目录

[ASP.NET MVC 音乐商店–0.概览 2](#_Toc323399544)

[ASP.NET MVC 音乐商店 - 1 创建项目 7](#_Toc323399545)

[ASP.NET MVC 音乐商店 - 2.控制器 10](#_Toc323399546)

[ASP.NET MVC 音乐商店 - 3. 视图与模型 19](#_Toc323399547)

[ASP.NET MVC 音乐商店 - 4. 数据访问 41](#_Toc323399552)

[ASP.NET MVC 音乐商店 - 5. 通过支架创建编辑表单 54](#_Toc323399566)

[ASP.NET MVC 音乐商店 – 5.通过支架创建编辑表单 续 70](#_Toc323399570)

[ASP.NET MVC 音乐商店 - 6. 使用 DataAnnotations 进行模型验证 88](#_Toc323399584)

[ASP.NET MVC 音乐商店 - 7.成员管理和授权 93](#_Toc323399587)

[ASP.NET MVC 音乐商店 - 8. 使用 Ajax 更新的购物车 101](#_Toc323399591)

[ASP.NET MVC 音乐商店 - 9. 注册和结账 120](#_Toc323399597)

[ASP.NET MVC 音乐商店 - 10. 完成导航和站点的设计 135](#_Toc323399604)

# ASP.NET MVC 音乐商店–0.概览

MVC 音乐店是用来介绍和展示使用 ASP.NETMVC 以及 Visual Studio 进行 Web 开发的示例应用程序，我们将从基础知识开始，一步一步地进行介绍，因此，初学者也可以去的很好的学习体验。

这个示例程序使用 ASP.NET MVC3 完成，视图部分使用 **Razor** 引擎，数据库访问使用 EF CodeFirst。

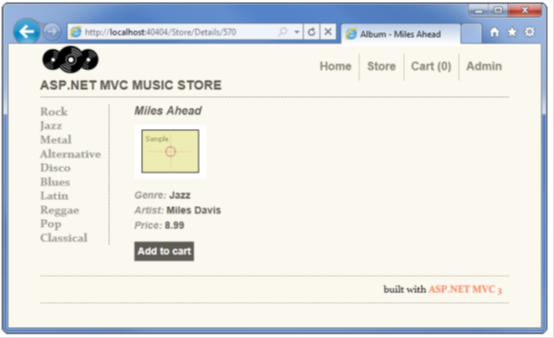
这个程序将会创建一个音乐商店，这个程序包括三个主要的部分：购物，结账和管理。



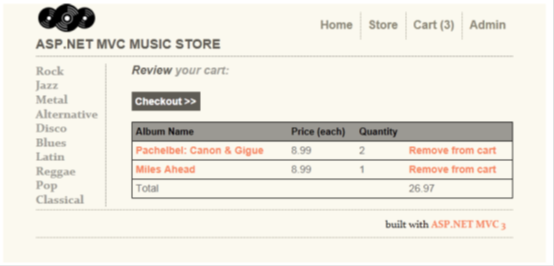
用户可以通过分类浏览唱片。



可以查看某张唱片，然后加入到购物车中。



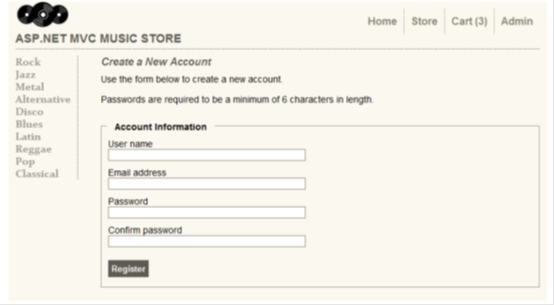
可以查看购物车，删除其中不再希望购买的项目。



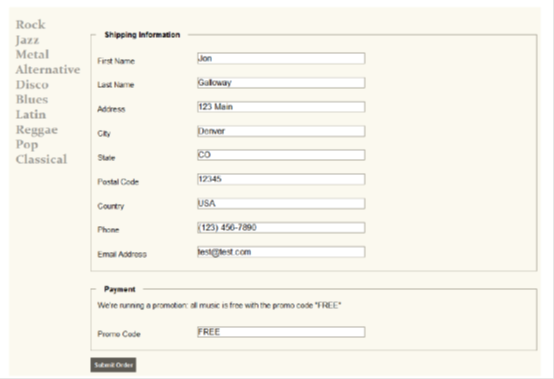
在结账的时候，就会提示用户登录，或者注册一个新的账号。



创建新帐号



创建账号之后，用户可以填写完整的送货和结账信息，为了让这个程序简单一些，如果你在优惠券号中输入 FREE 的话，我们会提示一切免费。



下完订单之后，用户会看到一个简单的确认提示。



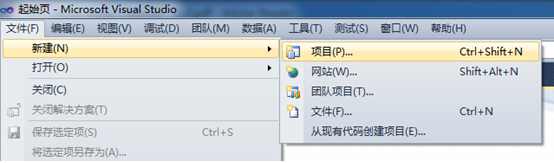
另外，我们还会提供一个管理员用来创建、修改、以及删除唱片的功能。



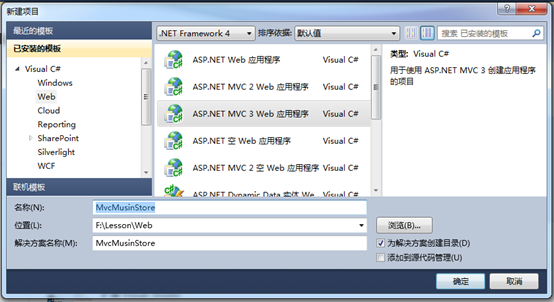
这个项目我们将使用 ASP.NET MVC3 和 Visual Studio 完成。其中的知识涉及到数据访问，表单的提交，数据验证，母版页，布局，以及 AJAX 的更新和验证，用户的登录等等。

# ASP.NET MVC 音乐商店 - 1 创建项目

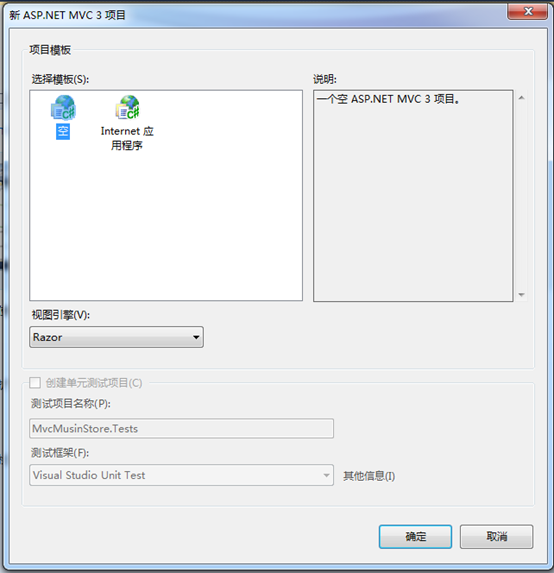
我们的项目从在 Visual Studio 中的文件菜单中选择“新建”，选择“项目”开始。



然后，选择 C# 中的 Web 模板组，在右边的项目模板中选择 ASP.NET MVC3 Web 应用程序，在项目的名称输入框中，输入 MvcMusicStore ，点击确定。

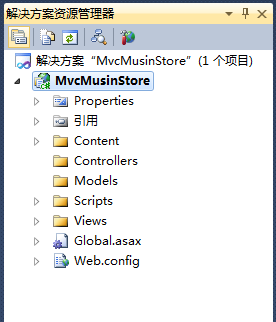


这时，你会到第二个对话框，允许我们设置这个项目关于 MVC 的一些设置，确认选中了“空”项目模板，视图引擎选中 Razor ，点击确定。



这样我们的项目就创建成功了！

我们看一下在这个项目都创建了哪些内容。



在 ASP.NET MVC 中使用了下面的一些基本的命名约定

|  |  |
| --- | --- |
| 文件夹 | 功能 |
| /Controllers | 控制器接受来自浏览器的请求，进行处理，然后向用户返回回应 |
| /Views | 视图文件夹保存用户界面的模板 |
| /Models | 这个文件夹定义处理的数据 |
| /Content | 我们的图片，CSS以及其他任何的静态内容放在这里 |
| /Scripts | 放置脚本文件 |
| /App\_Data | 数据库文件 |

这些文件夹在一个空的 ASP.NET MVC 应用中也会存在，因为 ASP.NET MVC 的框架默认使用“约定胜于配置”的原则，已经假定这些文件夹有着特定的用途。例如，控制器将会在 Views 文件夹中寻找相应的视图，而不需要你在代码中显式设置，这样可以节省大量的编程工作，也可以使其他的开发人员更加容易理解你的程序。在创建这个程序的过程 中，我们将会详细的说明这些约定。

# ASP.NET MVC 音乐商店 - 2.控制器

在典型的 Web 应用中，用户请求的 URL 地址通常映射到保存在网站中的文件上，例如，当用户请求 /Products.aspx 的时候，或者 /Products.php 的时候，很可能是在通过处理 Products.aspx 或者 Products.php 文件来完成任务。

ASP.NET MVC 的处理方式则不同，它没有映射到文件上，相反，将这些 URL 地址映射到类的方法上，这些类被称为“控制器”，控制器用来接受 HTTP 请求，处理用户的输入，获取或者保存数据，其中的处理方法称为 Action，然后将回应发送到客户端，可能是显示了一个 HTML 的网页，下载一个文件，重定向到另外一个地址等等。

**默认的路由配置**

在新创建的项目中，打开 Global.asax.cs 文件，可以看到如下的代码。

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

using System.Web.Routing;

namespace MvcMusicStore

{  
 // 注意: 有关启用 IIS6 或 IIS7 经典模式的说明，  
 // 请访问 http://go.microsoft.com/?LinkId=9394801  
 public class MvcApplication : System.Web.HttpApplication  
 {  
 public static void RegisterGlobalFilters(GlobalFilterCollection filters)  
 {  
 filters.Add(new HandleErrorAttribute());  
 }

public static void RegisterRoutes(RouteCollection routes)  
 {  
 routes.IgnoreRoute("{resource}.axd/{\*pathInfo}");

**routes.MapRoute(**  
 **"Default", // 路由名称  
 "{controller}/{action}/{id}", // 带有参数的 URL  
 new { controller = "Home", action = "Index", id = UrlParameter.Optional } // 参数默认值  
 );**

}

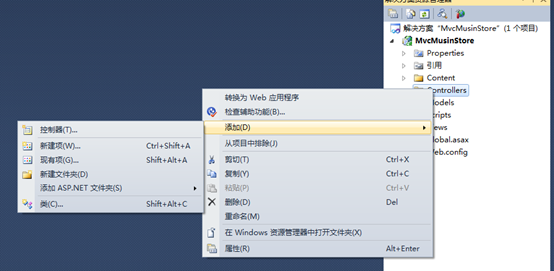
// 一般用来进行网站的初始化  
 protected void Application\_Start()  
 {  
 System.Data.Entity.Database.SetInitializer(new MvcMusicStore.Models.SampleData());  
 AreaRegistration.RegisterAllAreas();  
 RegisterGlobalFilters(GlobalFilters.Filters);  
 **RegisterRoutes(RouteTable.Routes);**  
 }  
 }  
}

其中 RegisterRoutes 方法注册了默认的路由配置，在方法中的 routes.MapRoute 语句中，请求地址将会被看成三个部分组成，{controller}/{action}/{id}，第一部分称为控制器，如果没有提供的话，默认为 **Home**, 第二部分称为 Action 方法，如果没有提供的话，默认为 **Index**, 第三部分称为 id, 通常用来提供数据的标识，没有默认值。这样，当请求 / 地址的时候，系统将会把请求映射到名为 Home 的控制器进行处理，调用其中名为 Index 的方法处理请求。

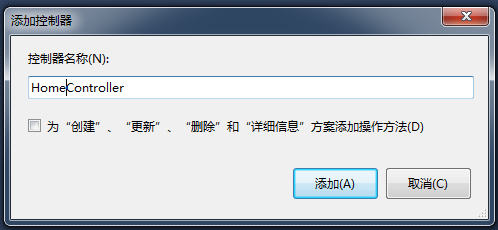
**增加 HomeController**

我们的应用商店从增加一个首页的控制器开始，使用默认的命名约定，控制器的名称应该以 Controller 作为后缀，我们将这个控制器命名为 HomeController。

在 Controller 文件夹上右击，然后选择“添加”，“控制器(T)…”



在弹出的对话框中，输入控制器的名字 HomeController，按下“添加”按钮。



这样，将会创建一个名为 HomeController.cs 的文件，其中的代码如下：

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace MvcMusinStore.Controllers

{  
 public class HomeController : Controller  
 {  
 //  
 // GET: /Home/  
 public ActionResult Index()  
 {  
 return View();  
 }  
 }  
}

为了尽可能地简单，我们让 Index 方法简单地返回一个字符串，这个字符串将作为回应内容直接返回浏览器，我们做下面的两个简单修改。

* 将方法的返回类型修改为string
* 将返回语句修改为 return "Hello form Home";

这样，我们的方法将会变成如下的内容

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

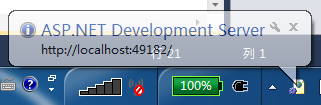
**运行程序**

现在，可以运行一下了，我们可以启动服务器，使用下面的任何一种方法都可以。

* 选择“调试”，“启动调试”菜单
* 点击工具栏上的绿色的箭头按钮
* 使用键盘快捷方式，F5

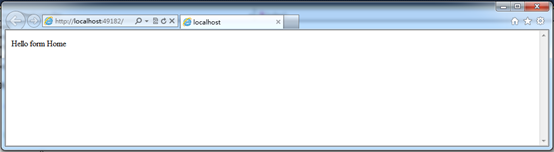


使用上面的任何一种方法都会导致编译项目，启动 Visual Studio 中内建的 ASP.NET 开发服务器.



在屏幕的右下角会弹出一个启动 ASP.NET 开发服务器的提示，

Visual Studio 将自动打开一个浏览器窗口，其中的地址指向我们的 Web 服务器，这使得我们可以快速使用我们的程序。



Ok, 非常的简单，我们已经创建了一个 Web 站点，写了一个三行的函数，在浏览器中得到了这行文字， 不是很震撼，但这仅仅只是开始。

**增加 StoreController**

我们已经为站点增加一个简单的 HomeController 作为首页，现在，增加另外一个控制器，可以用来浏览我们的音乐商店，我们的商店控制器将要支持三个场景：

* 列出商店中的唱片的分类
* 浏览商店中某个分类中的唱片列表
* 显示特定唱片的详细信息

我们从增加一个新的 StoreController 开始，先停止刚才的程序，可以直接关闭浏览器，或者也可以在调试菜单中，选择停止调试。

现在，增加一个新的 StoreController，像在 HomeController 中做的一样，我们在 Controllers 文件夹上右健，选择“添加”，“控制器”.

新的 StoreController 控制器中已经包含了 Index 方法，我们使用这个方法来实现列出所有分类的列表，我们还会再增加两个额外的方法来实现其他的场景：浏览和明细。

这些包含在控制器中的方法，我们称为控制器中的 Action，像你在前面看到的，HomeController 中的 Index 方法就是一个 Action，这些 Action 的作用就是处理请求，然后返回对请求的处理结果。

对于我们的 StoreController ，首先让 Index 这个 Action 返回一个“Hello” 串，然后，增加两个方法：Browse() 和 Detials()

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace MvcMusinStore.Controllers

{

public class StoreController : Controller

{

//

// GET: /Store/

public string Index()

{

return "Hello from Store.Index()";

}

public string Browse()

{

return "Hello from Store.Browse()";

}

public string Details()

{

return "Hello from Store.Details()";

}

}

}

重新运行程序，现在已经可以访问这些地址了。

* /Store
* /Store/Browse
* /Store/Details

太棒了，但是现在仅仅能够返回一些常量的字符串，让我们将它们变成动态的，我们从 URL 中获取一些信息，然后把他们显示在返回的页面中。

首先，修改我们的 Browse 这个 Action，使得它可以从 URL 地址中获取查询信息，为方法增加一个名为“genre” 的字符串类型参数，当我们这样做的时候，ASP.NET MVC 就会自动把任何名为 genre 的请求参数的值赋予这个参数。

//  
// GET: /Store/Browse?genre=?Disco  
public string Browse(string genre)

{

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

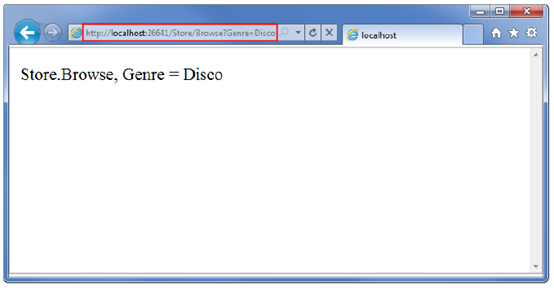
return message;

}

注意：

我们使用了HttpUtility.HtmlEncode 方法来处理用户的输入，这样可以防止用户的脚本注入攻击。例如：/Store/Browse?Genre=<script>window.location=’http://hackersite.com’</script>.

现在，我们在浏览器中访问一下：/Store/Browse?Genre=Disco



下一步，我们处理 Details 这个 Action，使它能够处理名为 ID 的整数类型参数。这次，我们不再在请求参数中传递这个整数，而是嵌在请求的 URL 地址中。例如：/Store/Details/5。

在 ASP.NET MVC 中，我们可以轻易地完成这个任务而不需要配置任何东西，ASP.NET MVC 默认的路由约定会将跟在 Action 方法之后的部分看作名为 ID 的参数的值，如果你的 Action 方法有一个名为 ID 的参数，那么，ASP.NeT MVC 就会自动将这部分作为参数传送给 Action 方法，需要注意的是，MVC 可以帮助你完成数据类型之间的转换，所以，地址的第三部分一定要可以转换为整数。

//  
// GET: /Store/Details/5  
public string Details(int id)

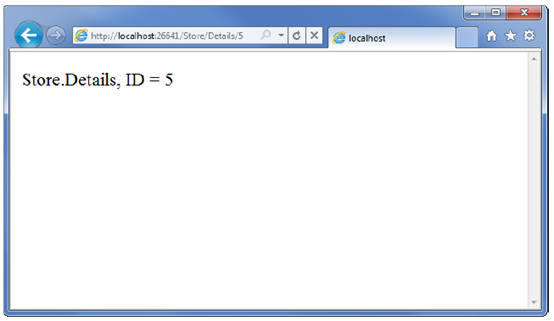
{

string message = "Store.Details, ID = " + id;

return message;

}

再次运行程序访问/Store/Details/5



总结一下我们完成的任务：

* 创建了一个 ASP.NET MVC 项目
* 讨论了基本的项目文件夹
* 学习了如何运行开发服务器
* 创建了两个控制器 HomeController 和 StoreController
* 为控制器增加了 Action 方法。

# ASP.NET MVC 音乐商店 - 3. 视图与模型

上一篇中有同学提到为什么不使用视图，而使用字符串，这一篇我们就开始使用视图来处理。

我们已经可以从控制器的 Action 中返回一个字符串，这可以帮助我们更好地理解 Controller 是如何工作的。但是对于创建一个 Web 程序来说还是不够的。下面我们使用更好的方法来生成 HTML，主要是通过模板来生成需要的 HTML，这就是视图所要做的。

**增加视图模板**

为了使用视图模板，我们需要将HomeController 中的 Index 这个 Action 的返回类型修改为 ActionResult，然后，让它像下面一样返回一个视图。

public class HomeController : Controller

{

//

// GET: /Home/  
 public ActionResult Index()

{

return View();

}

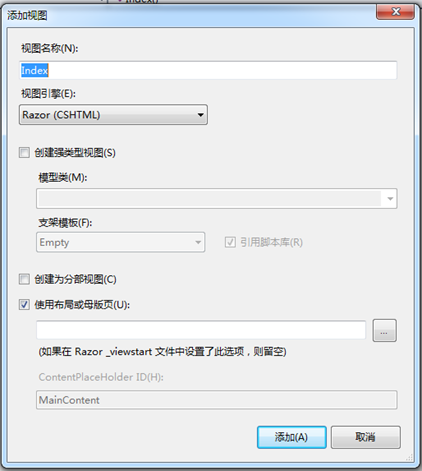
}

上面的修改表示我们将使用视图来替换掉原来的字符串，以便生成返回的结果。

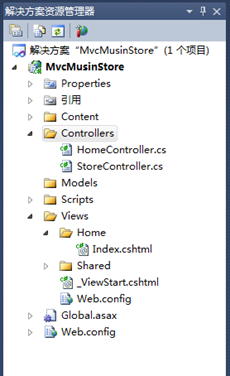
现在为我们的项目增加一个视图，为达到这个目的，我们将光标移到 Index 方法内，然后，点击鼠标的右键，在右键菜单中选择“添加视图(D)…”，这样将会弹出增加视图的对话框。



添加视图的对话框允许我们快速，简单地创建一个视图模板，默认情况下，视图的名称使用当前 Action 的名字。因为我们是在 Index 这个 Aciton 上添加模板，所以添加视图对话框中，视图的名字就是 Index，我们不需要修改这个名字，点击添加。



在点击添加之后，Visual Studio 将会创建一个名为 Index.cshtml的视图模板，放置在 \Views\Home 目录中，如果没有这个目录，MVC 将会自动创建它。



Index.cshtml 所在文件夹的名称和位置是很重要的，它是根据ASP.NET MVC 的约定来指定的。目录名称 \Views\Home ，匹配的控制器就是 HomeController ，视图模板的名字 Index，匹配将要使用这个视图的 Action 方法的名字。

当使用默认的约定的时候，ASP.NET MVC 允许我们不用显式设置这些名字和位置，当我们的代码如下所示的时候，将会默认寻找 \Views\Home\Index.cshtml。

public class HomeController : Controller

{

//

// GET: /Home/  
 public ActionResult Index()

{

return View();

}

}

Visutal Studio 创建并打开了Index.cshtml 视图模板，其中的内容如下：

@{

ViewBag.Title = "Index";

}

<h2>Index</h2>

视图使用了 Razor 语法，这比 Web Form 视图引擎的语法更加简单。

前三行使用 ViewBag.Title 设置了页面的标题，我们马上就可以看到这样做的效果，但是，首先，我们我们替换一下网页的内容，将 <h1> 标记中的内容修改为 This is the Home Page 。

@{

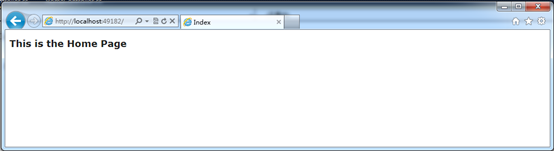
ViewBag.Title = "Index";

}

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

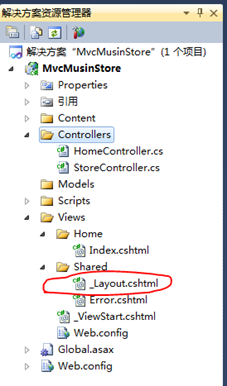
复制代码

现在，我们的首页应该变成下面的样子。



## 为页面的公共内容使用布局

大多数的网站在页面之间有许多共享的内容：导航，页首，页脚，公司的 Logo，样式表等等。Razor 引擎默认使用名为 \_Layout.cshtml 的布局来自动化管理，它保存在 /Views/Shared 文件夹中。



打开之后，可以看到下列内容：

<!DOCTYPE html>

<html>

<head>

<title>@ViewBag.Title</title>

<link href="@Url.Content("~/Content/Site.css")" rel="stylesheet" type="text/css" />

<script src="@Url.Content("~/Scripts/jquery-1.4.4.min.js")" type="text/javascript"></script>

</head>

<body>

**@RenderBody()**

</body>

</html>

来自内容视图中的内容，将会被通过 **@RenderBody()** 来显示，任何出现在网页中的公共内容就加入到 \_Layout.cshtml 中，我们希望我们的 MVC 音乐商店有一个公共的页首，其中含有链接到我们的首页和商店区域的链接，所以，我们将这些内容直接添加到这个布局中。

<!DOCTYPE html>

<html>

<head>

<title>@ViewBag.Title</title>

<link href="@Url.Content("~/Content/Site.css")" rel="stylesheet" type="text/css"/>

<script src="@Url.Content("~/Scripts/jquery-1.4.4.min.js")" type="text/javascript"></script>

</head>

<body>

<div id="header">

<h1>

ASP.NET MVC MUSIC STORE</h1>

<ul id="navlist">

<li class="first"><a href="/" id="current">Home</a></li>

<li><a href="/Store/">Store</a></li>

</ul>

</div>

**@RenderBody()**

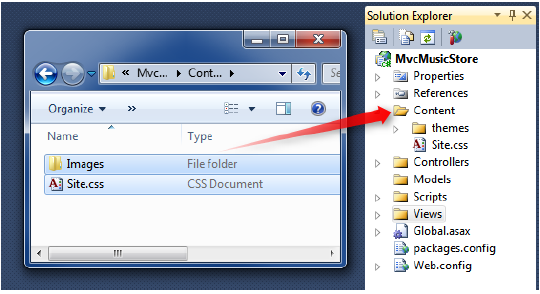
</body>

</html>

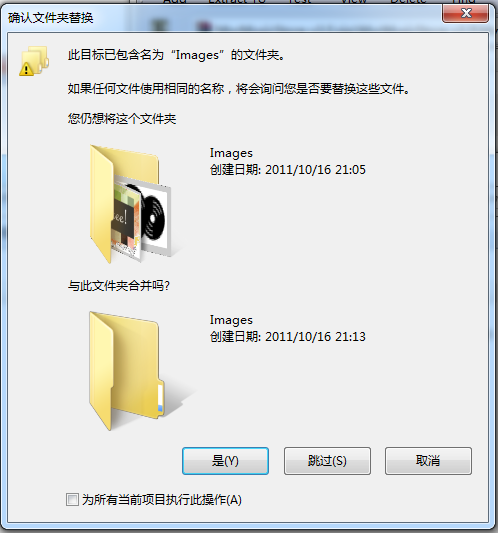
## 更新样式表

在创建项目使用的空项目模板中，仅仅包含很简单的用来显示验证信息的样式。我们的设计师提供了一些额外的 CSS 样式和图片来改进网站的观感，现在我们就使用它们。

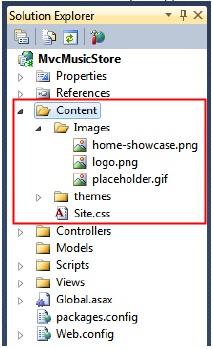
首先，到网站 [mvcmusicstore.codeplex.com](http://mvcmusicstore.codeplex.com/) 下载 [MvcMusicStore-v3.0.zip](http://mvcmusicstore.codeplex.com/releases/view/64379)，这里面有一个文件夹 MvcMusicStore-Assets，将这个文件夹的Content 文件夹的内容复制到项目的 Content 文件夹中。



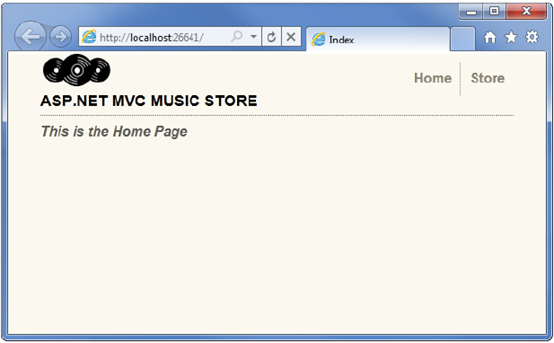
你会被询问是否需要覆盖存在的文件，选择是。



现在，网站的 Content 文件夹中的内容如下所示：



重新运行程序，现在的页面变成了这样。



我们回顾一下，什么发生了变化：HomeController 的 Index 的 Action 方法寻找并通过 \Views\Home\Index.cshtml 模板生成输出结果，代码中是通过 return View() 实现的，因为默认的命名约定，Index 这个 Action 方法将会默认使用 Index 视图输出。

而 Index 视图使用了我们的 \_Layout.cshtml 模板，所以，欢迎信息被包含在标准的 HTML 布局中。

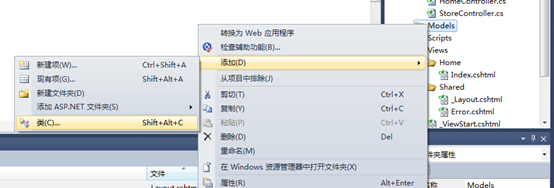
## 使用模型为视图传递信息

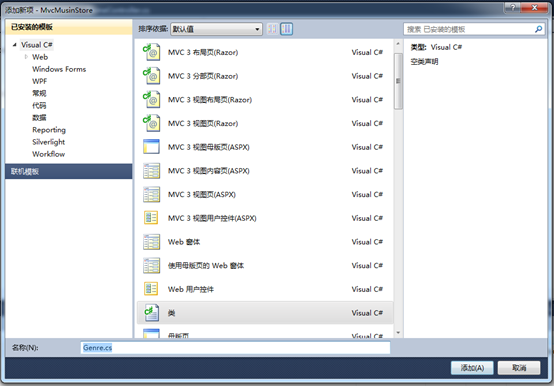
仅仅使用硬编码的 HTML 不能创建令人感兴趣的网站，创建动态网站，我们需要从控制器的 Action 传送信息给视图模板。

在 MVC 模式中，术语 Model 表示应用表现的数据，通常，模型对象用来表示数据库中保存在表中的数据，也不一定如此。

控制器的 Action 方法通过返回的 ActionResule 可以传送模型对象给视图。这就允许控制器可以将所有生成回应需要的数据打包，然会传送给视图模板，以便生成适当的 HTML 回应，在 Action 方法中可以很容易理解，让我们开始吧。

首先，我们将创建一些模型类来表示商店中的唱片类型和专辑类型，从创建类型 Genre 类开始，在项目中，右击模型 Models 文件夹，然后选择增加类选项，然后命名为 Genre.cs。





在新创建的类中增加一个属性。

public class Genre

{

public string Name { get; set; }

}

复制代码

注意：这里的 { get; set; } 是 C# 的自动属性特性，这使得我们不需要在创建属性的时候，先创建一个成员字段

现在，用同样的方法创建专辑类 Album，它有两个属性：Title 和 Genre .

public class Album

{

public string Title { get; set; }

public Genre Genre { get; set; }

}

现在，我们修改 StoreController 通过模型来使视图显示动态信息，为了演示方便，我们定义专辑基于一个唯一的标识 Id, 我们将在视图中显示这个标识。

我们从修改 Details 这个 Action 使得可以显示单个的专辑开始，在 StoreController.cs 的开始部分增加一些 using 语句来包含 MvcMusicStore.Models 命名空间，这使得我们不用总是输入这个命名空间。

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

using MvcMusicStore.Models;

复制代码

然后，我们更新 Details Action ，使得返回 ActionResult 类型的结果而不是字符串，就像在 HomeController 中的 Index 方法中做得一样。

public ActionResult Details(int id)

现在，修改方法的处理逻辑，返回一个专辑对象到视图中，在这个项目最后，显示的数据将会来自数据库，现在我们仅仅填充一些数据而已。

public ActionResult Details(int id)

{

var album = new Album { Title = "Album " + id };

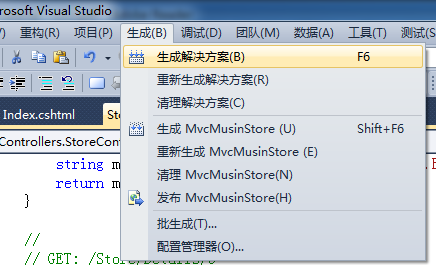
return View(album);

}

复制代码

如果你对 c# 不太熟悉，可能你会认为使用 var 定义变量使用了迟绑定，这是不正确的，C# 编译器使用赋予变量的值来推定变量的类型，所以，实际上变量的类型就是 Album 类型，因此不仅在编译时， Visual Studio 的代码编辑器中也会有类型支持。

下面创建一个使用专辑来生成 HTML 的模板，在这样做之前，我们需要编译项目，以便增加视图的对话框知道我们新创建的专辑类型。你可以通过菜单“生成”的“生成解决方案”来完成。

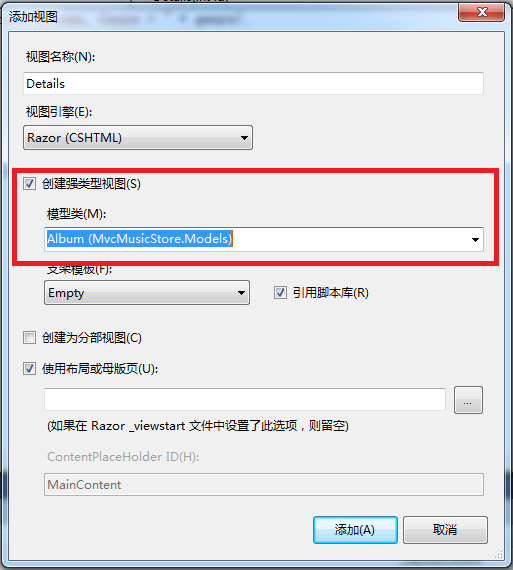


另外，也可以通过热键 Ctrl – Shift – B 来编译项目。

已经可以创建视图模板了，在 Details 方法中右键选择“增加视图…”



像以前一样，我们看到创建视图的对话框，不一样的是，我们要选中“创建强类型视图”，然后在下面的列表中选择“Album”类，这样视图将会期望得到一个 Album 类型的对象。



在点击增加之后，我们的视图模板 \Views\Store\Details.cshtml 被创建了，其中包含的如下的代码：

@model MvcMusicStore.Models.Album

@{

ViewBag.Title = "Details";

}

<h2>Details</h2>

复制代码

注意第一行，表示视图使用强类型的 Album 类。Rozer 视图引擎理解传送来的 Album 对象，所以我们可以容易地访问模型的属性，在 Visual Studio 中得到智能感知的帮助。

更新 <h2> 标记，使得可以显示专辑的 Title 属性

@model MvcMusicStore.Models.Album

@{

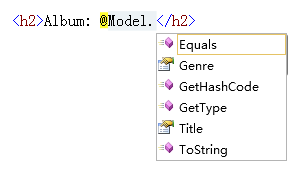
ViewBag.Title = "Details";

}

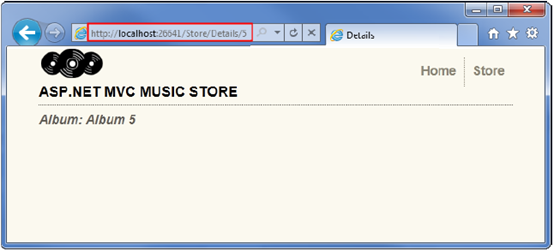
<h2>Album: @Model.Title</h2>

复制代码

注意，智能感知使得可以提示 Album 的属性和方法。



再次运行并访问 /Store/Details/5，可以得到下面的结果。



现在，我们继续修改 Browse 方法，更新方法返回 ActionResult 类型的结果，修改方法的处理，返回一个 Genre 类型的对象实例。

public ActionResult Browse(string genre)

{

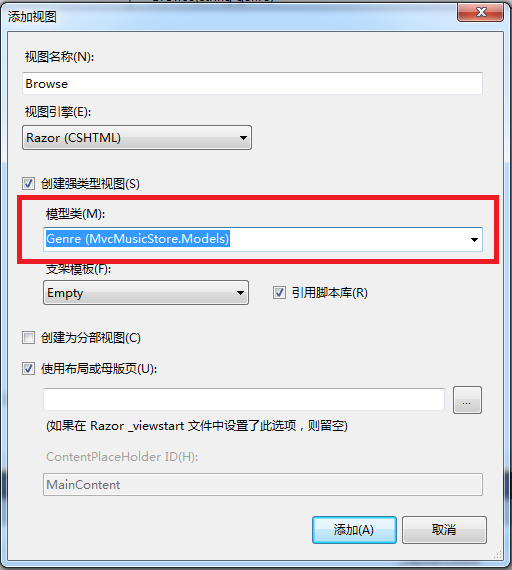
var genreModel = new Genre { Name = genre };

return View(genreModel);

}

复制代码

在方法上右击，选择“增加视图…”，增加一个强类型的视图。



修改 <h2> 标记显示 Genre 的信息

@model MvcMusicStore.Models.Genre

@{

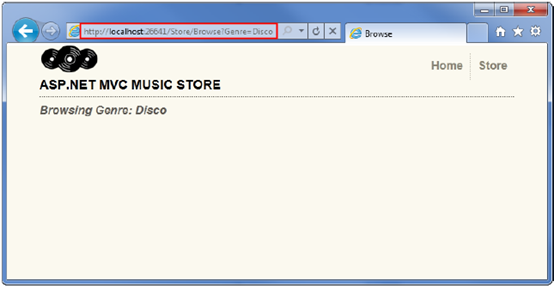
ViewBag.Title = "Browse";

}

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

复制代码

重新运行，访问 /Store/Browse?Genre=Disco，可以看到如下的显示



最后，将 Index 也修改为强类型的视图，显示所有唱片的类别，我们使用 Genre 的一个列表，而不是单个的 Genre 对象。

public ActionResult Index()

{

var genres = new List<Genre>

{

new Genre { Name = "Disco"},

new Genre { Name = "Jazz"},

new Genre { Name = "Rock"}

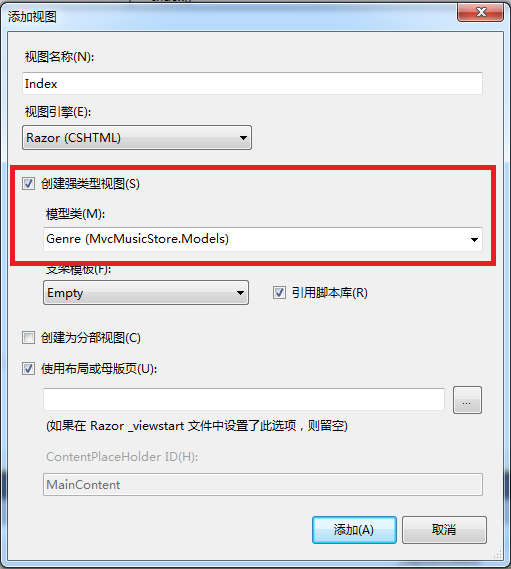
};

return View(genres);

}

复制代码

创建一个强类型的视图



首先，我们将期望得到多个 Genre 对象而不是一个，将第一行修改为如下内容。

@model IEnumerable<MvcMusicStore.Models.Genre>

这告诉视图引擎模式是一个包含多个 Genre 对象的集合，我们使用 IEnumerable<Genre> 而不是 List<Genre>，因为这样更通用，可以允许我们在以后改变集合为任何实现 IEnumerable 接口的集合。

现在，我们遍历集合中的 Genre 对象进行处理。

@model IEnumerable<MvcMusicStore.Models.Genre>

@{

ViewBag.Title = "Store";

}

<h3>

Browse Genres</h3>

<p>

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

<ul>

@foreach (var genre in Model)

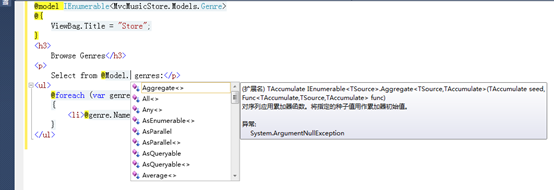
{ <li>@genre.Name</li>

}

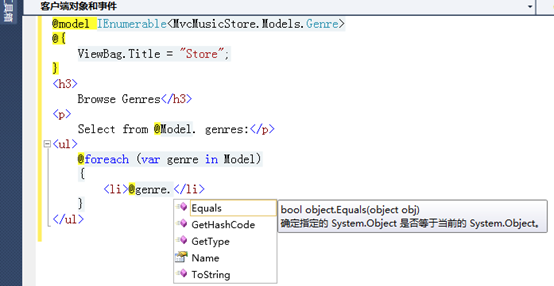
</ul>

复制代码

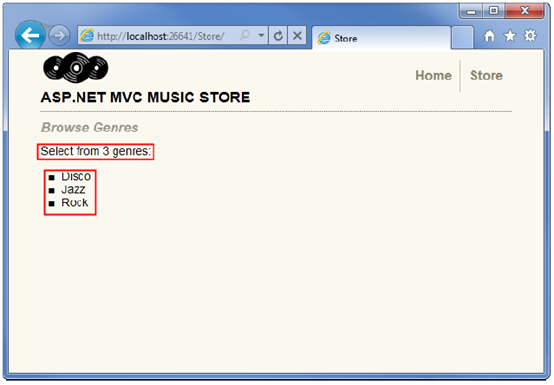
注意，此时有完全的智能提示



在 foreach 循环中，也同样有提示。



再次运行程序，我们可以看到如下的结果。



## 增加页面之间的链接

现在，我们的 /Store 可以使用纯文本列出当前的分类名称，下一步，我们将这些纯文本替换成可以链接到浏览分类的链接 /Store/Browse 上，这样，当用户点击音乐分类“Disco”将会被导航到 /Store/Browse?genre=Disco 的 URL 地址上。我们再次更新\Views\Store\index.cshtml 视图模板，先看一下，一会我们还会再次改进。

<ul>

@foreach (var genre in Model)

{

<li><a href=/Store/Browse?genre=@genre.Name>@genre.Name</a></li>

}

</ul>

复制代码

这样就可以工作了，但是这里使用了硬编码的字符串，如果我们希望修改控制器的名称，那么，我们就要找到所有这样的位置进行修改

更好的处理方式是使用 HTMLHelper 的助手方法，ASP.NET MVC 包含了一个 HTML 的助手类，其中的方法专门用于在视图模板中完成多种常见的任务，其中的Html.ActionLink() 助手方法就是常用的一个，这使得可以容易地创建 <a> ，包括关于链接的一些细节处理，像地址需要进行 URL 编码之类。

Html.ActionLink() 有多个重载用于多种情况，在简单的情况下，你只需要提供提示的文本，以及指向的 Action 方法即可，在客户端，举个例子，我们希望链接到 /Store 的 Index 方法，提示文本为 Go to the Store Index，那么下面的代码就可以。

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

注意：在这个例子中，我们不需要再特别指定控制器的名称，因为我们在使用同一个控制器的不同 Action 方法。

我们的链接还需要一些参数，我们可以使用另外一种重载来传递三个参数。

1. 链接的提示文本，这里显示分类的名称

2. 控制器的名称，Browse

3. 路由参数，提供名字 genre 和值，genre 的名字

合在一起，下面就是需要写在视图模板中的内容

<ul> @foreach (var genre in Model)

{

<li>@Html.ActionLink(genre.Name, "Browse", new { genre = genre.Name })</li>

}

</ul>

复制代码

现在，当我们运行程序，访问 /Store 的时候，将会看到一个分类的列表，每一个分类都是一个超级链接，当点击链接的时候，将会被导航到 /Store/Browse?genre=[genre] 的地址

页面中生成的分类链接如下：

<ul>

<li><a href="/Store/Browse?genre=Disco">Disco</a></li>

<li><a href="/Store/Browse?genre=Jazz">Jazz</a></li>

<li><a href="/Store/Browse?genre=Rock">Rock</a></li>

</ul>

# ASP.NET MVC 音乐商店 - 4. 数据访问

上一次，我们使用了模拟的数据从控制器发送到视图模板。现在，我们开始使用真正的数据库，在这个教程中，我们将指导如何使用 SQL Server Compact 版的数据库，它经常被称为 SQL CE, 来作为数据库引擎，SQL CE 是一个免费的，嵌入式的，基于文件的数据库系统，不需要任何的安装或者配置，很适合本地的开发使用。

注意：可能需要单独安装 SQL Server Compact 4.0 数据库以及 Entity Framework。在我的机器上，这两个软件都是单独安装的。

当然，你还可以使用熟悉的 SQL Server 数据库。

## Microsoft SQL Server Compact 4.0

Microsoft SQL Server Compact 4.0 是一种免费的嵌入式数据库，也就是说，你不需要安装数据库系统。软件开发人员可以使用它来构建 ASP.NET 网站和 Windows 桌面应用程序。SQL Server Compact 4.0 的优点是：占用的空间小，支持在应用程序文件夹中专门部署其二进制文件，轻松地使用 Visual Studio 和 WebMatrix 进行应用程序开发，以及无缝地将架构和数据迁移到 SQL Server。

SQL Server Compact 4.0 安装文件的下载地址：<http://www.microsoft.com/downloads/zh-cn/details.aspx?familyid=033cfb76-5382-44fb-bc7e-b3c8174832e2&displaylang=zh-cn>

不过，你完全可以不使用这个数据库，还继续使用原来的 SQL Server, 在我的机器上，我不能直接使用服务器资源管理器来查看 SQL CE 4 的的数据。如果使用原来的 SQL Server 就不会有这个问题了。

## ****ADO.NET Entity Framework 4.1****

**EF 4.1有哪些新玩新儿？**

1. 首先当然是DbContext API，它是基于以前版本中的ObjectContext和其他一些类型抽象出的一个简单的API，针对常用开发场景和编程模式进行了优化。DbContext可以被于Database First, Model First, Code First三种开发模式。

2. Code First是基于Entity Framework的新的开发模式，原先只有Database First和Model First两种。Code First顾名思义，就是先用C#/VB.NET的类定义好你的领域模型，然后用这些类映射到现有的数据库或者产生新的数据库结构。Code First同样支持通过Data Annotations或fluent API进行定制化配置。

这里的数据访问就使用 Code First 完成，它会使你的数据访问惊人地简单。

关于ADO.NET Entity Framework 4.1，我以前翻译了一个系列，地址： <http://www.cnblogs.com/haogj/archive/2011/05/06/2038965.html>

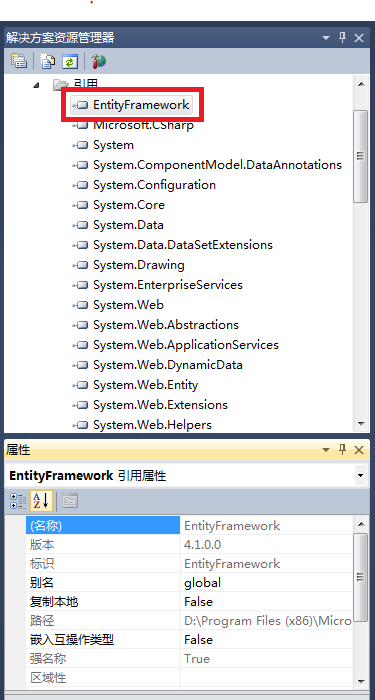
ADO.NET Entity Framework 4.1 安装文件下载地址：<http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=8363>

## 使用 Entity Framework Code-First 进行数据访问

我们将使用包含在 ASP.NET MVC3 中的 Entity Framework (EF) 支持进行查询和更新数据库中的数据。EF 是一个灵活的进行数据访问的对象关系映射 API，允许开发人员使用面向对象的方式对数据库中的数据进行查询和更新。

Entity Framework 4 支持一种称为代码优先的开发模式，代码有限允许你通过编写简单的类来创建模型对象（也被称为 POCO, 简单的，老的 CLR 对象），然后通过类来创建数据。

注意，需要在你的项目中引用程序集 EntityFramework，在你安装 Entity Framework 的文件夹中可以找到这个程序集。



## 修改我们的模型类

我们将延后数据库的创建工作，在完成这个任务之前，我们先修改我们得模型类，增加我们需要的内容。

## 增加艺术家 Artist 类

我们的专辑将要关联到艺术家，所以，我们需要增加一个简单的类来描述艺术家，增加一个新的名为 Artist 的类。

namespace MvcMusicStore.Models

{

public class Artist

{

public int ArtistId { get; set; }

public string Name { get; set; }

}

}

复制代码

## 更新现有的模型类

namespace MvcMusicStore.Models

{

public class Album

{

public int AlbumId { get; set; }

public int GenreId { get; set; }

public int ArtistId { get; set; }

public string Title { get; set; }

public decimal Price { get; set; }

public string AlbumArtUrl { get; set; }

public Genre Genre { get; set; }

public Artist Artist { get; set; }

}

}

然后更新 Genre 类

namespace MvcMusicStore.Models

{

public class Genre

{

public int GenreId { get; set; }

public string Name { get; set; }

public string Description { get; set; }

public List<Album> Albums { get; set; }

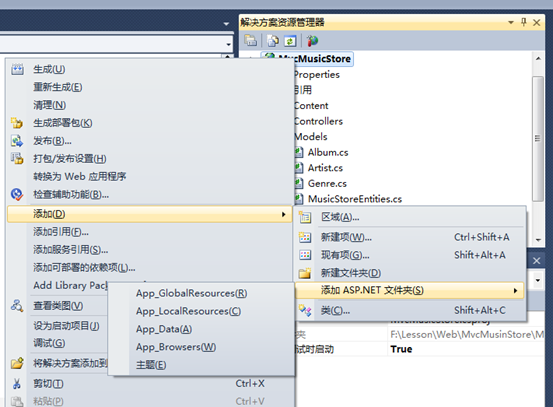
}

}

## 增加 App\_Data 文件夹

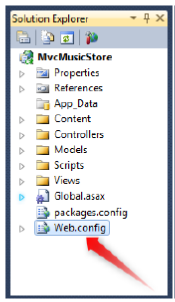
这一步并不必要，如果使用 SQL CE4，可以用来保存数据库文件，如果使用 SQL Server 则不一定需要。

我们在项目中增加 App\_Data 文件夹用来保存数据库文件，App\_Data 是一个 ASP.NET 中特殊的文件夹，已经被网站对其中数据地访问进行了安全限制，从项目的菜单中，选择增加 ASP.NET 文件夹，然后，选择 App\_Data.



## 在 Web.config 中创建数据库连接串

我们需要在网站的配置文件中增加一些行，以便 Entity Framework 知道如何连接到我们的数据库，双击 Web.config 文件。



卷到文件的最后，然后增加一个 <connectionStrings> 的配置节，如何所示：

<connectionStrings>

<add name="MusicStoreEntities"

connectionString="Data Source=|DataDirectory|MvcMusicStore.sdf"

providerName="System.Data.SqlServerCe.4.0"/>

</connectionStrings>

复制代码

注意，这里数据库连接串的名称很重要，以后使用 EF Code-First 的时候，通过它来找到数据库，这里的链接串种使用了 Data Source=|DataDirectory|MvcMusicStore.sdf，这里的 DataDirectory 指的就是项目中的 App\_Data 文件夹夹。

如果使用 SQL Server, 可以使用如下的链接串。注意 providerName 也要替换成 SQLServer 使用的提供器。

<!-- 数据库连接串的配置 -->

<connectionStrings>

<add name="MusicStoreEntities"

connectionString="server=.\sqlexpress;database=musicstore;integrated security=true;"

providerName="System.Data.SqlClient"/>

</connectionStrings>

## 增加上下文类

在模型文件夹上右键点击，然后，增加一个新的名为 MusicStoreEntities.cs 的文件。 需要注意的是，这个类的名称必须与数据库连接串的名称一致。



这个类将反映 Entity Framework 数据库的上下文，用来处理创建，读取，更新和删除的操作，代码如下所示：

using System.Data.Entity;

namespace MvcMusicStore.Models

{

public class MusicStoreEntities:

DbContext

{

public DbSet<Album> Albums { get; set; }

public DbSet<Genre> Genres { get; set; }

public DbSet<Artist> Artists { get; set; }

}

}

注意，这里使用了 System.Data.Entity 命名空间。记得要 using 一下。

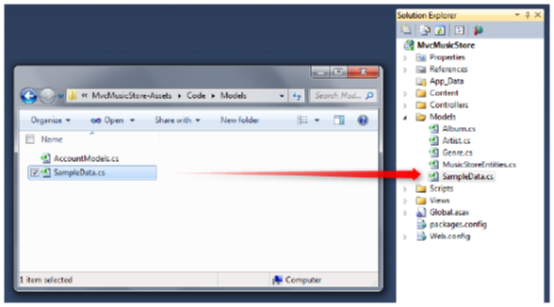
不需要其他的配置，特定的接口等等，通过扩展 DbContext 基类，我们得 MusicStoreEntities 类就可以处理我们对数据库的操作了，现在，我们就开始，先为我们的类增加一些属性来从数据库获取额外的信息。

## 增加我们的商店分类数据

对于 Code First 来说，我们首先定义模型，然后通过模型来创建数据库，甚至也不需要写 Insert 语句，我们可以通过标准的 C# 代码来创建表中的记录。

我们先通过一些种子数据通过 Entity Framework 为新创建的数据库增加一些数据。先创建我们的商店分类，这需要通过一个 Genres 的列表完成，还有专辑。在 MvcMusicStore-Asset.zip 文件中，已经包含了用来简单地创建数据的文件，有一个保存在 Code 文件夹中的类来完成。

在 Code 中的 Models 文件夹中，找到 SampleData.cs 文件，将它加入到 Models 文件夹中，如下所示。



现在，我们需要增加一些代码来告诉 Entity Framework 关于 SampleData 类的事情。双击 Global.asax 文件，打开它，在 Application\_Start 方法中，增加如下的行。

// 一般用来进行网站的初始化  
protected void Application\_Start()

{

**System.Data.Entity.Database.SetInitializer(new MvcMusicStore.Models.SampleData());**

AreaRegistration.RegisterAllAreas();

RegisterGlobalFilters(GlobalFilters.Filters);

RegisterRoutes(RouteTable.Routes);

}

这个方法用来初始化数据库，然后填充一些数据。

这样，我们就完成了配置 Entity Framework 的工作。

## 查询数据库

现在，我们更新一下我们的 StoreController 以便取代以前模拟的数据，通过调用我们的数据库来查询实际数据。我们先在 StoreController 中定义一个字段来访问我们的 MusicStoreEneities 类的对象实例，它命名为 storeDB。

using MvcMusicStore.Models;

namespace MvcMusicStore.Controllers

{

public class StoreController : Controller

{

MusicStoreEntities storeDB = new MusicStoreEntities();

## 更新 Index Action 查询数据库

MusicStoreEntities 类通过 Entity Framework 提供了数据库中数据表的集合，更新一下 StoreController 的 Index Action 方法来获取全部的分类数据。我们原来使用硬编码的数据，现在，我们可以使用 Entity Framework 的 Generes 集合来取代它了。

对于 EF 的使用，我建议你了解一下 Repository 模式。

//  
// GET: /Store/  
public ActionResult Index()

{

var genres = storeDB.Genres.ToList();

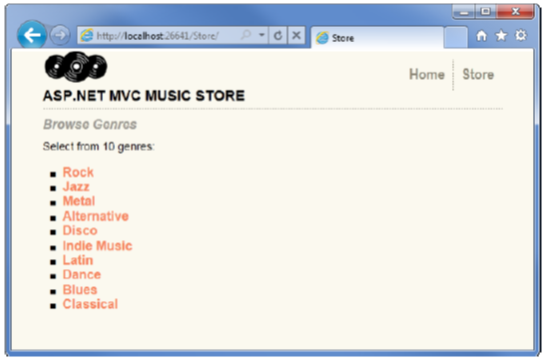
return this.View( genres );

}

复制代码

对于视图模板不需要任何修改，我们仍然返回同样的 StoreIndexViewModel 。

运行程序，访问 /Store 地址的时候，我们现在可以看到数据库中分类的列表。



## 使用数据库中的数据更新浏览和明细页面

当在首页通过 /Store/Browse?genre=[some-genre] 链接访问 Browse 这个 Action 的时候，我们需要通过流派的名称来获取相应的专辑，对于我们的音乐店来说，每个流派的名称是唯一的，可以通过 LINQ 中的 Single 扩展方法来获取查询结果中的唯一的流派对象。

var example = storeDB.Genres.Single(g => g.Name == “Disco”);

Single 方法使用一个 Lambda 表达式作为参数，表示我们希望获取匹配指定值的单个流派对象，在上面的例子中，我们将会获得名为 Disco 的流派对象。

通过 EF ，在获得流派对象的同时，我们还可以获取流派相关的对象，例如属于这个流派的专辑集合，我们可以提前获取相关的专辑信息，这就需要我们修改一下上面的查询，包含专辑信息。通过 Include 方法可以指定我们希望获取的相关信息，这种方式非常有效，这样，我们就可以在一次数据访问中，既可以获取流派对象，也可以同时获取相关的专辑对象。

这样更新之后，我们的 Action 方法将会成为下面的样子。

// /Store/Browse?genre=DISCO  
public ActionResult Browse( string genre )

{

var genreModel = storeDB.Genres.Include("Albums").Single(g => g.Name == genre);

return this.View(genreModel);

}

复制代码

注意，在流派 Genre 的属性中有一个名为 Albums 的集合属性。

然后，我们可以更新一下 Store 的 Browse 视图来显示相应的专辑，打开视图模板，增加一个列表。

@model MvcMusicStore.Models.Genre

@{

ViewBag.Title = "Browse";

}

<h2>

Browsing Genre: @Model.Name</h2>

<ul>

@foreach (var album in Model.Albums)

{

<li>

@album.Title

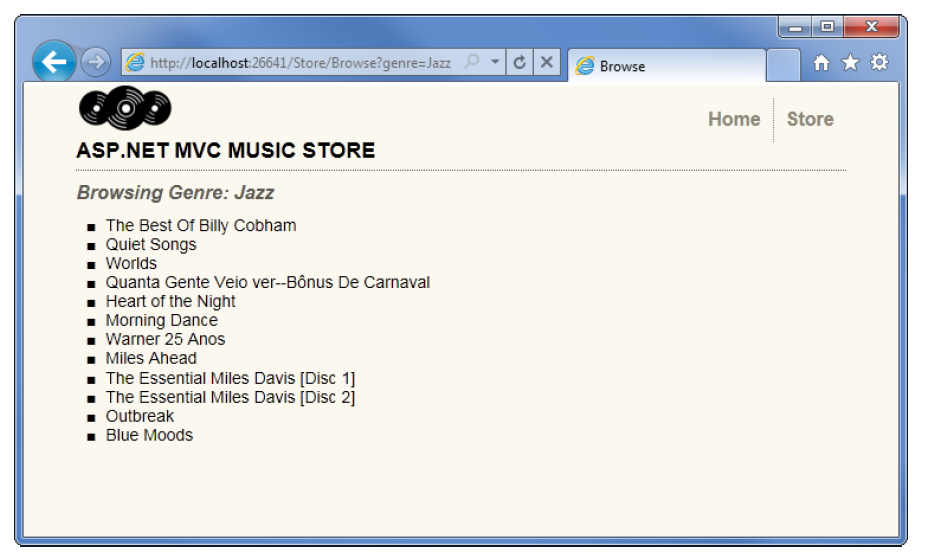
</li>

}

</ul>

复制代码

运行程序，浏览 /Store/Browse?genre=Jazz，现在就可以看到保存在数据库中的的专辑数据了。



同样，我们还可以修改一下 Details ,通过传递的参数来获取专辑对象。修改后的方法如下所示。

// /Store/Details/5  
public ActionResult Details(int id)

{

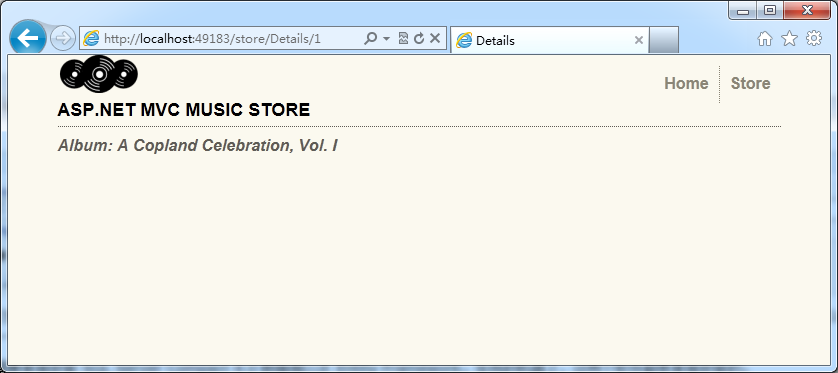
var album = storeDB.Albums.Find(id);

return View(album);

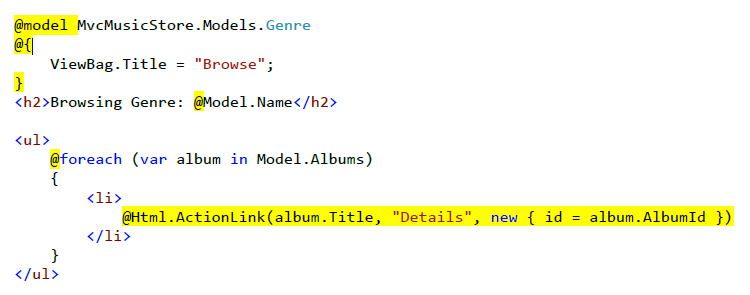
}

复制代码

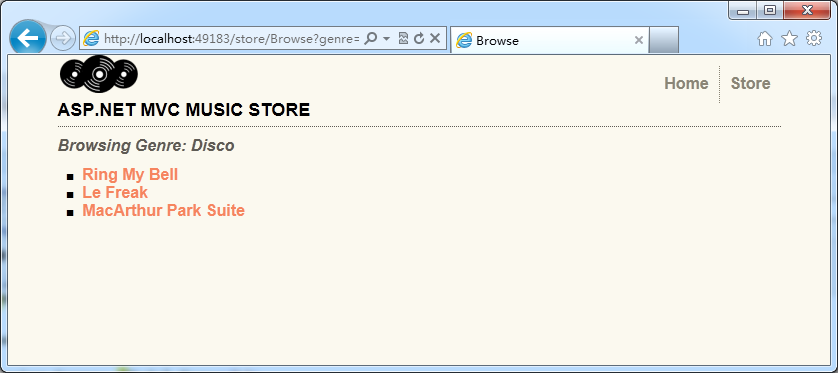
运行程序，访问 /Store/Details/1，应该可以看到下面的内容。



我们更新一下 Browse 视图，提供链接到明细页面的超级链接，这里，我们使用 ActionLink 方法，修改后的视图如下所示。



再次浏览 Browse 的时候，每个专辑应该已经成为了一个链接，如图所示：

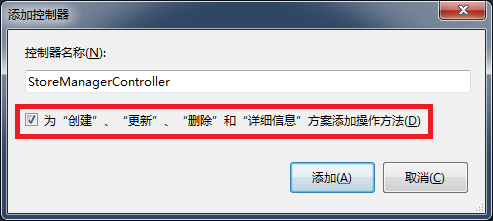


# ASP.NET MVC 音乐商店 - 5. 通过支架创建编辑表单

在上一章，我们已经从数据库获取数据，然后显示出来，这一章，我们将允许编辑数据。

## 创建 StoreManagerController 控制器

我们将要创建称为 StoreManager 的控制器，对于这个控制器，我们将通过使用 ASP.NET MVC3 中提供的脚手架功能来实现。在添加控制器的窗口中，注意需要选中 为”创建”、”更新”、”删除”和”详细信息” 方案添加操作方法。



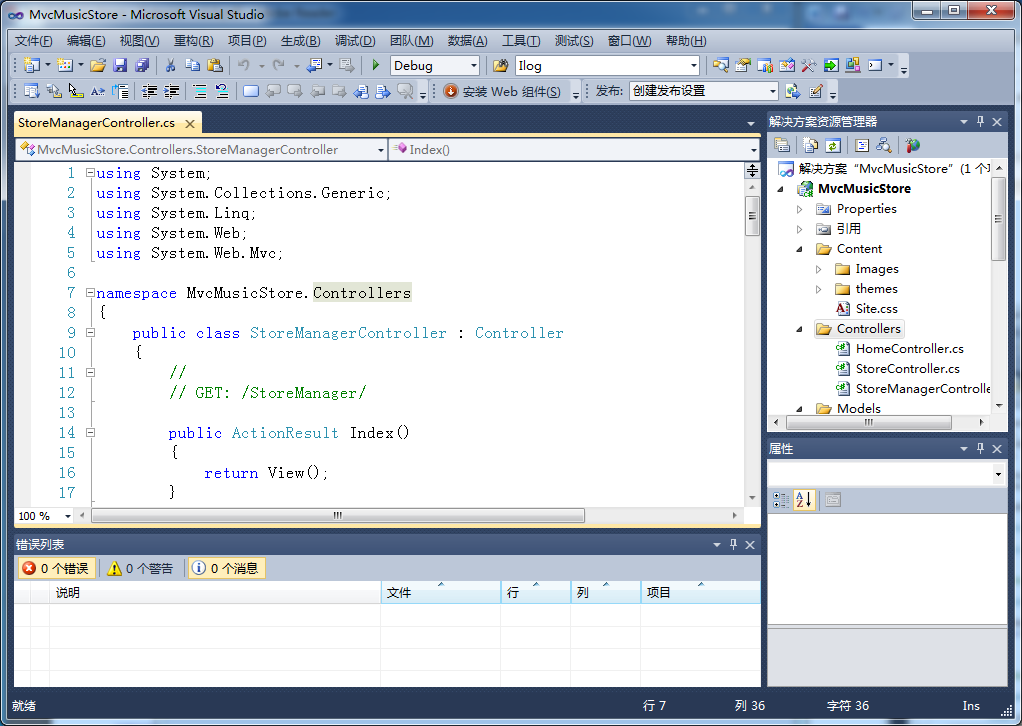
在我们点击添加按钮之后，你将会看到 ASP.NET MVC3 的支架机制已经为你在 Controllers文件夹中添加了一个名为 StoreManagerController 的控制器。

原文中说：创建了一个带有本地实体框架变量的 StoreManagerController 控制器

但是，在我的 VS2010 中实际并没有创建实体框架变量，在前面的数据访问部分，我机器上的实体框架也是手工下载，并安装的。

原文还说，创建了 Create.cshtml，Delete.cshtml，Details.cshtml，Edit.cshtml，以及 Index.cshtml 视图。 这些视图使用了强类型的 Album 类型。

但是，在对话框中并没有选择实体类型的选项，所以，在我这里，实际上也没有创建相应的视图，更没有强类型一说了，不过，这些其实都是小问题。



新的 StoreManager 控制器包含了 CRUD （创建，读取，更新，删除）控制器 Action ，原文中说：这些控制器知道如何使用 Album 模型和实体框架的上下文对象来访问数据。

由于并没有创建实体上下文对象，所以，我这里也没有实际的数据访问代码存在，在下面的说明中，我们将手工创建这一部分。

所以，我们手工在控制器中增加数据访问的实体上下文对象。

namespace MvcMusicStore.Controllers

{

public class StoreManagerController : Controller

{

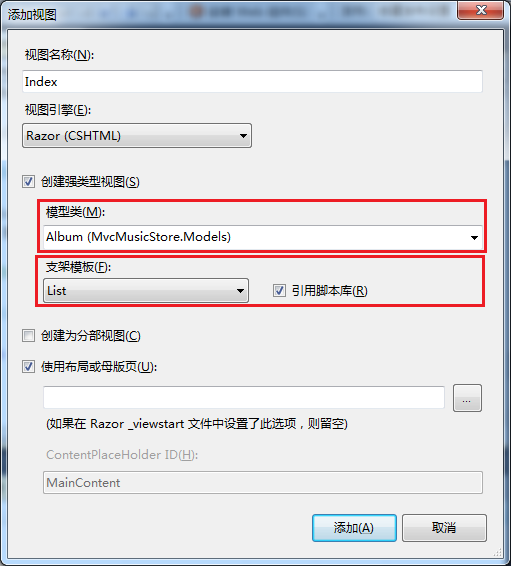
MvcMusicStore.Models.MusicStoreEntities storeDB

= new MvcMusicStore.Models.MusicStoreEntities();

## 修改视图

这里我们先手工生成视图，然后再进行修改。

首先为 Index 增加强类型的视图。需要在对话框中选中 创建强类型视图，然后，选中模型类，在支架模板中选择 List，这是因为我们需要在 Index 视图中处理专辑的列表。

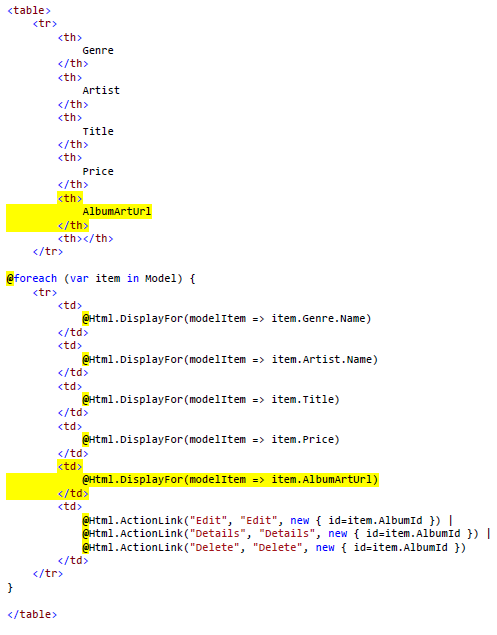


使用 List 支架之后，创建的视图中，模型的类型将会生成为如下的实现。

@model IEnumerable<MvcMusicStore.Models.Album>

记住，虽然脚手架可以为我们自动生成代码，但只是标准的 ASP.NET MVC 代码， 就像我们的这个教程可以节省你的学习时间一样，支架可以节省你手工创建控制器，以及强类型视图的时间，但是，具体的细节内容还是需要自己来处理。

所以，让我们来快速编辑一下 StoreManager 的 Index 视图（/Views/StoreManager/Index.cshtml）。这个视图通过一个 table 表格显示专辑的列表，包含专辑的公共属性，还带有进行编辑/详细内容/删除的链接。我们要删除专辑艺术家的链接，我们不需要显示这个值，在视图中的 <table> 部分，删除 AlbumArtUrl 相关的 <th> 和 <td> 元素，下面的代码中高亮显示的部分。



这样， Index 视图的代码应该如下所示：

@model IEnumerable<MvcMusicStore.Models.Album>

@{

ViewBag.Title = "Index";

}

<h2>

Index</h2>

<p>

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

</p>

<table>

<tr>

<th>

Genre

</th>

<th>

Artist

</th>

<th>

Title

</th>

<th>

Price

</th>

<th>

</th>

</tr>

@foreach (var item in Model)

{

<tr>

<td>

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

</td>

<td>

@Truncate(item.Artist.Name, 25)

</td>

<td>

@Truncate(item.Title, 25)

</td>

<td>

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

</td>

<td>

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

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

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

</td>

</tr>

}

</table>

## Store Manager 的第一印象

视图虽然修改完成了，但是 Index 方法中还没有向视图提供数据的代码，修改 Index 方法，增加从数据库中获取数据的处理逻辑。

public ActionResult Index()

{

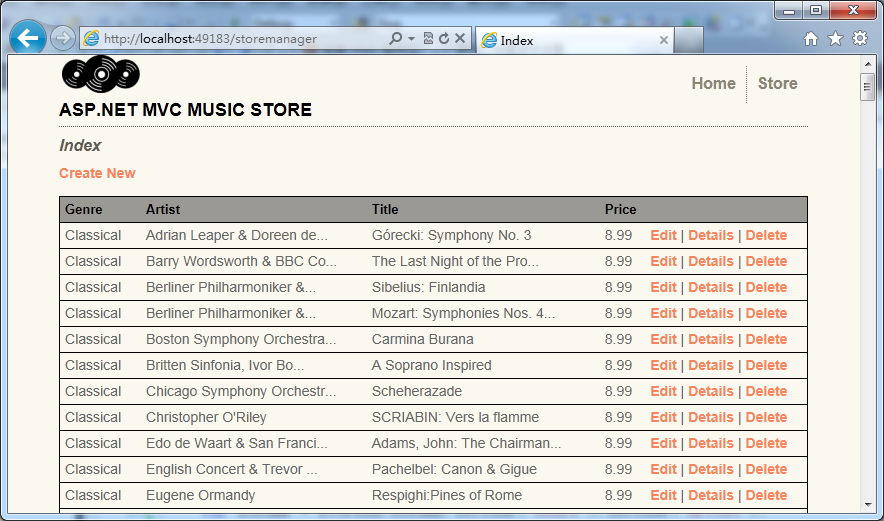
var albums = storeDB.Albums.Include("Genre").Include("Artist");

return View(albums.ToList());

}

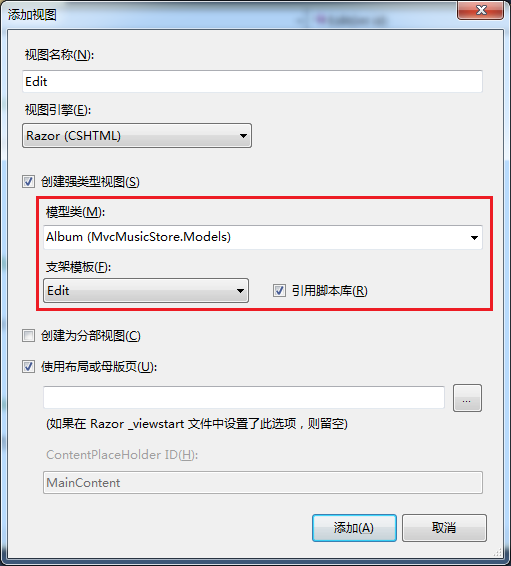
复制代码

现在，运行程序，浏览 /StoreManager，注意，在路由的配置中，默认的 Action 将会使用 Index，修改之后的视图显示了带有编辑，详细内容和删除链接的专辑列表。



需要注意的是，编辑、详细内容和删除链接的视图需要我们手工创建之后才能使用。

我们依次创建这些视图。将光标放置在 Edit 的 Action 方法内，单击鼠标右键，在弹出菜单中选择添加视图。



创建编辑视图的对话框，注意使用的支架为 Edit。

同样，修改 Edit 的 Action 方法。

需要注意的是，你会看到在 Controller 中存在两个同名的 Edit 方法，第二个方法前面使用了 [HttpPost] 标签，这个标签标识当请求类型为 Post 方式的时候，将会由这个方法进行请求的处理，否则，使用第一个 Edit 方法进行处理。

在 ASP.NET MVC 中，已经不再使用视图状态了，在需要用户编辑数据的时候，我们首先提供一个编辑表单，用户得到这个编辑表单的方式一般是通过某个超级链接，这样的请求方式将是 GET 请求，当这样的请求到达服务器的时候，我们向客户端返回编辑页面，允许用户编辑数据。

在编辑数据的窗体中，我们提供一个 form 表单，这个表单的提交方式设置为 Post 方式，用户在提交表单的时候，将填写的数据提交到服务器。由于此时的提交方式成为 Post 方式，这就允许我们在服务器上通过请求的提交方式区分出来请求的类型。

这样，我们就可以在 Controlller 中提供同名的 Action 来处理用户的编辑操作，Get 方式的 Action 用来提供编辑表单，而 Post 方式的 Action 用来获取用户提交的数据。这种方式在 ASP.NET MVC 中使用很多。

修改后的 Edit 方法如下所示：

public ActionResult Edit(int id)

{

MvcMusicStore.Models.Album album = storeDB.Albums.Find(id);

return View(album);

}

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

[HttpPost]

public ActionResult Edit(int id, FormCollection collection)

{

try

{

// TODO: Add update logic here  
 MvcMusicStore.Models.Album album = storeDB.Albums.Find(id);

if (this.TryUpdateModel<MvcMusicStore.Models.Album>(album))

{

return RedirectToAction("Index");

}

return View();

}

catch

{

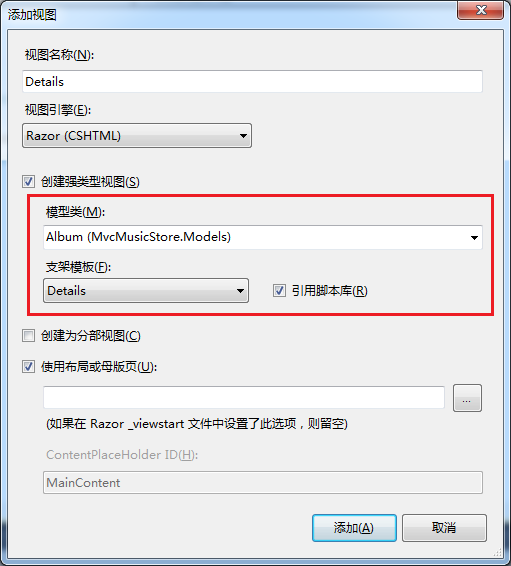
return View();

}

}

复制代码

创建详细内容视图。注意使用的支架。



相应修改 Details 方法的处理逻辑。

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

public ViewResult Details(int id)

{

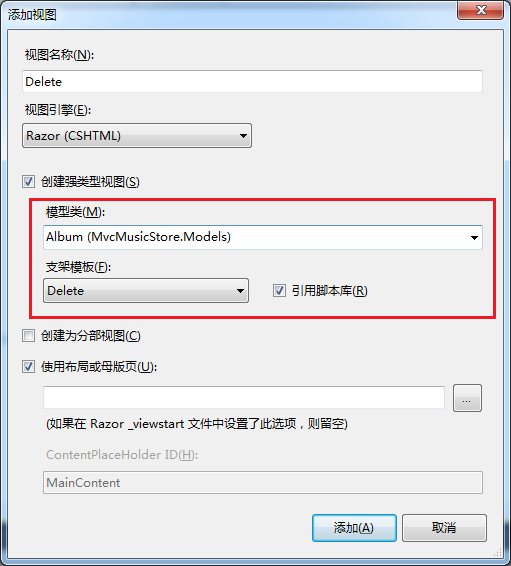
MvcMusicStore.Models.Album album = storeDB.Albums.Find(id);

return View(album);

}

复制代码

创建删除视图的对话框，注意使用的支架模板。



同样， Delete 方法也有两个，一个用来处理 Get 方式的请求，一个用来处理点击确认删除之后的请求。

public ActionResult Delete(int id)

{

MvcMusicStore.Models.Album album = storeDB.Albums.Find(id);

return View(album);

}

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

[HttpPost]

public ActionResult Delete(int id, FormCollection collection)

{

try

{

// TODO: Add delete logic here  
 MvcMusicStore.Models.Album album = storeDB.Albums.Find(id);

storeDB.Albums.Remove(album);

return RedirectToAction("Index");

}

catch

{

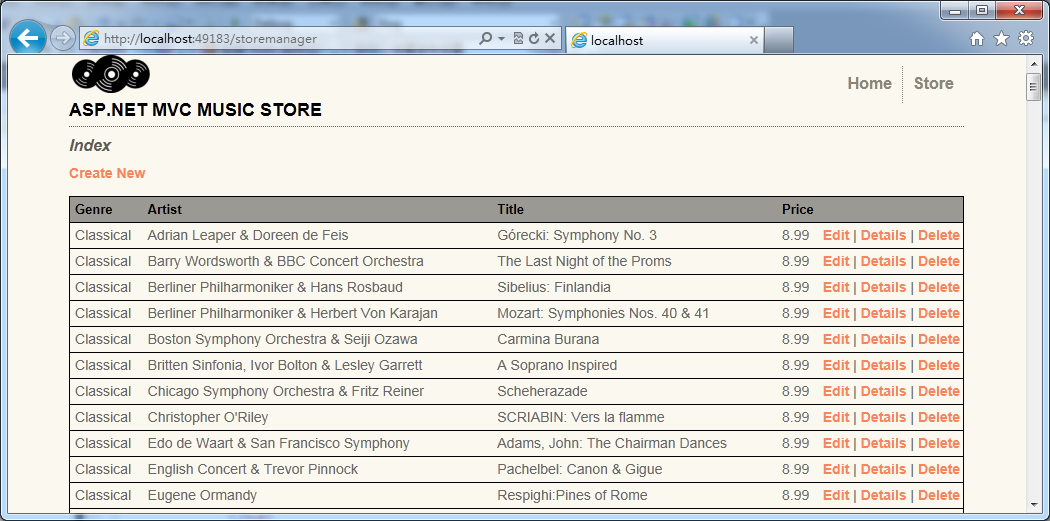
return View();

}

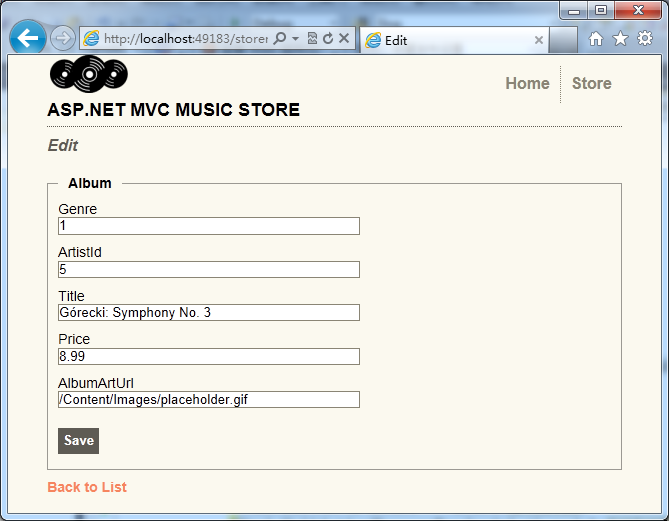
}

复制代码

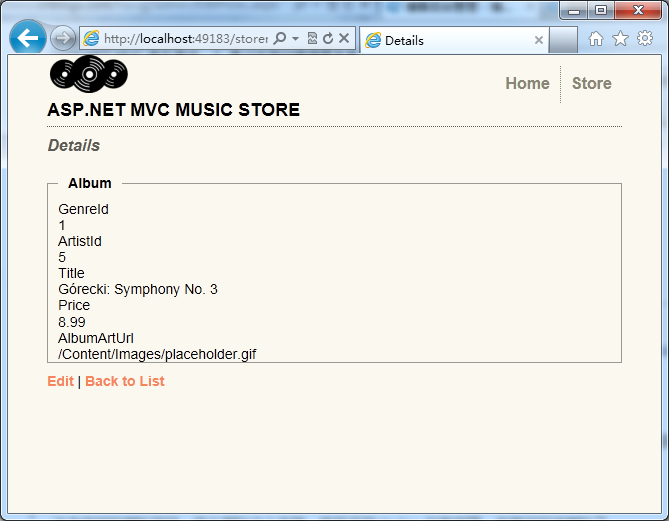
现在，可以运行一下了，访问 /StoreManager 可以得到如下的结果。



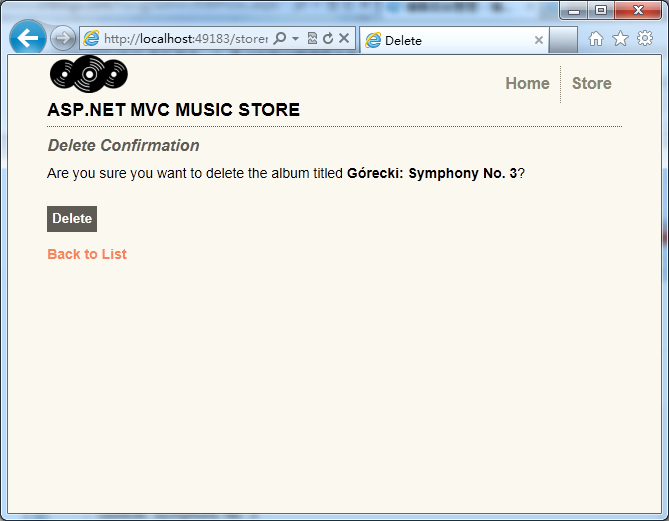
点击编辑链接，将会显示一个带有专辑字段的编辑表单。



点击位于底部的返回列表链接，然后，点击详细内容链接， 将会显示单个专辑的详细信息。



再来一次，回到列表，点击删除链接，现在将会看到一个确认对话框，显示了专辑的详细内容，请你确认是否真的需要删除。



点击底部的删除按钮，将会删除这个专辑，然后返回 Index 列表视图，专辑已经被删除了。

我们的工作还没有完成，我们要开始对控制器和视图的 CRUD 操作进行处理了。

# ASP.NET MVC 音乐商店 – 5.通过支架创建编辑表单 续

## 查看 StoreManager 控制器的代码

现在，Store Manager 控制器中已经包含了一定数量的代码，我们从头到尾重新过一下。

首先，在控制器中包含了标准的 MVC 控制器的代码，为了使用方便，还可以引用我们的模型类所在的命名空间 MvcMusicStore.Models。控制器还拥有了一个私有的 MusicStoreEntities 的私有成员，以方便控制器中的每个 Action 访问数据库。

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

using MvcMusicStore.Models;

namespace MvcMusicStore.Controllers

{

public class StoreManagerController : Controller

{

MvcMusicStore.Models.MusicStoreEntities storeDB

= new MvcMusicStore.Models.MusicStoreEntities();

复制代码

### Index 和 Details Action

Index 视图获取专辑的列表，包含每一个专辑引用的流派和艺术家信息，向我们在前面 Store控制器的 Browse时候看到的，Index 视图中需要包含对于链接到的流派和艺术家对象来显示相关的信息，所以，在 Index 的 Action 方法中，需要包含这些数据。

//  
// GET: /StoreManager/

public ActionResult Index()

{

var albums = storeDB.Albums.Include("Genre").Include("Artist");

return View(albums.ToList());

}

复制代码

在 Details 方法中，类似于 Store 控制器的 Details 方法，通过专辑的 Id 来获取专辑对象，这里使用 Find() 方法完成，最后，把这个对象传递给视图。

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

public ViewResult Details(int id)

{

MvcMusicStore.Models.Album album = storeDB.Albums.Find(id);

return View(album);

}

复制代码

### 创建方法 Create

与前面看到的不同，Create 方法需要处理表单，当用户第一次访问地址 /StoreManager/Create 的时候，用户将会看到一个空的表单，HTML 页面中包含一个 <form> 元素，其中包含了下拉列表和文本框等输入元素，用户可以借助他们输入专辑的详细信息。

当用户填写了专辑的信息之后，可以通过点击 “保存” 按钮来提交表单信息到服务器，我们de应用程序可以获取这些信息保存到数据库中。在用户点击 “保存” 的时候，浏览器发出一个 Http 的 Post 请求，到 /StoreManager/Create 地址，表单的内容作为这个 Post 请求的一部分发送回服务器。

ASP.NET MVC 允许我们容易地分割这两种同样对于 Create 方法的请求处理，通过提供两个同名的 Create 方法，一个用来处理 Http Get 请求，一个用来处理 Http Post 请求，区分的方式是在处理 Post 请求的方法前面增加一个 [HttpPost] 的标签。如果增加 [HttpGet] 标签，则表示这个方法仅仅处理 Http Get 请求。通常没有这个标签，则表示无论是 Get 请求还是 Post 请求都可以由这个 Action 方法处理。

#### 使用 ViewBag 向视图传递信息

我们曾经在这个教程的开始使用过 ViewBag, 但是，没有深入讨论它，ViewBag 允许我们向视图传递信息而不需要首先定义强类型的 Model，在创建专辑的 Action 中，因为在表单中需要两个列表框，我们需要向视图传递列表所需要的数据，以便在视图中填充下拉列表框，最简单的方式就是通过 ViewBag 来实现。

ViewBag 是动态对象，这意味着你可以使用 ViewBag.Foo 或者 ViewBag.YourNameHere 形式的属性而不需要预先定义这些属性，控制器中的代码使用 ViewBag.GenreId 和 ViewBag.Artisid 传递流派和艺术家的信息以便生成表单中下拉列表的值，以后，用来在新创建的专辑中使用。

传递到视图的下拉列表的值使用 SelectList 对象表示，对于我们的要求，使用下面的代码。

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

方法中的三个参数被用于创建这个对象：

用来生成下拉列表中信息的集合，注意，不是一个字符串集合，我们这里是流派对象的集合。

方法的第二个参数提供下拉列表中的值，这是一个字符串，实际上是流派对象的一个属性 GenreId。

最后的参数提供下拉列表中显示出来的值，这里使用流派的 Name 属性。

理解了上面的内容，代码就很简单了，两个 SelectList 对象通过 ViewBag 传递给视图，没有使用模型对象，我们也没有创建这样用途的模型。

//  
// GET: /StoreManager/Create  
public ActionResult Create()

{

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

ViewBag.ArtistId = new SelectList(storeDB.Artists, "ArtistId", "Name");

return View();

}

### 在 Create 视图中使用 HtmlHelper 来显示下拉列表

下拉列表的数据已经被传递到了视图中，现在，我们看一下视图如何使用这些数据显示一个下拉列表，在 Create 视图中，将原来显示流派标识的代码

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

修改为

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

这里使用了 HtmlHelper ，一个用于视图中生成 Html 的实用工具，使用 HtmlHelper 可以保持我们的视图清楚和易读。Html.DropDownList 助手方法由 ASP.NET MVC 提供，后面可以看到，我们也可以提供自己的助手方法进行扩展。

Html.DropDownList 方法需要两个参数，从哪里获取显示用的列表，和哪一个值需要被预先选中，方法的第一个参数， GenreId，告诉DropDownList 从模型对象或者 ViewBag 对象中寻找名为 GenreId 的属性值，第二个参数用来指出下拉列表默认选中的值。这是创建专辑的表单，所以，没有需要预先选中的值，这里传递了一个 String.Empty 。

### 获取 Post 表单中的值

我们在前面讨论过，对于一个表单可以有两个对应的处理方法，一个处理 Http Get 请求显示表单，另外一个用于处理 Http Post 请求，用于处理提交的表单数据，注意，在控制器中，处理 Http Post 请求的方法需要通过标签 [HttpPost] 进行标注，这样，这个方法将会被 ASP。NET 仅仅用来处理 Post 请求。

//  
// POST: /StoreManager/Create

[HttpPost]

public ActionResult Create(Album album)

{

if (ModelState.IsValid)

{

storeDB.Albums.Add(album);

storeDB.SaveChanges();

return RedirectToAction("Index");

}

ViewBag.GenreId = new SelectList(storeDB.Genres, "GenreId", "Name", album.GenreId);

ViewBag.ArtistId = new SelectList(storeDB.Artists, "ArtistId", "Name", album.ArtistId);

return View(album);

}

这个 Action 方法完成四个任务：

1. 读取表单的数据
2. 检查表单的数据是否通过了验证规则
3. 如果表单通过了验证，保存数据，然后显示更新之后的专辑列表
4. 如果表单没有通过验证，重新显示带有验证提示信息的表单。

### 通过模型绑定获取表单数据

控制器处理的表单提交中包含了流派的标识 GenreId 和 艺术家标识 ArtistId，这些来自下拉列表框，以及通过文本框输入的 Title，Price 等等数据，虽然可以直接通过 FormCollection 来访问表单数据，但是，更好的做法是使用 ASP.NET MVC 内置提供的模型绑定。

当 Action 方法的参数是模型类型的时候，ASP.NET MVC 将会试图使用表单中的数据来填充对象的属性，它通过检查表单参数的名字是否匹配模型对象的属性来进行，例如，对于专辑对象的 GenreId 属性来说，它将会在表单数据中查找名为 GenreId 的值赋予它。当使用标准的模型方式生成视图的时候，表单会使用模型对象的属性名称来生成表单输入项目的名称，这样，在发出表单的时候，请求参数就会正好匹配模型的属性了。

### 验证模型

通过访问 ModelState 的 IsValid 属性就可以检查模型是否通过验证。这里我们还没有为 Album 提供任何的验证规则，后面我们就会提供，所以，这里的验证并没有实际的作用。验证的重要性取决于我们定义的在模型上的验证规则，如果我们已经提供了对于 ModelState 的 IsValid 检查的话，以后增加验证规则就不需要修改这里的代码。

### 保存提交的数据

如果提交的数据通过了验证，下一步就可以保存到数据库中，使用 EF，仅仅需要将模型对象加入到 Albums 的集合中，然后调用 SaveChanges 方法就可以了。

db.Albums.Add(album);

db.SaveChanges();

EF 将会生成适当的 SQL 命令来持久化对象，保存数据之后，我们重定向到专辑列表的页面，以便看到更新之后的内容。这可以通过调用 RedirectToAction 方法，并提供 Action 方法的名称来完成。在这里，我们使用 Index 。

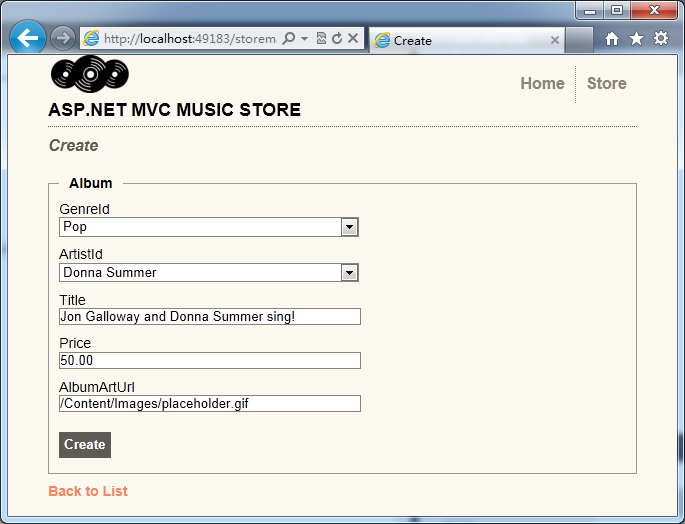
### 显示带有验证提示信息的表单

如果没有通过验证，与 GET 处理的时候相同，我们通过 ViewBag 传递下拉列表的值，通过模型传递其他的数据，验证的提示信息由 @HtmlValidationMessageFor 助手方法自动显示在页面上。

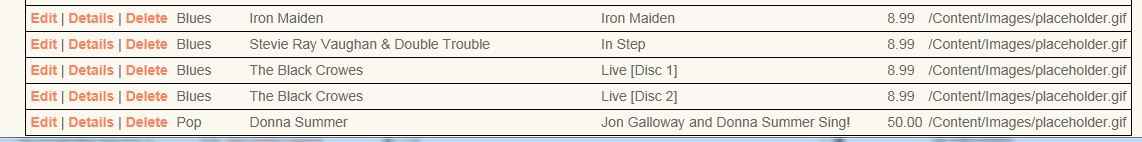
### 测试创建表单

测试一下，运行程序，访问 /StoreManager/Create，你将会看到一个由 StoreManagerController 的 Create 方法处理 Http Get 请求生成的空白表单。

填写一些内容，然后点击创建按钮提交表单。



可以看到内容已经被添加。



### 处理编辑

编辑处理的一对 Action 方法非常类似我们刚刚看到的 Create 处理。由于编辑的情况需要对已经存在的唱片进行处理，在编辑的 Get 方法中，我们使用唱片的 Id 来加载原有的唱片，这个参数通过路由传递过来，实际的代码类似在 Details 中看到的处理。除了专辑对象，同时还有处理下拉列表，所以，这里也通过 ViewBag 来处理，这样就允许我们在传递一个 Model 的同时还通过 ViewBag 传递了两个额外的 SelectList。

//  
// GET: /StoreManager/Edit/5  
public ActionResult Edit(int id)

{

Album album = storeDB.Albums.Find(id);

ViewBag.GenreId = new SelectList(storeDB.Genres, "GenreId", "Name", album.GenreId);

ViewBag.ArtistId = new SelectList(storeDB.Artists, "ArtistId", "Name", album.ArtistId);

return this.View(album);

}

复制代码

处理 Post 请求的 Edit 方法也非常类似于 Create 的 Post 处理方法，仅有的不同就是不用创建一个新的专辑对象加入到集合中，而是将现有的专辑对象，注意已经通过模型绑定获取了请求参数，将这个对象的状态属性 State 修改为 Modified ，这就回告诉 EF 我们正在修改一个存在的专辑对象，而不是创建一个新的。

// POST: /StoreManager/Edit/5  
[HttpPost]

public ActionResult Edit(Album album)

{

if (ModelState.IsValid)

{

storeDB.Entry(album).State = System.Data.EntityState.Modified;

storeDB.SaveChanges();

return RedirectToAction("Index");

}

ViewBag.GenreId = new SelectList(storeDB.Genres, "GenreId", "Name", album.GenreId);

ViewBag.ArtistId = new SelectList(storeDB.Artists, "ArtistId", "Name", album.ArtistId);

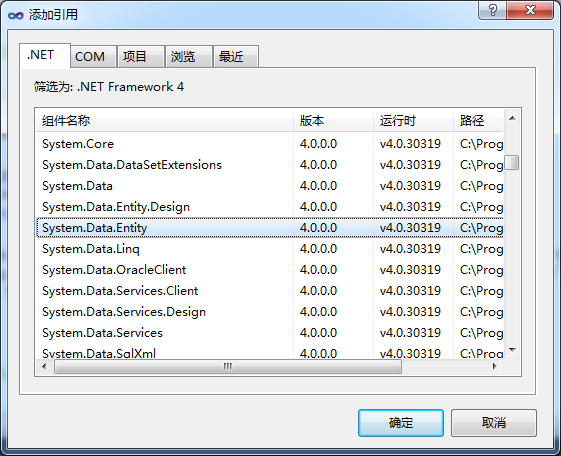
return View(album);

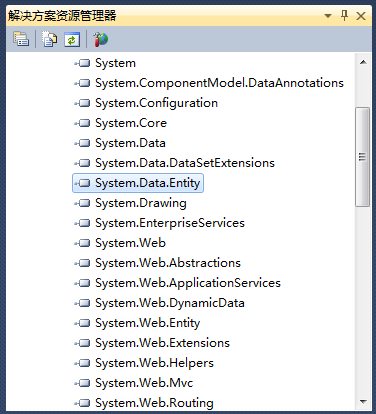
}

注意，这个时候，编译器会提示一个错误：

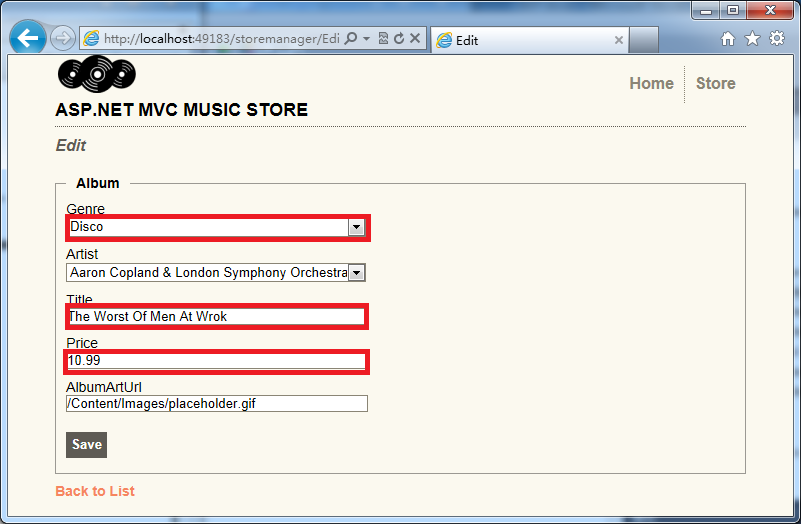
类型“System.Data.EntityState”在未被引用的程序集中定义。必须添加对程序集“System.Data.Entity, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089”的引用。 F:\Lesson\MVC\MvcMusicStore\MvcMusicStore\Controllers\StoreManagerController.cs 76 17 MvcMusicStore

添加这个程序集的引用。

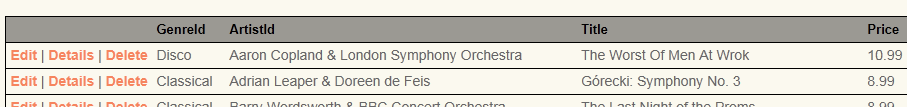




下面我们运行程序测试一下，浏览/StoreManager，然后点击 Edit 链接。



点击 Save, 回到专辑列表，可以看到值已经被更新了。



### 处理删除

删除的处理模式与 Edit 和 Create 一样，使用一个控制器的 Action 显示确认信息，使用另外一个 Action 来处理提交。

删除的 Get 处理非常类似于前面的 Details 处理。

//  
// GET: /StoreManager/Delete/5  
public ActionResult Delete(int id)

{

Album album = storeDB.Albums.Find(id);

return View(album);

}

使用 Delete 视图模板显示强类型的 Album 表单。

默认的 Delete 视图使用模型来显示信息，但是，我们仅仅需要简单地显示确认信息就可以了，把这个视图修改一下，变成下面的样子。

@model MvcMusicStore.Models.Album

@{

ViewBag.Title = "Delete"; }

<h2>

Delete Confirmation</h2>

<p>

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

</p>

@using (Html.BeginForm())

{

<p>

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

</p>

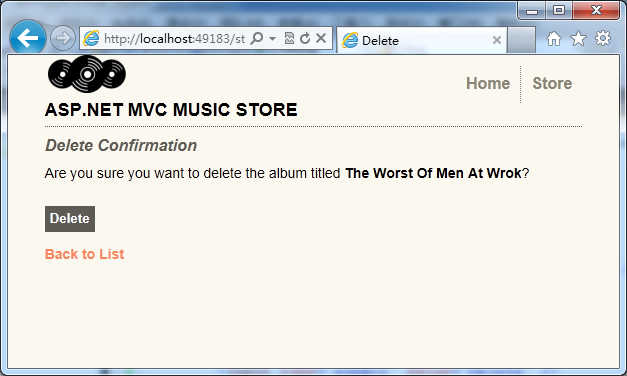
<p>

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

</p>

}

这样，确认页面就成了下面的样子。



点击 Delete 按钮之后，将导致表单发送到服务器，执行 DeleteConfirmed 的 Action 处理方法。

//  
// POST: /StoreManager/Delete/5  
[HttpPost, ActionName("Delete")]

public ActionResult DeleteConfirmed(int id)

{

Album album = storeDB.Albums.Find(id);

storeDB.Albums.Remove(album);

storeDB.SaveChanges();

return RedirectToAction("Index");

}

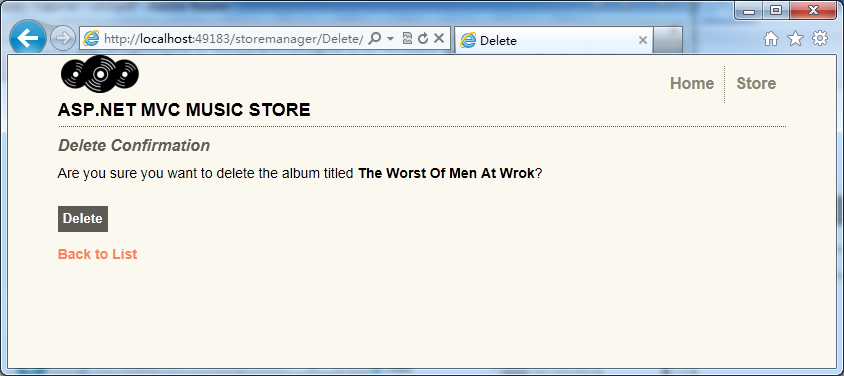
在我们的 Post 处理中，完成了下面的任务

1. 通过专辑的 Id 加载专辑对象
2. 删除专辑，然后保存所做的修改
3. 重新定向到 Index, 显示删除专缉之后的列表。

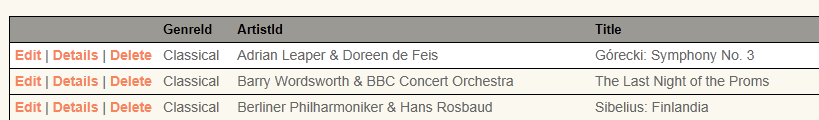
测试一下这些工作，运行程序，选择一个专辑，然后删除它。



点击删除之后，显示了第一张专辑的删除确认页面。



点击删除之后，原来的专辑已经消失了。



### 使用自定义的 HtmlHelper 截断文本内容

我们的管理首页存在一个潜在的问题，我们的专辑名称和艺术家名称可能比较长，这会影响到我们的页面。下面我们创建一个自定义的 HtmlHelper 扩展来使得我们可以轻易地截断比较长的字符串。在视图中显示成这样的效果

Rozor 的 @helper 可以使这个工作变得很容易实现，打开 /View/StoreManager/Index.cshtml视图，直接在 @model 的下面增加下面的代码。

@helper Truncate(string input, int length)

{

if (input.Length <= length)

{

@input

}

else

{

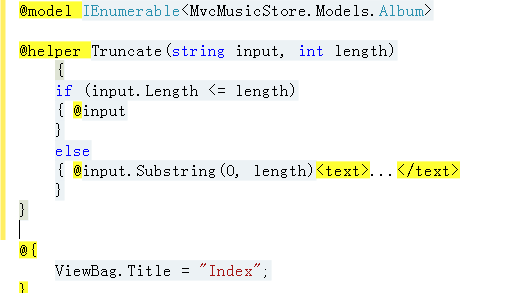
@input.Substring(0, length)<text>...</text>

}

}

复制代码

页面将会变成下面的样子。



这个助手方法取得一个字符串和一个最大长度，如果字符串长度小于指定的长度，就直接返回这个串，如果大于最大长度，那么，就截断字符串，然后在剩下的最后增加 “…”。

现在，我们使用这个截断字符串的助手方法来确保专辑名称和艺术家名称小于 25 个字符，完整的视图内容如下所示。

@model IEnumerable<MvcMusicStore.Models.Album>

@helper Truncate(string input, int length)

{

if (input.Length <= length)

{

@input

}

else

{

@input.Substring(0, length)<text>...</text>

}

}

@{

ViewBag.Title = "Index";

}

<h2>

Index</h2>

<p>

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

</p>

<table>

<tr>

<th>

Genre

</th>

<th>

Artist

</th>

<th>

Title

</th>

<th>

Price

</th>

<th>

</th>

</tr>

@foreach (var item in Model)

{

<tr>

<td>

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

</td>

<td>

@Truncate(item.Artist.Name, 25)

</td>

<td>

@Truncate(item.Title, 25)

</td>

<td>

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

</td>

<td>

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

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

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

</td>

</tr>

}

</table>

注意，这里展示了简单地在一个视图中使用 HtmlHelper 的方式，关于更多的信息，请参考：

<http://bit.ly/mvc3-helper-options>

如果希望在多个视图中使用这个扩展的助手方法，可以如下进行。

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) + "...";

}

}

}

}

注意，使用扩展方法必须 using 其命名空间

@{

ViewBag.Title = "Home Page";

}

@using MvcMusicStores.Helpers  
<h2>@Html.Truncate(ViewBag.Message as string, 8)</h2>

你也可以在配置文件 web.config 中统一定义。

<system.web.webPages.razor>

<host factoryType="System.Web.Mvc.MvcWebRazorHostFactory, System.Web.Mvc, Version=3.0.0.0, Culture=neutral, PublicKeyToken=31BF3856AD364E35"/>

<pages pageBaseType="System.Web.Mvc.WebViewPage">

<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>

</system.web.webPages.razor>

# ASP.NET MVC 音乐商店 - 6. 使用 DataAnnotations 进行模型验证

在前面的创建专辑与编辑专辑的表单中存在一个问题：我们没有进行任何验证。字段的内容可以不输入，或者在价格的字段中输入一些字符，在执行程序的时候，这些错误会导致数据库保存过程中出现错误，我们将会看到来自数据库的错误信息。

通过为模型类增加数据描述的 DataAnnotations ，我们可以容易地为应用程序增加验证的功能。DataAnnotations 允许我们描述希望应用在模型属性上的验证规则，ASP.NET MVC 将会使用这些 DataAnnotations ，然后将适当的验证信息返回给用户。

### 为专辑表单增加验证

我们将会使用下列的 DataAnnotations

* Required 必须 – 表示这个属性是必须提供内容的字段
* DisplayName 显示名 – 定义表单字段的提示名称
* StringLength 字符串长度 – 定义字符串类型的属性的最大长度
* Range 范围 – 为数字类型的属性提供最大值和最小值
* Bind 绑定 – 列出在将请求参数绑定到模型的时候，包含和不包含的字段
* ScaffoldColumn 支架列 - 在编辑表单的时候，需要隐藏起来的的字符

注意：更多关于模型验证的信息，请参考：[http://msdn.microsoft.com/zh-cn/library/ee256141(VS.100).aspx](http://msdn.microsoft.com/zh-cn/library/ee256141%28VS.100%29.aspx)

打开 Album 类，首先增加下面的 using 语句，这些语句引用了 DataAnnotations 使用的命名空间。

using System.ComponentModel;

using System.ComponentModel.DataAnnotations;

using System.Web.Mvc;

然后，更新属性，增加显示和验证的 DataAnnotations

namespace MvcMusicStore.Models

{

[Bind(Exclude = "AlbumId")]

public class Album

{

[ScaffoldColumn(false)]

public int AlbumId { get; set; }

[DisplayName("Genre")]

public int GenreId { get; set; }

[DisplayName("Artist")]

public int ArtistId { get; set; }

[Required(ErrorMessage = "An Album Title is required")]

[StringLength(160)]

public string Title { get; set; }

[Required(ErrorMessage = "Price is required")]

[Range(0.01, 100.00,

ErrorMessage = "Price must be between 0.01 and 100.00")]

public decimal Price { get; set; }

[DisplayName("Album Art URL")]

[StringLength(1024)]

public string AlbumArtUrl { get; set; }

public virtual Genre Genre { get; set; }

public virtual Artist Artist { get; set; }

}

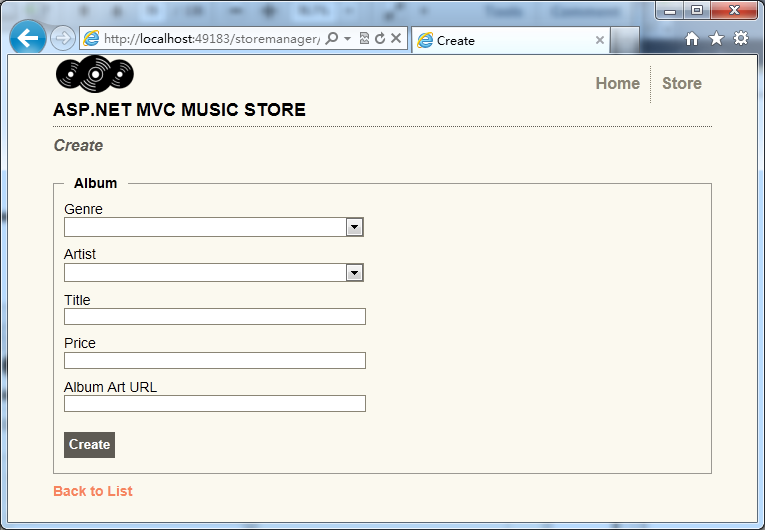
}

然后，将专辑 Album 的属性 Genre 和 Artist 设置为虚拟的 virtual ，这将会使 EF-Code First 使用延迟加载。

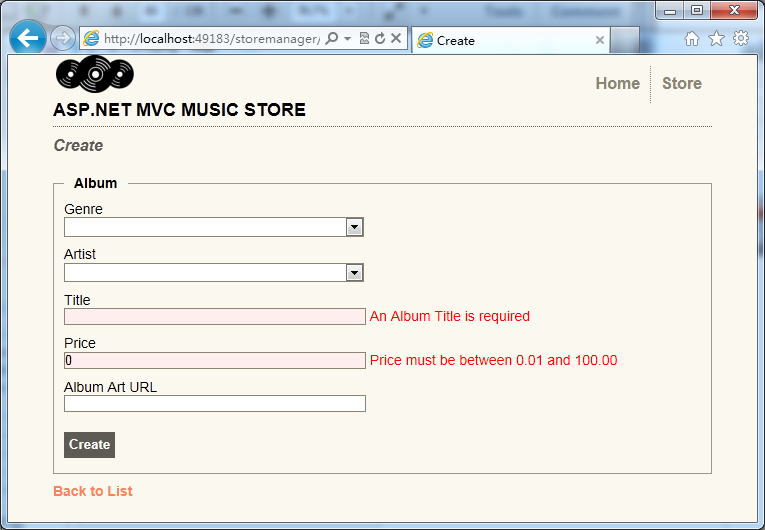
public virtual Genre Genre { get; set; }

public virtual Artist Artist { get; set; }

为专缉修改完成之后，我们的创建和编辑界面立即就会验证字段，并且使用我们提供的显示名称，例如 AlbumArtUrl 将会成为 Album Art URL 等等。运行程序，浏览 /StoreManager/Create.



下一步，我们特意输入一些破坏验证规则的数据，在价格字段中输入 0, 将标题字段的内容保留为空白，当我们点击创建的时候，我们将会看到表单中不符合验证规则的字段显示了验证的错误提示信息。



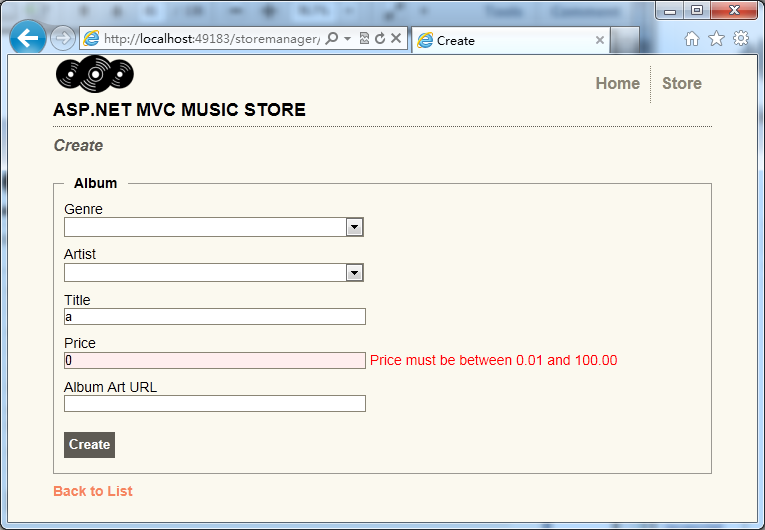
### 测试客户端验证

对于应用程序来说，服务器端验证非常重要，因为用户可能绕过了客户端验证，实际上，Web 页面仅仅实现服务器端验证存在三个显著的问题：

1. 在提交表单的时候，用户必须等待，验证在服务器端进行，需要将验证的结果发送回浏览器。
2. 用户不能在输入错误的时候立即得到回应，以便通过验证规则的检查。
3. 我们把可以在浏览器完成的工作交给了服务器，浪费了服务器的资源。

幸运的是，ASP.NET MVC3 支架模板还提供了内建的客户端验证，不需要我们做额外的工作就可以使用。

在必须输入的标题字段中输入一个字符，验证的错误提示信息立即就消失了。



这里，我们要注意几点：

页面中已经引用了 jQuery 的脚本。

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

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

在 web.config 中，已经默认支持了客户端验证。

<appSettings>

<add key="ClientValidationEnabled" value="true"/>

<add key="UnobtrusiveJavaScriptEnabled" value="true"/>

</appSettings>

# [ASP.NET MVC 音乐商店 - 7.成员管理和授权](http://www.cnblogs.com/haogj/archive/2011/11/18/2253140.html)

目前，我们的 Store Manager 可以被任何人访问，让我们限制一下对站点管理的访问。

### 增加 AccountController 和 相应的视图

在全功能的 ASP.NET MVC3 Wb 应用程序与空的 ASP.NET MVC3 应用程序模板之间的区别在于，空的应用程序模板中没有包含账号控制器，我们可以从新创建的全功能的 ASP.NET MVC 应用程序中复制相应的文件，来增加账号控制器。

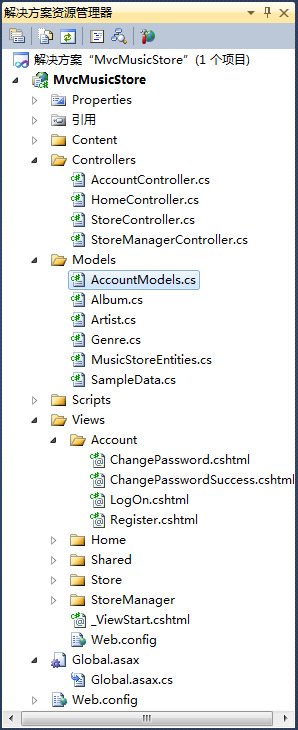
另外，在你下载的 MvcMusicStore-Assets.zip 文件中，也包含了账号管理的文件。

复制下面的内容到你的网站中。

1. 复制 AccountController.cs 到 Controllers 目录中
2. 复制 AccountModels.cs 到 Models 目录中
3. 在 Views 目录中创建 Account 目录，然后复制相应的四个视图。

注意修改控制器和模型类的命名空间为 MvcMusicStore。AccountController 类应该为 MvcMusicStore.Controllers 命名空间，AccountModels 类应用使用 MvcMusicStore.Models 命名空间。

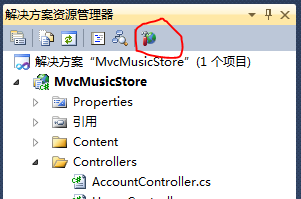
更新之后的解决方案看起来如下所示：



### 使用 ASP.NET 站点配置工具增加管理员账号

在我们授权访问网站之前，需要先创建一个管理员账号，最简单的方式就是使用内建的 ASP.NET 站点管理工具创建。

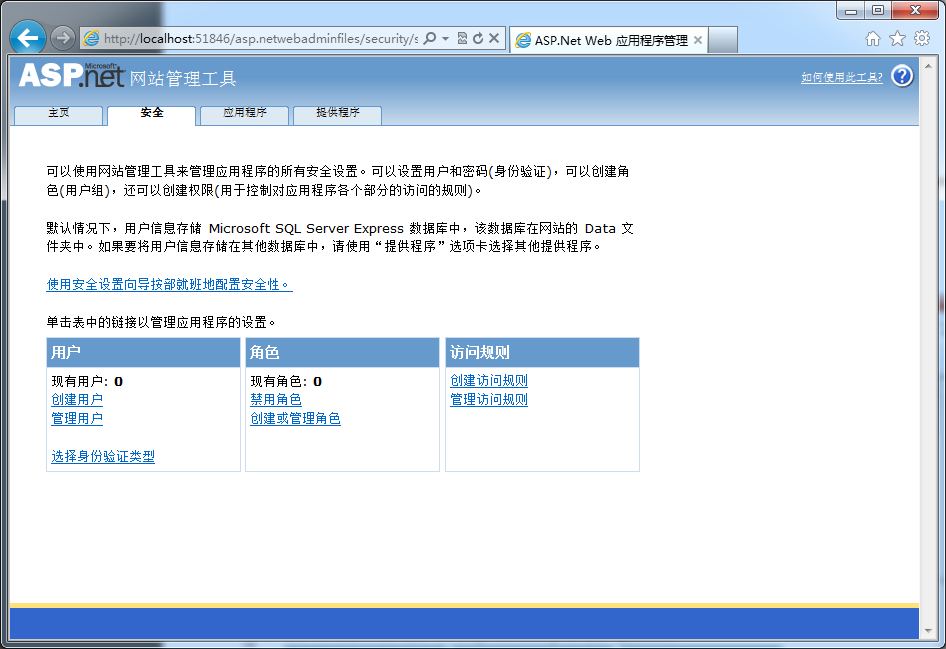
在解决方案管理器上，点击站点配置工具



稍等片刻，将会弹出一个浏览器窗口，点击首页中的安全选项卡，然后，点击屏幕中间的“启用角色”链接。



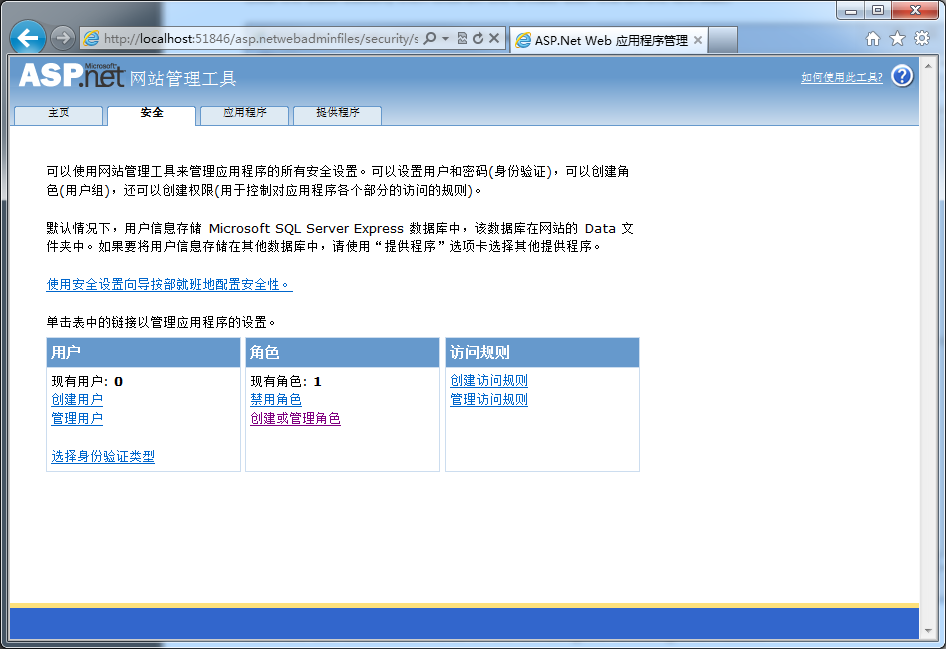
点击“创建或管理角色”链接。



在角色名称的输入框中输入 “Administrator”，点击增加角色按钮。



点击返回按钮，点击屏幕左边创建用户的链接。



使用下面的信息填写用户资料。

 Field 　　　　　         Value

User Name 　　          Administrator

Password 　　  　　　  password123!

Confirm Password 　　password123!

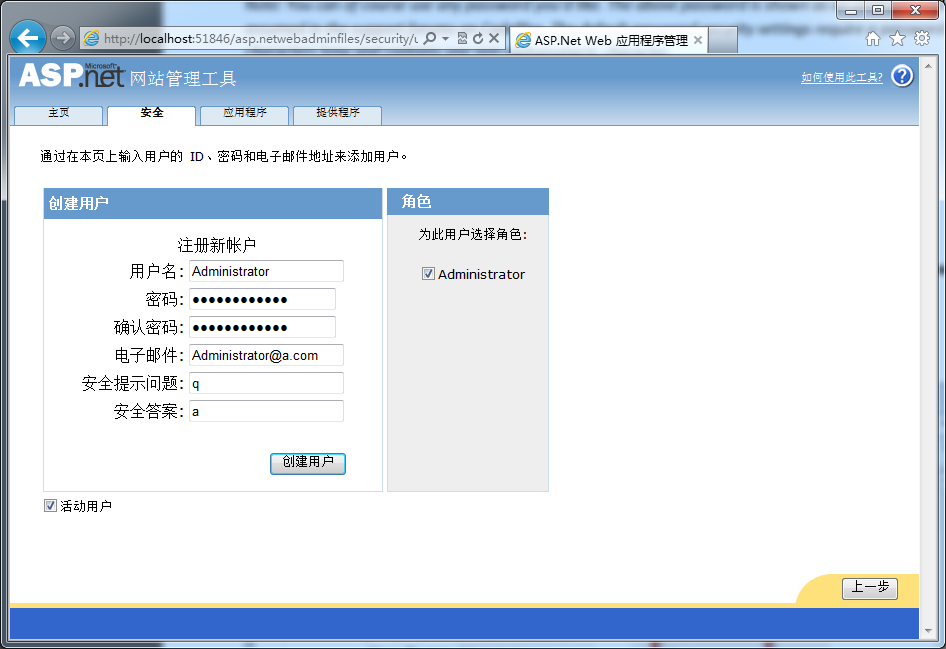
E-mail 　　　　　　     (any e-mail address will work)

Security Question 　　(whatever you like)

Security Answer 　　  (whatever you like)

注意：你可以使用任何你愿意的口令，但是，默认的密码规则要求口令至少为 7 个字符，其中包含至少一个非字母和数字的字符。

注意选择 Administrator 角色，



用户创建成功了。



现在，你可以关掉这个窗口。

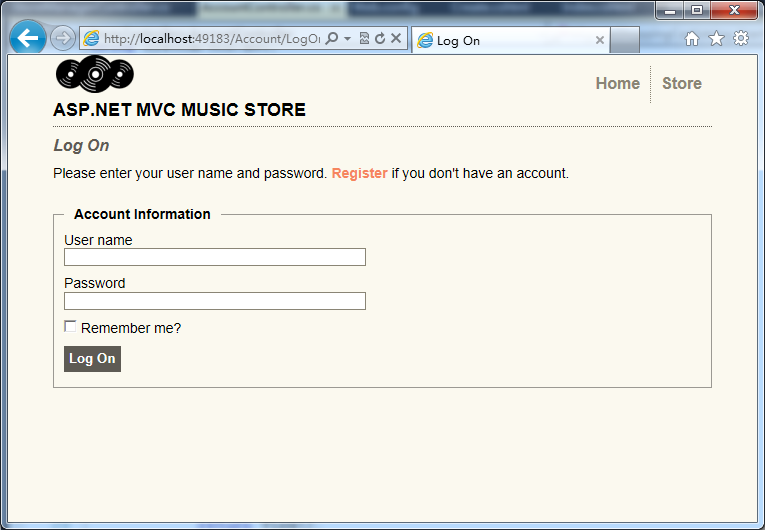
### 基于角色的授权

现在，我们可以使用 [Authorize] 标注来限制对 StoreManager 控制器的访问了，设置访问 StoreManager 任何 Action 的用户必须拥有 Administrator 的角色。

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

注意，[Authorize] 也可以用在 Action 方法上。

现在浏览 /StoreManager ，将会被导航到登陆页面上。



 使用具有 Administrator 角色的账号登陆之后，就可以进入 StoreManager 了。

# [ASP.NET MVC 音乐商店 - 8. 使用 Ajax 更新的购物车](http://www.cnblogs.com/haogj/archive/2011/11/20/2255515.html)

在这个项目中，我们将允许用户在没有注册登录的情况下将专辑加入购物车，但是，在完成结 账的时候必须完成注册工作。购物和结账将会被分离到两个控制器中：一个 ShoppingCart 控制器，允许匿名用户使用购物车，另一个 Checkout 控制器处理结账。我们先从购物车的控制器开始，然后在下一部分来处理结帐。

### 加入购物车，订单和订单明细的模型类

在购物车和结账的处理中将会使用到一些新的类，在 Models 文件夹上右键，然后使用下面的代码增加一个新的类 Cart.

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

这个类非常类似我们前面使用的类，除了 RecordId 属性上的[Key] 标注之外。我们的购物车拥有一个字符串类型的名为 CartId 的标识，用来允许匿名用户使用购物车，但是,CartId 并不是表的主键，表的主键是整数类型的名为 RecordId的字段，根据约定，EF CodeFirst 将会认为表的主键名为 CartId 或者 Id，不过，如果需要的话，我们可以很容易地通过标注或者代码来重写这个规则。这里例子演示了在使用 EF CodeFirst 的时候。当我们的表不是约定的样子时，我们也不必被约定所局限。

下一步，使用下面的代码增加订单 Order 类。

using System.Collections.Generic;  
namespace MvcMusicStore.Models  
{  
 public partial class Order  
 {  
 public int OrderId { get; set; }  
 public string Username { get; set; }  
 public string FirstName { get; set; }  
 public string LastName { get; set; }  
 public string Address { get; set; }  
 public string City { get; set; }  
 public string State { get; set; }  
 public string PostalCode { get; set; }  
 public string Country { get; set; }  
 public string Phone { get; set; }  
 public string Email { get; set; }  
 public decimal Total { get; set; }  
 public System.DateTime OrderDate { get; set; }  
 public List<OrderDetail> OrderDetails { get; set; }  
 }  
}

这个类跟踪订单的汇总和发货信息，它的结构也不复杂，订单依赖我们这里还没有定义的一个类，通过 OrderDetails 属性来表示订单的明细。我们来定义一下这个 OrderDetail 类。

[复制代码](javascript:void(0);)

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

[复制代码](javascript:void(0);)

把我们的 MusicStoreEntities 更新一下，以便包含我们新定义的模型类，包括艺术家 Artist，更新之后的 MusicStoreEntities 如下所示。

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

### 管理购物车业务逻辑

下一步，我们在 Models 文件夹中创建 ShoppingCart 类，ShoppingCart 模型类处理  Cart 表的数据访问，另外，它还需要处理在购物车中增加或者删除项目的业务逻辑。

因为我们并不希望用户必须登录系统才可以使用购物车，对于没有登录的用户，我们需要为他们创建一个临时的唯一标识，这里使用 GUID，或者被称为全局唯一标识符，对于已经登录的用户，我们直接使用他们的名称，这个表示我们保存在 Session 中。

注意：Session 会话可以很方便地存储用户的信息，在用户离开站点之后，这些信息将会过期，滥用 Session 信息会对大型站点产生影响，我们这里使用 Session 达到演示目的。

ShoppingCart 类提供了如下的方法：

**AddToCart**, 将专辑作为参数加入到购物车中，在 Cart 表中跟踪每个专辑的数量，在这个方法中，我们将会检查是在表中增加一个新行，还是仅仅在用户已经选择的专辑上增加数量。

**RemoveFromCart**，通过专辑的标识从用户的购物车中将这个专辑的数量减少 1，如果用户仅仅剩下一个，那么就删除这一行。

**EmptyCart**，删除用户购物车中所有的项目。

**GetCartItems**，获取购物项目的列表用来显示或者处理。

**GetCount**，获取用户购物车中专辑的数量

**GetTotal**，获取购物车中商品的总价

**CreateOrder**，将购物车转换为结账处理过程中的订单。

**GetCart** ，这是一个静态方法，用来获取当前用户的购物车对象，它使用 GetCartId 方法来读取保存当前 Session 中的购物车标识，GetCartId 方法需要 HttpContextBase 以便获取当前的 Session。

实际的代码如下：

namespace MvcMusicStore.Models  
{  
 public partial class ShoppingCart  
 {  
 MusicStoreEntities storeDB = new MusicStoreEntities();  
  
 string ShoppingCartId { get; set; }  
 public const string CartSessionKey = "CartId";  
  
 public static ShoppingCart GetCart(HttpContextBase context)  
 {  
 var cart = new ShoppingCart();  
 cart.ShoppingCartId = cart.GetCartId(context);  
 return cart;  
 }  
 // Helper method to simplify shopping cart calls  
 public static ShoppingCart GetCart(Controller controller)  
 {  
 return GetCart(controller.HttpContext);  
 }  
 public void AddToCart(Album album)  
 {  
 // Get the matching cart and album instances  
 var cartItem = storeDB.Carts.SingleOrDefault(  
 c => c.CartId == ShoppingCartId  
 && c.AlbumId == album.AlbumId);  
 if (cartItem == null)  
 {  
 // Create a new cart item if no cart item exists  
 cartItem = new Cart  
 {  
 AlbumId = album.AlbumId,  
 CartId = ShoppingCartId,  
 Count = 1,  
 DateCreated = DateTime.Now  
 };  
 storeDB.Carts.Add(cartItem);  
 }  
 else  
 {  
 // If the item does exist in the cart, then add one to the quantity  
 cartItem.Count++;  
 }  
 // Save changes  
 storeDB.SaveChanges();  
 }  
  
 public int RemoveFromCart(int id)  
 {  
 // Get the cart  
 var cartItem = storeDB.Carts.Single(  
 cart => cart.CartId == ShoppingCartId  
 && cart.RecordId == id);  
  
 int itemCount = 0;  
 if (cartItem != null)  
 {  
 if (cartItem.Count > 1)  
 {  
 cartItem.Count--;  
 itemCount = cartItem.Count;  
 }  
 else  
 {  
 storeDB.Carts.Remove(cartItem);  
 }  
 // Save changes  
 storeDB.SaveChanges();  
 }  
 return itemCount;  
 }  
 public void EmptyCart()  
 {  
 var cartItems = storeDB.Carts.Where(cart => cart.CartId == ShoppingCartId);  
 foreach (var cartItem in cartItems)  
 {  
 storeDB.Carts.Remove(cartItem);  
 }  
 // Save changes  
 storeDB.SaveChanges();  
 }  
 public List<Cart> GetCartItems()  
 {  
 return storeDB.Carts.Where(cart => cart.CartId == ShoppingCartId).ToList();  
 }  
 public int GetCount()  
 {  
 // Get the count of each item in the cart and sum them up  
 int? count = (from cartItems in storeDB.Carts  
 where cartItems.CartId == ShoppingCartId  
 select (int?)cartItems.Count).Sum();  
 // Return 0 if all entries are null  
 return count ?? 0;  
 }  
  
 public decimal GetTotal()  
 {  
 // Multiply album price by count of that album to get  
 // the current price for each of those albums in the cart  
 // sum all album price totals to get the cart total  
  
 decimal? total = (from cartItems in storeDB.Carts  
 where cartItems.CartId == ShoppingCartId  
 select (int?)cartItems.Count \* cartItems.Album.Price).Sum();  
 return total ?? decimal.Zero;  
 }  
 public int CreateOrder(Order order)  
 {  
 decimal orderTotal = 0;  
 var cartItems = GetCartItems();  
 // Iterate over the items in the cart, adding the order details for each  
 foreach (var item in cartItems)  
 {  
 var orderDetail = new OrderDetail  
 {  
 AlbumId = item.AlbumId,  
 OrderId = order.OrderId,  
 UnitPrice = item.Album.Price,  
 Quantity = item.Count  
 };  
 // Set the order total of the shopping cart  
 orderTotal += (item.Count \* item.Album.Price);  
 storeDB.OrderDetails.Add(orderDetail);  
 }  
 // Set the order's total to the orderTotal count  
 order.Total = orderTotal;  
 // Save the order  
 storeDB.SaveChanges();  
 // Empty the shopping cart  
 EmptyCart();  
 // Return the OrderId as the confirmation number  
 return order.OrderId;  
 }  
  
 // We're using HttpContextBase to allow access to cookies.  
 public string GetCartId(HttpContextBase context)  
 {  
 if (context.Session[CartSessionKey] == null)  
 {  
 if (!string.IsNullOrWhiteSpace(context.User.Identity.Name))  
 {  
 context.Session[CartSessionKey] = context.User.Identity.Name;  
 }  
 else  
 {  
 // Generate a new random GUID using System.Guid class  
  
 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();  
 }  
 }  
}

[复制代码](javascript:void(0);)

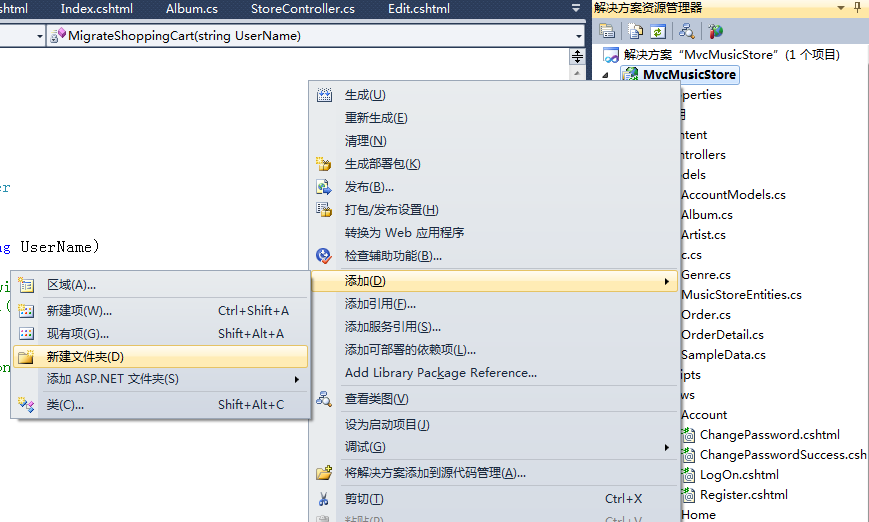
### 视图模型

我们的 ShoppingCart 控制器需要向视图传递复杂的信息，这些信息与现有的模型并不完全匹配，我们不希望修改模型来适应视图的需要；模型类应该表示领域信息，而不是用户界面。一 个解决方案是使用 ViewBag 来向视图传递信息，就像我们在 Store Manager 中的下列列表处理中那样，但是通过 ViewBag 来传递大量信息就不好管理了。

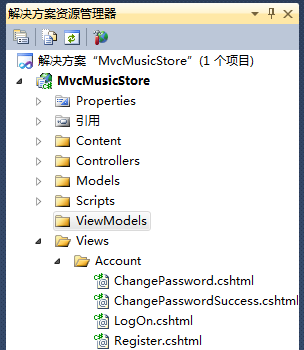
另外一个解决方案是使用视图模型模式，使用这个模式，我们需要创建强类型的用于视图场景的类来表示信息，这个类拥有视图所需要的值或者内容。我们的控制器填充信息，然后传递这种类的对象供视图使用，这样就可以得到强类型的、编译时检查支持，并且在视图模板中带有智能提示。

我们将会创建两个视图模型用于我们的 ShoppingCart 控制器：ShoppingCartViewModel 将会用于用户的购物车，而 ShoppingCartRemoveViewModel 会用于在购物车中删除内容时的确认提示信息。

首先在项目中创建 ViewModels 文件夹来组织我们的项目文件，在项目上点击鼠标的右键，然后选择添加 –〉新文件夹。



命名为 ViewModels



下一步，在 ViewModels 文件夹中增加 ShoppingCartViewModel 类，它包括两个属性，一个 CartItem 的列表，另外一个属性是购物中的总价。

using System.Collections.Generic;  
using MvcMusicStore.Models;  
  
namespace MvcMusicStore.ViewModels  
{  
 public class ShoppingCartViewModel  
 {  
 public List<Cart> CartItems { get; set; }  
 public decimal CartTotal { get; set; }  
 }  
}

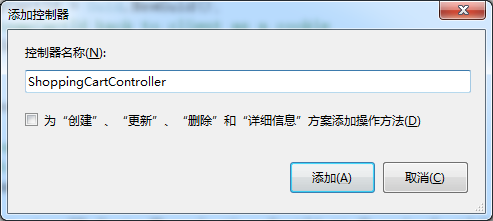
然后，增加 ShoppingCartRemoveViewModel 类，它包括五个属性。

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

### Shopping Cart 控制器

Shopping Cart 控制器有三个主要的目的：增加项目到购物车，从购物车中删除项目，查看购物车中的项目。控制器使用到我们刚刚创建的三个 类：ShoppingCartViewModel，ShoppingCartRemoveViewModel 和 ShoppingCart，像 StoreController 和 StoreManagerController 一样，我们在控制器中增加一个 MusicStoreEntities 字段来操作数据。

在项目中使用空的控制器模板创建 ShoppingCart 控制器



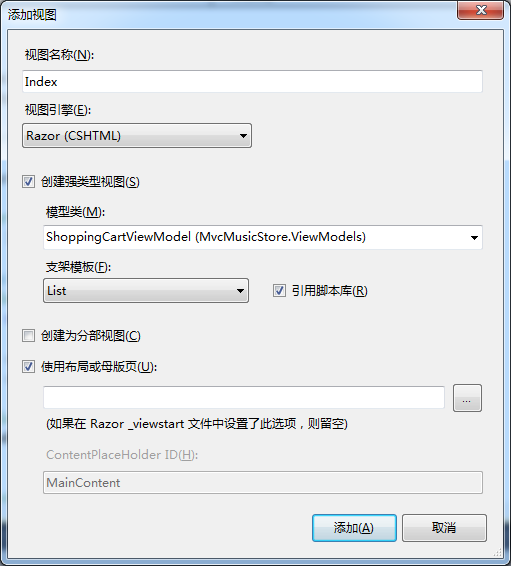
下面是已经完成的控制器代码，Index 和 Add 方法看起来非常熟悉。Remove 和 CartSummary 这两个 Action 方法处理两种特定的场景，我们将在后面讨论。

using MvcMusicStore.Models;  
using MvcMusicStore.ViewModels;  
  
namespace MvcMusicStore.Controllers  
{  
 public class ShoppingCartController : Controller  
 {  
 MusicStoreEntities storeDB = new MusicStoreEntities();  
 //  
 // GET: /ShoppingCart/  
 public ActionResult Index()  
 {  
 var cart = ShoppingCart.GetCart(this.HttpContext);  
 // Set up our ViewModel  
 var viewModel = new ShoppingCartViewModel  
 {  
 CartItems = cart.GetCartItems(),  
 CartTotal = cart.GetTotal()  
 };  
 // Return the view  
 return View(viewModel);  
 }  
 //  
 // GET: /Store/AddToCart/5  
 public ActionResult AddToCart(int id)  
 {  
 // Retrieve the album from the database  
 var addedAlbum = storeDB.Albums  
 .Single(album => album.AlbumId == id);  
 // Add it to the shopping cart  
 var cart = ShoppingCart.GetCart(this.HttpContext);  
 cart.AddToCart(addedAlbum);  
 // Go back to the main store page for more shopping  
 return RedirectToAction("Index");  
 }  
 //  
 // AJAX: /ShoppingCart/RemoveFromCart/5  
 [HttpPost]  
 public ActionResult RemoveFromCart(int id)  
 {  
 // Remove the item from the cart  
 var cart = ShoppingCart.GetCart(this.HttpContext);  
 // Get the name of the album to display confirmation  
 string albumName = storeDB.Carts  
 .Single(item => item.RecordId == id).Album.Title;  
 // Remove from cart  
 int itemCount = cart.RemoveFromCart(id);  
 // Display the confirmation message  
 var results = new ShoppingCartRemoveViewModel  
 {  
 Message = Server.HtmlEncode(albumName) +  
 " has been removed from your shopping cart.",  
 CartTotal = cart.GetTotal(),  
 CartCount = cart.GetCount(),  
 ItemCount = itemCount,  
 DeleteId = id  
 };  
 return Json(results);  
 }  
 //  
 // GET: /ShoppingCart/CartSummary  
 [ChildActionOnly]  
 public ActionResult CartSummary()  
 {  
 var cart = ShoppingCart.GetCart(this.HttpContext);  
 ViewData["CartCount"] = cart.GetCount();  
 return PartialView("CartSummary");  
 }  
 }  
}

[复制代码](javascript:void(0);)

### 使用 jQuery 进行 Ajax 更新

下面我们将创建 ShoppingCart 的 Index Action 视图，这个视图使用强类型的 ShoppingCartViewModel ，像以前的视图一样，使用 List 视图模板。



在这里，我们不使用 Html.ActionLink 从购物车中删除项目，我们将会使用 JQuery 来包装客户端使用 RemoveLink 的类所有超级链接元素的事件，不是提交表单，而是通过客户端的事件向 RemoveFromCart 控制器方法发出 Ajax 请求，然后 RemoveFromCart 返回 JSON 格式的结果，这个结果被发送到我们在 AjaxOptions 的 OnSucess 参数中创建的 JavaScript 函数，在这里是 handleUpdate，handleUpdate 函数解析 JSON 格式的结果，然后通过 jQuery 执行下面的四个更新。

1. 从列表中删除专辑
2. 更新头部的购物车中的数量
3. 向用户显示更新信息
4. 更新购物车中的总价

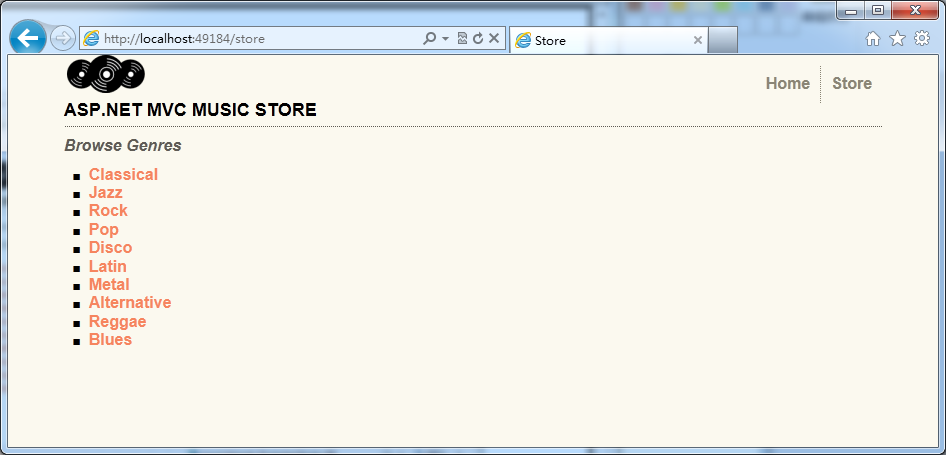
因为在 Index 视图中我们处理了删除的场景，我们就不再需要为 RemoveFromCart 方法增加额外的视图。下面是视图的完整代码。

@model MvcMusicStore.ViewModels.ShoppingCartViewModel   
@{   
 ViewBag.Title = "Shopping Cart";   
}  
<script src="/Scripts/jquery-1.4.4.min.js" type="text/javascript"></script>  
<script type="text/javascript">  
 $(function () {  
 // Document.ready -> link up remove event handler  
 $(".RemoveLink").click(function () {  
 // Get the id from the link  
 var recordToDelete = $(this).attr("data-id");  
 if (recordToDelete != '') {  
 // Perform the ajax post  
 $.post("/ShoppingCart/RemoveFromCart", { "id": recordToDelete },  
function (data) {  
 // Successful requests get here  
 // Update the page elements  
 if (data.ItemCount == 0) {  
 $('#row-' + data.DeleteId).fadeOut('slow');  
 } else {  
 $('#item-count-' + data.DeleteId).text(data.ItemCount);  
 }  
 $('#cart-total').text(data.CartTotal);  
 $('#update-message').text(data.Message);  
 $('#cart-status').text('Cart (' + data.CartCount + ')');  
});  
 }  
 });  
 });  
 function handleUpdate() {  
 // Load and deserialize the returned JSON data  
 var json = context.get\_data();  
 var data = Sys.Serialization.JavaScriptSerializer.deserialize(json);  
 // Update the page elements  
 if (data.ItemCount == 0) {  
 $('#row-' + data.DeleteId).fadeOut('slow');  
 } else {  
 $('#item-count-' + data.DeleteId).text(data.ItemCount);  
 }  
 $('#cart-total').text(data.CartTotal);  
 $('#update-message').text(data.Message);  
 $('#cart-status').text('Cart (' + data.CartCount + ')');  
 }  
</script>  
<h3>  
 <em>Review</em> your cart:  
</h3>  
<p class="button">  
 @Html.ActionLink("Checkout >>", "AddressAndPayment", "Checkout")  
</p>  
<div id="update-message">  
</div>  
<table>  
 <tr>  
 <th>  
 Album Name  
 </th>  
 <th>  
 Price (each)  
 </th>  
 <th>  
 Quantity  
 </th>  
 <th>  
 </th>  
 </tr>  
 @foreach (var item in Model.CartItems)  
 { <tr id="row-@item.RecordId">  
 <td>  
 @Html.ActionLink(item.Album.Title, "Details", "Store", new { id = item.AlbumId }, null)  
 </td>  
 <td>  
 @item.Album.Price  
 </td>  
 <td id="item-count-@item.RecordId">  
 @item.Count  
 </td>  
 <td>  
 <a href="#" class="RemoveLink" data-id="@item.RecordId">Remove from cart</a>  
 </td>  
 </tr>  
 }  
 <tr>  
 <td>  
 Total  
 </td>  
 <td>  
 </td>  
 <td>  
 </td>  
 <td id="cart-total">  
 @Model.CartTotal  
 </td>  
 </tr>  
</table>

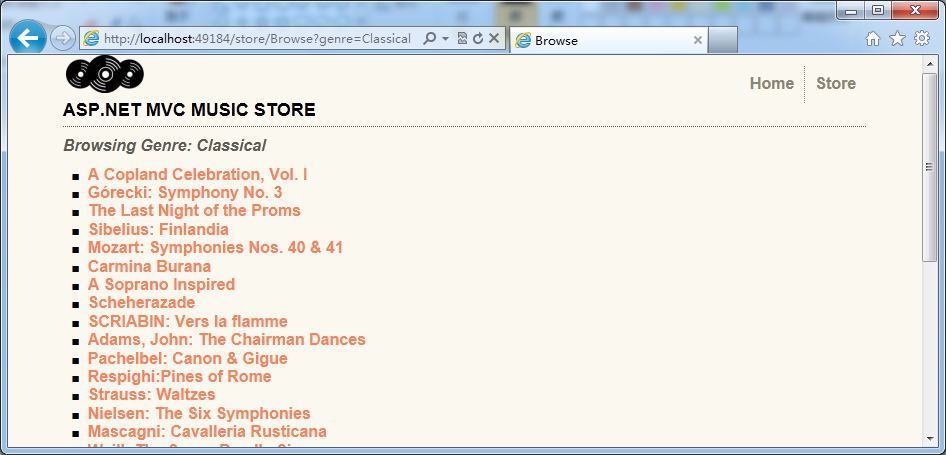
为了测试一下，我们需要向购物车中增加一些项目，更新 Store 的 Details 视图包含添加到购物车按钮，在这里，我们还需要包含我们后来增加的专辑的一些额外信息，流派，艺术家，价格等等。更新后的视图如下所示。

@model MvcMusicStore.Models.Album  
@{  
 ViewBag.Title = "Album - " + Model.Title;   
}  
<h2>@Model.Title</h2>  
<p>  
 <img alt="@Model.Title" src="@Model.AlbumArtUrl" />  
</p>  
<div id="album-details">  
 <p>  
 <em>Genre:</em> @Model.Genre.Name  
 </p>  
 <p>  
 <em>Artist:</em> @Model.Artist.Name  
 </p>  
 <p>  
 <em>Price:</em> @String.Format("{0:F}", Model.Price)  
 </p>  
 <p class="button">  
 @Html.ActionLink("Add to cart", "AddToCart", "ShoppingCart", new { id = Model.AlbumId }, "")  
 </p>  
</div>

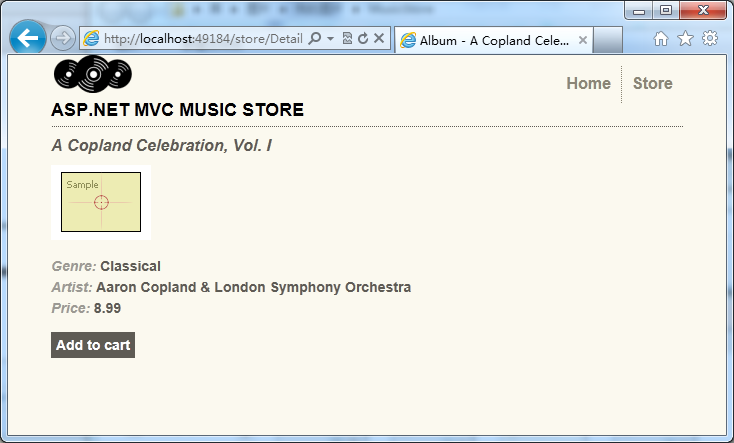
现在，我们可以在商店中通过购物车来购买和删除一些项目了。运行程序，浏览 Store 控制器的 Index 。



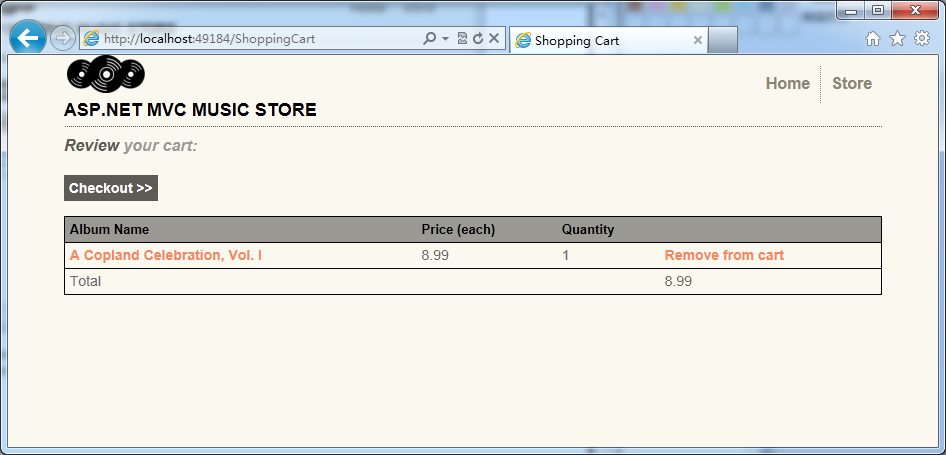
然后，点击某个分类来查看专辑的列表。



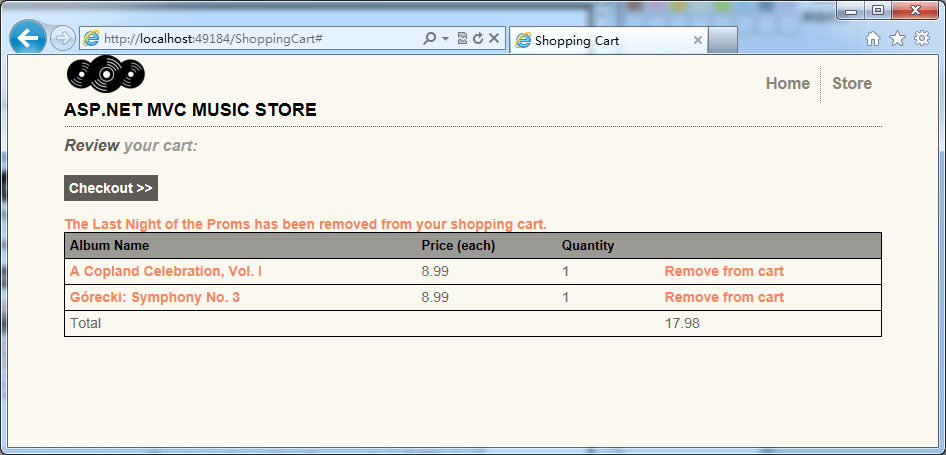
点击某个专辑来显示专辑的详细内容，现在已经有了加入购物车的按钮。



点击加入购物车之后，可以在购物车中看到。



在购物车中，可以点击从购物车中删除的链接，将会看到 Ajax 更新购物车的效果。

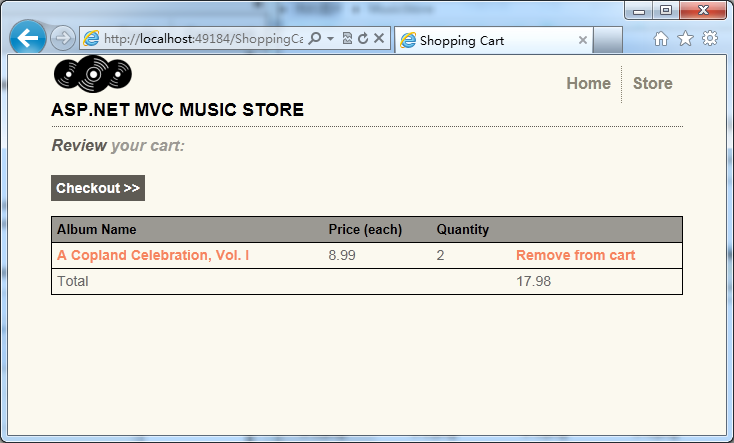


现在的购物车允许没有注册的用户使用购物车添加项目，在下一部分，我们将允许匿名用户注册和完成结账的处理。

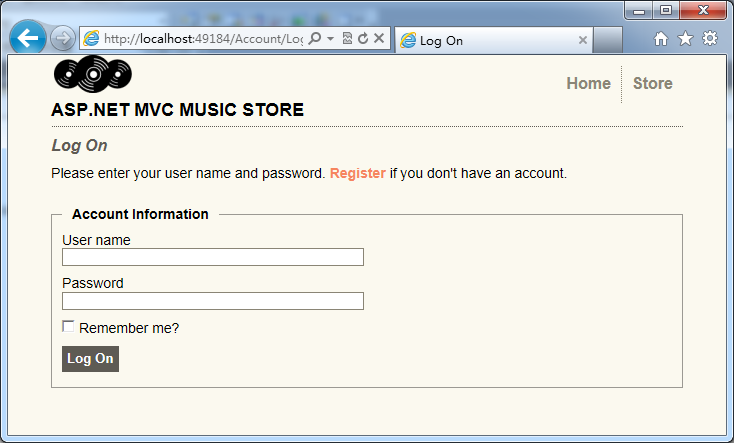
# [ASP.NET MVC 音乐商店 - 9. 注册和结账](http://www.cnblogs.com/haogj/archive/2011/11/20/2255675.html)

在这一节，我们将创建结账的控制器 CheckoutController 来收集用户的地址和付款信息，我们需要用户在结账前注册账户，因为这个控制器需要授权。

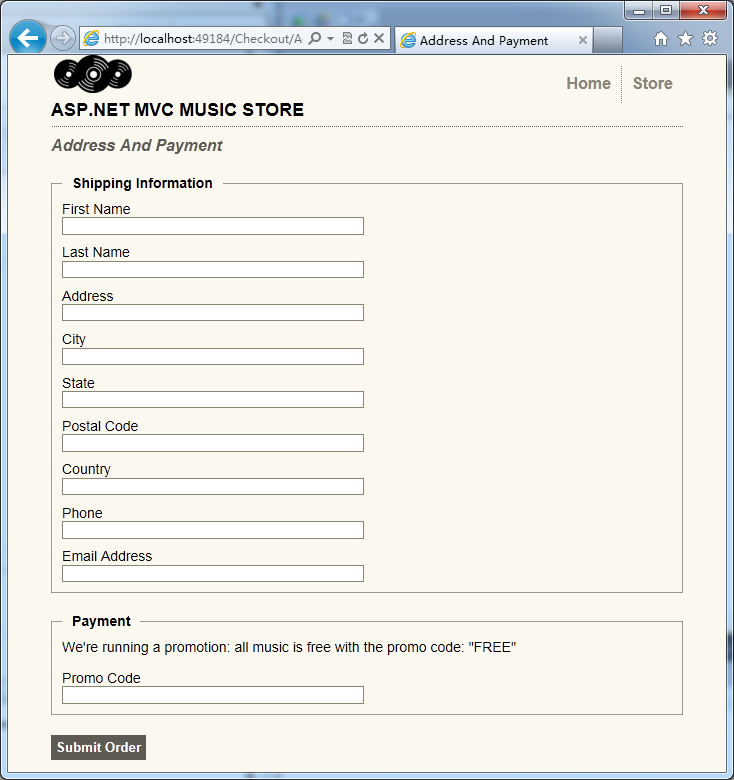
当用户点击结账 Checkout 按钮的时候，用户将会被导航到结账的处理流程中。



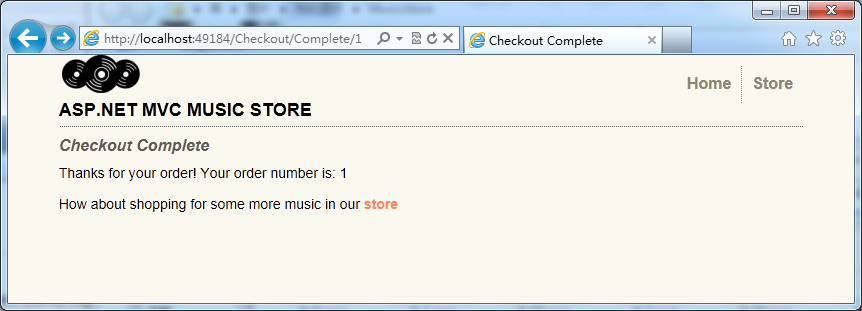
如果用户没有登录，将会被提示需要登录。



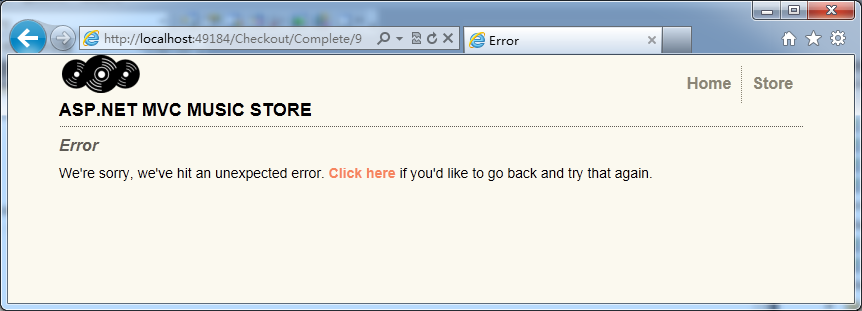
一旦用户成功登陆，用户就可以看到地址和付款的视图。



一旦用户填写了这个表单并提交，他们将会看到订单的确认页面。



视图访问不存在的订单，或者不属于你的订单，将会看到错误页面。



### 合并购物车

在匿名购物的时候，当用户点击结账 Checkout 按钮，用户会被要求注册和登陆，用户会希望继续使用原来的购物车，所以，在匿名用户登录之后，我们需要维护购物车。

实际上非常简单，因为 ShoppingCart 类已经提供了一个方法，通过当前的用户名来获取购物车中所有的项目，在用户注册登录以后，我们只需要调用这个方法。

打开在成员管理和授权中添加的 AccountController 类，增加一个 using 来引用 MvcMusicStore.Models，然后，增加 MigrateShoppingCart 方法。

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;  
}

然后，修改 LonOn 的 Post 处理方法，在用户通过验证之后，调用 MigrateShoppingCart  方法。

//  
// POST: /Account/LogOn  
[HttpPost]  
public ActionResult LogOn(LogOnModel model, string returnUrl)  
{  
 if (ModelState.IsValid)  
 {  
 if (Membership.ValidateUser(model.UserName, model.Password))  
 {  
 MigrateShoppingCart(model.UserName);  
 FormsAuthentication.SetAuthCookie(model.UserName, model.RememberMe);  
 if (Url.IsLocalUrl(returnUrl) && returnUrl.Length > 1 && returnUrl.StartsWith("/")  
 && !returnUrl.StartsWith("//") && !returnUrl.StartsWith("/\\"))  
 {  
 return Redirect(returnUrl);  
 }  
 else  
 {  
 return RedirectToAction("Index", "Home");  
 }  
 }  
 else  
 {  
 ModelState.AddModelError("", "The user name or password provided is incorrect.");  
 }  
 }  
 // If we got this far, something failed, redisplay form  
 return View(model);  
}

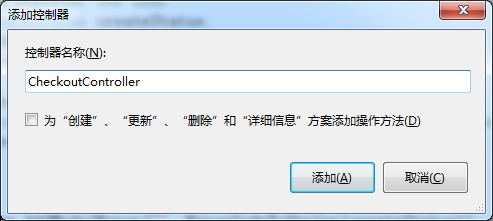
在 Register 的 Post 处理方法中，一旦用户成功创建帐户，也进行类似的修改，

//  
// POST: /Account/Register  
[HttpPost]  
public ActionResult Register(RegisterModel model)  
{  
 if (ModelState.IsValid)  
 {  
 // Attempt to register the user  
 MembershipCreateStatus createStatus;  
 Membership.CreateUser(model.UserName, model.Password, model.Email, "question", "answer", true, null, out createStatus);  
 if (createStatus == MembershipCreateStatus.Success)  
 {  
 MigrateShoppingCart(model.UserName);  
 FormsAuthentication.SetAuthCookie(model.UserName, false /\* createPersistentCookie \*/);  
 return RedirectToAction("Index", "Home");  
 }  
 else  
 {  
 ModelState.AddModelError("", ErrorCodeToString(createStatus));  
 }  
 }  
 // If we got this far, something failed, redisplay form  
 return View(model);  
}

就这样，现在匿名用户登录之后，购物车将会被自动传送过来。

### 创建结账 CheckoutController 控制器

在 Controller 文件夹上右键，添加一个新的控制器，命名为 CheckoutController ，使用空的控制器模板。



首先，在控制器上增加授权的标注 [Authorize]，来确定用户必须在登录之后才能访问。

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

注意：这一步很像我们前面在 StoreManager 控制器上的工作，但是，在那个时候，我们要求用户必须拥有 Administrator 的角色。在结账控制器中，我们不需要用户必须是 Administrator ，而是必须登录。

出于简化的考虑，在这个教程中没有处理付款的信息，作为替代，我们允许用户输入一个促销代码，这里促销代码定义在常量 PromoCode。

像在 StoreController 中一样，在控制器中，我们也需要定义 MusicStoreEntities 的字段，将它命名为 storeDB，结账的开始部分如下。

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

结账的控制器将包含下面的控制器方法：

**AddressAndPayment ( Get )** 用来显示一个用户输入信息的表单

**AddressAndPayment ( Post )** 验证用户的输入，处理订单。

Complete 用来在在用户完成订单之后显示，这个视图包含用户的订单账号和确认信息。

首先，将 Index 方法改名为 AddressAndPayment， 这个 Action 方法用来显示结账表单，所以，不需要任何的模型信息。

//  
// GET: /Checkout/AddressAndPayment  
public ActionResult AddressAndPayment()  
{  
 return View();  
}

AddressAndPayment 的 Post 处理方法使用我们在 StoreManagerController 中类似的模式：如果成功了就完成订单，如果失败了就重新显示表单。

在验证了表单之后，我们将会直接检查促销代码，假设所有的信息都是正确的，我们将会在订单中保存信息，告诉购物车对象完成订单处理，最后，重定向到完成的 Complete Action 方法。

[HttpPost]  
public ActionResult AddressAndPayment(FormCollection values)  
{  
 var order = new Order();  
 TryUpdateModel(order);  
 try  
 {  
 if (string.Equals(values["PromoCode"], PromoCode,  
 StringComparison.OrdinalIgnoreCase) == false)  
 {  
 return View(order);  
 }  
 else  
 {  
 order.Username = User.Identity.Name;  
 order.OrderDate = DateTime.Now;  
 //Save Order  
 storeDB.Orders.Add(order);  
 storeDB.SaveChanges();  
 //Process the order  
 var cart = ShoppingCart.GetCart(this.HttpContext);  
 cart.CreateOrder(order);  
 return RedirectToAction("Complete",  
 new { id = order.OrderId });  
 }  
 }  
 catch  
 {  
 //Invalid - redisplay with errors  
 return View(order);  
 }  
}

一旦完成了结账处理，用户将被重定向到 Complete 方法， 这个 Action 方法将会进行简单的检查，在显示订单号之前，检查订单是否属于当前登录的用户。

//  
// 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");  
 }  
}

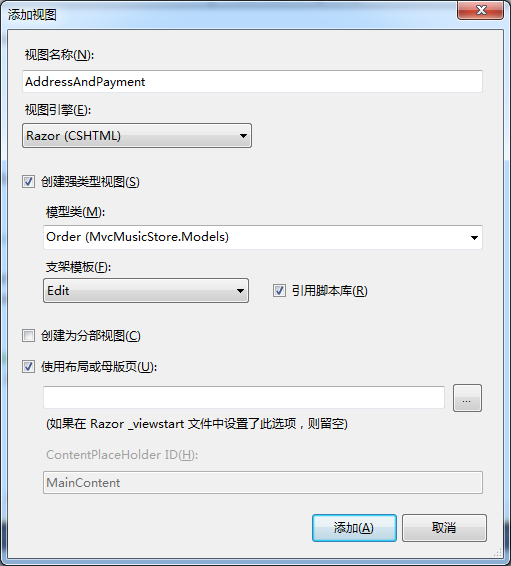
注意，错误视图创建项目的时候，保存在  /Views/Shared 文件夹中的 error.cshtml 生成。

完整的 CheckoutController 如下所示.

using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Web;  
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();  
 }  
  
 [HttpPost]  
 public ActionResult AddressAndPayment(FormCollection values)  
 {  
 var order = new Order();  
 TryUpdateModel(order);  
 try  
 {  
 if (string.Equals(values["PromoCode"], PromoCode,  
 StringComparison.OrdinalIgnoreCase) == false)  
 {  
 return View(order);  
 }  
 else  
 {  
 order.Username = User.Identity.Name;  
 order.OrderDate = DateTime.Now;  
 //Save Order  
 storeDB.Orders.Add(order);  
 storeDB.SaveChanges();  
 //Process the order  
 var cart = ShoppingCart.GetCart(this.HttpContext);  
 cart.CreateOrder(order);  
 return RedirectToAction("Complete",  
 new { id = order.OrderId });  
 }  
 }  
 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");  
 }  
 }  
 }  
}

### 增加 AddressAndPayment 视图

现在，我们创建 AddressAndPayment 视图，在 AddressAndPayment 控制器的某个 Action 中点击鼠标的右键，增加名为 AddressAndPayment 的强类型 Order 视图，使用编辑模板。



这个视图使用我们在 StoreManager 的 Edit 视图中使用的两个技术：

1.使用 Html.EditorForModel() 来显示订单模型的字段

2.使用 Order 模型的验证标签定义验证规则

我们先使用 Html.EditorForModel() 方法，然后，增加额外的输入框用来输入促销码，完成的视图如下。

@model MvcMusicStore.Models.Order  
@{  
 ViewBag.Title = "Address And Payment";  
}  
<script src="@Url.Content("~/Scripts/jquery.validate.min.js")" type="text/javascript"></script>  
<script src="@Url.Content("~/Scripts/jquery.validate.unobtrusive.min.js")" type="text/javascript"></script>  
@using (Html.BeginForm())  
{  
 <h2>  
 Address And Payment</h2>  
 <fieldset>  
 <legend>Shipping Information</legend>  
 @Html.EditorForModel()  
 </fieldset>  
 <fieldset>  
 <legend>Payment</legend>  
 <p>  
 We're running a promotion: all music is free with the promo code: "FREE"</p>  
 <div class="editor-label">  
 @Html.Label("Promo Code")  
 </div>  
 <div class="editor-field">  
 @Html.TextBox("PromoCode")  
 </div>  
 </fieldset>  
 <input type="submit" value="Submit Order" />  
}

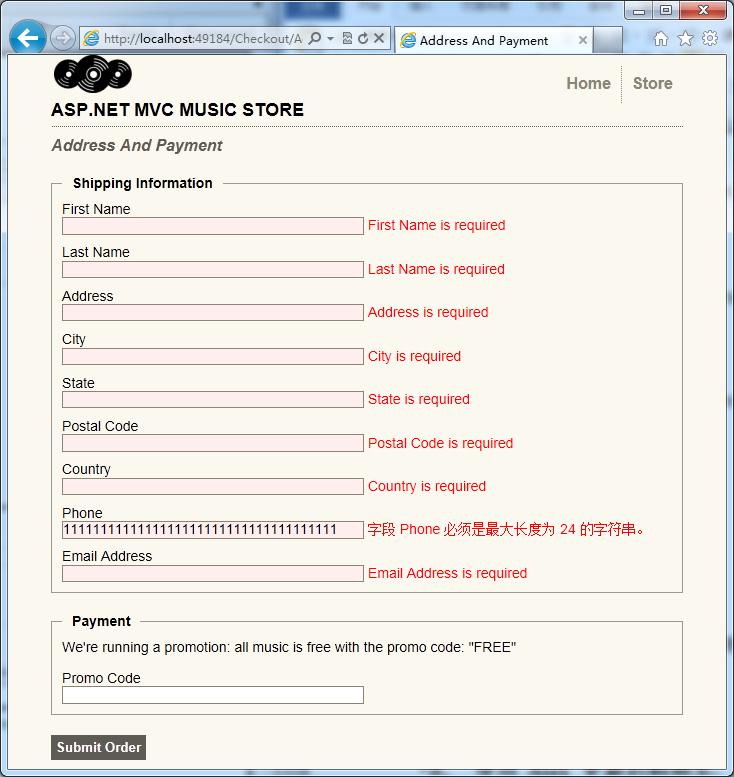
### 定义订单的验证规则

现在，我们的视图已经创建了，类似于专辑，我们定义订单的验证规则。在 Models 文件夹上点击鼠标的右键，增加名为 Order 的模型类，我们使用专辑中使用过的验证标注，我们还将使用正则表达式来验证用户的电子邮件地址。

[复制代码](javascript:void(0);)

using System.Collections.Generic;  
using System.ComponentModel;  
using System.ComponentModel.DataAnnotations;  
using System.Web.Mvc;  
  
namespace MvcMusicStore.Models  
{  
 [Bind(Exclude = "OrderId")]  
 public partial class Order  
 {  
 [ScaffoldColumn(false)]  
 public int OrderId { get; set; }  
 [ScaffoldColumn(false)]  
 public System.DateTime OrderDate { get; set; }  
 [ScaffoldColumn(false)]  
 public string Username { get; set; }  
 [Required(ErrorMessage = "First Name is required")]  
 [DisplayName("First Name")]  
 [StringLength(160)]  
 public string FirstName { get; set; }  
 [Required(ErrorMessage = "Last Name is required")]  
 [DisplayName("Last Name")]  
 [StringLength(160)]  
 public string LastName { get; set; }  
 [Required(ErrorMessage = "Address is required")]  
 [StringLength(70)]  
 public string Address { get; set; }  
 [Required(ErrorMessage = "City is required")]  
 [StringLength(40)]  
 public string City { get; set; }  
 [Required(ErrorMessage = "State is required")]  
 [StringLength(40)]  
 public string State { get; set; }  
 [Required(ErrorMessage = "Postal Code is required")]  
 [DisplayName("Postal Code")]  
 [StringLength(10)]  
 public string PostalCode { get; set; }  
 [Required(ErrorMessage = "Country is required")]  
 [StringLength(40)]  
 public string Country { get; set; }  
 [Required(ErrorMessage = "Phone is required")]  
 [StringLength(24)]  
 public string Phone { get; set; }  
 [Required(ErrorMessage = "Email Address is required")]  
 [DisplayName("Email Address")]  
 [RegularExpression(@"[A-Za-z0-9.\_%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}",  
 ErrorMessage = "Email is is not valid.")]  
 [DataType(DataType.EmailAddress)]  
 public string Email { get; set; }  
 [ScaffoldColumn(false)]  
 public decimal Total { get; set; }  
 public List<OrderDetail> OrderDetails { get; set; }  
 }  
}

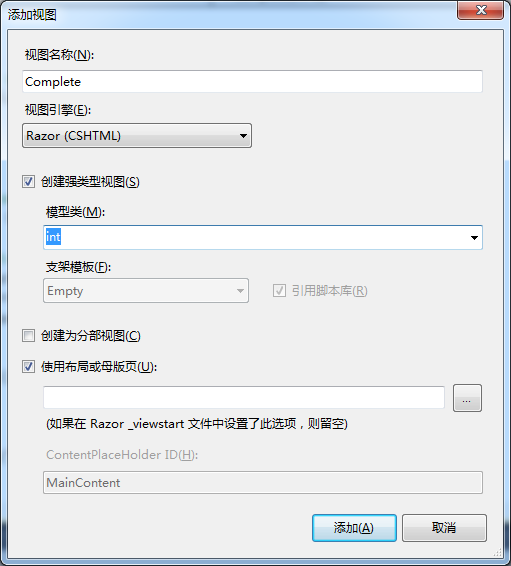
在没有提交必要的信息或者提交错误信息的时候，我们将会看到客户端的验证信息。



现在，我们已经完成了结账中最困难的部分，还有一些小的工作要完成。我们增加两个简单的视图，还需要注意在登录过程中购物车信息。

### 增加完成结账视图

完成结账的视图非常简单，仅仅需要显示订单的编号，在控制器中的 Complete 方法上点击右键，增加名为 Complete 的强类型 int 视图。



现在，修改视图显示订单的编号。

[复制代码](javascript:void(0);)

@model int  
@{  
 ViewBag.Title = "Checkout Complete";   
}  
<h2>  
 Checkout Complete</h2>  
<p>  
 Thanks for your order! Your order number is: @Model</p>  
<p>  
 How about shopping for some more music in our @Html.ActionLink("store", "Index", "Home")  
</p>

[复制代码](javascript:void(0);)

### 更新错误视图

项目的默认模板中，包含了定义在 /Shared Views 文件夹中的错误页面，可以在整个站点中使用。这个页面仅仅包含简单的信息，也没有使用我们的布局，我们更新一下。

由于这是通用的错误页面，内容非常简单，我们仅仅包含一个提示信息，和用来重做工作的导航到上一个页面的链接。

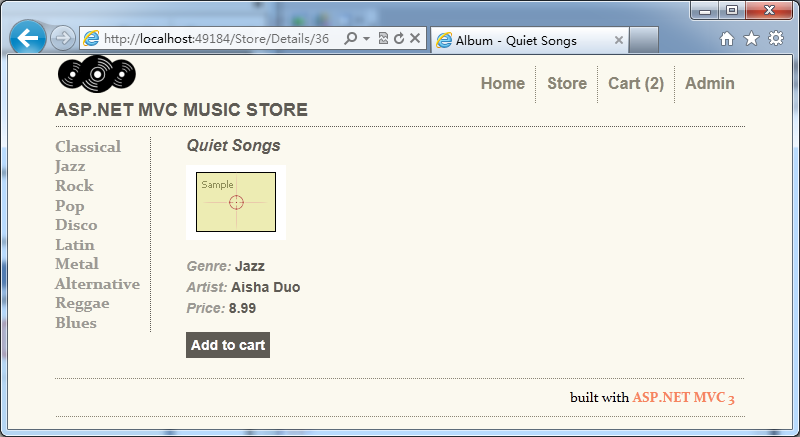
@{ ViewBag.Title = "Error"; }  
<h2>  
 Error</h2>  
<p>  
 We're sorry, we've hit an unexpected error. <a href="javascript:history.go(-1)">Click  
 here</a> if you'd like to go back and try that again.

# [ASP.NET MVC 音乐商店 - 10. 完成导航和站点的设计](http://www.cnblogs.com/haogj/archive/2011/11/21/2255680.html)

我们已经完成了网站的大部分工作，但是，还有一些添加到站点的导航功能，主页，以及商店的浏览页面。

### 创建购物车汇总部分视图

我们希望在整个站点的页面上都可以看到购物车中的数量。

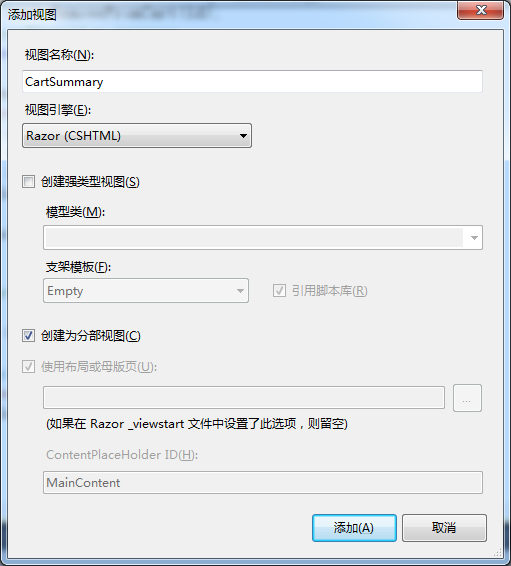


通过创建一个部分视图，然后添加到网站的布局中就可以容易地完成，

前面看到，在 ShoppingCart 控制器中包含了一个名为 CartSummary 的 Action 方法返回分部视图。

//  
// GET: /ShoppingCart/CartSummary  
[ChildActionOnly]  
public ActionResult CartSummary()  
{  
 var cart = ShoppingCart.GetCart(this.HttpContext);  
 ViewData["CartCount"] = cart.GetCount();  
 return PartialView("CartSummary");  
}

在这个 Action 方法上点击鼠标右键，或者在 Views/ShoppingCart 文件夹夹上点击鼠标右键，选择创建新视图，命名为 CartSummary ，注意选中创建分部视图的复选框。



CartSummary 分部视图非常简单，仅仅链接到 ShoppingCart 的 Index ，显示当前购物车中的数量，完整的代码如下：

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

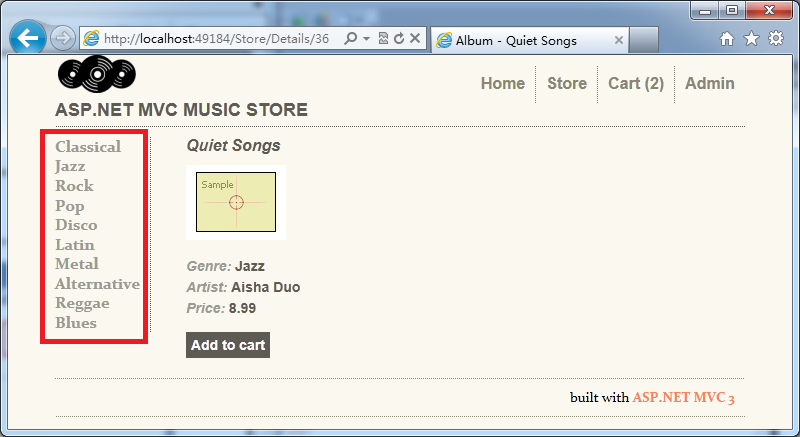
在网站的任何页面中都可以包含分部视图，使用 Html.RenderAction 方法就可以。RenderAction 需要指定 Action 的名字，这里是 CartSummary，以及控制器的名字，这里是 ShoppingCart。

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

在将这个分部视图加入到布局之前，我们还要创建一个流派的菜单，这样我们可以一次更新完站点的 Site.master。

### 创建流派菜单的分部视图

通过在页面上增加一个流派的菜单，可以是用户在站点内导航的时候更加容易。



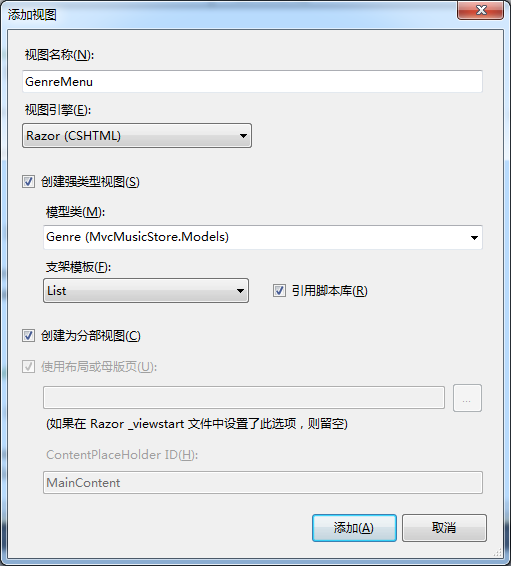
我们可以使用类似前面的步骤来创建流派菜单的分部视图，把两个分部视图一起添加到站点的布局中，首先，在 StoreController 中增加 GenreMenu 的控制器方法。

//  
// GET: /Store/GenreMenu  
[ChildActionOnly]  
public ActionResult GenreMenu()  
{  
 var genres = storeDB.Genres.ToList();  
 return PartialView(genres);  
}

这个方法返回流派的列表，在后面创建的视图中用来生成菜单。

注意：在 Action 方法上我们增加了 [ChildActionOnly] 标注，这意味着我们仅仅可以通过分部视图来访问这个 Action，这可以防止通过浏览 /Store/GenreMenu 来访问，对于分部视图来说，这不是必须的，但是一个很好的实践，因为我们希望我们的控制器方法被我们希望的方式使用，这里我们还返回了一个分部视图而不是 一个普通的视图，这用来告诉视图引擎，不需要对这个视图使用布局，它将会被包含在其他的视图中。

创建分部视图，使用强类型的 Genre 作为模型类型。使用 List 模板。



更新生成的视图，显示一个列表。

@model IEnumerable<MvcMusicStore.Models.Genre>  
<ul id="categories">  
 @foreach (var genre in Model)  
 { <li>@Html.ActionLink(genre.Name,  
 "Browse", "Store",  
 new { Genre = genre.Name }, null)  
 </li>  
 }  
</ul>

### 更新站点的布局显示我们的分部视图

现在，可以在布局中加入分部视图了，在 /Views/Shared/\_Layout.cshtml 中通过调用 Html.RenaderAction() 方法可以调用分部视图，把两个分部视图都加入到布局中，如下所示：

<!DOCTYPE html>  
<html>  
<head>  
 <title>@ViewBag.Title</title>  
 <link href="@Url.Content("~/Content/Site.css")" rel="stylesheet"  
type="text/css" />  
 <script src="@Url.Content("~/Scripts/jquery-1.4.4.min.js")"  
type="text/javascript"></script>  
</head>  
<body>  
 <div id="header">  
 <h1>  
 <a href="/">ASP.NET MVC MUSIC STORE</a></h1>  
 <ul id="navlist">  
 <li class="first"><a href="@Url.Content("~")" id="current">Home</a></li>  
 <li><a href="@Url.Content("~/Store/")">Store</a></li>  
 <li>  
 @{Html.RenderAction("CartSummary", "ShoppingCart");}  
 </li>  
 <li><a href="@Url.Content("~/StoreManager/")">Admin</a></li>  
 </ul>  
 </div>  
 @{Html.RenderAction("GenreMenu", "Store");}  
 <div id="main">  
 @RenderBody()  
 </div>  
 <div id="footer">  
 built with <a href="http://asp.net/mvc">ASP.NET MVC 3</a>  
 </div>  
</body>  
</html>

### 更新 Store 的 Browse 页面

商店的浏览页面现在看来还不太好，我们更新这个页面在一个更好地布局中显示专辑，如下更新我们的视图。

@model MvcMusicStore.Models.Genre   
@{ ViewBag.Title = "Browse Albums"; }  
<div class="genre">  
 <h3>  
 <em>@Model.Name</em> Albums</h3>  
  
 <ul id="album-list">  
 @foreach (var album in Model.Albums)  
 { <li><a href="@Url.Action("Details", new { id = album.AlbumId })">  
 <img alt="@album.Title" src="@album.AlbumArtUrl" />  
 <span>@album.Title</span> </a></li> }  
 </ul>  
</div>

这里，我们将使用 Url.Action 来代替 Html.ActionLink ，以便显示格式化信息，包括艺术家的插画。

注意：我们显示专辑的封面，这些信息保存在数据中，可以通过 StoreManager 进行编辑，也欢迎你加入你的插图。

现在，当我们浏览流派的时候，我们将会看到带有封面的专辑显示在一个网格中。



### 更新主页来显示畅销专辑

我们希望在首页上增加畅销专辑来增进销售，我们在 HomeController 中增加一下内容来实现，然后增加一些额外的图片来变得更好。

首先，在我们的专辑中增加一个导航属性，以便与 EF 知道关联的的信息。专辑中最后的一行就是新增加的。

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

注意：这里使用了泛型的集合，需要在代码的前面使用 using 来引用 System.Collections.Generic 命名空间。

首先，我们将要增加 storeDB 的字段和引用 MusicStore.Models 命名空间，类似于其他的控制器。

然后，我们在 HomeController 中增加下面的方法，来查询数据库根据 OrderDetails 找到畅销的唱片。

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();  
}

这是私有方法，因为我们不希望直接可以访问到，这里为了简单将它写在了 HomeController 中，实际开发的时候，可能需要移到后台的逻辑服务中。

这里，我们更新 Index 来访问前面定义的方法，查询销售前 5 名的专辑，然后将他们传递到视图中。

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

完整的代码如下：

using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Web;  
using System.Web.Mvc;  
  
using MvcMusicStore.Models;  
  
namespace MvcMusicStore.Controllers  
{  
 public class HomeController : Controller  
 {  
 private Models.MusicStoreEntities storeDB = new Models.MusicStoreEntities();  
 //  
 // GET: /Home/  
  
 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();  
 }  
 }  
}

最后，我们需要更新我们的 Home 控制器的 Index 视图，访问模型在后面加入专辑的列表，借助这个时机，我们还要增加一个标头和一个促销的节。

@model List<MvcMusicStore.Models.Album>   
@{  
 ViewBag.Title = "ASP.NET MVC Music Store";   
}  
<div id="promotion">  
</div>  
<h3>  
 <em>Fresh</em> off the grill</h3>  
<ul id="album-list">  
 @foreach (var album in Model)  
 { <li><a href="@Url.Action("Details", "Store",  
new { id = album.AlbumId })">  
 <img alt="@album.Title" src="@album.AlbumArtUrl" />  
 <span>@album.Title</span> </a></li>  
 }  
</ul>

现在，当运行程序的时候，我们将会看到更新后的主页，带有畅销的专辑和我们的促销信息。

