



ASP.NET MVC Music Store Tutorial

Version 0.8

Jon Galloway - Microsoft

4/28/2010

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

ASP.NET MVC Music Store Tutorial

Contents

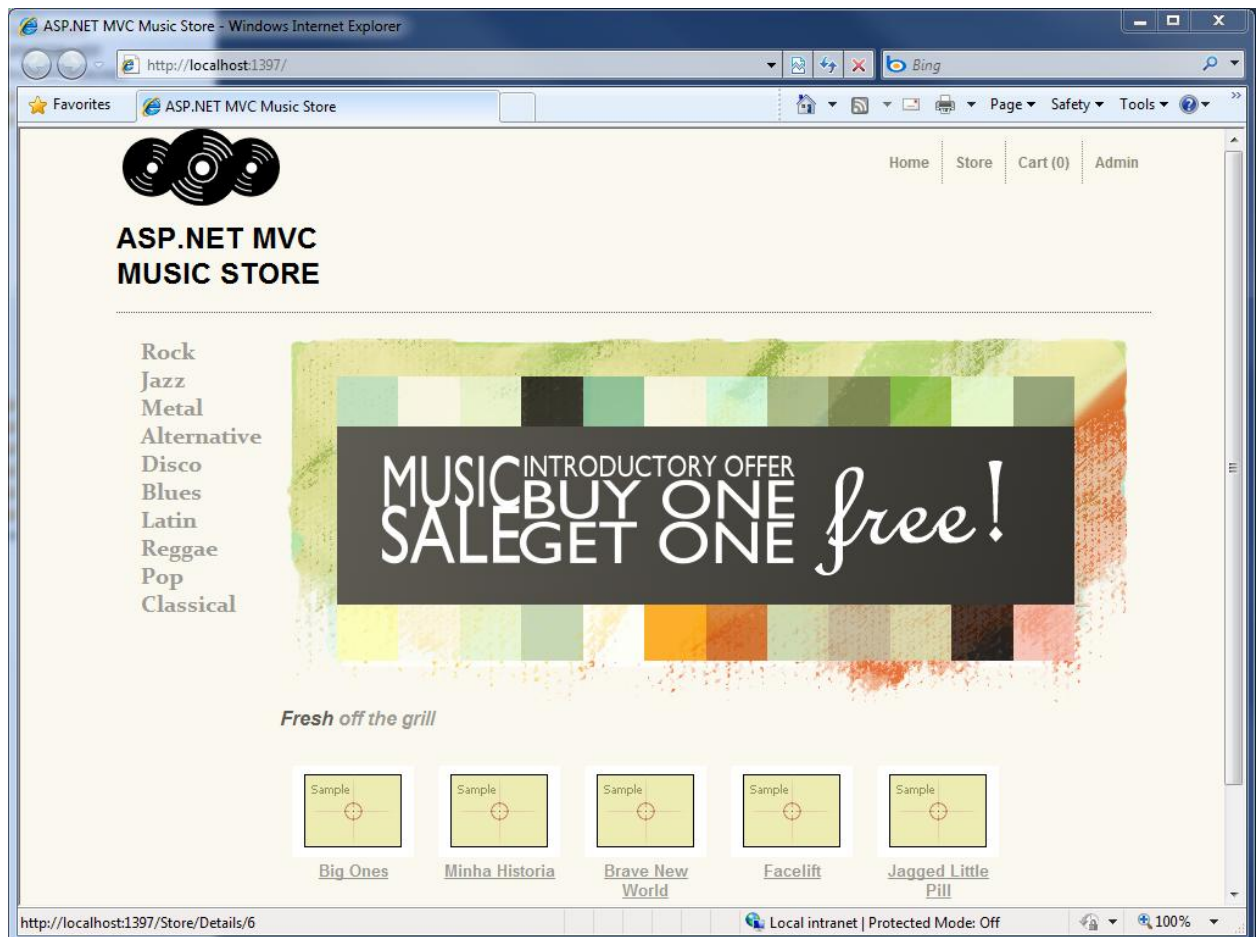
Overview	3
File / New Project.....	8
Controllers.....	10
Adding a HomeController	10
Running the Application	12
Views and ViewModels	17
Using a MasterPage for common site elements	17
Adding a View template	19
Using a ViewModel to pass information to our View.....	21
More complex ViewModels for Store Browse and Index	27
Adding Links between pages	31
Models and Data Access.....	33
Querying the Database	39
Store Index using a LINQ Query Expression.....	39
Store Browse, Details, and Index using a LINQ Extension Method	40
Edit forms and Validation	41
Customizing the Store Manager Index.....	43
Scaffold View templates	44
Using a custom HTML Helper to truncate text.....	46
Creating the Edit View	49
Creating the StoreManagerViewModel.....	50
Writing HTTP-GET Edit Controller Action	50
Creating a Shared Album Editor Template.....	50
Creating the Edit View	52
Implementing Dropdowns on the Album Editor Template.....	53
Implementing the HTTP-POST Edit Action Method	55
Adding the Create View	55
Handling Deletion	58
Using Data Annotations for Model Validation	60

Using MetaData Partial Classes with Entity Framework.....	61
Adding Validation to our Album Forms.....	61
Using Client-Side Validation.....	63
Membership and Authorization	64
Adding the AccountController and Views	64
Adding an Administrative User with the ASP.NET Configuration site	65
Role-based Authorization	68
Registration and Checkout	69
Managing the Shopping Cart business logic	70
The Shopping Cart Controller.....	73
AJAX Updates using Ajax.ActionLink.....	75
Showing the Cart summary in the header with RenderPartial.....	77
Adding in Site style and Conclusion.....	78
Conclusion	79

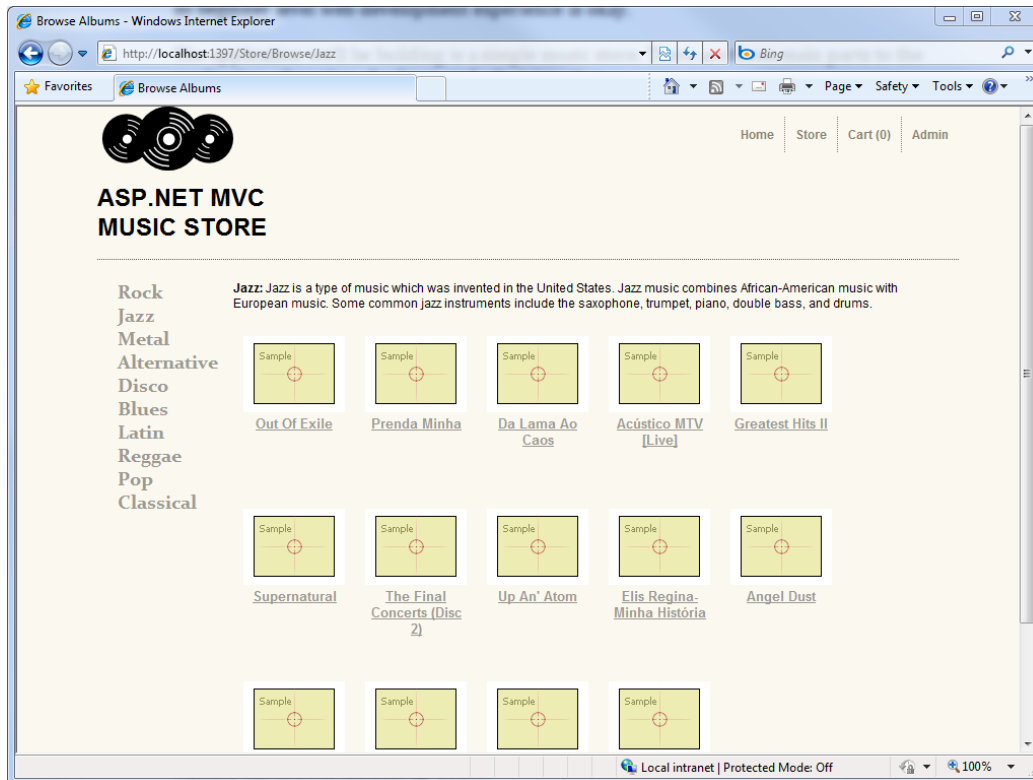
Overview

This tutorial is an introduction to ASP.NET MVC 2. We'll be starting slowly, so beginner level web development experience is okay.

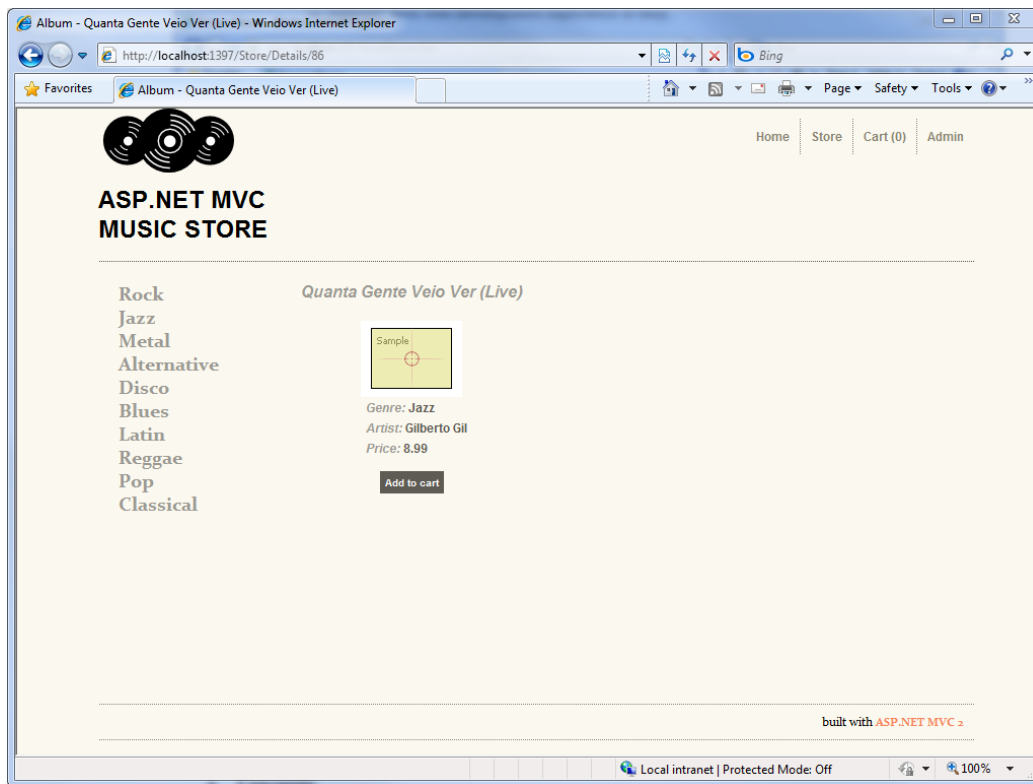
The application we'll be building is a simple music store. There are three main parts to the application: shopping, checkout, and administration.



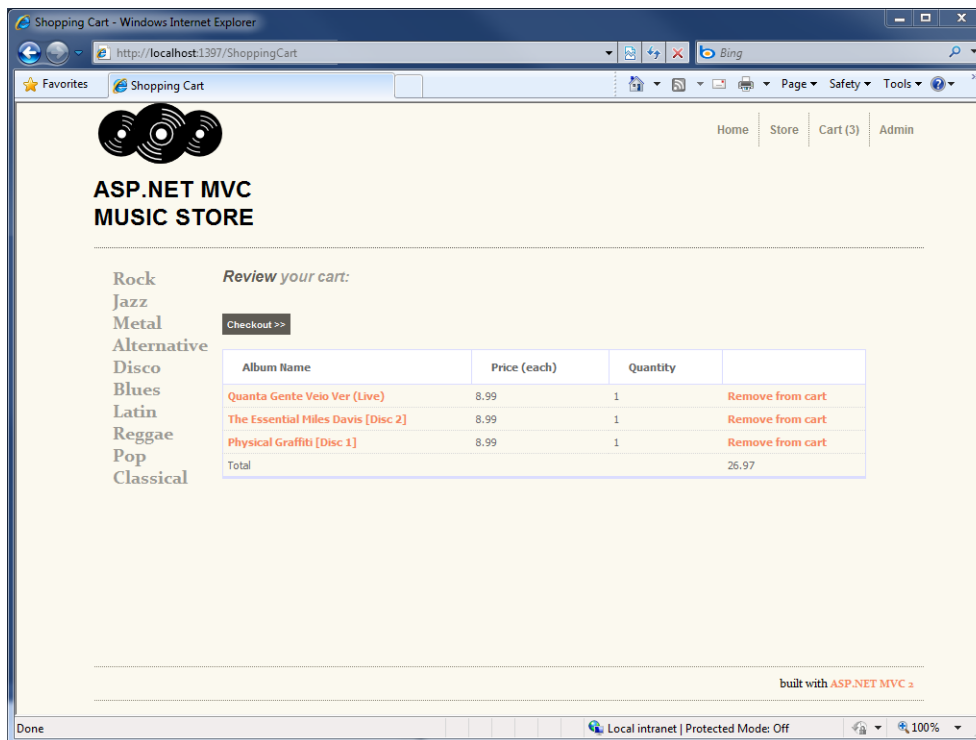
Visitors can browse Albums by Genre:



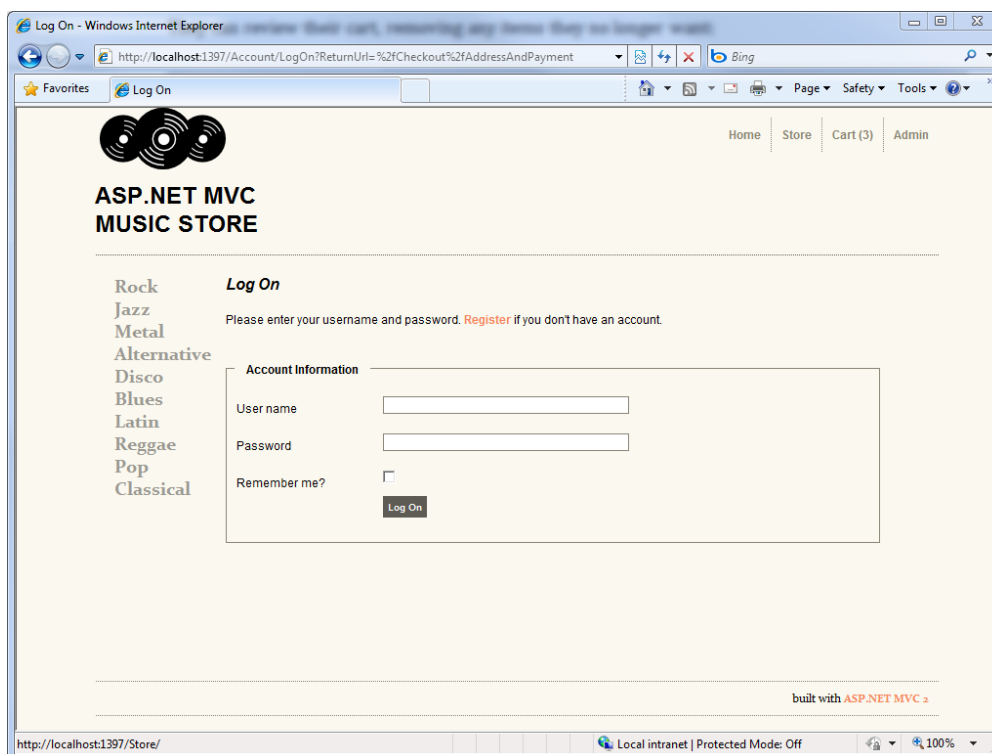
They can view a single album and add it to their cart:

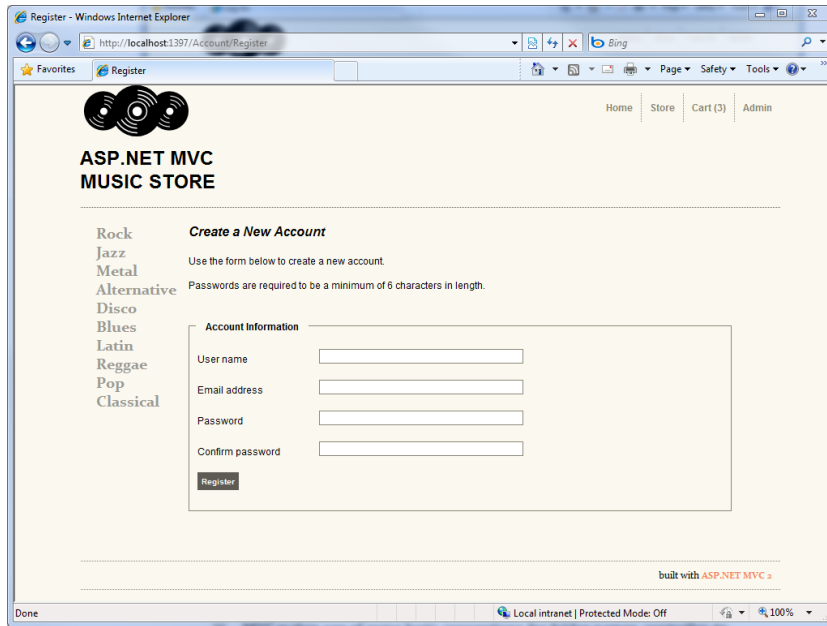


They can review their cart, removing any items they no longer want:

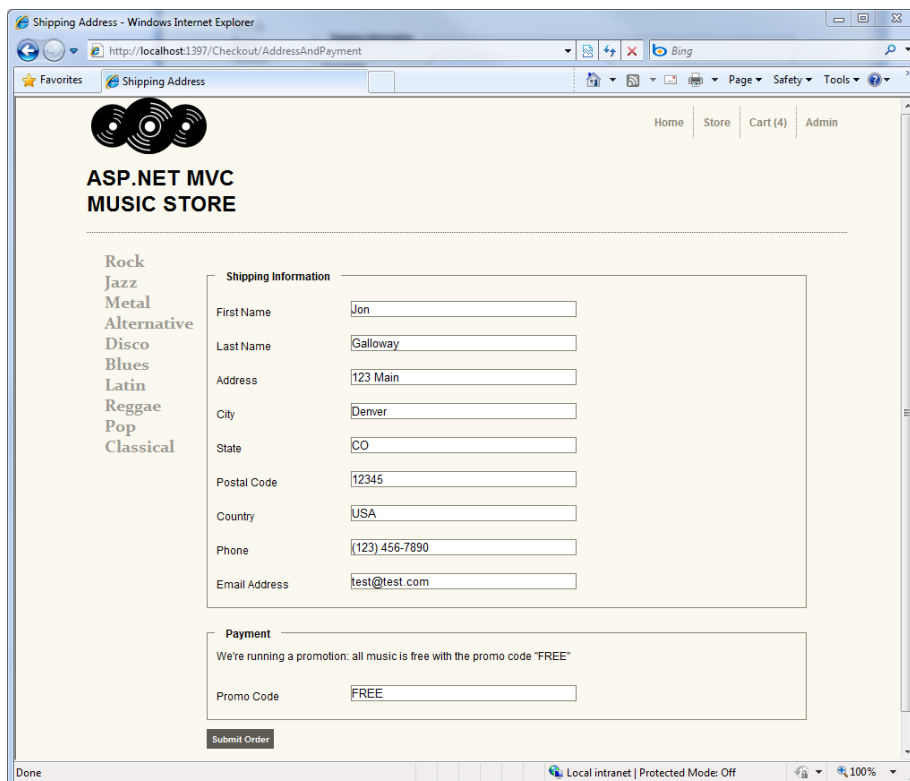


Proceeding to Checkout will prompt them to login or register for a user account.

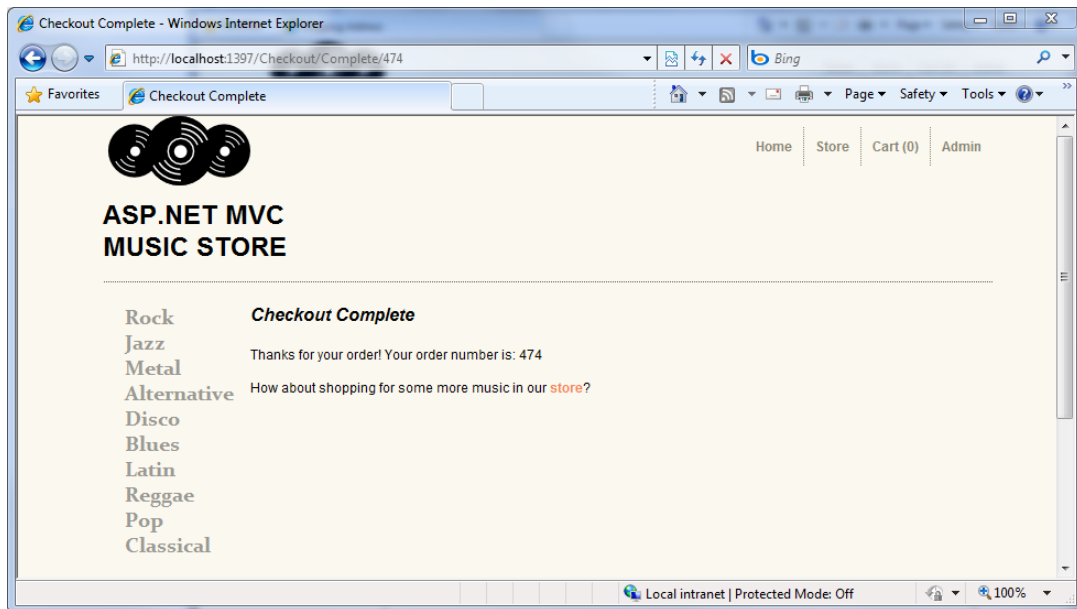




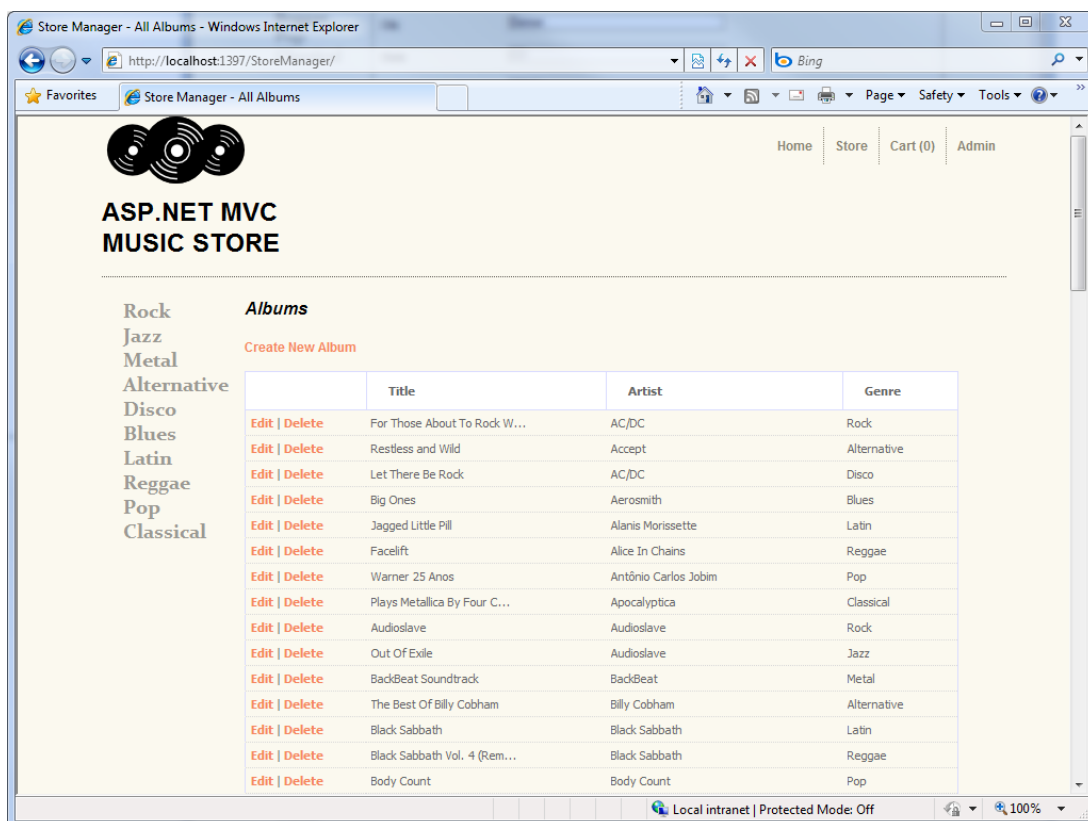
After creating an account, they can complete the order by filling out shipping and payment information. To keep things simple, we're running an amazing promotion: everything's free if they enter promotion code "FREE"!



After ordering, they see a simple confirmation screen:



The Administration page shows a list of albums from which Administrators can Create, Edit, and Delete albums:



We'll begin by creating a new ASP.NET MVC 2 project in Visual Studio 2010, and we'll incrementally add features to create a complete functioning application. Along the way, we'll cover database

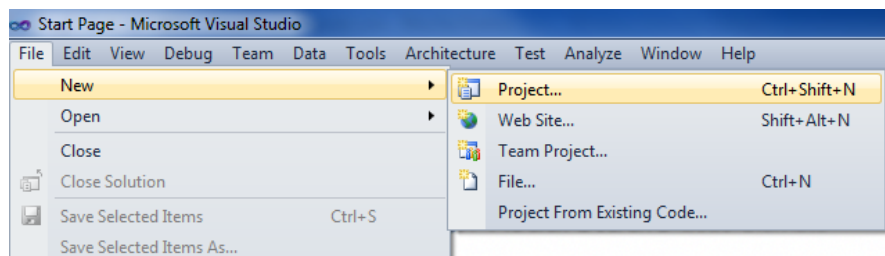
access, list and details views, data update pages, data validation, using master pages for consistent page layout, AJAX for page updates and validation, user membership, and more.

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

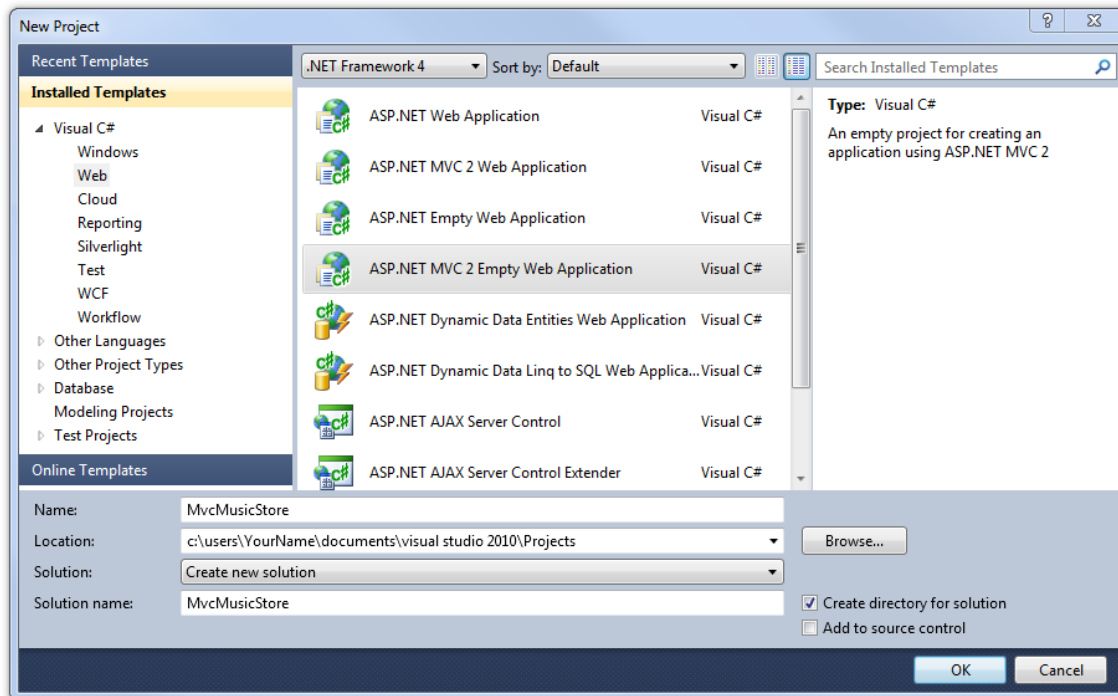
You can use either Visual Studio 2010 or the free Visual Web Developer 2010 Express to build the application. You can use either SQL Server or the free SQL Server Express to host the database. The Express tools can be installed using the Web Platform Installer here: <http://www.microsoft.com/web/platform/tools.aspx>

File / New Project

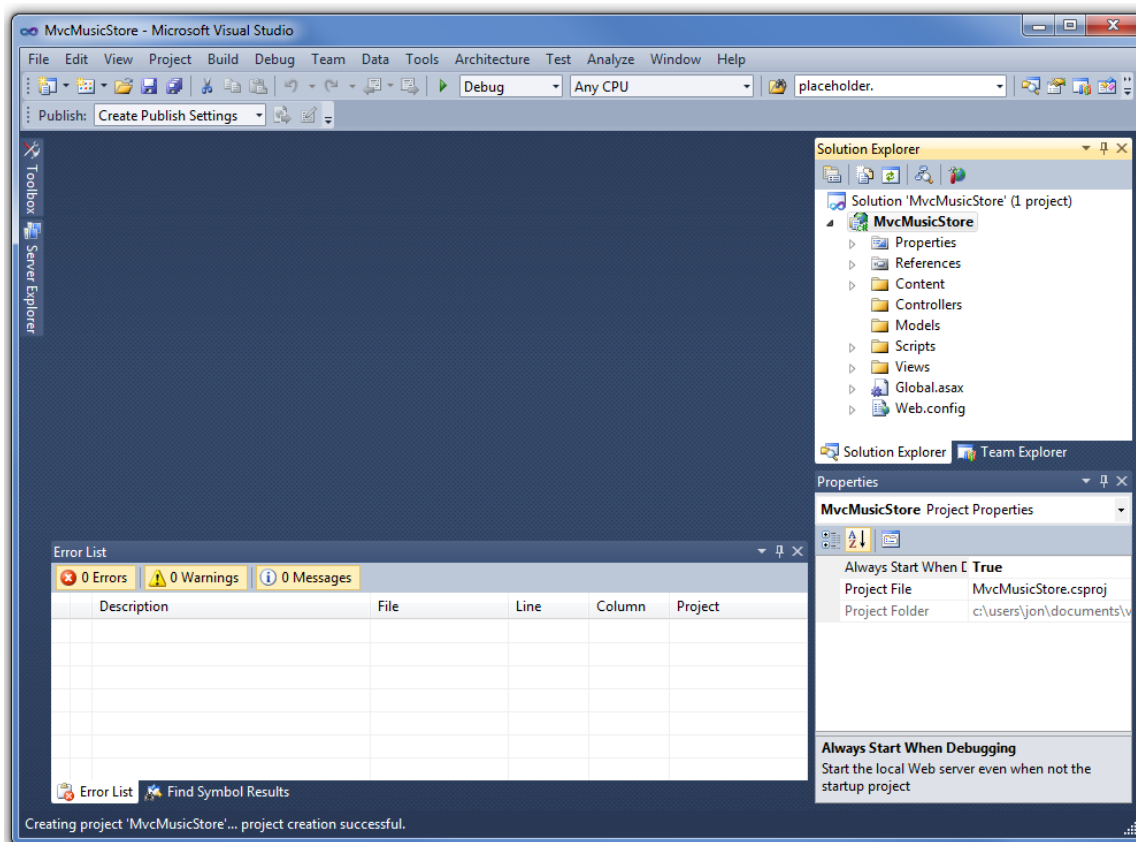
We'll start by selecting the New Project from the File menu in Visual Studio. This brings up the New Project dialog.



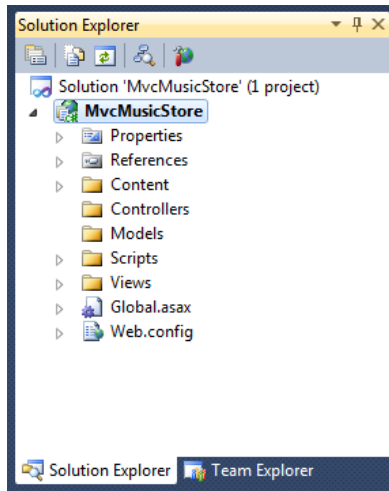
We'll select the Visual C# / Web Templates group on the left, then choose the "ASP.NET MVC 2 Empty Web Application" template in the center column. Name your project MvcMusicStore and press the OK button.



This will create our project. Let's take a look at the folders that are included in our application in the Solution Explorer on the right side.



The Empty MVC 2 Solution isn't completely empty – it adds a basic folder structure:



ASP.NET MVC makes use of some basic 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 data files which can be read and updated by the application

These folders are included even in an Empty ASP.NET MVC application because the framework makes some assumptions based on folder in naming conventions. For instance, controllers look for views in the Views folder by default, so sticking with the basic conventions not only makes it easier for other developers to understand your project, it simplifies your code.

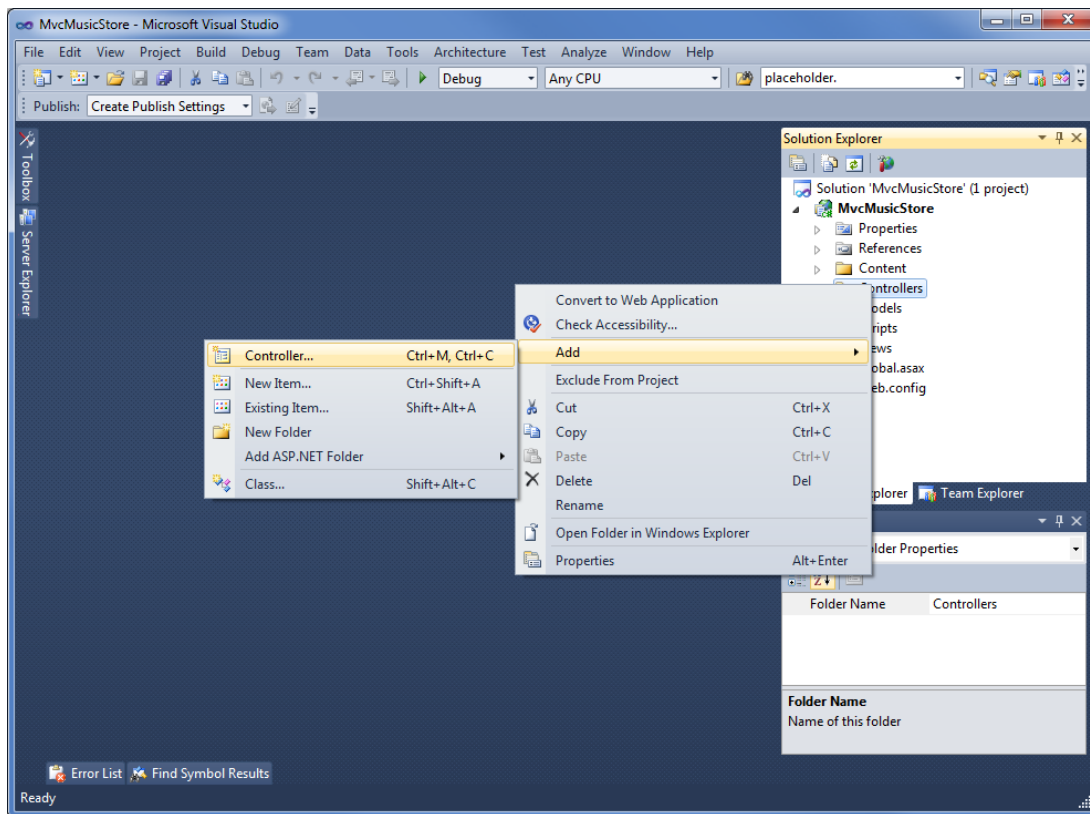
ASP.NET MVC makes extensive use of conventions to simplify the development process; we'll point them out as we go.

Controllers

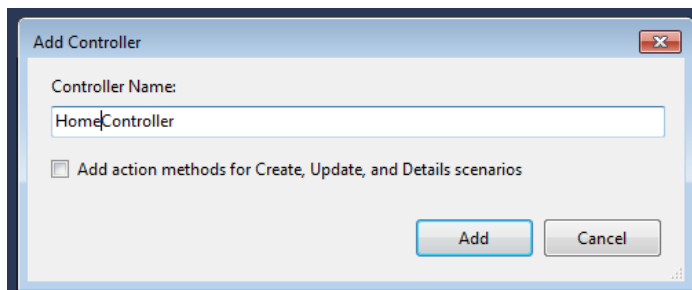
Controllers run the show in an MVC application, so we'll begin there. We'll want to get our site started with a Home page, so we'll add a Controller to handle the Home page of the site. We'll follow the convention and call it HomeController.

Adding a HomeController

Right-click the controllers folder and select "Add", then "Controller..."



Change the Controller Name to 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 that with a simple method that just returns a string. Let's 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:

```

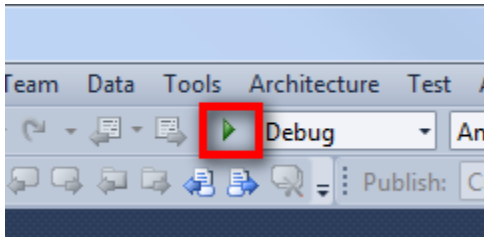
public string Index()
{
    return "Hello from Home";
}

```

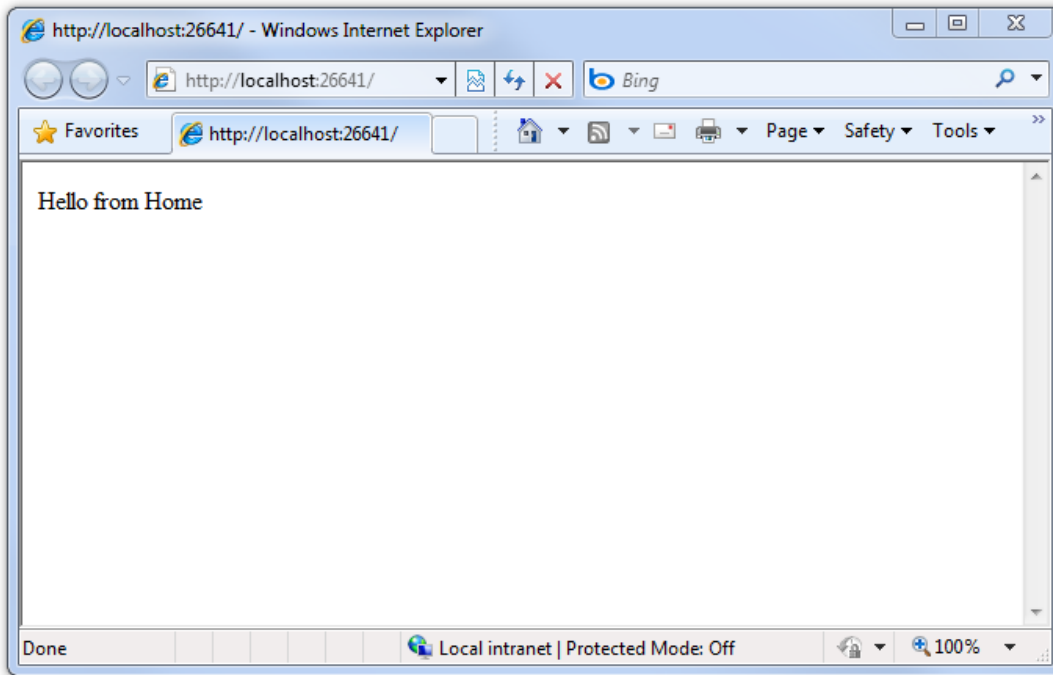
Running the Application

Now we can run the site. We can do 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.



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 we’ve got a start.

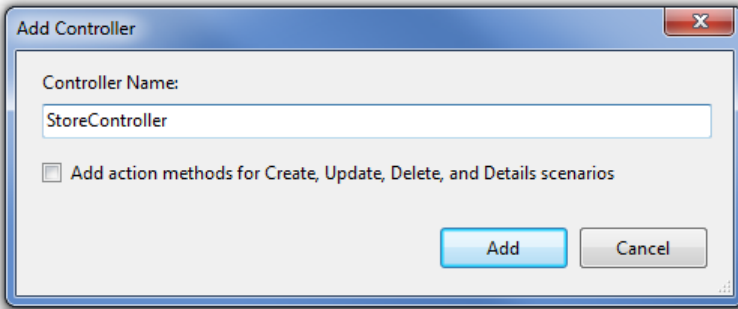
Note: Visual Studio includes the ASP.NET Development Server, which will run your website on a random free “port” number. In the screenshot above, the site is running at `http://localhost:26641/`, so it’s using port 26641. Your port number will be different. When we talk about URL’s like `/Store/Browse` 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`.

Now let’s set up a controller for our Store. The store has three levels:

- The Store Index lists the genres our store carries
- The Browse page lists all albums in a genre
- The Details page shows information for a specific album

We’ll start by adding a new StoreController, just like we created the HomeController. 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:



The new controller will already have an Index method. We'll also need to add methods to handle the two other pages in the store: Browse, and Details. These methods are called Controller Actions, and as you've already seen with the HomeController Index() Controller Action, their job is to respond to URL requests and (generally speaking) put content on a page.

We'll start by changing the StoreController.Index 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()";
        }

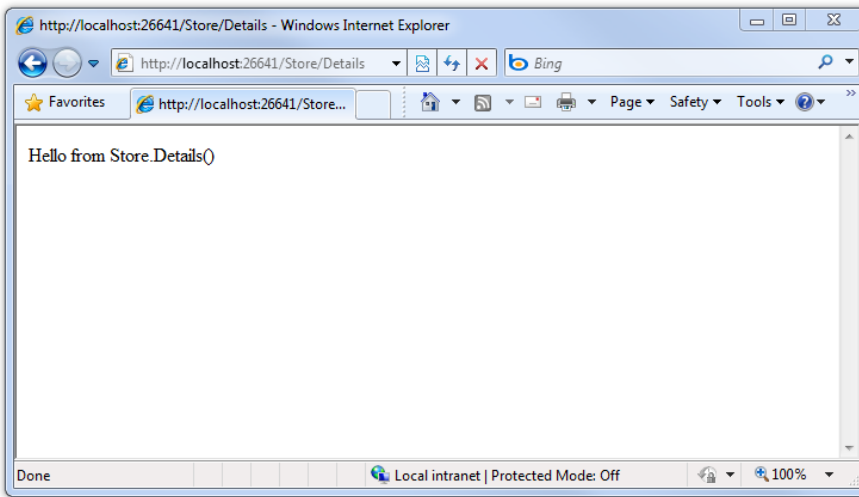
        //
        // GET: /Store/Browse

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

        //
        // GET: /Store/Details

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

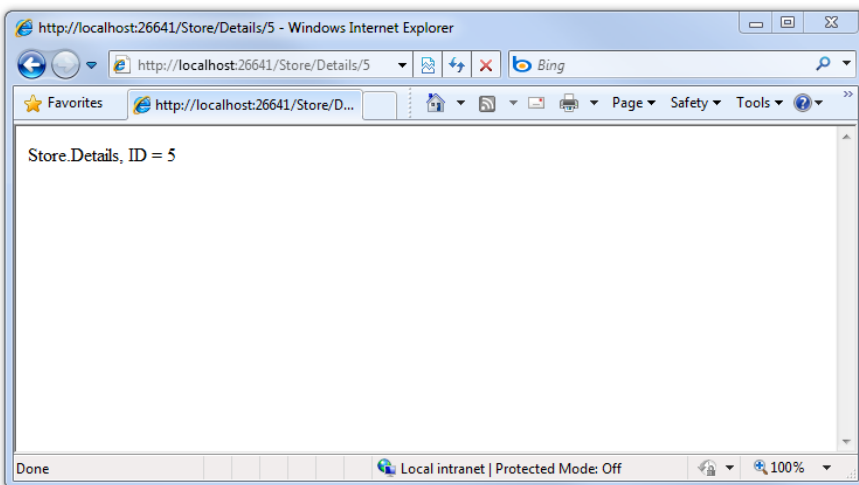
Run the project again and browse to /Store/Details:



That's great, but these are just simple 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 Details action to read and display an input parameter named ID.

```
//  
// GET: /Store/Details/5  
public string Details(int id)  
{  
    string message = "Store.Details, ID = " + id;  
    return Server.HtmlEncode(message);  
}
```

Run the application and browse to /Store/Details/5. The controller action read the ID from the URL and filled it into the string that we wrote out to the browser:

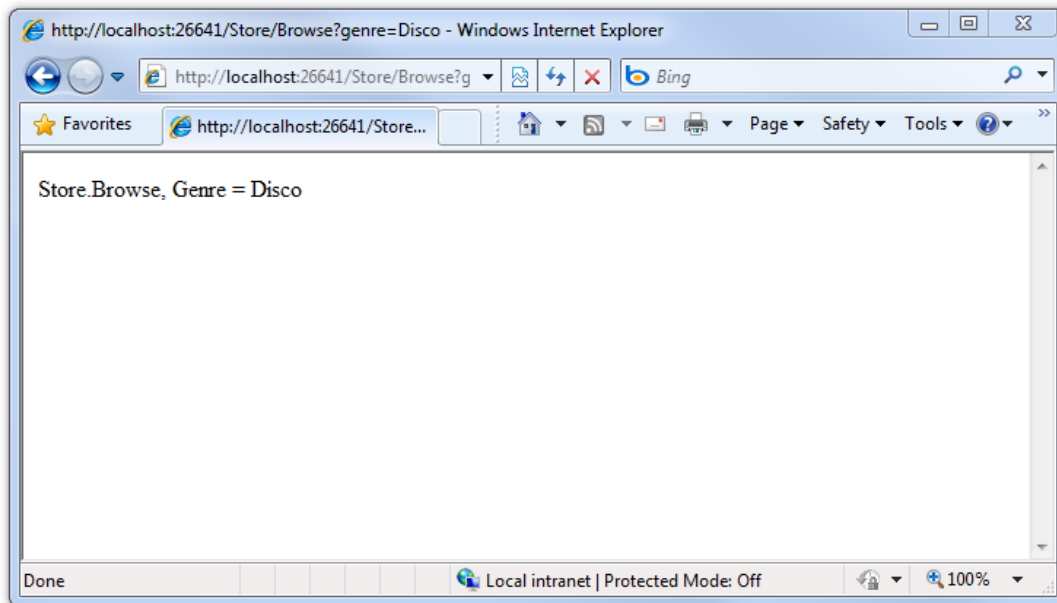


That was especially easy because ID is a special case. ASP.NET MVC uses a system called routing to map URL values to controller action parameters, and it includes a default “route” with an ID parameter already set up in all new projects. That’s configurable, and we’ll look at that more later.

We can also read querystring values from a URL without any routing changes. Alter the Browse Controller action to accept and Genre string parameter:

```
//  
// GET: /Store/Browse/  
public string Browse()  
{  
    string message = "Store.Browse, Genre = " +  
        Server.HtmlEncode(Request.QueryString["genre"]);  
    return Server.HtmlEncode(message);  
}
```

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 recap what we’ve done so far:

- We’ve created a new project in Visual Studio
- We’ve overviewed 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 Controllers with four Controller Actions which respond to URL requests and return text to the browser

Views and ViewModels

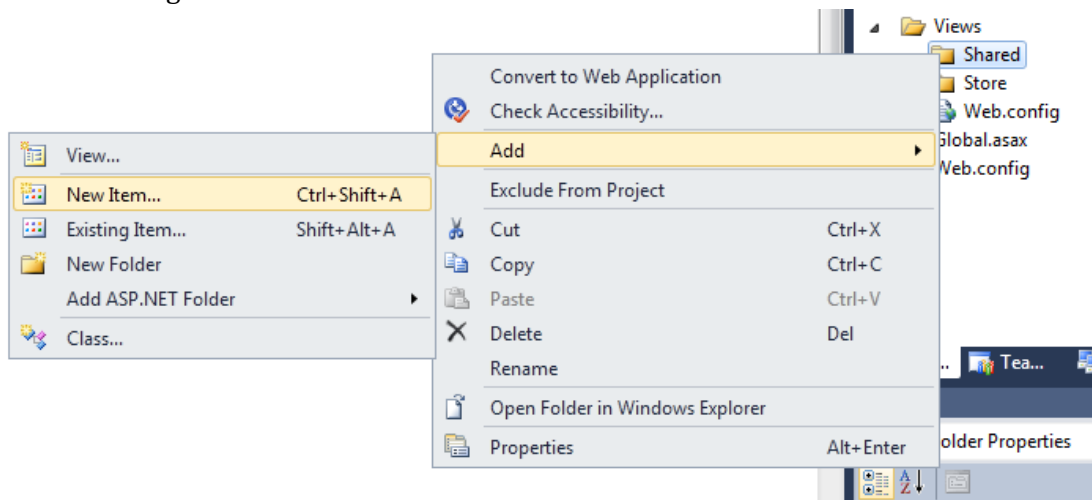
- *Concepts:*
 - *Views allow us to template content (rather than just writing out strings)*
 - *Views follow a naming convention: /Controller/Action*
 - *ViewModels help us to pass information from our Controller Action to our View*
 - *ActionResult*

So far we've just been returning strings from controller actions. That's a great way to get an idea of how controllers work, but it's not how you'd want to build a complete web application. We need a way to split our HTML out into a separate template. That's exactly what Views do.

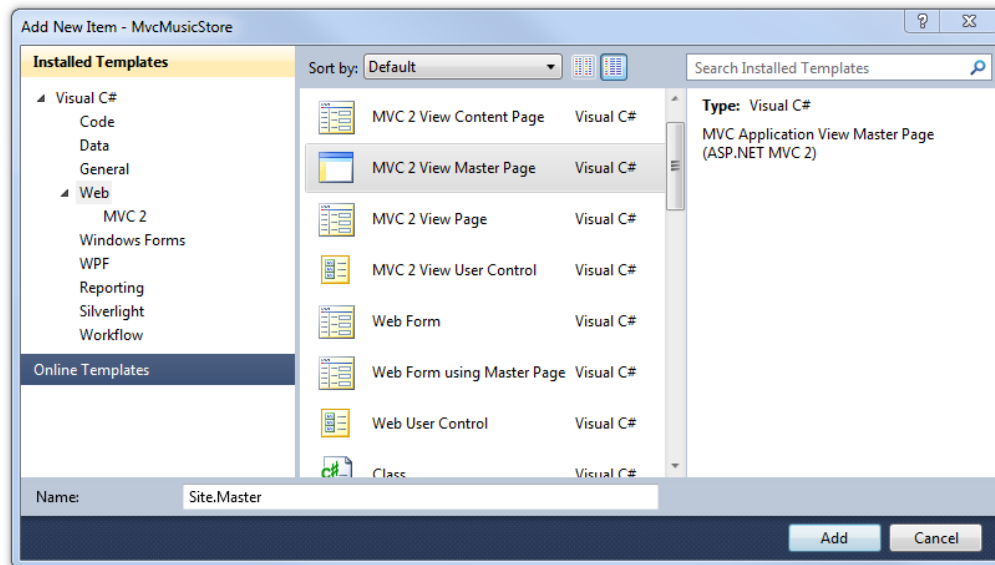
Using a MasterPage for common site elements

We're going to convert our Home page to use a view, but before we do that we'll want to add a CSS stylesheet and a MasterPage to our site. ASP.NET MasterPages allow us to set a template for common user interface elements that will use across the entire website. These are similar to include files, but a lot more powerful.

Since the MasterPage is shared by all pages in the site, we'll create it in 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.



Since we'll want to reference our Stylesheet on all pages in our site, we'll add a reference to it in our MasterPage. Add a <link> element directly below the <head> tag, so the head of your MasterPage looks like this:

```
<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 a header with links to our Home page and Store area on all pages in the site, so we'll add that directly below the opening <div> tag. 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>
```

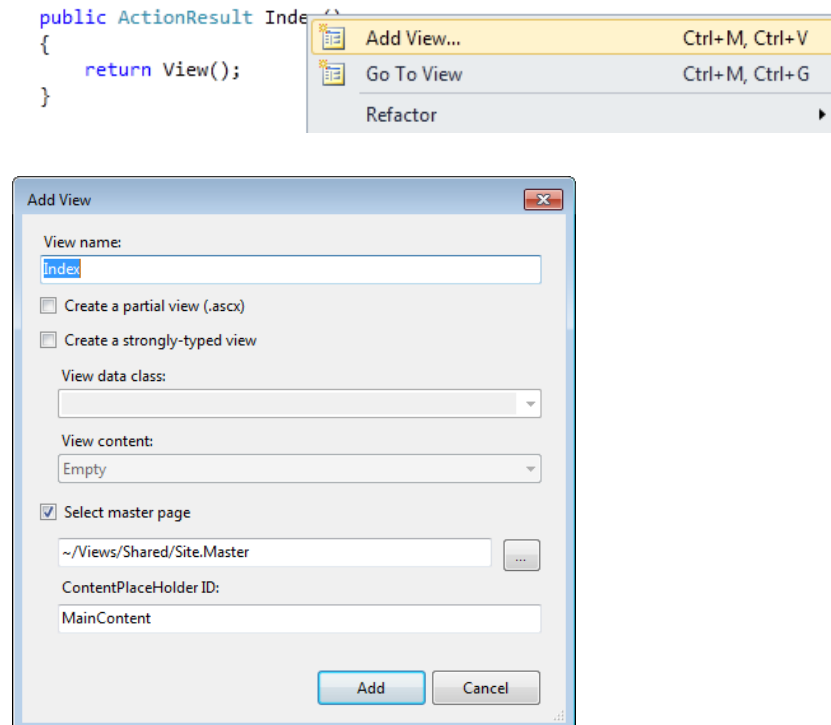
Adding a View template

Now we're ready to add a View for the Home page. Change the HomeController Index method to return an ActionResult, and have it return View(), like this:

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

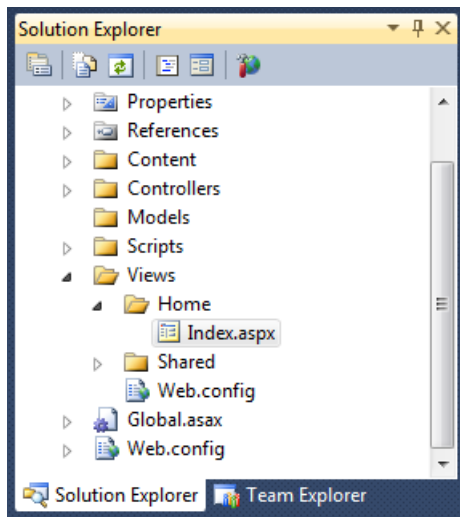
    public ActionResult Index()
    {
        return View();
    }
}
```

Now we're ready to add the view file to the project. Position the text cursor within the controller action method, the right-click and select "Add View". This will bring up the Add View dialog:



By default the dialog pre-populates the name of the View to match the action method. Since we've got a MasterPage set up, it also pre-fills that value for us.

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.



Both the file name and location are important, since they follow a convention. The directory name, Home, matches the controller, which is named HomeController. The view template name, Index, matches the controller action which will be displaying the view. This allows us to keep our controller code simple since we don't need to specify the name or location of the view as long as we're following the convention.

Visual Studio also automatically opens the Index.aspx view template in the code-editor when it is created. Let's update the Title to "Home", and change the heading 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" %>

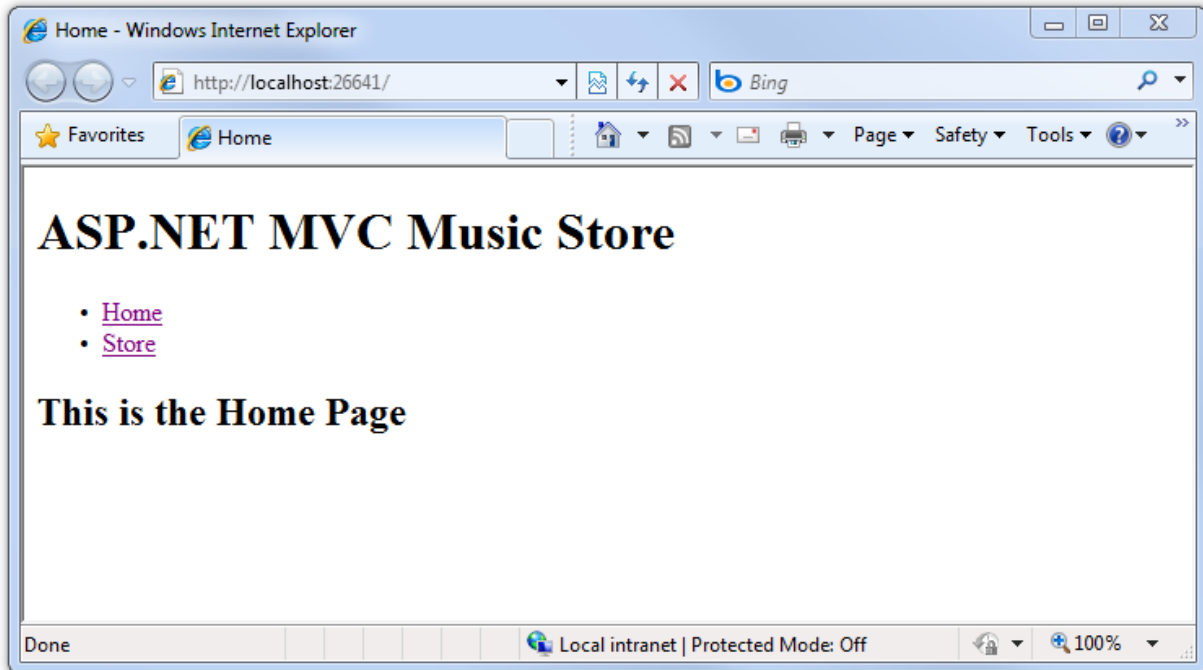
<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
    Home
</asp:Content>

<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.



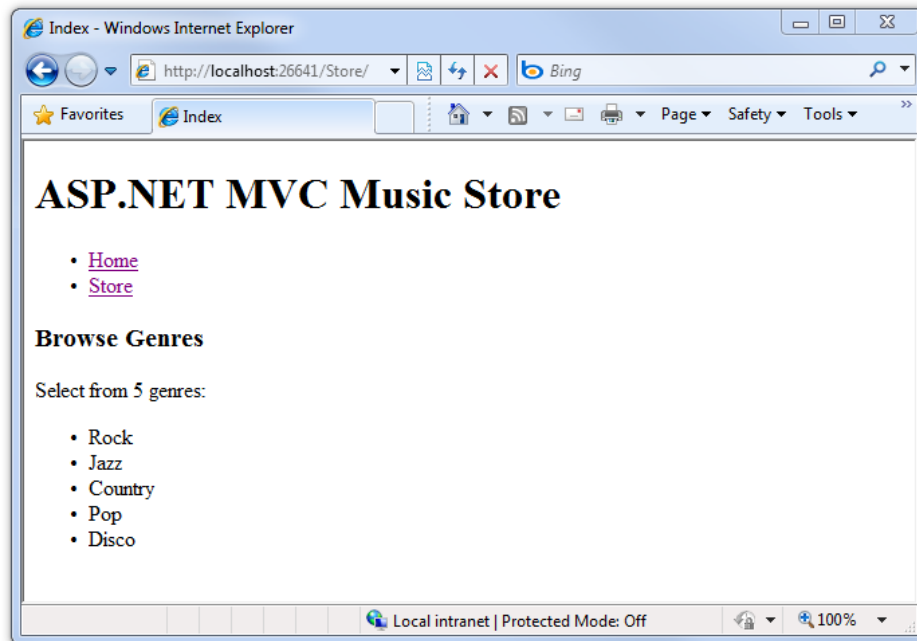
Let's review what's changed:

- The Home Page is using our MasterPage template, so the header content is displaying.
- The HomeController's Index action method found and displayed the correct View template, even though our code called "return View()", because our View template followed the standard naming convention.

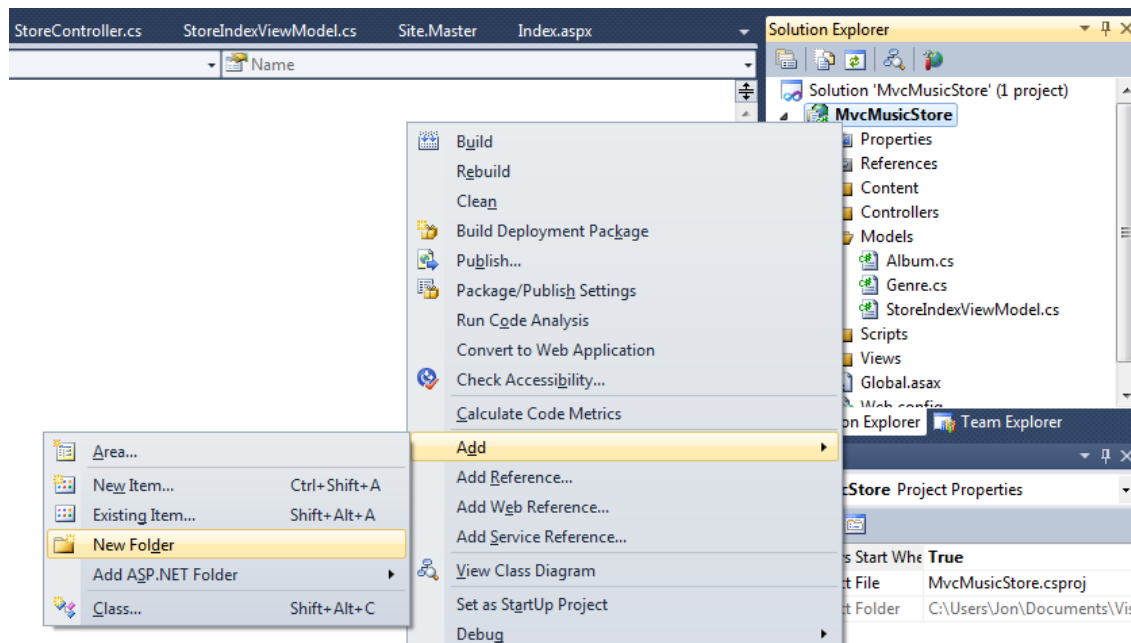
Using a ViewModel to pass information to our View

A View template that just displays hardcoded HTML isn't going to make for very interesting web site. To create a dynamic web site, we'll need to pass information from our controller actions to the view templates. One common technique is to use a ViewModel, which allows us to cleanly package all the information our View template will need to display.

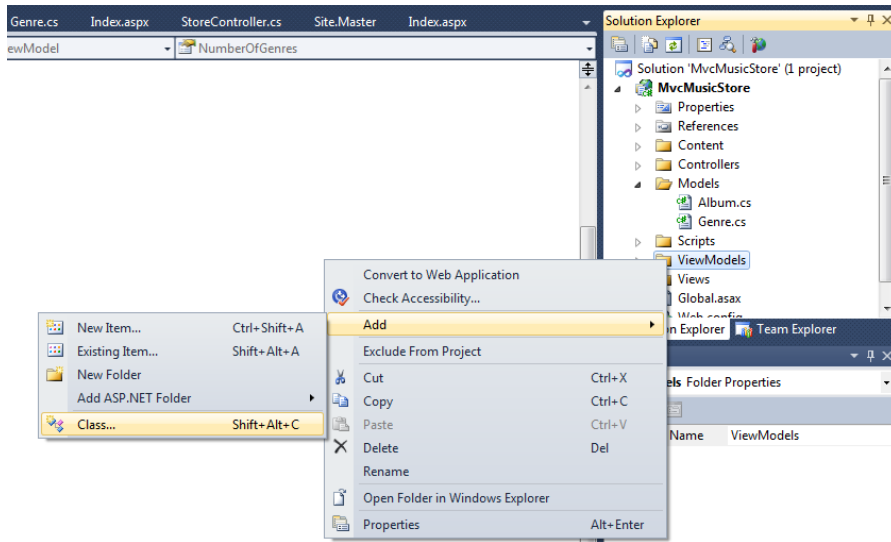
We'll first change our Store Index page to list the number of genres we carry, so it looks like this:



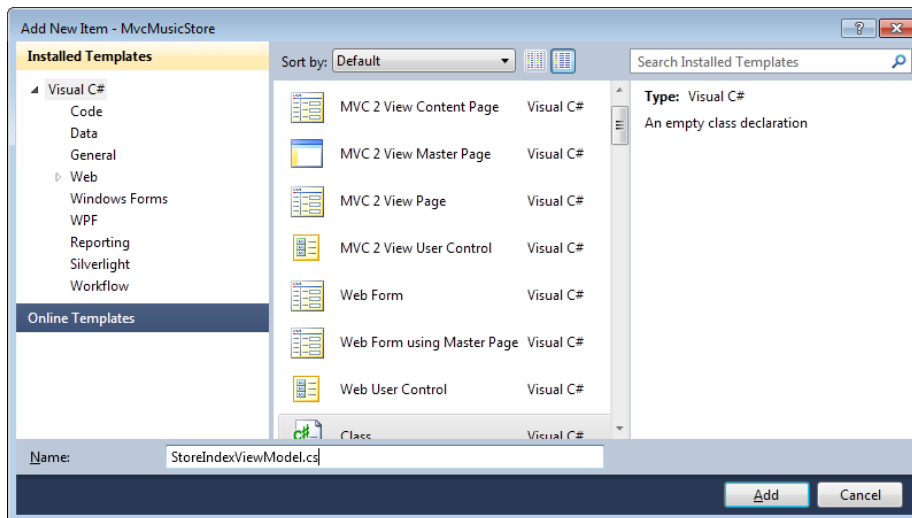
We'll create a ViewModel directory to hold our ViewModels by right-clicking the project and selecting Add⇒New Folder, then naming the folder ViewModels.



Next, we'll create a ViewModel for our Store Index page. Right-click on the ViewModels folder and select Add⇒Class...



Name the class StoreIndexViewModel and press the Add button:



We'll add an integer property named NumberOfGenres and a List of strings to handle Genres:

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

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.

In order to use the `StoreIndexViewModel` from our `StoreController`, we need to add the following line to the top of the `StoreController` code:

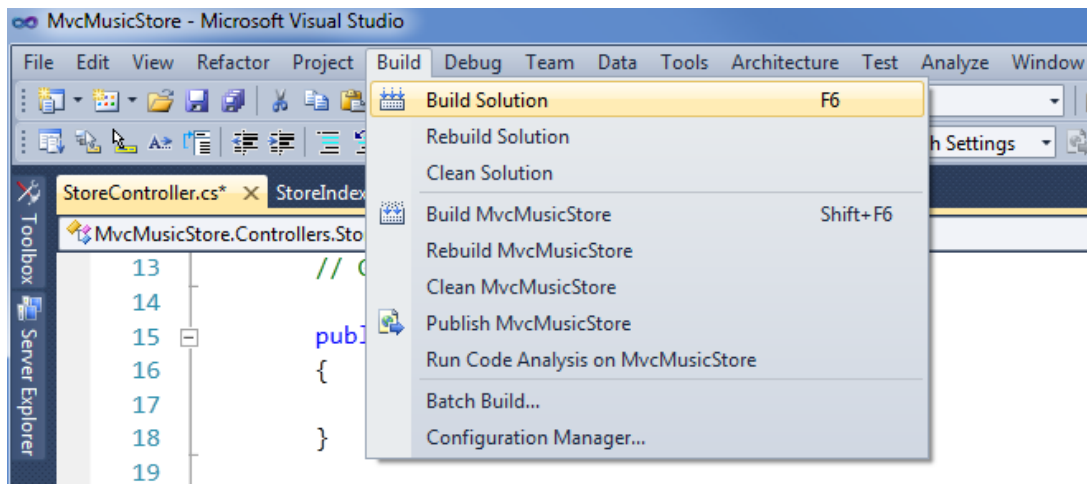
```
using MvcMusicStore.ViewModels;
```

Now we'll change the `StoreController`'s `Index` action method to return that `ViewModel`. We'll use Object Initializer syntax to quickly create objects and set their properties with one line of code.

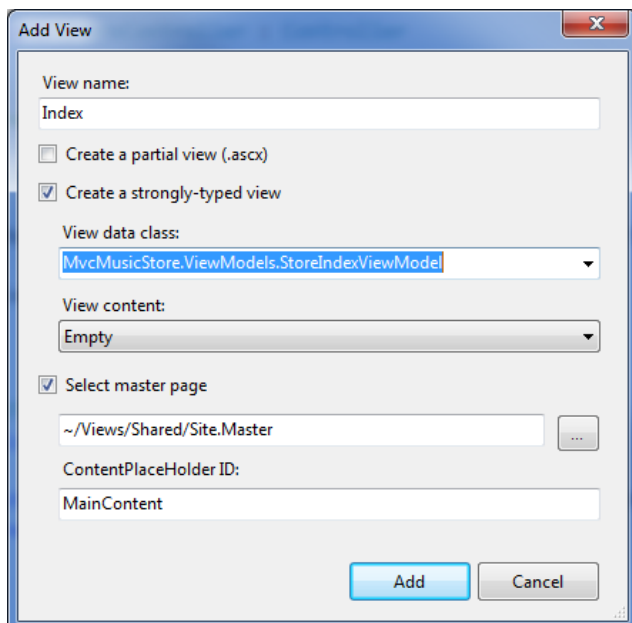
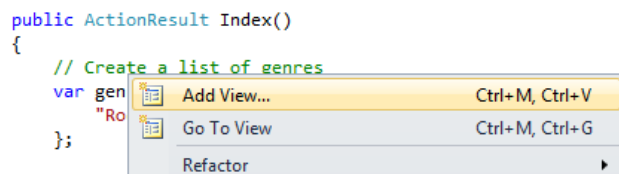
```
//  
// GET: /Store/  
  
public ActionResult Index()  
{  
    // Create a list of genres  
    var genres = new List<string> { "Rock", "Jazz", "Country", "Pop", "Disco" };  
  
    // Create our view model  
    var viewModel = new StoreIndexViewModel {  
        NumberOfGenres = genres.Count(),  
        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.

Next we'll want to add a view, but before we do that we need to build the project so that the Add View dialog knows about our `StoreIndexViewModel`. You can build the project by selecting the Build⇒Build Solution menu item.



Right-click `Store.Index()` and Add View. Create a strongly-typed view based on `StoreIndexViewModel`.

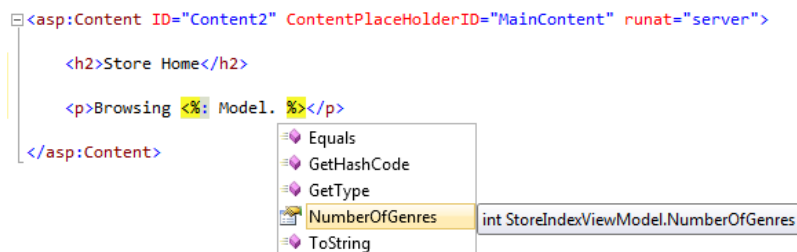


Now we'll update the Store Index page to add the NumberOf Genres value. We'll be using the <%: %> syntax (often referred to as "code nuggets") to execute code within our View template. There are two main ways you'll see this used:

- 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 always use the <%: %> syntax instead, since it will automatically HTML Encode the results, similar to how we used Server.HtmlEncode() in our controller action earlier.

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 values. You can just select "NumberOfGenres" off the list rather than typing it in.



Next we'll list the Genres in the list with 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 of type StoreIndexViewModel. The View engine exposes the page type as a property named Model, so we can access our ViewModel's values using the Model values.

```
<%@ 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">
    Index
</asp:Content>

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

    <h3>Browse Genres</h3>

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

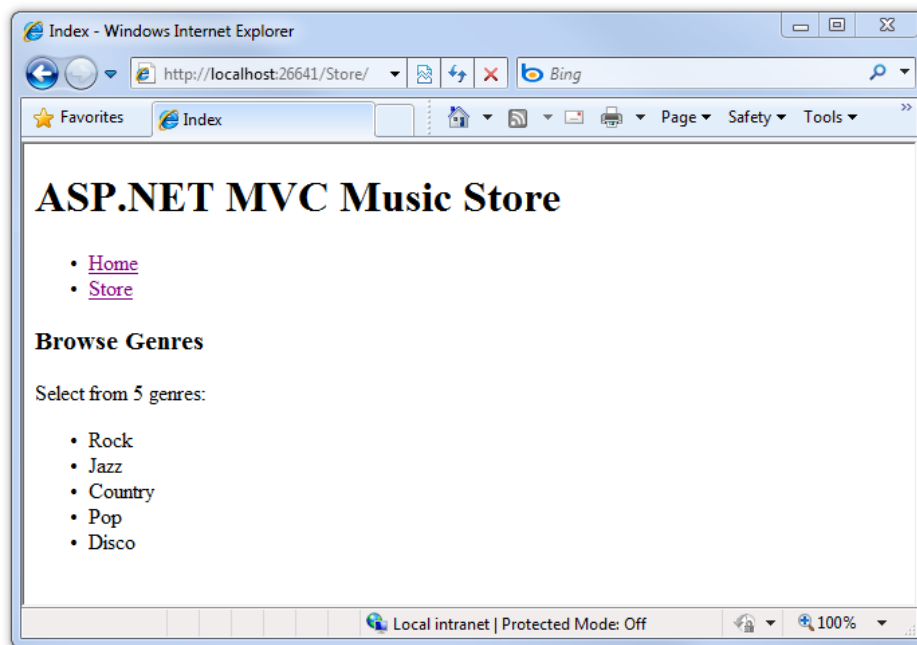
```

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

</asp:Content>

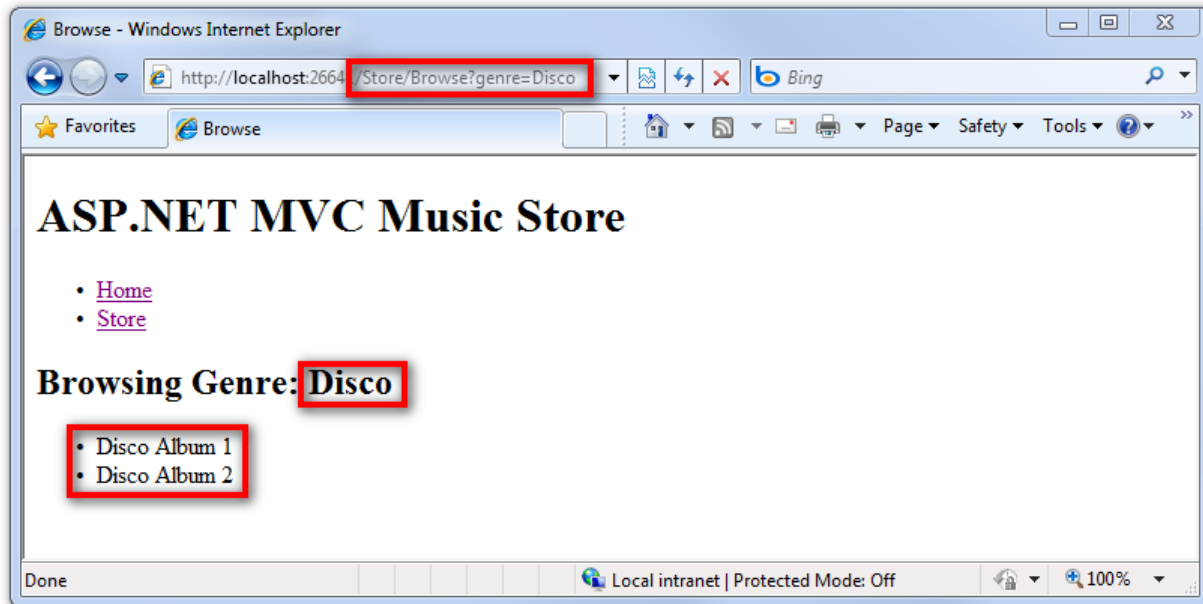
```

Run and click on the Store link. Note that the number of Genres was filled in.



More complex ViewModels for Store Browse and Index

Now let's take a look at a slightly more complex example with the Store Browse page. This page reads the Genre name from the URL and displays the Genre and a few Albums using the name it was passed, 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 won't have a Genre page in the application, but we'll be using the concept of a Genre repeatedly, so we'll want a model class to manage it. 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 and add a new Genre class and add a string Name property.

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

Now we'll create an Album class that has a Title and a Genre.

```
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 (again, right-clicking the ViewModels folder and selecting Add⇒Class).

We'll add a using 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 Store Browse and Details action methods to use our new ViewModels. We'll start by adding a using MvcMusicStore.Models statement to the using statements list at the top of the controller, then modify the Browse and Details methods to appear as follows:

```

// GET: /Store/Browse/
public ActionResult Browse()
{
    string genreName =
        Server.HtmlEncode(Request.QueryString["genre"]);

    var genre = new Genre {
        Name = genreName
    };

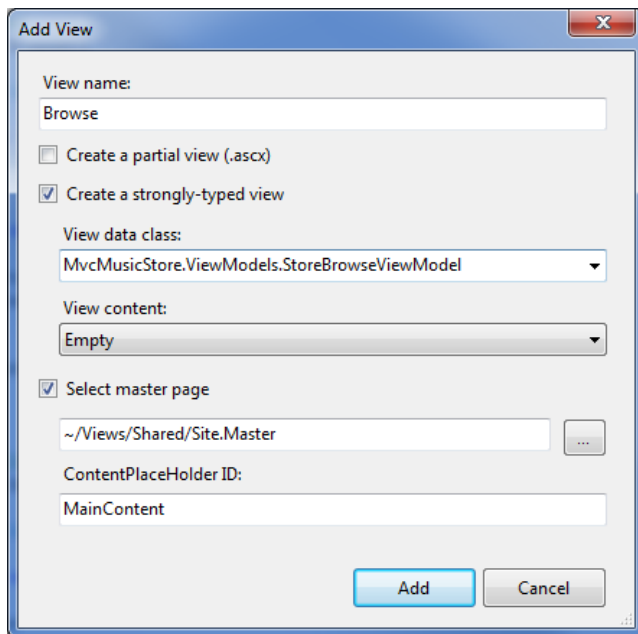
    var albums = new List<Album>();
    albums.Add(new Album { Title = genreName + " Album 1" });
    albums.Add(new Album { Title = genreName + " Album 2" });

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

    return View(viewModel);
}

```

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, then a strongly typed Details view:



Now we can modify the Browse page content to display the Genre information, accessing the ViewModel info information:

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

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

    <ul>

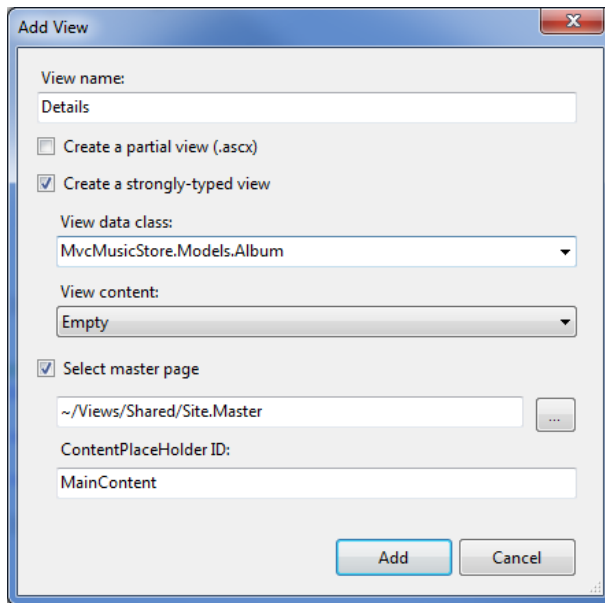
        <% foreach (var album in Model.Albums) { %>
            <li><%: album.Title %></li>
        <% } %>

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

Now that we have these basics down, we can quickly create a Store Details page to display a single album. We could create a StoreDetailsViewModel that holds a single Album, but in this case we'll just return an Album since we're certain that's all our Details view will be displaying. We'll need to add a using statement that references the MvcMusicStore.Models namespace, then we can update our Store Details controller action as follows:

```
//
// GET: /Store/Details/5
public ActionResult Details(int id)
{
    var album = new Album { Title = "Sample Album" };
    return View(album);
}
```

Now right-click on the Store Details controller action method and create a new Details view:



Now we can display the Album title in the Details page.

```
<%@ 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>
```

Adding Links between pages

Our Store Index page lists Genres as plain text. We can easily convert those to dynamic links to our Browse page, so clicking on Disco will navigate to /Store/Browse?genre=Disco. We could just hardcode those links, as follows:

```
<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.

Instead, we can make use 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 utility tasks just like this. You'll use the `Html.ActionLink` helper method pretty often, since it makes it easy to build links and takes care of annoying details like making your 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. For example, we can link to `"/Store/"` 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 five parameters:

1. Link text, which will display the Genre name
2. Controller action name (Browse)
3. Controller name (Store)
4. Route parameter values, specifying both the name (Genre) and the value (Genre name)
5. HTML Attributes for the link – in this case we don't need to specify any, so we'll just pass null

Putting that all together, here's how we'll write those links to the Browse page:

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

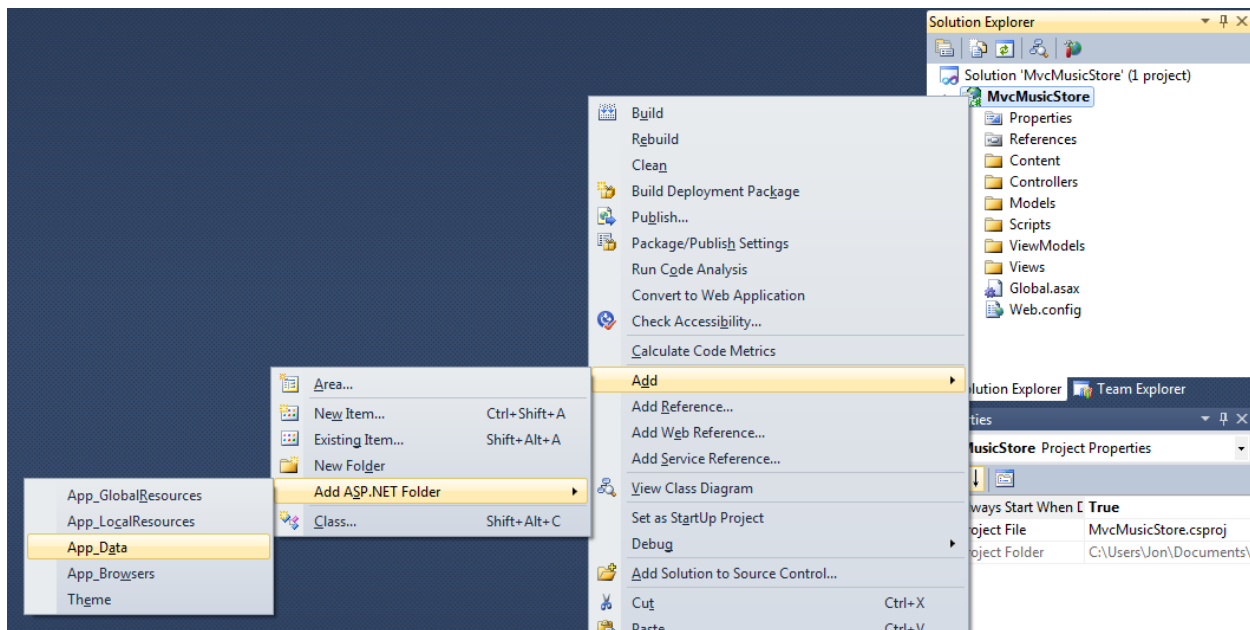
Models and Data Access

Concepts:

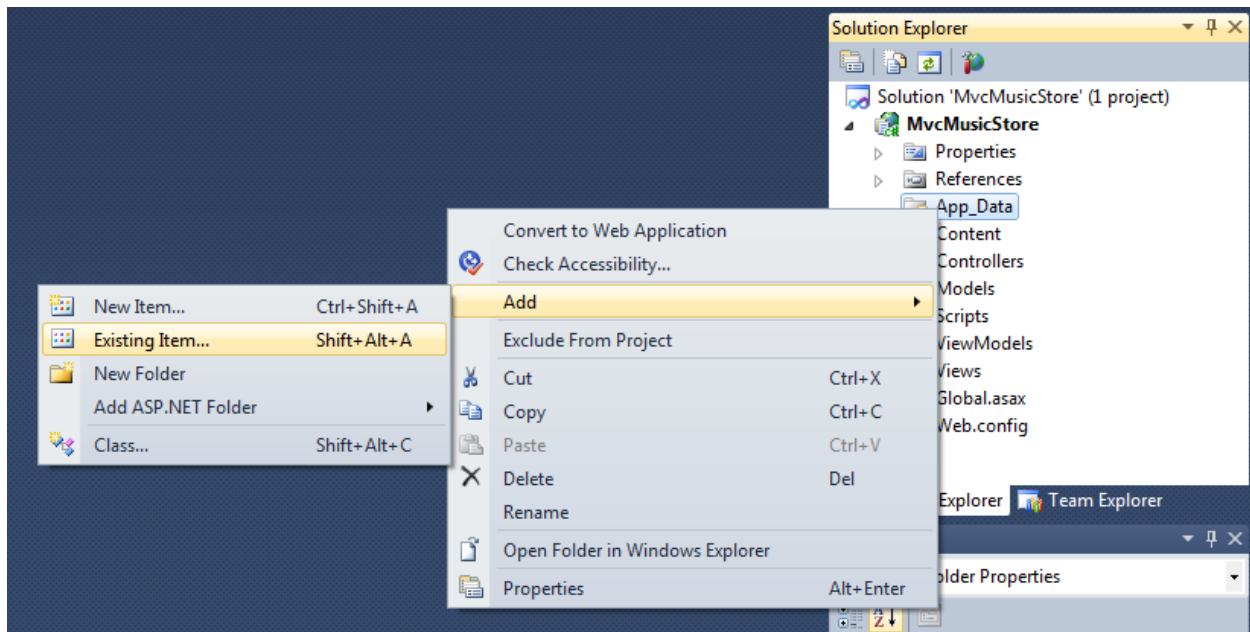
- *Use of EntityFramework for data access*
- *End-to-end parameter flow:*
 - *Parameter passed to Controller Action via Routing*
 - *Parameter passed to ViewModel in constructor*
 - *ViewModel returned to View via ActionResult*
- *Requested data displayed on screen*

So far, we've just been passing strings from URLs to Controller actions to Views. That's a great way to lay the foundations, but now we're ready to hook up a database. The tutorial text will cover creating the database using the free SQL Server Express edition, but all our 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.

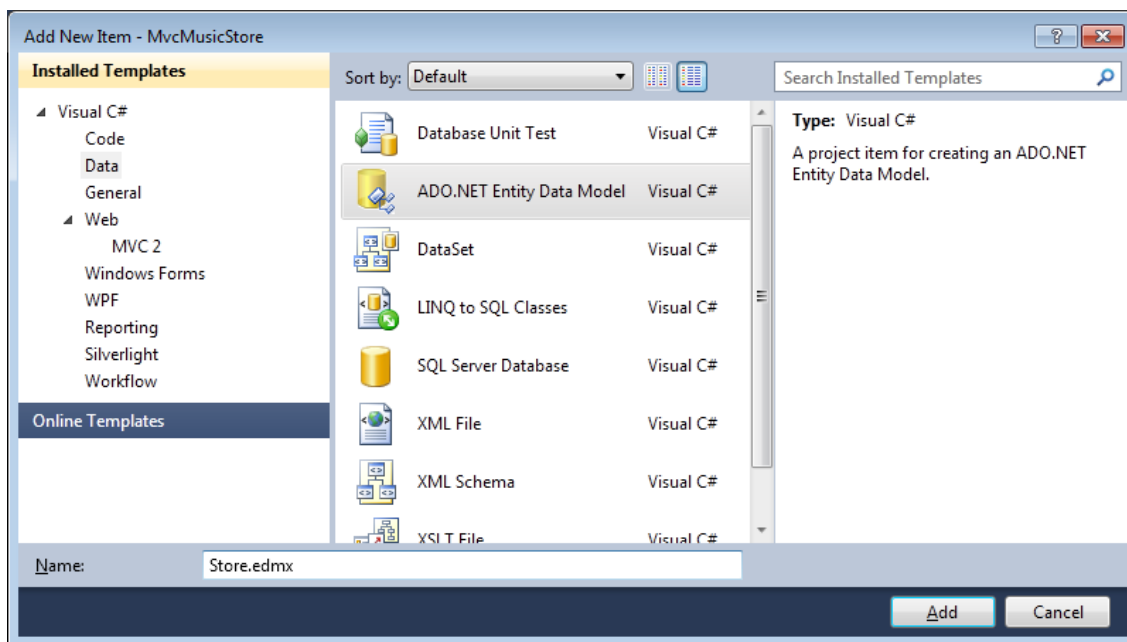
Right-click the project and select Add⇒Add ASP.NET Folder⇒App_Data.



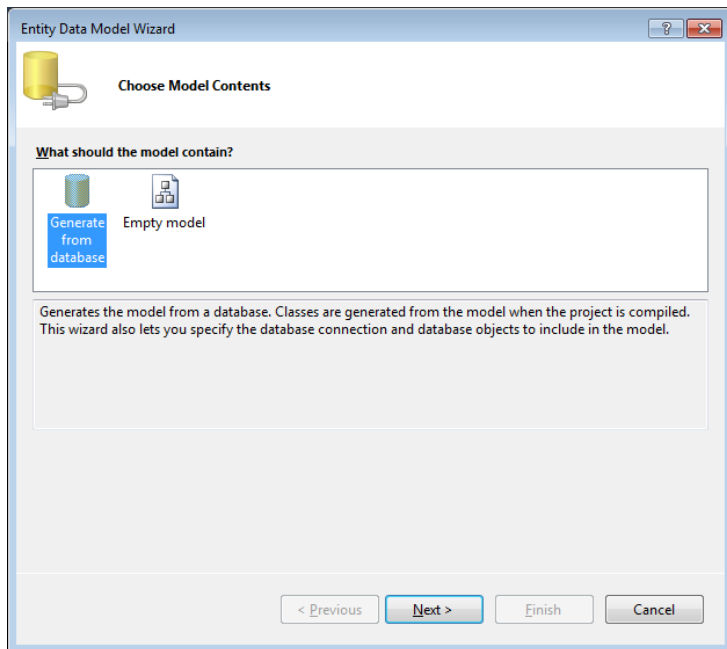
Now we'll add our database file. The database file is named MvcMusicStore.mdf and is included in the project downloads. Right-click the new App_Data folder, select Add⇒Existing Item... and browse to MvcMusicStore.mdf.



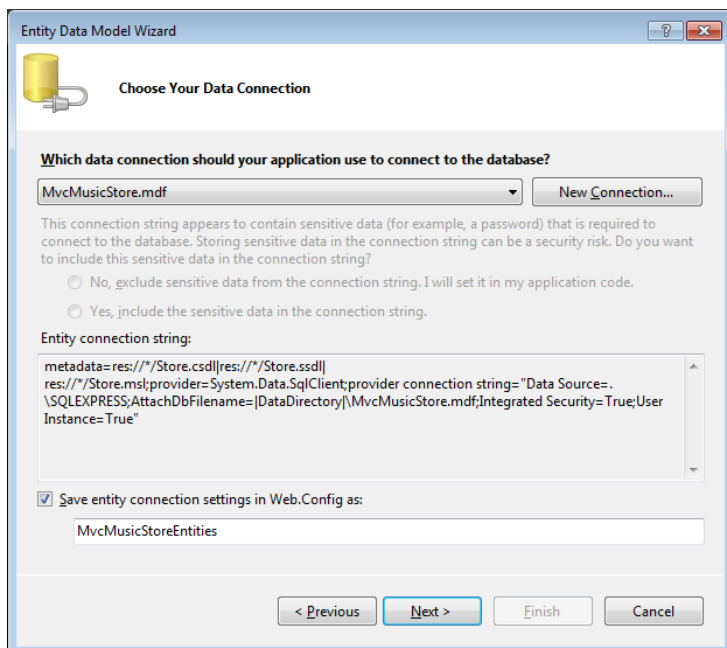
Now that the database has been added to the project, we'll need code to read and write to the database. We'll use Entity Framework data model to handle that. Right-click the Models directory and select Add⇒New Item⇒Data⇒ADO.NET Entity Data Model. Name the data model StoreDB.edmx and press the Add button.



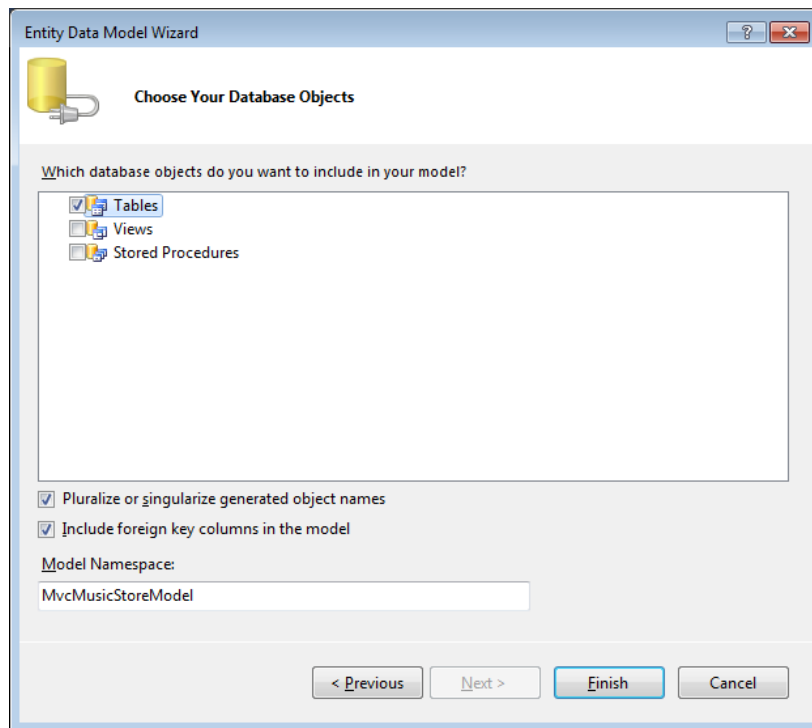
The Entity Data Model Wizard first asks if you want to generate a model from a database or create an empty model. Select “Generate from database” and click the Next button.



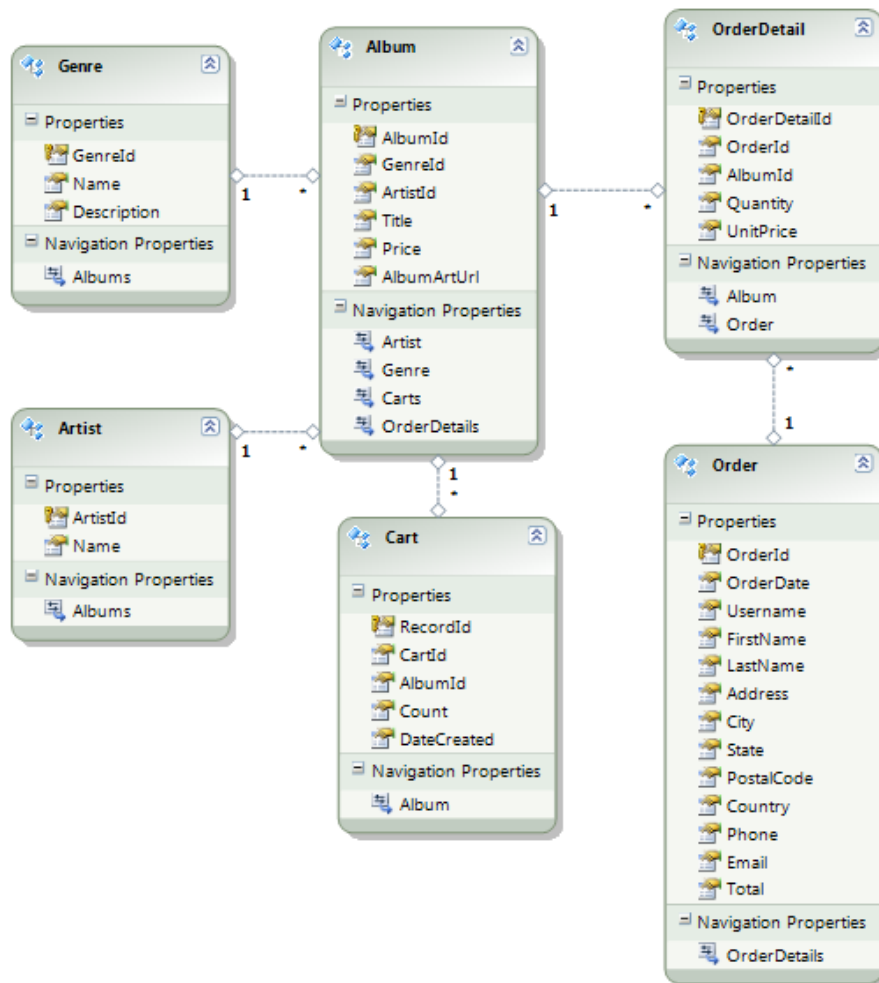
Since we're generating our model from a database, we'll need to specify which database we want to use. The wizard's smart enough to see that we've got a database in our App_Data folder, so it fills in the correct connection information for that database for us, and we can just click the Next button.



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.



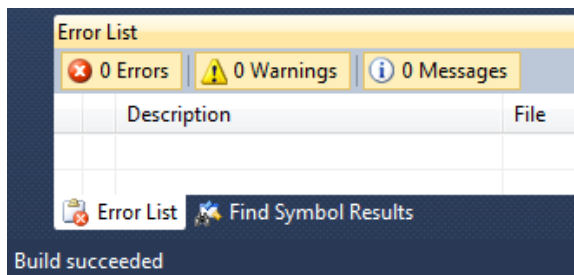
This brings up an entity diagram for our database.



On the music side of things, we're adding Artist to the Genre and Albums concepts we already looked at in the last chapter. Since we're running a store, we're also adding tables for Cart, Order, and OrderDetails.

Now that we have a real data model in place, we can delete the Album and Genre model classes we've been working with. Expand the Models directory, select Album.cs, and press the Delete key. Follow the same steps to delete Genre.cs.

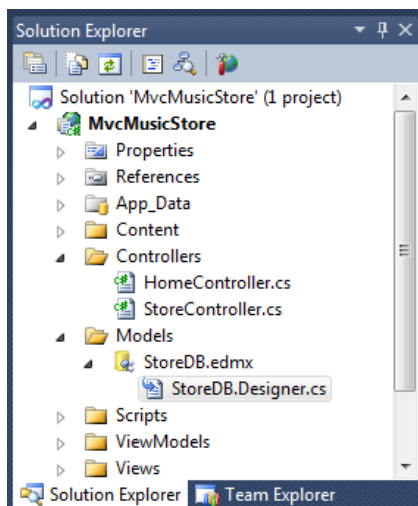
Despite having deleted the Albums and Genre class, the project still builds and the pages still work.



Why?

Since our database tables have fields which include all the properties we were using in our Album and Genre classes earlier, 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.



To demonstrate that (and because it'll make our code look nicer), we'll rename the object context class from MvcMusicStoreEntities to MusicStoreEntities. Right-click on the class name (MvcMusicStoreEntities) and select Refactor⇒Rename.

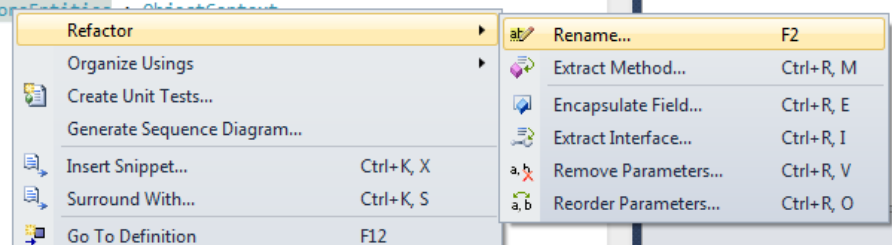
```

/// <summary>
/// No Metadata Documentation available.
/// </summary>
public partial class MvcMusicStoreEntities : DbContext
{
    Constructors

    Partial Methods

    ObjectSet Properties
    AddTo Methods
}

```



Change the class name to MusicStoreEntities and press the Apply button.

Querying the Database

Now let's change our Store controller actions to call into our database. 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 exposes collection class for each table in our database, so we can query our Genre table using LINQ (language integrated query) query expressions. First we'll update our Store Index page to read all Genre names in our database, as shown below:

```
//
// 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.

Browse Genres

Select from 10 genres:

- [Rock](#)
- [Jazz](#)
- [Metal](#)
- [Alternative](#)
- [Disco](#)
- [Blues](#)
- [Latin](#)
- [Reggae](#)
- [Pop](#)
- [Classical](#)

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

For the Store Browse, we'll demonstrate an alternative to the LINQ query expression syntax we just looked at - LINQ Extension Methods. These both do the same thing under the hood, so you can use whichever syntax seems more natural for a particular query.

In this controller action, 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, so we'll use the Single extension method on the Genre object set, like this:

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

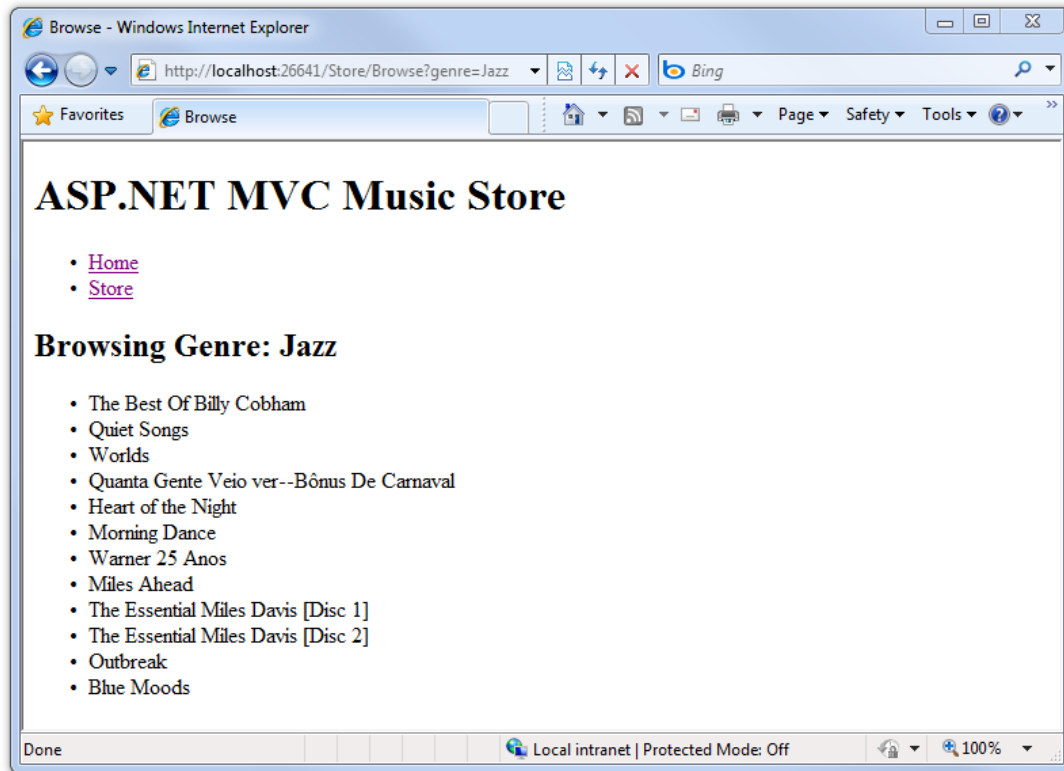
The Single method takes a Lambda expression as a parameter, which specifies that we want a single Genre object such that its name matches the value we've defined. In the case above, we'd be loading a single Genre object with a Name value matching Disco.

We'll take advantage of an Entity Framework feature that allows us to specify other related entities we want loaded as well, called Query Result Shaping. We want to load the Albums for the matching Genre, so we'll query 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 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.



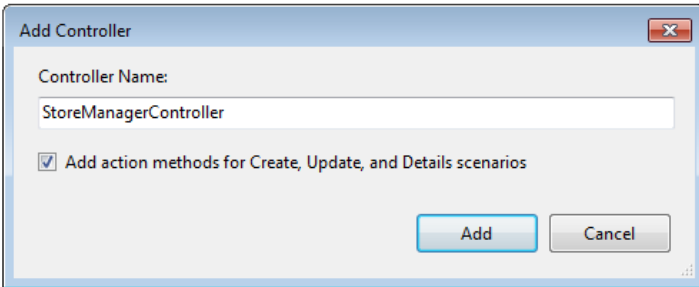
We'll make the same change to our Store Details page, replacing our dummy data with a database query which loads an Album whose ID matches the parameter value.

```
//  
// GET: /Store/Details/5  
  
public ActionResult Details(int id)  
{  
    var album = storeDB.Albums.Single(a => a.AlbumId == id);  
  
    return View(album);  
}
```

Edit forms and Validation

In the past chapter, we were loading data from our database for display. In this chapter, we'll be editing data.

We'll create a new controller called `StoreManagerController`. This controller will be handling Create and Update actions, so we'll check the checkbox to "Add action methods for Create, Update, and Details scenarios."



This generates a controller with stub methods for the controller actions we'll be building, 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 action, so we can delete it. 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.

```

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

```

Now we can get started with the Store Manager Index page. This page will display a list of albums, so the controller action logic will be pretty similar to the Store Index controller action. We'll use LINQ extension methods to retrieve all albums, including Genre and Artist information for display.

```

//
// GET: /StoreManager/

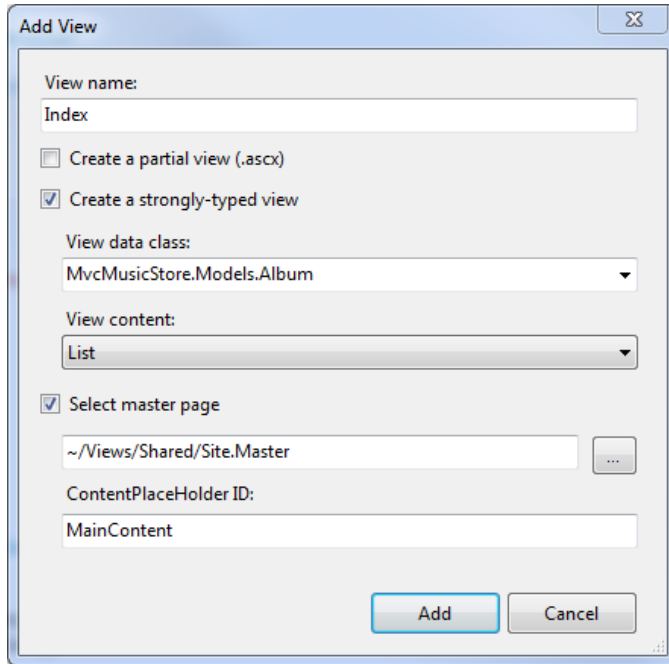
public ActionResult Index()
{
    var albums = storeDB.Albums
        .Include("Genre").Include("Artist")
        .ToList();

    return View(storeDB.Albums);
}

```

}

Now we can create the view. As before, we'll right-click on the controller action code to bring up the Add View dialog. This view will use the List template and will be strongly-typed to our Album class, as shown below.



Scaffold View templates

We're making use of the *scaffold* View template feature to quickly generate a simple View template which lists all fields in our Album model. It's a quick way to get started on a strongly typed view. Rather than having to add in all the fields by hand, we can just modify the generated code.

Let's look at the generated list of fields. Oh, and yes, this is using an HTML table. Remember that HTML tables, while a poor choice for site layout, are a perfect fit for displaying tabular data.

```
<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:

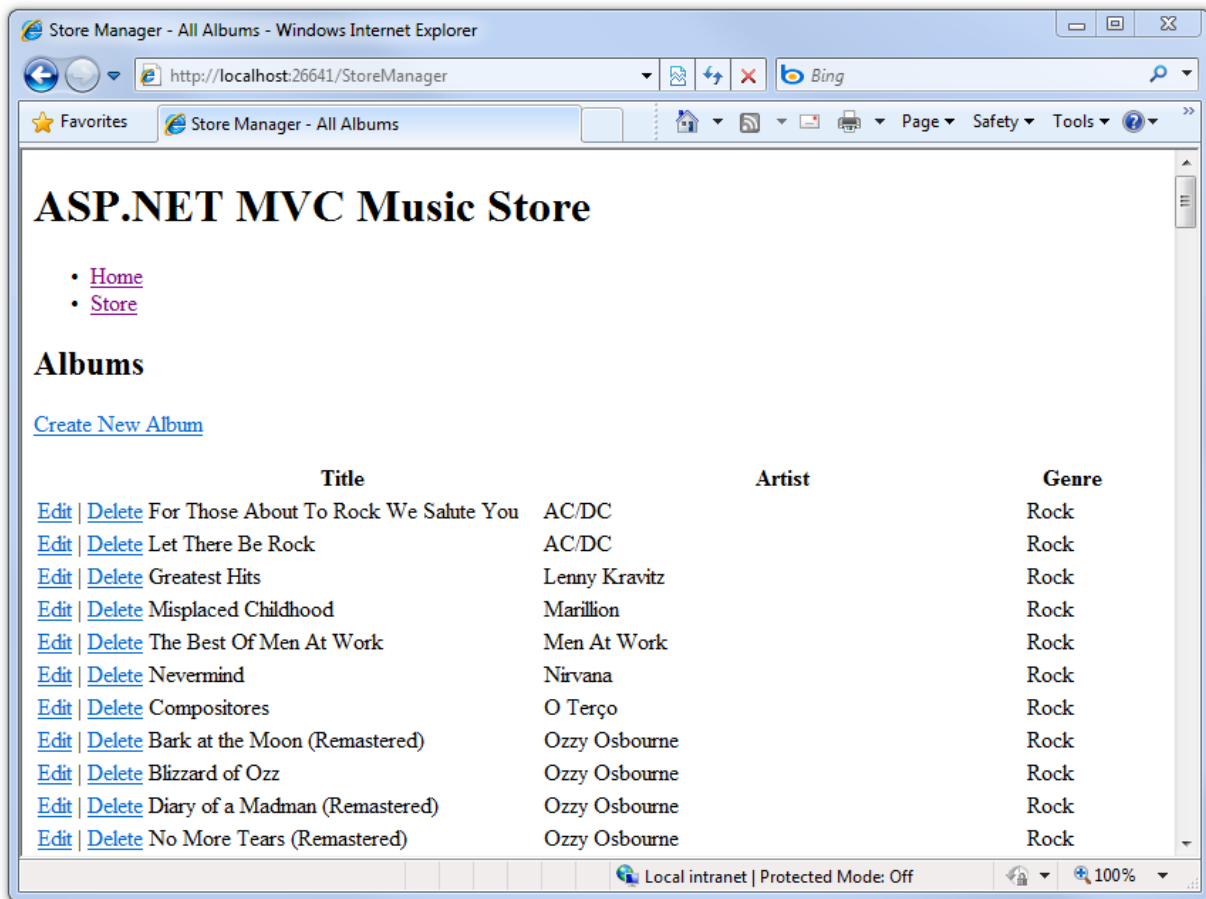
```

<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.

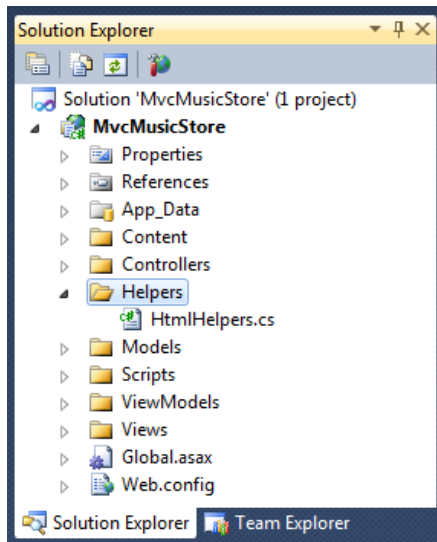


Using a custom HTML Helper to truncate text

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 quick HTML Helper to allow us to easily truncate these and other properties in our Views.

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 be extending `System.Web.Mvc.HtmlHelpers`, so we'll need to add a using statement referencing `System.Web.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. Now we need to let our view know about it by importing the namespace. We do that by using an `<%@ Import` statement, directly below the `<%@ Page %>` element, like this:

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

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


```

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

<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>

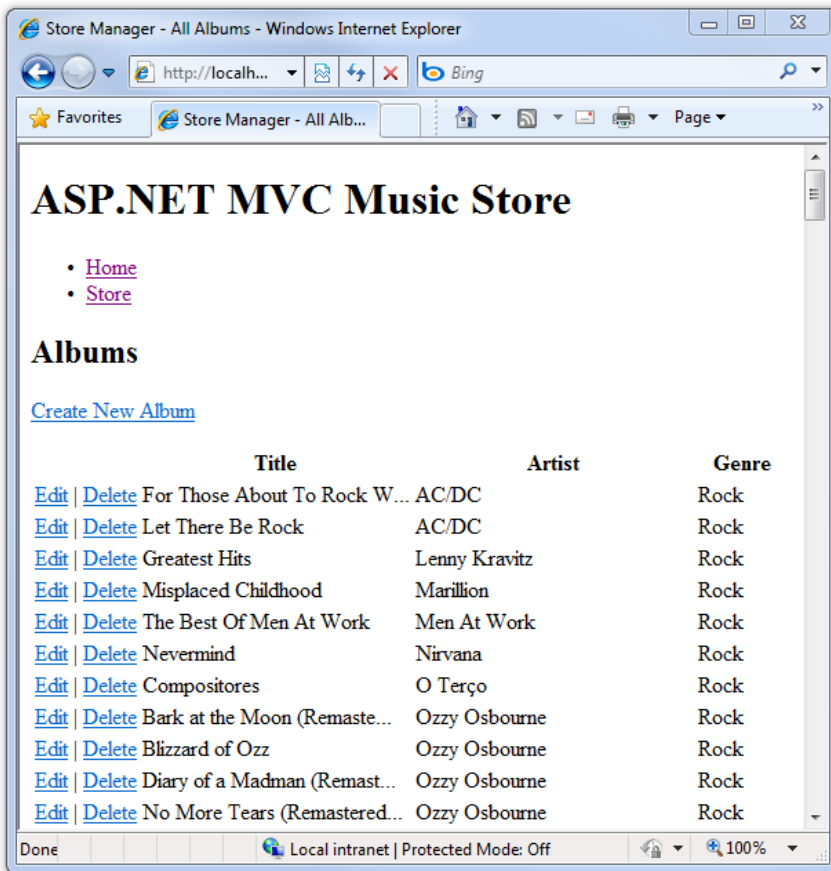
        <% } %>

    </table>

</asp:Content>

```

Now when we view the Store Manager Index, the albums and titles are kept below our maximum lengths.



Creating the Edit View

Next we'll create a form to allow editing an album details. It will include text fields for Album Title, Price, and Album Art URL. Additionally, we'll be building dropdowns to allow selecting the Artist and Genre.

Edit Album

Edit Album

Title

Price

Album Art URL

Artist

Genre

Creating the StoreManagerViewModel

Editing an Album will require selecting an Artist and a Genre, so we'll create a ViewModel. We can reuse this ViewModel for the Create controller action as well, so we'll call this the StoreManagerViewModel.

Create the StoreManagerViewModel class in the ViewModels folder and the following three properties.

```
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; }
    }
}
```

Writing HTTP-GET Edit Controller Action

We'll need two controller actions for our Edit form. When the form is first loaded, we'll be responding to an HTTP-GET request and displaying the Album information. When the user makes a change and clicks the Save button, the form information will be submitted to another controller action that handles the form HTTP-POST request.

The HTTP-GET Edit action method retrieves an album by ID, similar to the Store Details action method. We also need a list of Genres and Artists to fill the available values for the dropdown lists, as shown below.

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

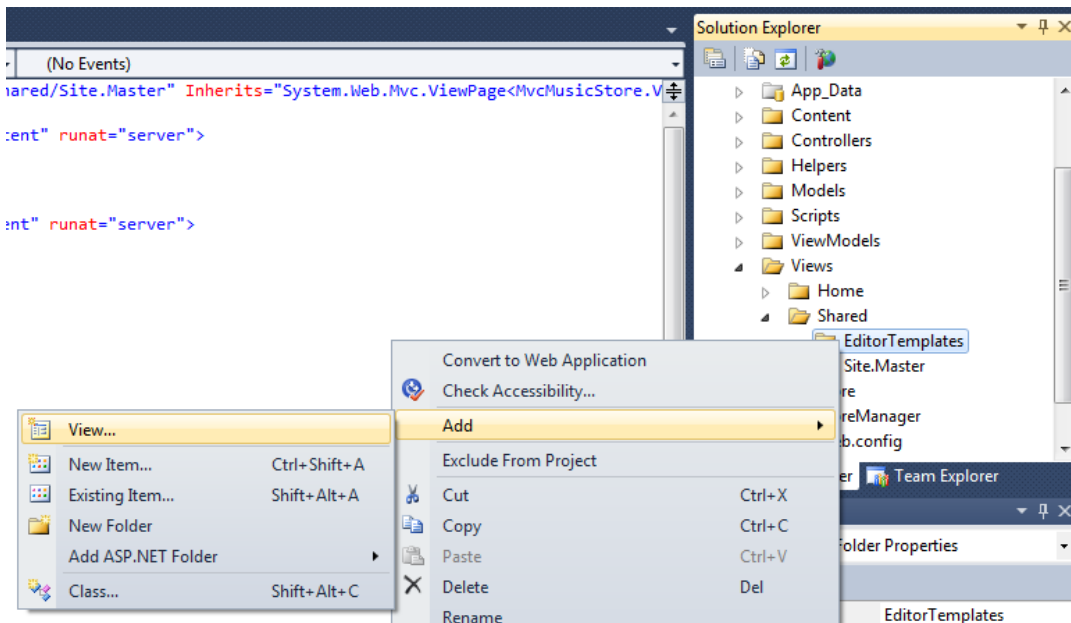
public ActionResult Edit(int id)
{
    var viewModel = new StoreManagerViewModel
    {
        Album = storeDB.Albums.Single(a => a.AlbumId == id),
        Genres = storeDB.Genres.ToList(),
        Artists = storeDB.Artists.ToList()
    };

    return View(viewModel);
}
```

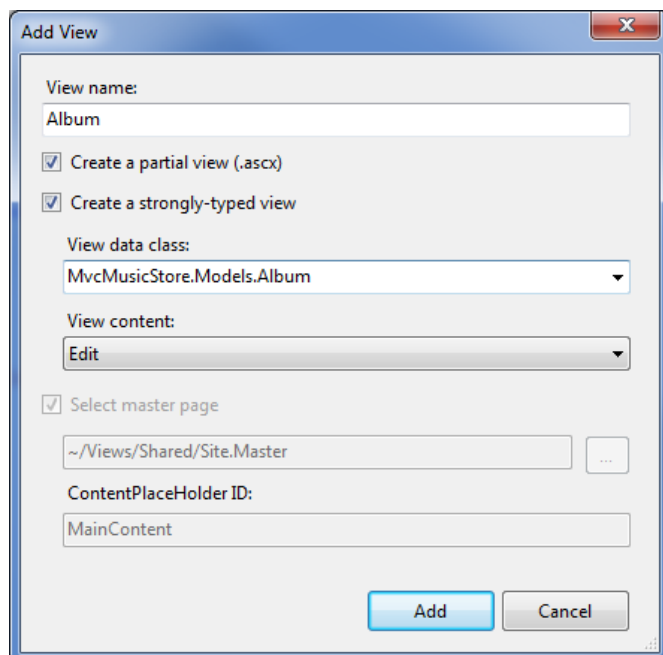
Creating a Shared Album Editor Template

The exact same form fields for Album Edit will be needed to handle the Album Create case, so we'll create a Shared Album Editor Template.

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.



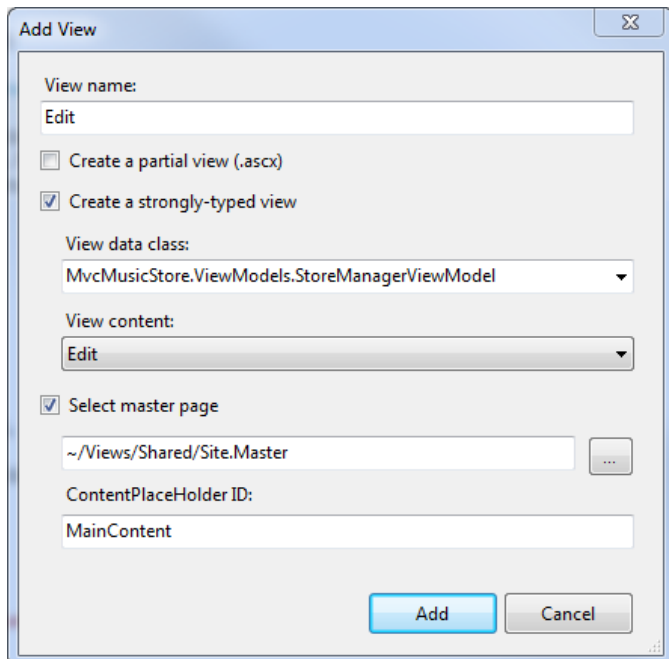
This view will be a little different than the views we've been creating so far. It will be a Partial View, meaning that it's intended to be displayed inside another view. Name the view Album, select Album for the View data class, set the View content template to Edit, and press the Add button.



We'll come back and customize this in a minute, but first we need to include it in the Edit view so we can view it.

Creating the Edit View

Create the Album Edit view by right-clicking on the Store Manager HTTP-GET Edit action method and selecting Add⇒View. The Edit form should be strongly typed to the StoreManagerViewModel and should use the Edit content template.



The generated Edit View template doesn't include any fields, because none of the properties in our StoreManagerViewModel were simple types like strings and integers. That's fine; we'll use our Album editor template.

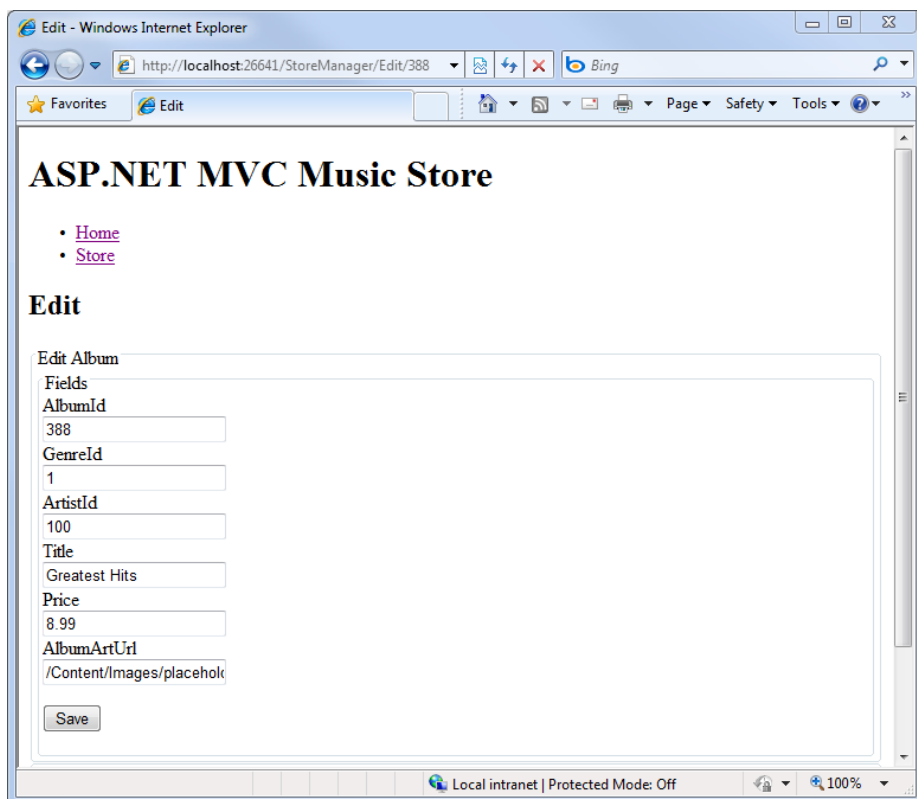
The `Html.EditorFor` helper method allows us to specify that we want an Editor for a specific type. In this case, we want an editor for our Album, so pass that in as the first parameter. We're using an overload that allows us to pass in our Artist and Genre lists for the dropdowns, but we'll need to do a little more work on the Album editor template to hook those up. The completed Edit form code appears below.

```
<% 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>

    <% } %>
```

Now our Edit form will display the Album editor template.

The screenshot shows a web browser window titled "Edit - Windows Internet Explorer". The address bar shows "http://localhost:26641/StoreManager/Edit/388". The page has a blue header with the title "ASP.NET MVC Music Store". Below the header are two links: "Home" and "Store". The main content area is titled "Edit" and contains a form titled "Edit Album". The form has several input fields: "AlbumId" (388), "GenreId" (1), "ArtistId" (100), "Title" (Greatest Hits), "Price" (8.99), and "AlbumArtUrl" (/Content/Images/placeholder). There is a "Save" button at the bottom of the form. The status bar at the bottom of the browser shows "Local intranet | Protected Mode: Off" and "100%".

Implementing Dropdowns on the Album Editor Template

We'll 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:

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

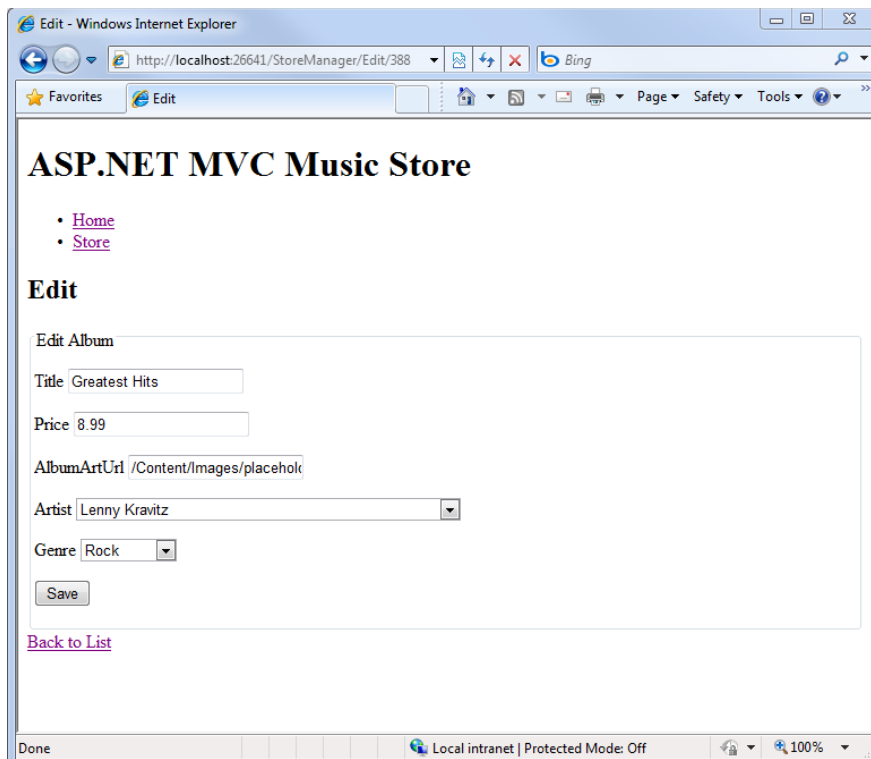
```

<%@ 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

The next step is handling form submissions. We'll do that by using an Edit action method which takes an ID (read from the route parameter values) and a FormCollection (read from the HTML Form).

This controller action method has three main steps:

1. Load the existing album from the database by the ID passed in the route parameter values
2. Try to update the Model using the values in the FormCollection, using the Controller's built-in UpdateModel method
3. Display the results to the user – either redisplay the form in case of an error, or redirect back to the list of albums in case of a successful update

```
//  
// 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  
    {  
        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.

Adding the Create View

Since we've created a shared editor template for the Album type, the create scenario is really easy. First we'll create a controller action method to display a new (empty) Album.

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



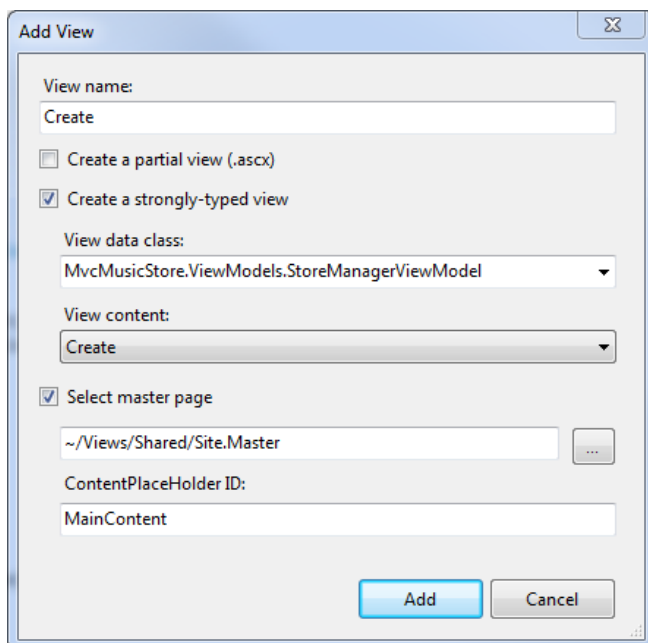
```

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

    return View(viewModel);
}

```

Next we right-click on that method and add a Create View that is strongly-typed to the StoreManagerViewModel.



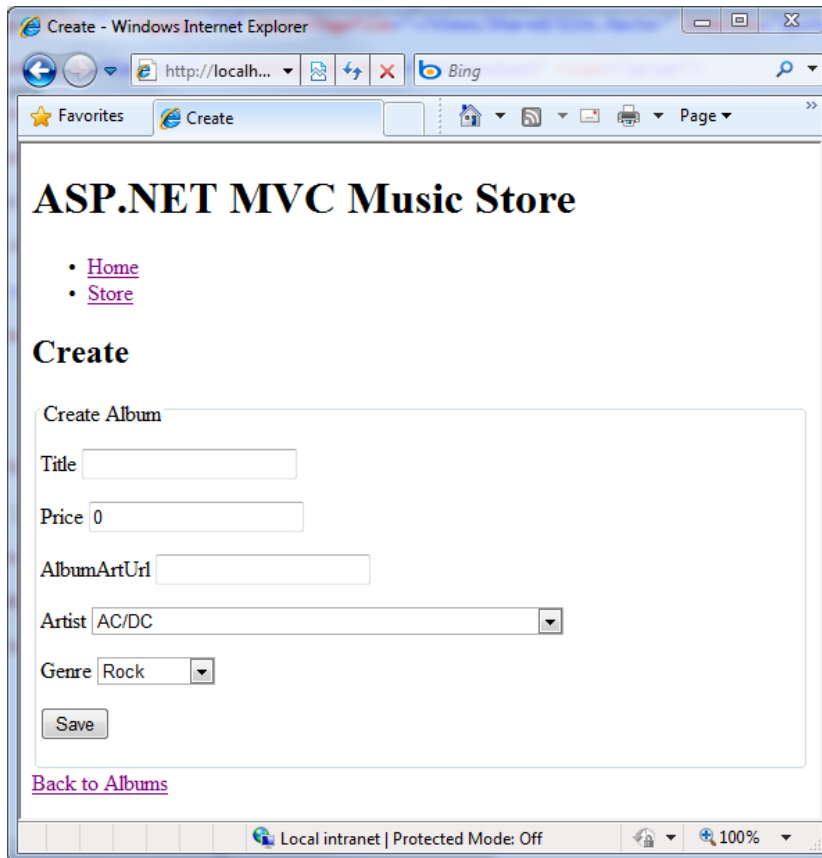
We need to add in our call to `Html.EditorFor`, just as in the Store Manager Edit form above.

```

<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>

```

Now the Create form looks just like the Edit form, but with default values.



And finally, we need to write our HTTP-POST Create controller action method, which follows the same pattern as our HTTP-POST Edit controller action method.

```
//  
// POST: /StoreManager/Create  
  
[HttpPost]  
public ActionResult Create(Album album)  
{  
    try  
    {  
        //Save Album  
        storeDB.AddToAlbums(album);  
        storeDB.SaveChanges();  
  
        return Redirect("/");  
    }  
    catch  
    {  
        //Invalid - redisplay with errors  
  
        var viewModel = new StoreManagerViewModel  
        {  
            Album = album,  
            Genres = storeDB.Genres.ToList(),  
            Artists = storeDB.Artists.ToList()  
        };  
    }  
}
```

```

        return View(viewModel);
    }
}

```

There's only one real difference here – instead of loading an existing Album and calling UpdateModel, we'll just accept an Album as the Action Method parameter and add it to the Albums list before saving.

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.

```

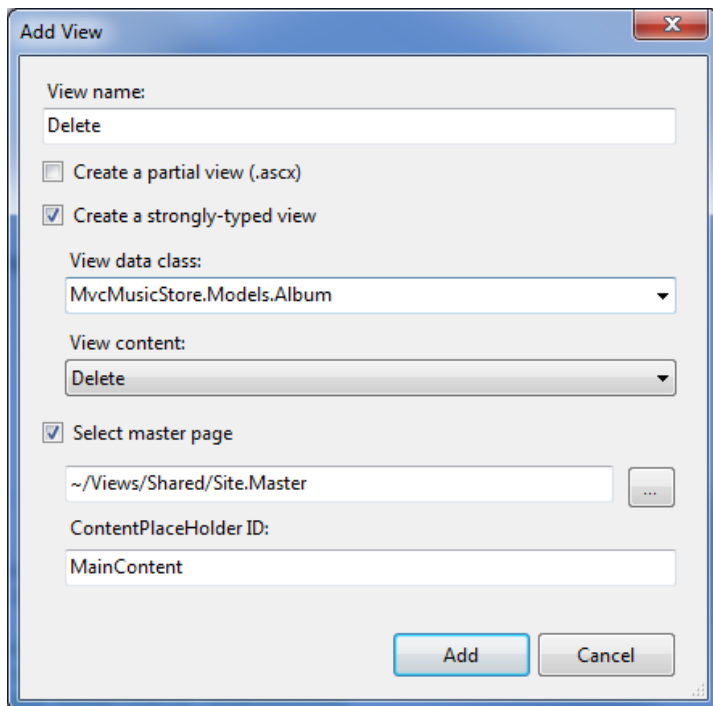
//
// GET: /StoreManager/Delete/5

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.

```

<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

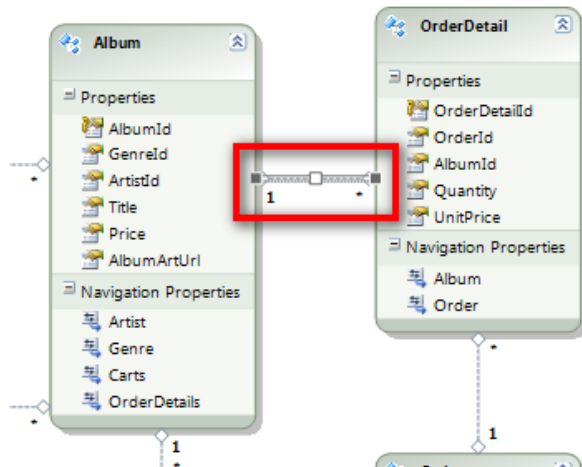
    storeDB.DeleteObject(album);
    storeDB.SaveChanges();

    return View("Deleted");
}

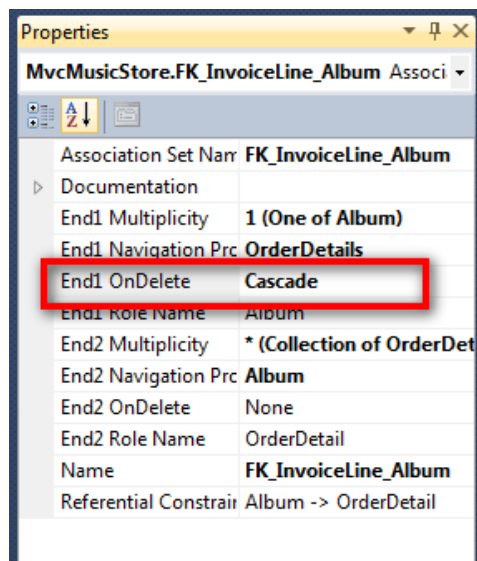
```

We'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.



Now in the properties window, set the OnDelete value for the associated table to Cascade.



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.DataAnnotations, 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

We can handle our validation by adding an Album class to the Models folder with the following code:

```
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; }

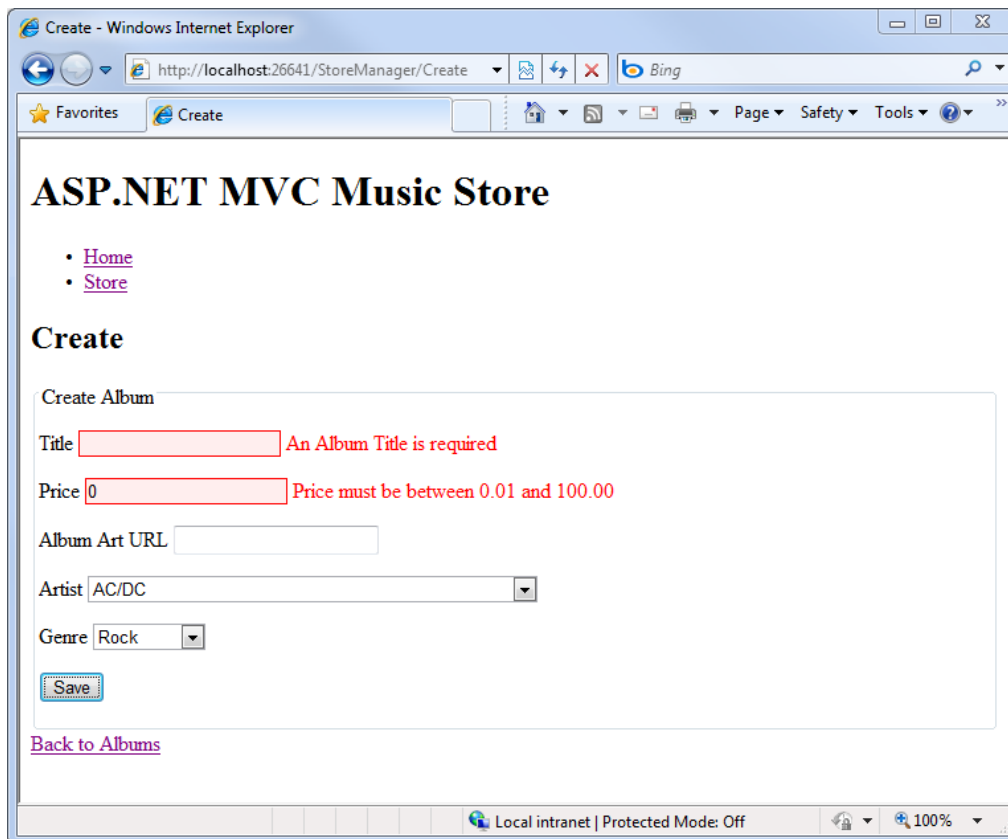
            [DisplayName("Artist")]
            public object ArtistId { get; set; }

            [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).



Using Client-Side Validation

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.

We'll add the scripts 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.

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

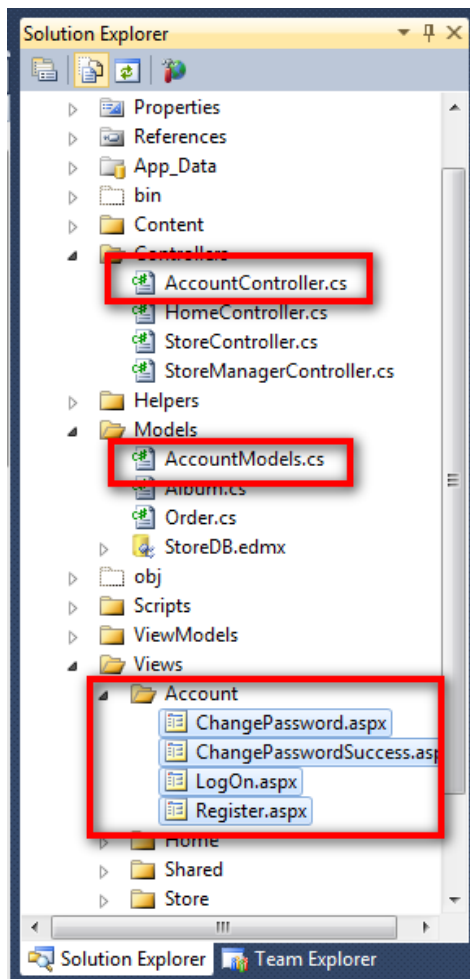
One difference between the full ASP.NET MVC 2 Web Application template and the ASP.NET MVC 2 Empty Web Application template is that the empty template doesn't include an Account Controller. We'll add an Account Controller by copying a few files from a new ASP.NET MVC application created from the full 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.

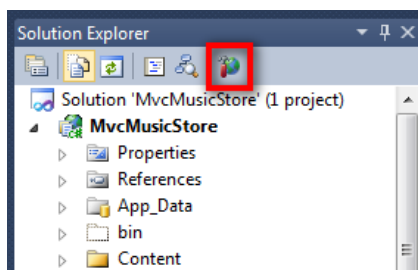
The updated solution should look like the following:



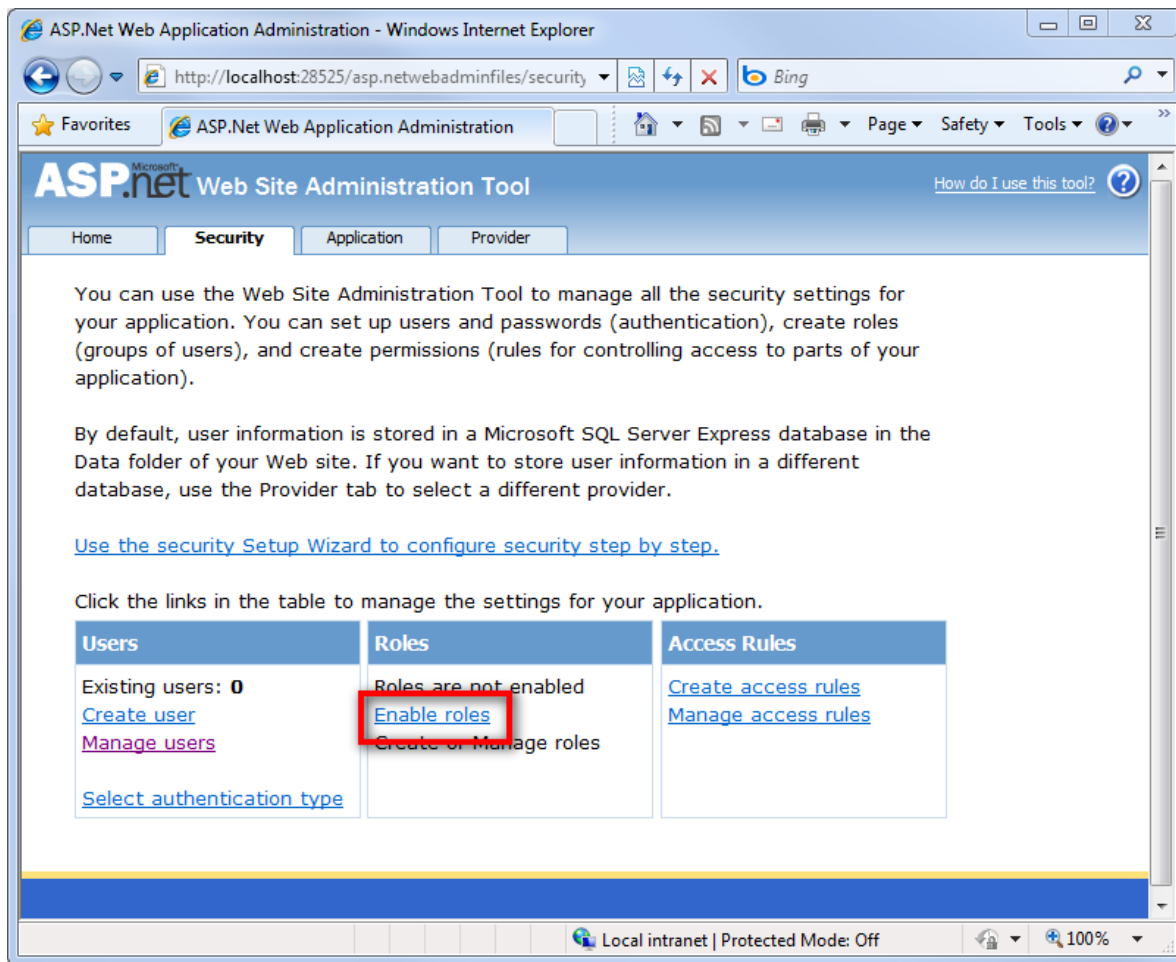
Adding an Administrative User with the ASP.NET Configuration site

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.

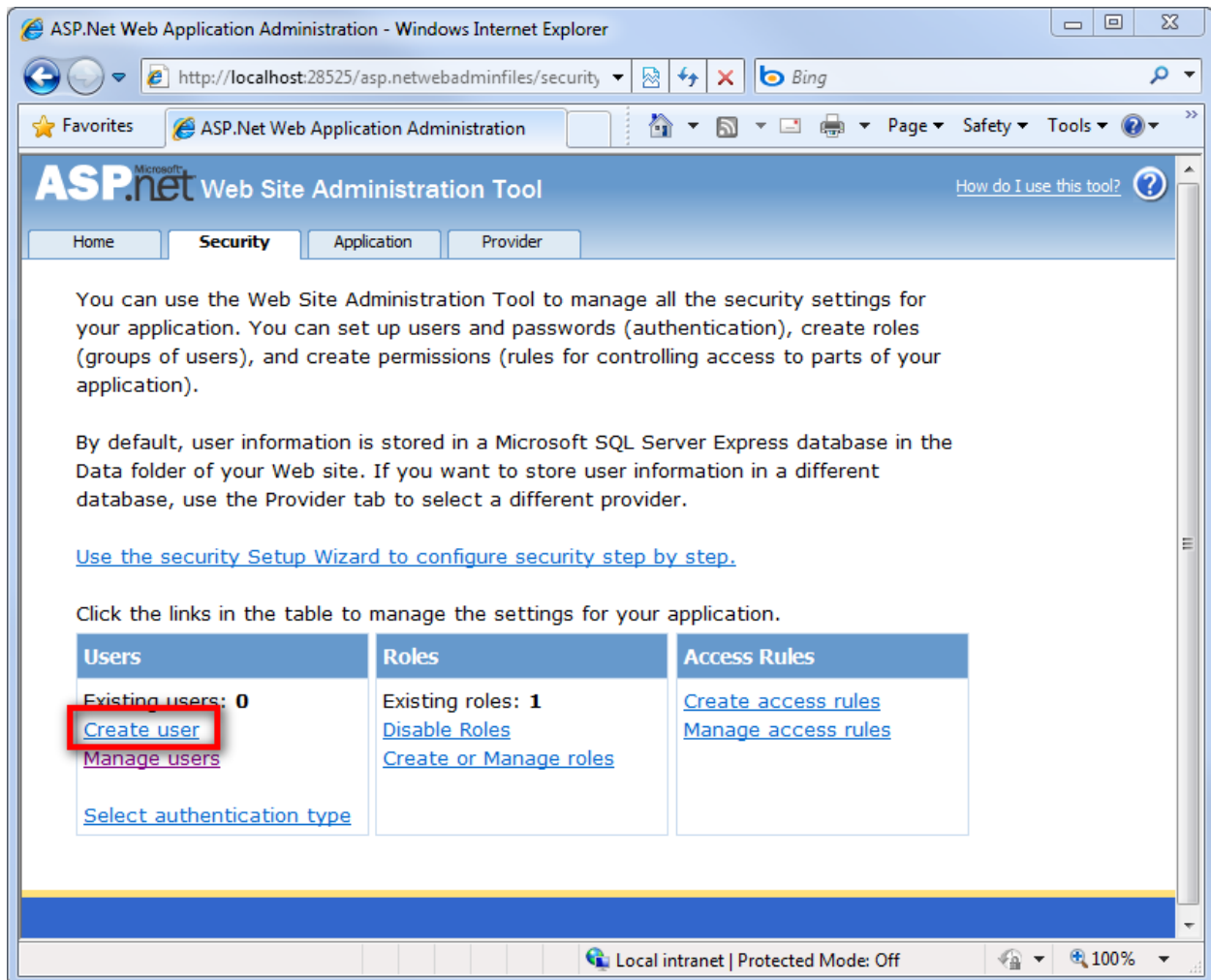
Launch the ASP.NET Configuration website by clicking following the icon in the Solution Explorer.



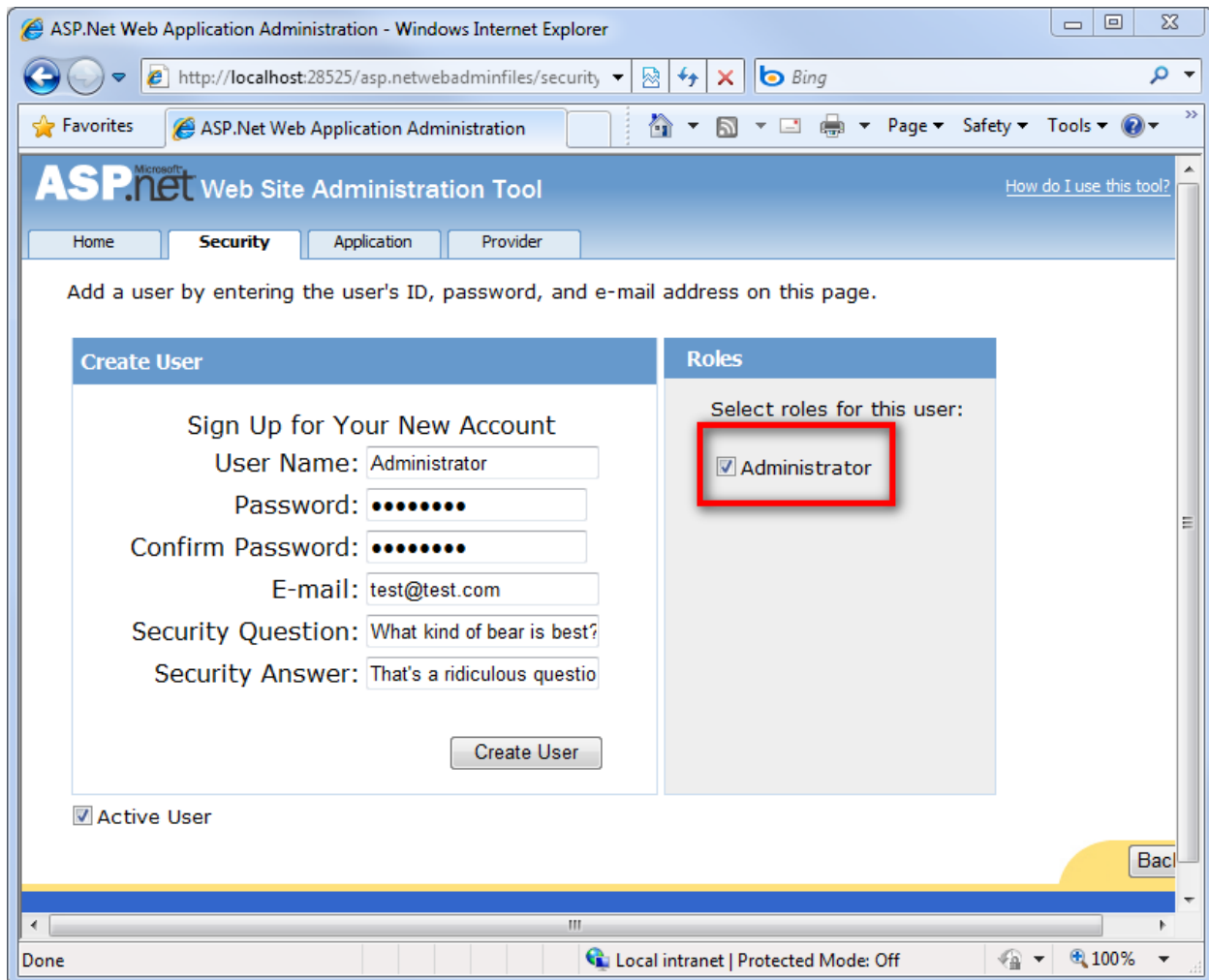
This launches a configuration website. Click on the Security tab on the home screen, then click the "Enable roles" link in the center of the screen.



Next, click on the Create user link on the left side.



Fill in the user information fields on the left, 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 close the form.

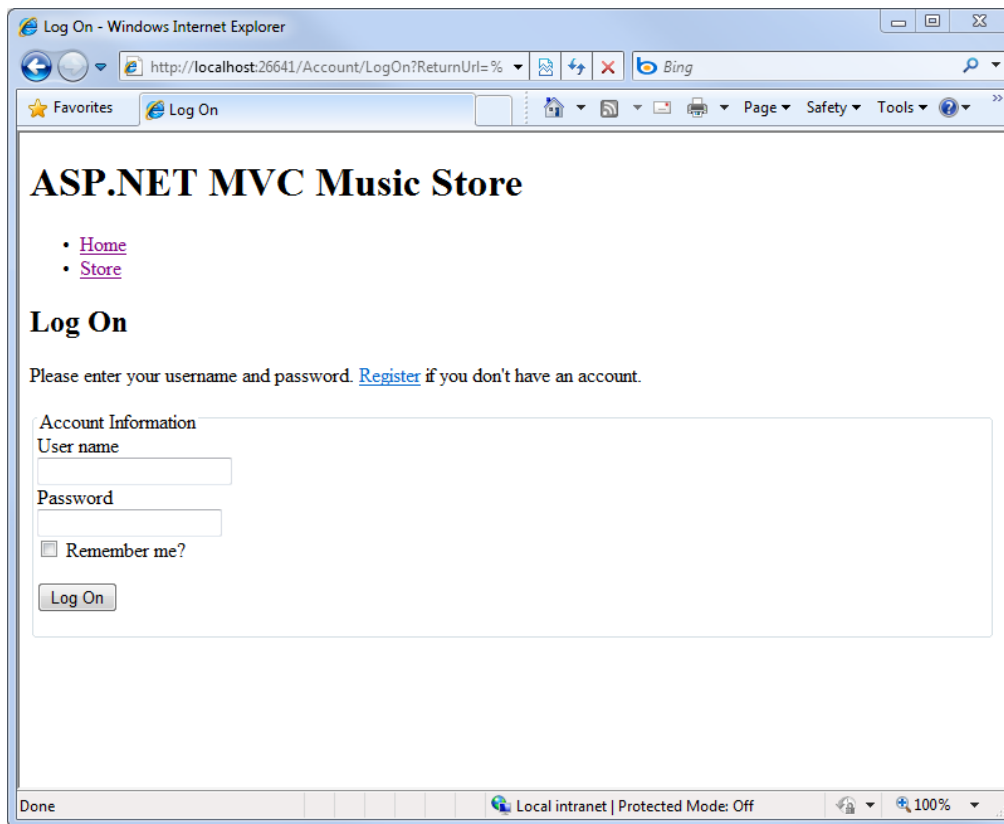
Role-based Authorization

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 the Store Manager 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.

Registration and Checkout

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.

First we'll create the ShoppingCartViewModel class. 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; }
    }
}
```

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, but as well as handles 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.IsNullOrEmpty(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 two additional classes: a

ShoppingCartViewModel and a ShoppingCart model class. As in the StoreController and StoreManagerController, we'll add a field to hold an instance of MusicStoreEntities.

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

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

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 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

```
<%@ 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>

```

Showing the Cart summary in the header with RenderPartial

We want to expose the number of items in the user's shopping cart across the entire site, and we can do that by using a PartialView.

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

```

//
// GET: /ShoppingCart/CartSummary

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

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

    return PartialView("CartSummary");
}

```

We can then include the cart summary in our Site master using the Html.RenderAction method:

```

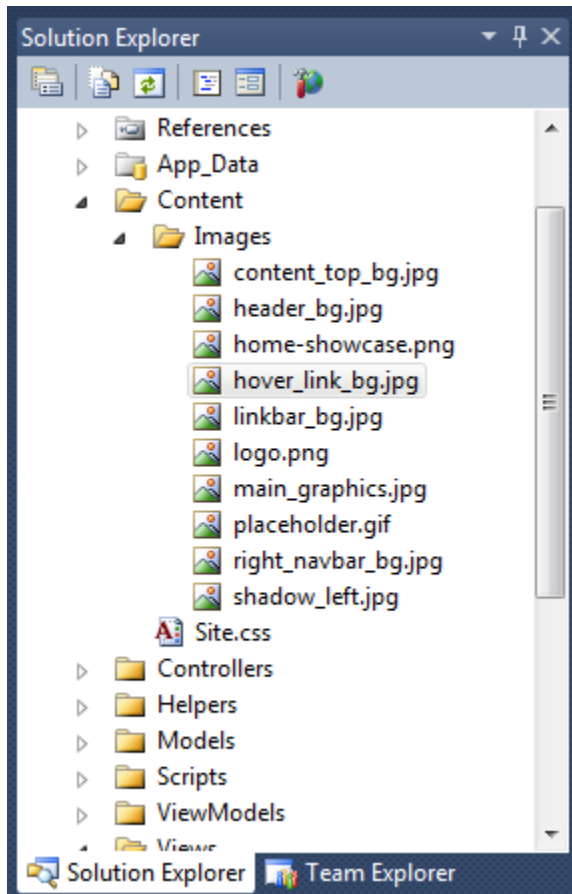
<% Html.RenderAction("CartSummary", "ShoppingCart"); %>

```

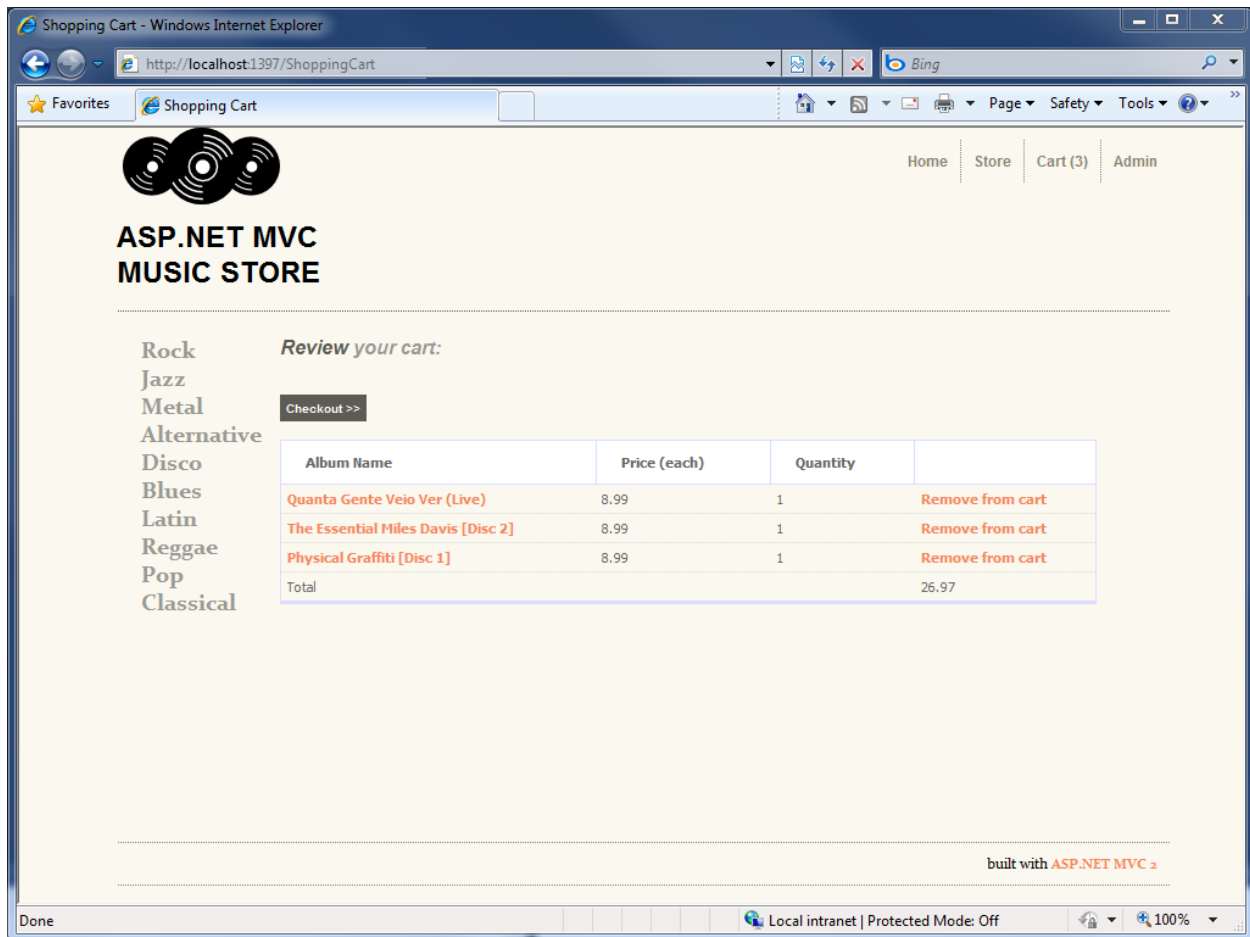
RenderAction requires us to specify the Action Name (“CartSummary”) and the Controller Name (“ShoppingCart”).

Adding in Site style and Conclusion

Since we started by adding a Site master page and built our site using clean markup, we can easily update the site look and feel. We’ll update the site appearance by copying the Site.css and Images folder, both in the Content folder, with the completed site design files included in the download.



Now browsing the site shows our update design:



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!