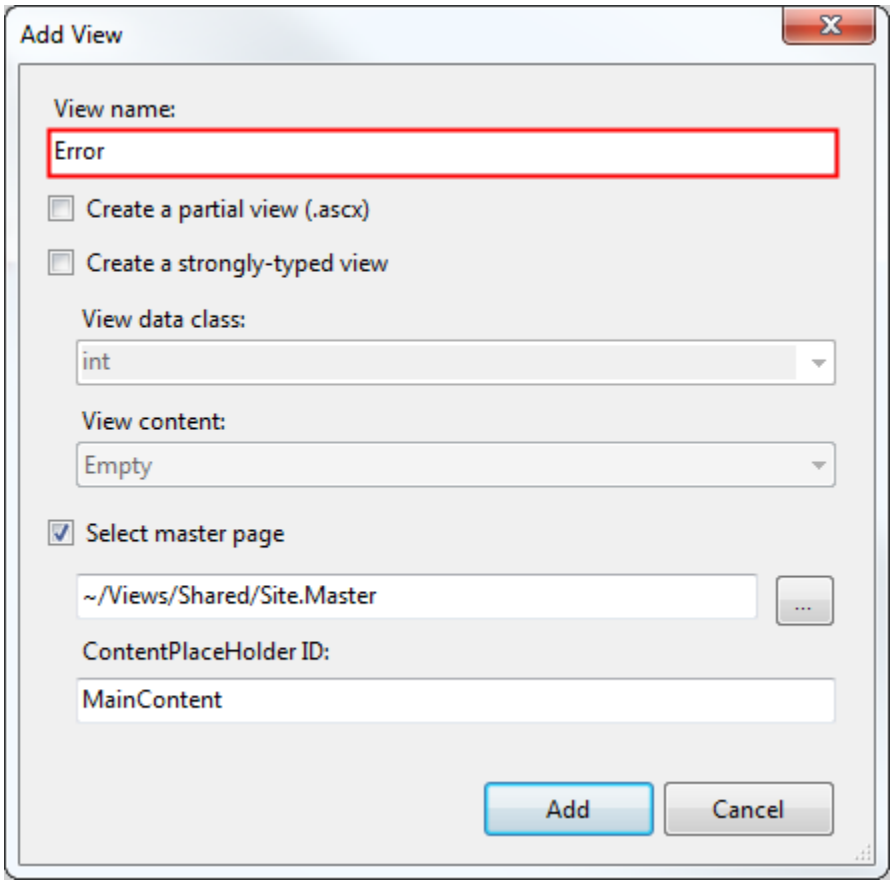
```
<%@ Page Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage<int>" %>

<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
    Checkout Complete
</asp:Content>
<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

    <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>

</asp:Content>
```

Adding The Error view

The Error view will be added to the Shared views folder so that it can be re-used elsewhere in the site. Right-click on the /Views/Shared folder and add a new view.



This view will be named Error, and will not be strongly typed.



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.

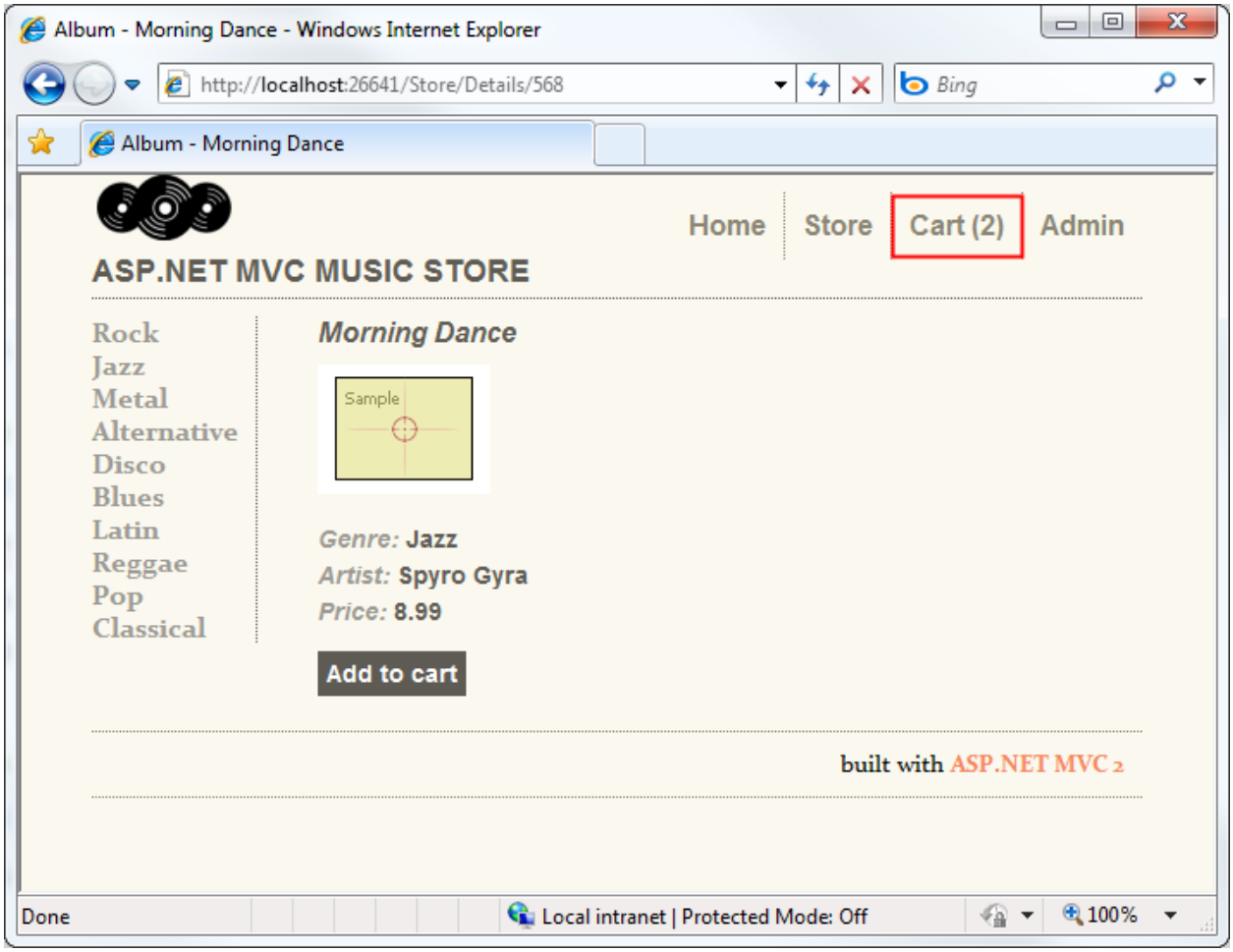
```
<%@ Page Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage" %>

<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
       Error
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

    <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>

</asp:Content>
```

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.



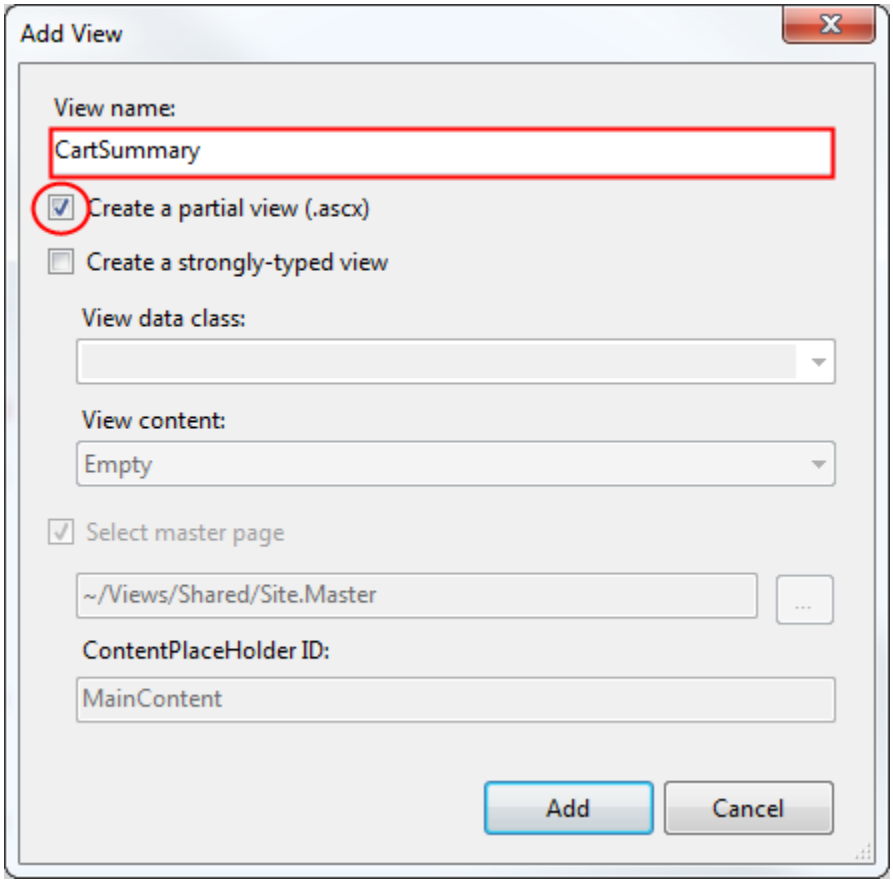
We can easily implement this by creating a partial view which is added to our Site.master.

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 (.ascx)" 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.ascx is as follows:

```
<%@ Control Language="C#" Inherits="System.Web.Mvc.ViewUserControl<dynamic>" %>

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

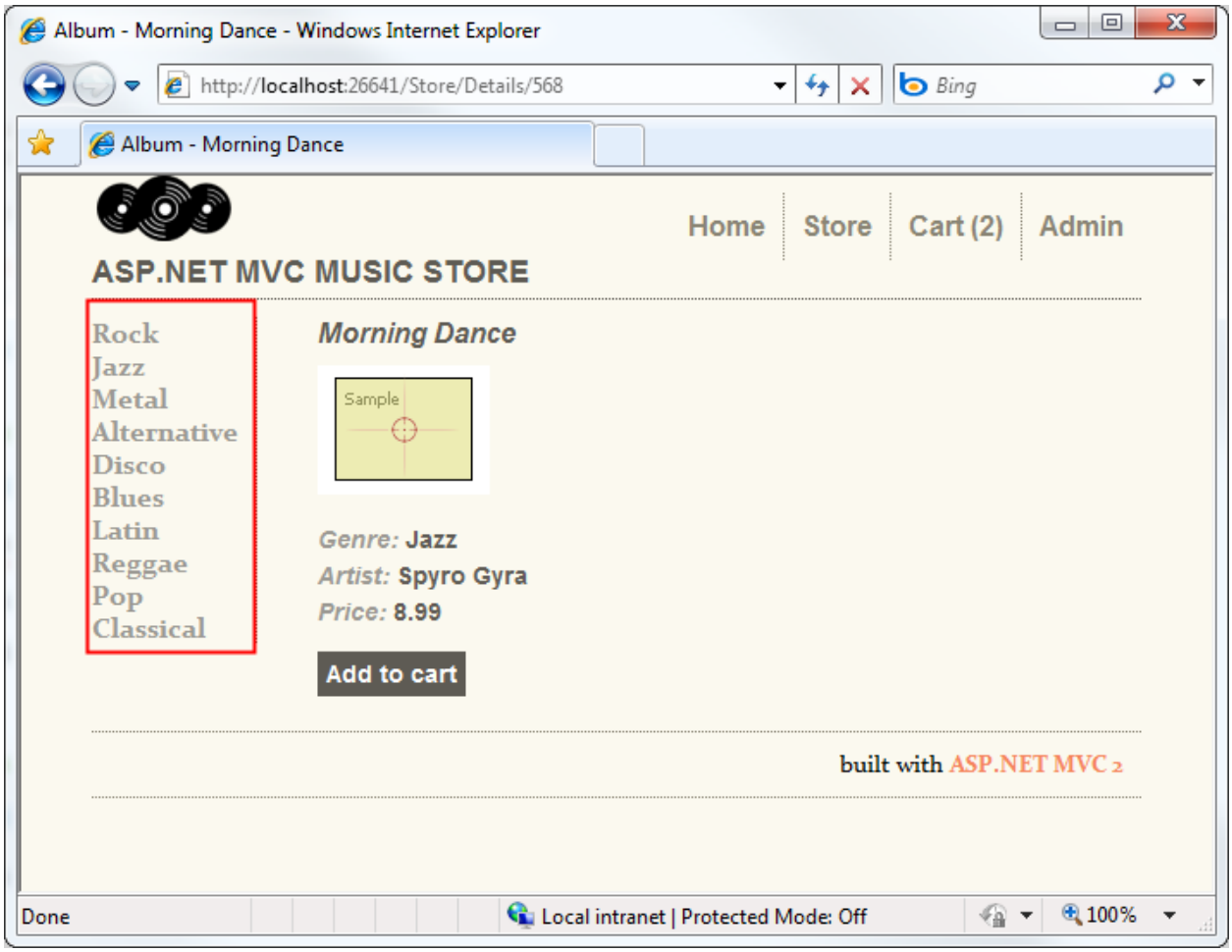
We can include a partial view in any page in the site, including the Site master, by using the Html.RenderAction method. RenderAction requires us to specify the Action Name ("CartSummary") and the Controller Name ("ShoppingCart") as below.

```
<% Html.RenderAction("CartSummary", "ShoppingCart"); %>
```

Before adding this to the Site.master, 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.



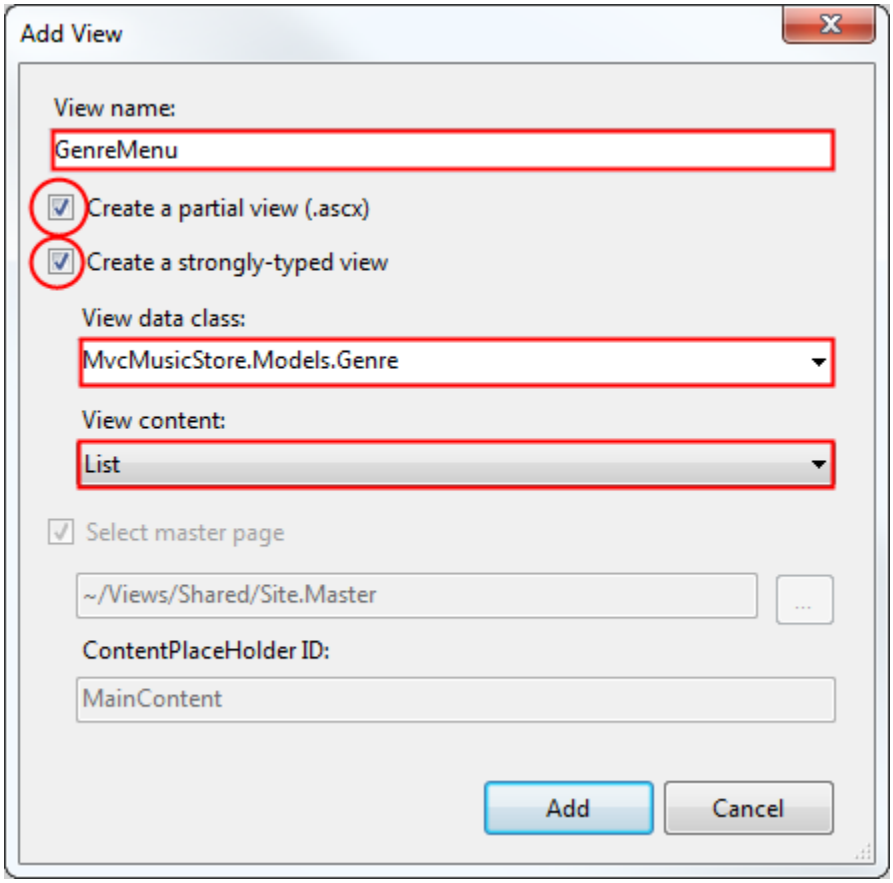
We will follow the same steps also create a GenreMenu partial view, and then we can add them both to the Site master. First, add the following GenreMenu controller action to the StoreController:

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

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

    return View(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.

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.

```
<%@ Control Language="C#"
Inherits="System.Web.Mvc.ViewUserControl<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 master to display our Partial Views

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

```
<%@ Master Language="C#" Inherits="System.Web.Mvc.ViewMasterPage" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">

    <head id="Head1" runat="server">
        <link href="/Content/Site.css" rel="stylesheet" type="text/css" />
        <title>
            <asp:ContentPlaceHolder ID="TitleContent" runat="server" />
        </title>
    </head>

    <body>
        <div id="container">

            <div id="header">
                <h1>
                    <a href="/">ASP.NET MVC MUSIC STORE</a>
                </h1>

                <ul id="navlist">
                    <li class="first"><a href="/" id="current">Home</a></li>
                    <li><a href="/Store/">Store</a></li>
                    <li>
                        <% Html.RenderAction("CartSummary", "ShoppingCart"); %>
                    </li>
                    <li><a href="/StoreManager/">Admin</a></li>
                </ul>
            </div>

            <% Html.RenderAction("GenreMenu", "Store"); %>

            <div id="main">
                <asp:ContentPlaceHolder ID="MainContent" runat="server" />
            </div>

            <div id="footer">
                built with <a href="http://asp.net/mvc">ASP.NET MVC 2</a>
            </div>

        </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.aspx) as follows:

```
<%@ Page Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
Inherits="System.Web.Mvc.ViewPage<MvcMusicStore.ViewModels.StoreBrowseViewModel>" %>

<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
    Browse Albums
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

    <div class="genre">
        <h3><em><%: Model.Genre.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>

</asp:Content>
```

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

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

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

Updating the Home Page to show Top Selling Albums

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

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

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

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

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

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

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

    return View(albums);
}
```

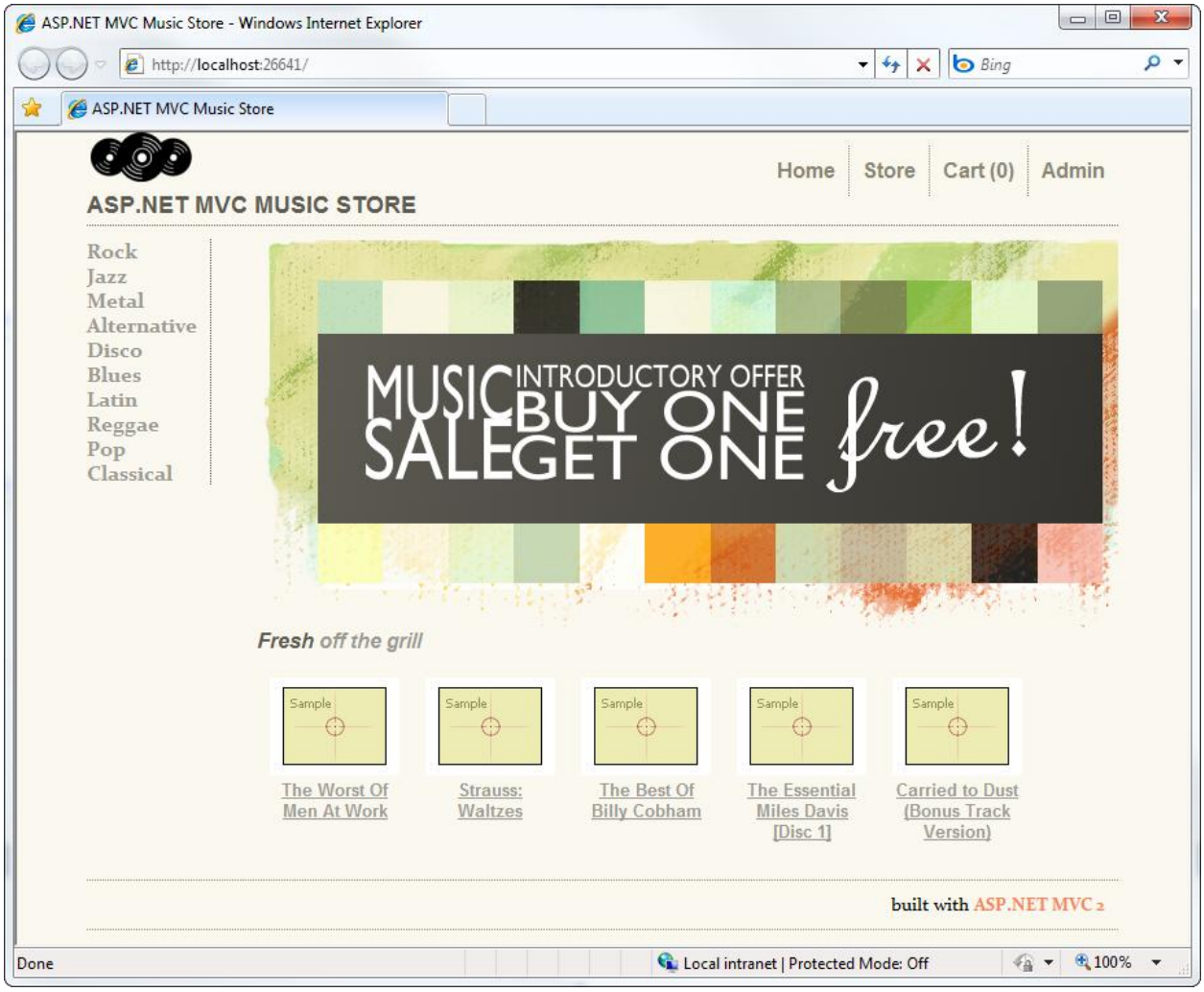
The complete code for the updated HomeController is as shown below.

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

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

        MusicStoreEntities storeDB = new MusicStoreEntities();

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

            return View(albums);
        }

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

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

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

```
<%@ Page Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
    Inherits="System.Web.Mvc.ViewPage<IEnumerable<MvcMusicStore.Models.Album>>" %>

<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
    ASP.NET MVC Music Store
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

    <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>

</asp:Content>
```

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



Conclusion

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