* 1. 

Hands-On Lab

ASP.NET MVC 3.0 Fundamentals

* 1. Lab version: 1.1.0
  2. Last updated: 1/11/2011

Contents

[Overview 4](#_Toc282507903)

[Exercise 1: Creating MusicStore ASP.NET MVC Web Application Project 14](#_Toc282507904)

[Task 1 – Creating the ASP.NET MVC Web Application Project 14](#_Toc282507905)

[Task 2 – Exploring the Solution Structure 16](#_Toc282507906)

[Task 3 – Adding a HomeController 18](#_Toc282507907)

[Task 4 – Running the Application 19](#_Toc282507908)

[Exercise 2: Creating a Controller 20](#_Toc282507909)

[Task 1 – Adding a New StoreController Class 21](#_Toc282507910)

[Task 2 – Modifying the StoreController’s Actions 21](#_Toc282507911)

[Task 3 – Running the Application 22](#_Toc282507912)

[Exercise 3: Passing parameters to a Controller 24](#_Toc282507913)

[Task 1 – Adding Genre Parameter to StoreController 24](#_Toc282507914)

[Task 2 – Running the Application 25](#_Toc282507915)

[Task 3 – Adding an Id Parameter Embedded in the URL 25](#_Toc282507916)

[Task 4 – Running the Application 26](#_Toc282507917)

[Exercise 4: Creating a View 27](#_Toc282507918)

[Task 1 – Adding a MasterPage 27](#_Toc282507919)

[Task 2 – Adding CSS Stylesheet 29](#_Toc282507920)

[Task 3 – Adding a View Template 31](#_Toc282507921)

[Exercise 4: Verification 34](#_Toc282507922)

[Exercise 5: Creating a View Model 36](#_Toc282507923)

[Task 1 – Creating a ViewModel Class 36](#_Toc282507924)

[Task 2 – Adding Properties to the ViewModel class 39](#_Toc282507925)

[Task 3 – Updating StoreController to use the StoreIndexViewModel 39](#_Toc282507926)

[Task 4 – Creating a View Template that Uses StoreIndexViewModel 41](#_Toc282507927)

[Task 5 – Updating the View Template 43](#_Toc282507928)

[Exercise 6: Using Parameters in View 46](#_Toc282507929)

[Task 1 – Adding Model Classes 46](#_Toc282507930)

[Task 2 – Adding a StoreBrowseViewModel 48](#_Toc282507931)

[Task 3 – Using the New ViewModel in the StoreController 50](#_Toc282507932)

[Task 4 – Adding a Browse View Template 51](#_Toc282507933)

[Task 5 – Running the Application 54](#_Toc282507934)

[Task 6 – Displaying information About a Specific Album 54](#_Toc282507935)

[Task 7 – Running the Application 56](#_Toc282507936)

[Task 8 – Adding Links Between Pages 56](#_Toc282507937)

[Task 9 – Running the Application 58](#_Toc282507938)

[Task 10 – Using Dynamic ViewModel Collection to Pass Values 58](#_Toc282507939)

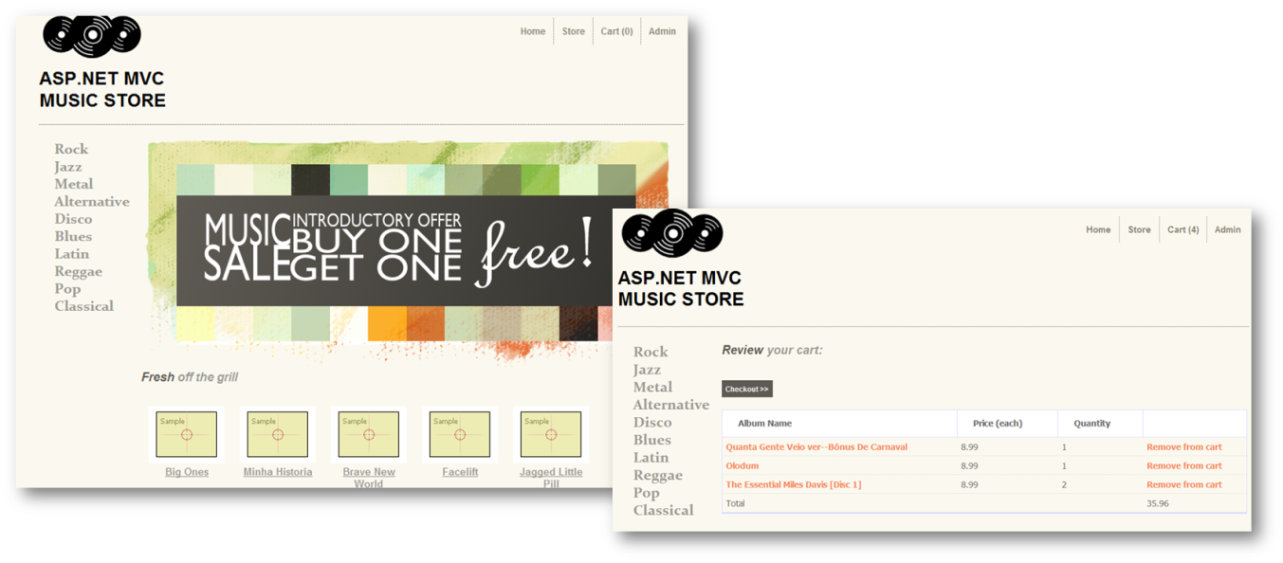
[Task 11 – Running the Application 61](#_Toc282507940)

[Summary 62](#_Toc282507941)

Overview

* 1. This Hands-On Lab is based on MVC (Model View Controller) Music Store, a tutorial application that introduces and explains step-by-step how to use ASP.NET MVC and Visual Web Developer 2010 Express (which is free). Throughout the lab you will learn the simplicity, yet power of using these technologies together. You will start with a simple application and will build it until you have a fully functional MVC Web Application.
  2. This Lab will work with ASP.NET MVC 3 and it uses new features as dynamic properties in views.
  3. If you wish to explore the ASP.NET MVC 2 version of the tutorial application, you will find it in <http://mvcmusicstore.codeplex.com/>.
  4. **Note:** This Hands-On Lab assumes that the developer has basic experience with Web Development and technologies such as HTML and JavaScript.
  5. cid:image001.png@01CB3593.8F0D4FB0 Before diving into ASP.NET MVC, if you would like more context on the ASP.NET 4 stack – watch [this short video](http://channel9.msdn.com/learn/courses/VS2010/ASPNET/ASPNET4/) where Jonathan Carter takes us through what’s new in ASP.NET 4.

# The Music Store application

* 1. The Music Store web application to be built throughout this Lab comprises three main parts: shopping, checkout, and administration. Visitors will be able to browse albums by genre, add albums to their cart, review their selection and finally proceed to checkout to login and complete the order. Additionally, store administrators will be able to manage the store’s available albums as well as their main properties.
  2. 
  3. Figure 1
  4. Music Store screens

# MVC Essentials

* 1. Music Store application will be built using **Model View Controller (MVC)**, an architectural pattern that separates an application into three main components:
  + **Models**: Model objects are the parts of the application that implement the domain logic. Often, model objects also retrieve and store model state in a database.
  + **Views:** Views are the components that display the application's user interface (UI). Typically, this UI is created from the model data. An example would be the edit view of Albums that displays text boxes and a drop-down list based on the current state of an Album object.
  + **Controllers:** Controllers are the components that handle user interaction, manipulate the model, and ultimately select a view to render the UI. In an MVC application, the view only displays information; the controller handles and responds to user input and interaction.
  1. The MVC pattern helps you to create applications that separate the different aspects of the application (input logic, business logic, and UI logic), while providing a loose coupling between these elements. This separation helps you manage complexity when you build an application, because it enables you to focus on one aspect of the implementation at a time. In addition to managing complexity, the MVC pattern makes it easier to test applications than it is to test a traditional ASP.NET Web application, encouraging the use of test-driven development (TDD) for creating an applications.
  2. Then, the **ASP.NET MVC** framework provides an alternative to the ASP.NET Web Forms pattern for creating MVC-based Web applications. The **ASP.NET MVC** framework is a lightweight, highly testable presentation framework that (as with Web-forms-based applications) is integrated with existing ASP.NET features, such as master pages and membership-based authentication so you get all the power of the core .NET framework. This is useful if you are already familiar with ASP.NET Web Forms because all of the libraries that you already use are available to you in ASP.NET MVC as well.
  3. In addition, the loose coupling between the three main components of an MVC application also promotes parallel development. For instance, one developer can work on the view, a second developer can work on the controller logic, and a third developer can focus on the business logic in the model.

# Objectives

* 1. In this Hands-On Lab, you will learn how to:
  + Create an ASP.NET MVC application from scratch based on the Music Store Application tutorial
  + Add Controllers to handle URLs to the Home page of the site and for browsing its main functionality
  + Add a View to customize the content displayed along with its style
  + Add Model classes to contain and manage data and domain logic
  + Use View Model pattern to pass information from controller actions to the view templates

# System Requirements

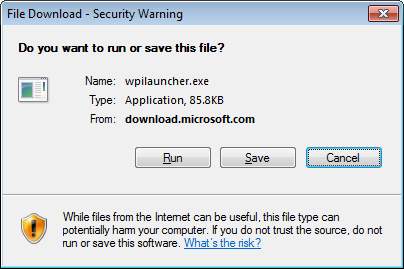
* 1. You must have the following items to complete this lab:
  + ASP.NET and ASP.NET MVC 3
  + Visual Studio 2010 Express

# Setup

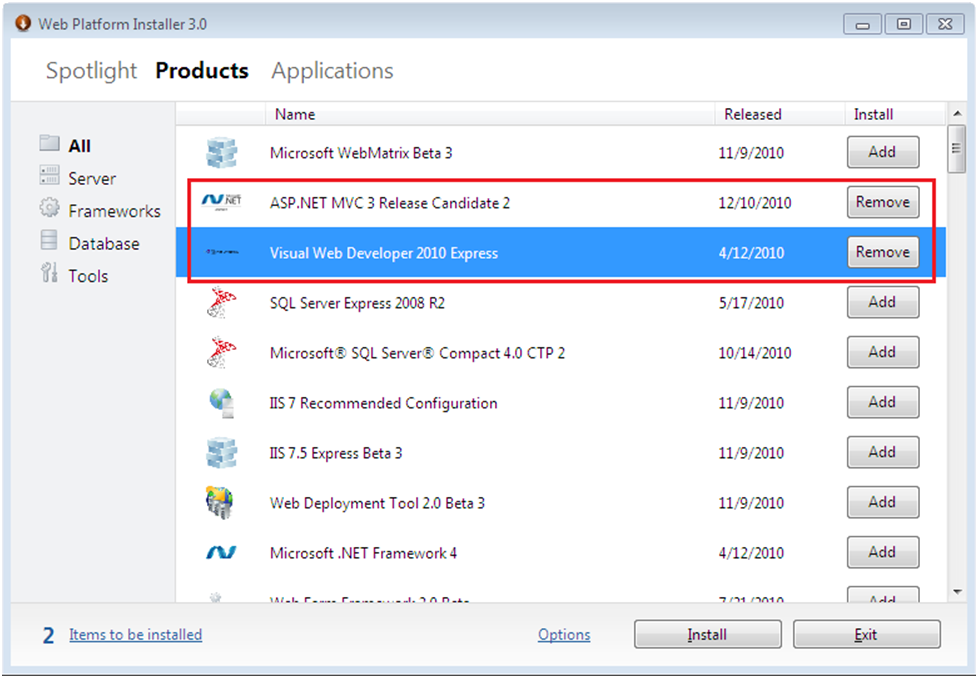
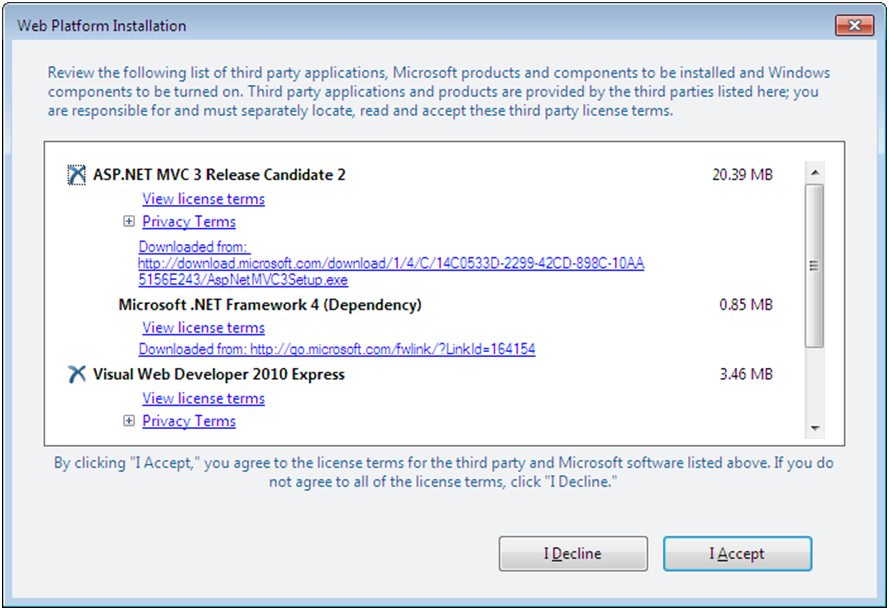
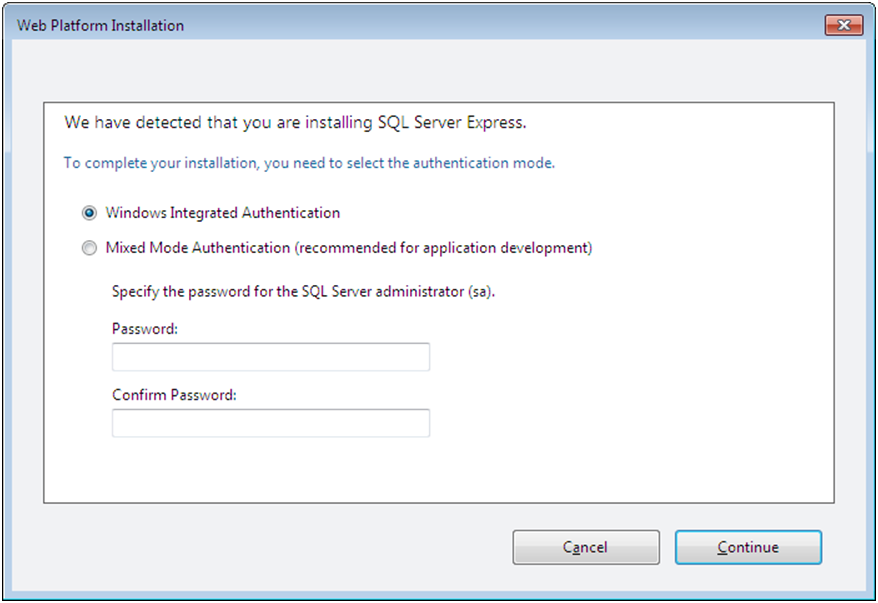
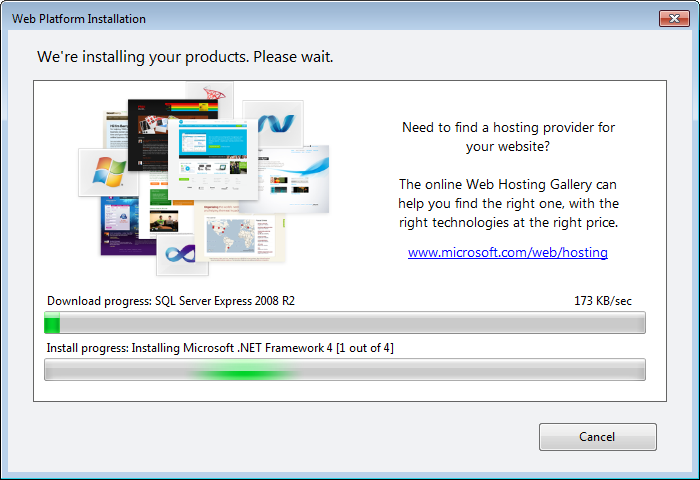
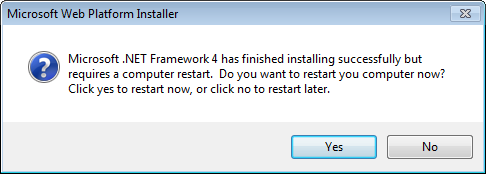
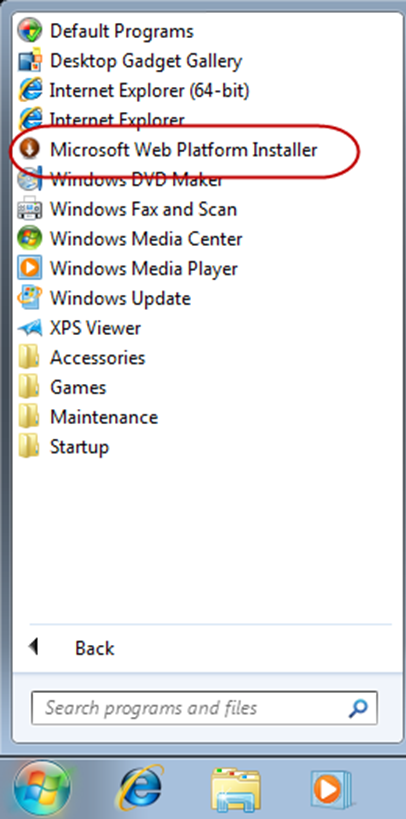
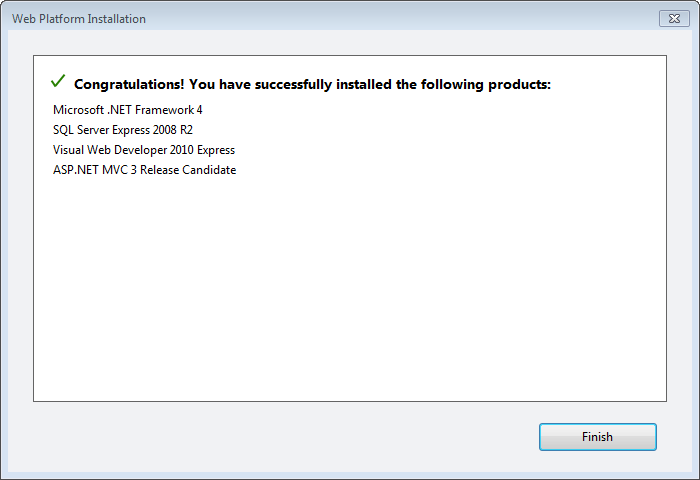
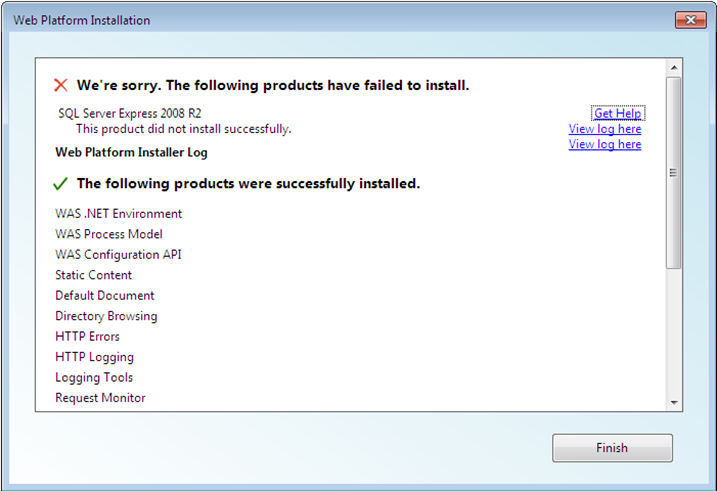
#### Installing Code Snippets

* 1. For convenience, much of the code you will be managing along this lab is available as Visual Studio code snippets. To install the code snippets run **.\Source\Assets\CodeSnippets.vsi** file.

#### Installing Web Platform Installer

* 1. This section assumes that you don’t have some or all the system requirements installed. In case you do, you can simply skip this section.
  2. Microsoft Web Platform Installer (WebPI) is a tool that manages the installation of the prerequisites for this Lab.
  3. **Note:** As well as the Microsoft Web Platform, WebPI can also install many of the open source applications that are available like Umbraco, Kentico, DotNetNuke and many more. These are very useful for providing a foundation from which to customize an application to your requirements, dramatically cutting down your development time.
  4. Please follow these steps to downloads and install Microsoft Visual Studio 2010 Express Edition and SQL Server 2008 Express Edition:
  5. Install the Web Platform Installer 3.0. To do this, Navigate to <http://go.microsoft.com/fwlink/?LinkID=194638> using a web browser. Click **Run** in the File Download Security Warning.
     1. 
     2. Figure 2
     3. File download pop up
  6. Allow the program to make changes to your computer by clicking **Yes** in the User Account Control pop up.

#### Running Web Platform Installer to install Lab prerequisites

* 1. The Web Platform Installer launches and shows a list of downloads.
  2. Choose the **Products** category at the top side of the window, then click the **All** folder at the left, and then clik **Add** for the following items:
     + **ASP.NET MVC 3**
     + **Visual Web Developer 2010 Express**
     1. 
     2. Figure 3
     3. Web Platform Installer window
  3. Click on **Install**. The **Web Platform Installer** displays the list of software to be installed. Accept by clicking **I Accept**.
     1. 
     2. Figure 4
     3. Web Platform Installation
  4. The SQL Server Express Edition could be configured to run with **Windows Integrated Authentication** or **Mixed Mode Authentication**. To simplify this Lab, select **Windows Integrated Authentication** and click on **Continue**. SQL Server will then use your Windows credentials to connect to the databases.
     1. 
     2. Figure 5
     3. Web Platform Installation – SQL Server Express Authentication mode
  5. The appropriate components will be downloaded and installed.
     1. 
     2. Figure 6
     3. Web Platform Installation – Download progress
  6. If your computer doesn’t have the **Microsoft.NET Framework 4.0**, a pop up will ask you to restart the computer. Click **Yes**.
     1. 
     2. Figure 7
     3. Web Platform Installation – Microsoft.NET Framework 4 installation
  7. After rebooting your PC, restart the Web Platform Installer by selecting it from the **Start** menu | **All Programs** | **Microsoft** **Web Platform Installer**.
     1. 
     2. Figure 8
     3. Starting Web Platform Installer
  8. The **Web Platform Installer** will resume downloading and installing the products. When this process is finished, the Installer will show the list of all the software installed. Click **Finish**.
     1. 
     2. Figure 9
     3. Web Platform Installer
     4. **Note:** If you get an error with SQL Server Express 2008 R2 installation like the following, please dismiss it. You will be able to work-around it in **ASP.NET MVC 2.0 Models and Data Access** Hands-on lab.
     5. 
     6. Figure 10
     7. SQL Server Express 2008 R2 installation error
  9. Finally the Web Platform Installer shows the installed products. Click **Exit** to finish the setup process.

# Exercises

* 1. This Hands-On Lab is comprised by the following exercises:
  2. Exercise 1: Creating MusicStore ASP.NET MVC Web Application Project
  3. Exercise 2: Creating a Controller
  4. Exercise 3: Passing parameters to a Controller
  5. Exercise 4: Adding Ajax for Searching Activities
  6. Exercise 5: Creating a View Model
  7. Exercise 6: Using parameters in View
  8. Estimated time to complete this lab: **45 minutes**.
  9. **Note:** Each exercise is accompanied by an **End** folder containing the resulting solution you should obtain after completing the exercises. You can use this solution as a guide if you need additional help working through the exercises.

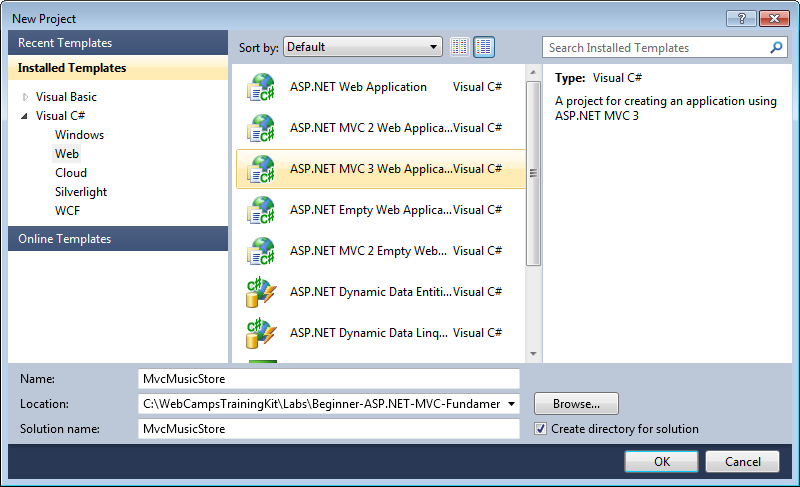
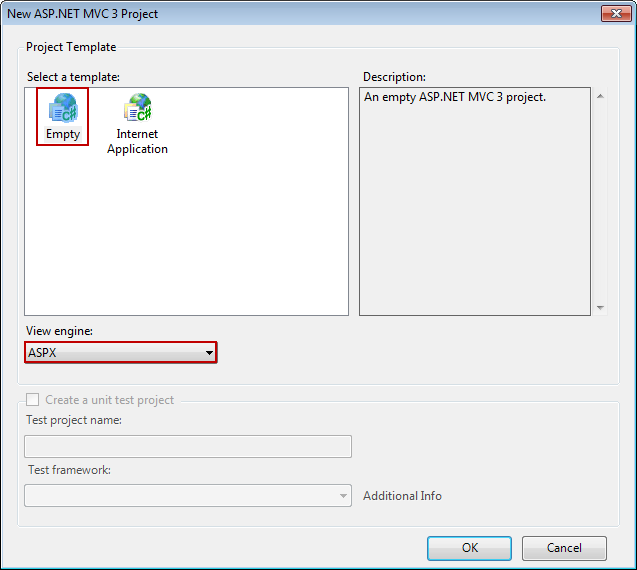
# Next Step

Exercise 1: Creating MusicStore ASP.NET MVC Web Application Project

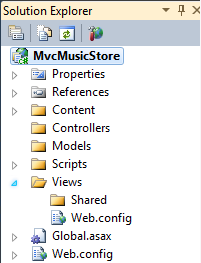
Exercise 1: Creating MusicStore ASP.NET MVC Web Application Project

* 1. In this exercise, you will learn how to create an ASP.NET MVC application in Visual Studio 2010 as well as its main folder organization. Additionally, you’ll learn how to add a new Controller and have it display a simple string at the Application screen.

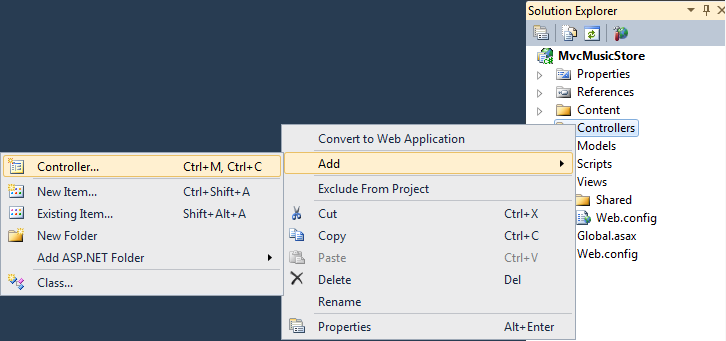
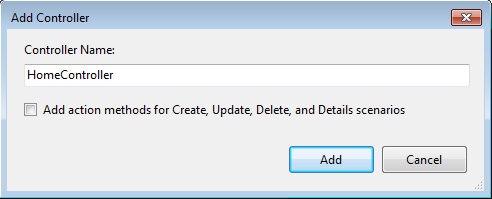
Task 1 – Creating the ASP.NET MVC Web Application Project

* 1. In this task, you will create an empty ASP.NET MVC application project using the MVC Visual Studio template.
  2. Start Microsoft Visual Web Developer 2010 Express from **Start** | **All Programs** | **Microsoft Visual Studio 2010 Express** | **Microsoft Visual Web Developer 2010 Express**.
  3. On the **File** menu, click **New Project**.
  4. In the **New Project** dialog box select the **ASP.NET MVC 3 Web Application** project type, located under **Visual C#,** **Web** template list.
  5. Change the **Name** to **MvcMusicStore**.
  6. Set the location of the solution inside a new **Begin** folder in this Exercise’s installation folder, for example **C:\WebCampsTrainingKit\Labs\Beginner-ASP.NET-MVC-Fundamentals MVC3\Source\Ex01-CreatingMusicStoreProject\Begin**. Click **OK**.
     1. 
     2. Figure 11
     3. Create New Project Dialog Box - C#
  7. In the **New ASP.NET MVC 3 Project** dialog box select the **Empty** template and make sure that the **View engine** selected is **ASPX**. Click **OK**.
     1. 
     2. Figure 12
     3. New ASP.NET MVC 3 Project Dialog Box - C#

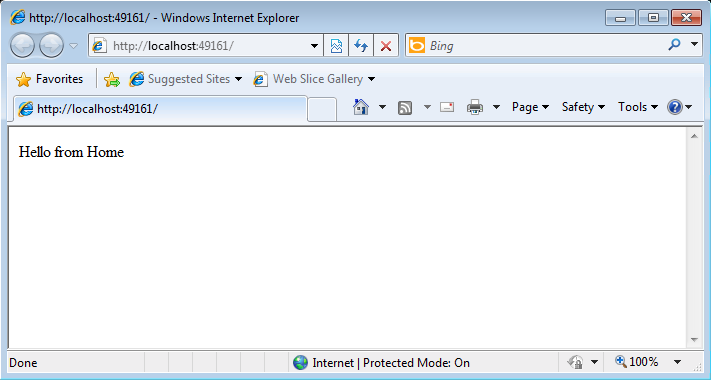
Task 2 – Exploring the Solution Structure

* 1. The ASP.NET MVC framework includes a Visual Studio project template that helps you create Web applications that are structured to support the MVC pattern. This template creates a new MVC Web application that is configured to have the required folders, item templates, and configuration-file entries.
  2. In this task, you will examine the solution structure to understand the involved elements and its relationships. The following folders are included even in an Empty ASP.NET MVC application because the ASP.NET MVC framework by default uses a “convention over configuration” approach and makes some default assumptions based on folder naming conventions.
  3. Once the project is created, review the folder structure that has been created in the Solution Explorer on the right side:
     1. 
     2. Figure 13
     3. ASP.NET MVC Folder structure in Solution Explorer
     4. **Content**. This folder is the recommended location to add content files such as cascading style sheet files, images, and so on. In general, the Content folder is used for static files.
     5. **Controllers.** This folder will contain the controller classes. In an MVC based application, controllers are responsible for handling end user interaction, manipulating the model, and ultimately choosing a view to render to display UI.
        1. **Note:** The MVC framework requires the names of all controllers to end with "Controller"—for example, HomeController, LoginController, or ProductController.
     6. **Models**. This folder is provided for classes that represent the application model for the MVC Web application. This usually includes code that defines objects and the logic for interacting with the data store. Typically, the actual model objects will be in separate class libraries. However, when you create a new application, you might include classes and then move them into separate class libraries at a later point in the development cycle.
     7. **Scripts**. This is the recommended folder to store JavaScript files in your application.
     8. **Views**. This folder is the recommended location for views, the components responsible for displaying the application's user interface. Views use .aspx, .ascx, .cshtml and .master files, in addition to any other files that are related to rendering views. Views folder contains a folder for each controller; the folder is named with the controller-name prefix. For example, if you have a controller named **HomeController**, the Views folder will contain a folder named Home. By default, when the ASP.NET MVC framework loads a view, it looks for an .aspx file with the requested view name in the Views\controllerName folder (**Views\[ControllerName]\[Action].aspx**) or (**Views\[ControllerName]\[Action].cshtml**) for Razor Views.
     9. **Views\Shared.** By default, there is also a folder named Shared in the Views folder, which does not correspond to any controller. The Shared folder is used for views that are shared across multiple controllers. For example, you can put the Web application's master page in the Shared folder.
        1. **Note:** In addition to the folders listed previously, an MVC Web application uses the **Global.asax** file to set global URL routing defaults, and it uses the **Web.config** file to configure the application.

Task 3 – Adding a HomeController

* 1. In ASP.NET applications that do not use the MVC framework, user interaction is organized around pages, and around raising and handling events from those pages. In contrast, user interaction with ASP.NET MVC applications is organized around controllers and their action methods.
  2. On the other hand, ASP.NET MVC framework maps URLs to classes that are referred to as controllers. Controllers process incoming requests, handle user input and interactions, execute appropriate application logic and determine the response to send back to the client (display HTML, download a file, redirect to a different URL, etc.). In the case of displaying HTML, a controller class typically calls a separate view component to generate the HTML markup for the request. In an MVC application, the view only displays information; the controller handles and responds to user input and interaction.
  3. In this task, you will add a Controller class that will handle URLs to the Home page of the Music Store site.
  4. Right-click the **Controllers** folder within the Solution Explorer, select **Add** and then the **Controller** command:
     1. 
     2. Figure 14
     3. Add a Controller Command
  5. The **Add Controller** dialog appears. Name the controller **HomeController** and press **Add**.
     1. 
     2. Figure 15
     3. Add Controller Dialog
  6. The file **HomeController.cs** is created in the **Controllers** folder. In order to have the **HomeController** returna string on its Index action, replace the **Index** method with the following code:
     1. (Code Snippet – ASP.NET MVC 3.0 Fundamentals – Ex1 HomeController Index – CSharp)
     2. C#
     3. **public string Index()**
     4. **{**
     5. **return "Hello from Home";**
     6. **}**

Task 4 – Running the Application

* 1. In this task, you will try out the Application in a web browser.
  2. Press **F5** to run the Application. The project gets compiled and the ASP.NET Web Server that is built-into Visual Web Developer starts. Visual Web Developer will automatically open a web browser pointing to the web-server URL. This will allow you to try out the web application.
     1. 
     2. Figure 16
     3. Application running in a web browser
     4. **Note**: ASP.NET Development Server will run the website on a random free port number. In the figure above, the site is running at http://localhost:49161/, so it’s using port 49161. Your port number may vary.
  3. Close the browser.

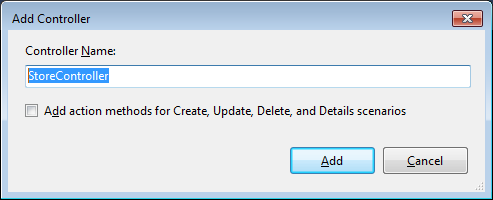
# Next Step

* 1. Exercise 2: Creating a Controller

Exercise 2: Creating a Controller

* 1. In this exercise, you will learn how to create a Controller to implement simple functionality of the Music Store. That controller will define action methods to handle each specific request:
  + A listing page of the music genres in the Music Store
  + A browse page that lists all of the music albums for a particular genre
  + A details page that shows information about a specific music album
  1. For the scope of this exercise, you will have those actions to simply return a string by now.

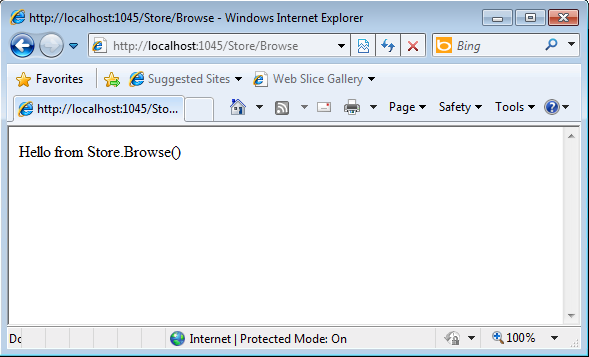
Task 1 – Adding a New StoreController Class

* 1. In this task, you will add a new Controller.
  2. If not already open, start Microsoft Visual Web Developer 2010 Express from **Start** | **All Programs** | **Microsoft Visual Studio 2010 Express** | **Microsoft Visual Web Developer 2010 Express**.
  3. In the **File** menu, choose **Open Project**. In the Open Project dialog, browse to Source\Ex02-CreatingAController\Begin, select **MvcMusicStore.sln** and click **Open**. Alternatively, you may continue with the solution that you obtained after completing the previous exercise.
  4. Add the new controller. To do this, right-click the **Controllers** folder within the Solution Explorer, select **Add** and then the **Controller** command. Change the **Controller** **Name** to **StoreController**, and click **Add**.
     1. 
     2. Figure 17
     3. Add Controller Dialog

Task 2 – Modifying the StoreController’s Actions

* 1. In this task, you will modify Controller methods that are called **actions**. These actions are responsible for responding to URL requests and determine the content that should be sent back to the browser or user that invoked the URL.
  2. The **StoreController** class already has an **Index** method. You will use it later in this Lab to implement the page that lists all genres of the music store. For now, just replace the **Index** method with the following code that returns a string “Hello from Store.Index()”:
     1. (Code Snippet – ASP.NET MVC 3.0 Fundamentals – Ex2 StoreController Index – CSharp)
     2. C#
     3. **public string Index()**
     4. **{**
     5. **return "Hello from Store.Index()";**
     6. **}**
  3. Add the **Browse** and **Details** methods. To do this, add the following code to the **StoreController**:
     1. (Code Snippet – ASP.NET MVC 3.0 Fundamentals – Ex2 StoreController BrowseAndDetails – CSharp)
     2. C#
     3. **//**
     4. **// GET: /Store/Browse**
     5. **public string Browse()**
     6. **{**
     7. **return "Hello from Store.Browse()";**
     8. **}**
     9. **//**
     10. **// GET: /Store/Details**
     11. **public string Details()**
     12. **{**
     13. **return "Hello from Store.Details()";**
     14. **}**

Task 3 – Running the Application

* 1. In this task, you will try out the Application in a web browser.
  2. Press **F5** to run the Application.
  3. The project starts in the Home page. Change the URL to verify each action’s implementation.
     1. **/Store**. You will see **“Hello from Store.Index()”**.
     2. **/Store/Browse**. You will see **"Hello from Store.Browse()"**.
     3. **/Store/Details**. You will see **“Hello from Store.Details()”**.
        1. 
        2. Figure 18
        3. Browsing /Store/Browse
  4. Close the browser.

# Next Step

* 1. Exercise 3: Passing parameters to a Controller

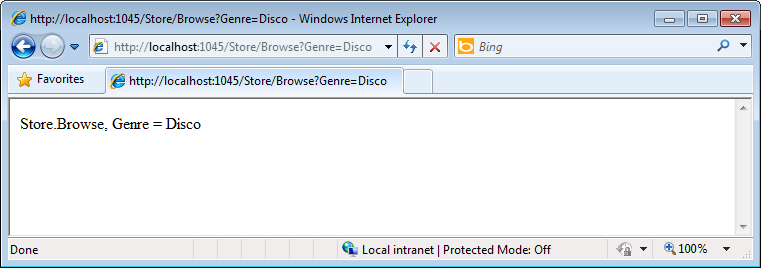
Exercise 3: Passing parameters to a Controller

* 1. Until now, you have been returning constant strings from the Controllers. In this exercise you will learn how to pass parameters to a Controller using the URL and querystring and then making the method actions respond with text to the browser.

Task 1 – Adding Genre Parameter to StoreController

* 1. In this task, you will use the **querystring** to send parameters to the **Browse** action method in the **StoreController**.
  2. If not already open, start Microsoft Visual Web Developer 2010 Express from **Start** | **All Programs** | **Microsoft Visual Studio 2010 Express** | **Microsoft Visual Web Developer 2010 Express**.
  3. In the **File** menu, choose **Open Project**. In the Open Project dialog, browse to Source\Ex03-PassingParametersToAController\Begin, select **MvcMusicStore.sln** and click **Open**. Alternatively, you may continue with the solution that you obtained after completing the previous exercise.
  4. Open **StoreController** class. To do this, in the **Solution Explorer**, expand the **Controllers** folder and double-click **StoreController.cs**.
  5. Change the **Browse** method, adding a string parameter to request for a specific genre. ASP.NET MVC will automatically pass any querystring or form post parameters named **genre** to this action method when invoked. To do this, replace the **Browse** method with the following code:
     1. (Code Snippet – ASP.NET MVC 3.0 Fundamentals – Ex3 StoreController BrowseMethod – CSharp)
     2. C#
     3. **//**
     4. **// GET: /Store/Browse?genre=Disco**
     5. **public string Browse(string genre)**
     6. **{**
     7. **string message = HttpUtility.HtmlEncode("Store.Browse, Genre = " + genre);**
     8. **return message;**
     9. **}**
     10. **Note:** You are using the **HttpUtility.HtmlEncode** utility method to prevents users from injecting Javascript into the View with a link like **/Store/Browse?Genre=<script>window.location=’http://hackersite.com’</script>**.
     11. For further explanation, please visit [this msdn article](http://msdn.microsoft.com/en-us/library/a2a4yykt(v=VS.80).aspx).

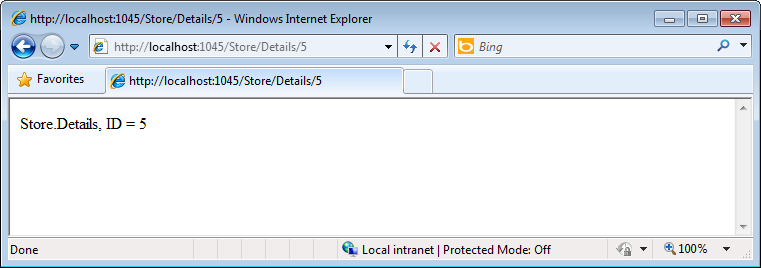
Task 2 – Running the Application

* 1. In this task, you will try out the Application in a web browser and use the **genre** parameter.
  2. Press **F5** to run the Application.
  3. The project starts in the Home page. Change the URL to **/Store/Browse?Genre=Disco** to verify that the action receives the genre parameter.
     1. 
     2. Figure 19
     3. Browsing /Store/Browse?Genre=Disco
  4. Close the browser.

Task 3 – Adding an Id Parameter Embedded in the URL

* 1. In this task, you will use the **URL** to pass an **Id** parameter to the **Details** action method of the **StoreController**. ASP.NET MVC’s default routing convention is to treat the segment of a URL after the action method name as a parameter named **Id**. If your action method has a parameter named Id then ASP.NET MVC will automatically pass the URL segment to you as a parameter. In the URL **Store/Detail/5**, **Id** will be interpreted as **5**.
  2. Change the **Details** method of the **StoreController**, adding an **int** parameter called **id**. To do this, replace the **Details** method with the following code:
     1. (Code Snippet – ASP.NET MVC 3.0 Fundamentals – Ex3 StoreController DetailsMethod – CSharp)
     2. C#
     3. **//**
     4. **// GET: /Store/Details/5**
     5. **public string Details(int id)**
     6. **{**
     7. **string message = "Store.Details, ID = " + id;**
     8. **return message;**
     9. **}**

Task 4 – Running the Application

* 1. In this task, you will try out the Application in a web browser and use the **Id** parameter.
  2. Press **F5** to run the Application.
  3. The project starts in the Home page. Change the URL to **/Store/Details/5** to verify that the action receives the id parameter.
     1. 
     2. Figure 20
     3. Browsing /Store/Details/5

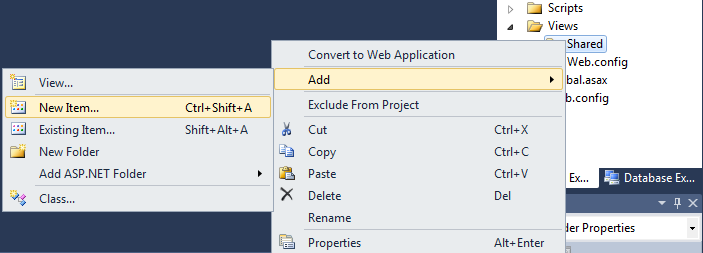
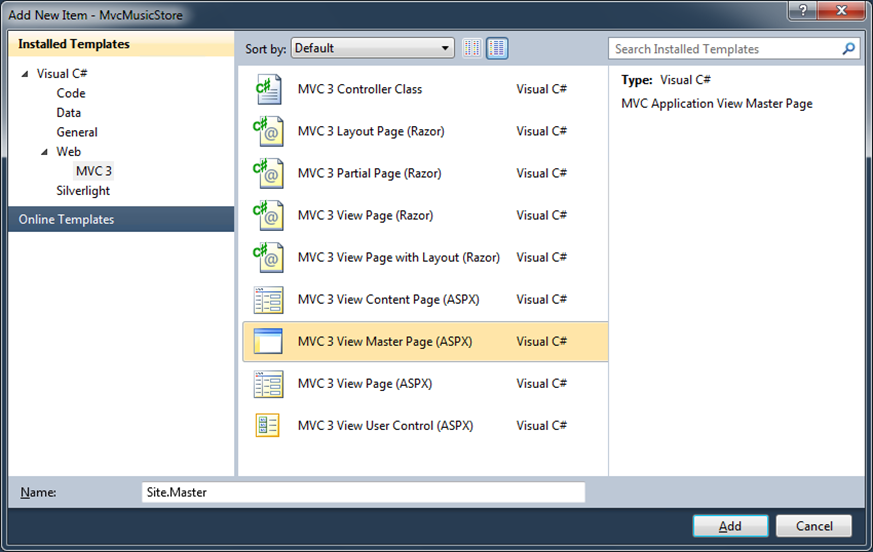
# Next Step

Exercise 4: Creating a View

Exercise 4: Creating a View

* 1. So far you have been returning strings from controller actions. Although that’s a useful way of understanding how controllers work, it’s not how you’d want to build a real web application. Views are components that provide a better approach for generating HTML back to the browser with the use of template files.
  2. In this exercise you will learn how to add a MasterPage to setup a template for common HTML content, a StyleSheet to enhance the look and feel of the site and finally a View template to enable HomeController to return HTML.

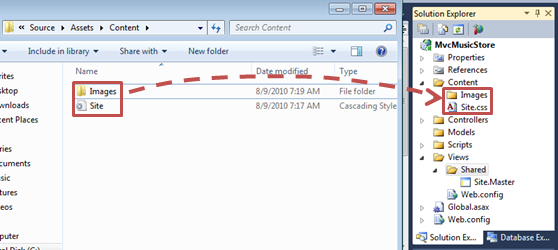
Task 1 – Adding a MasterPage

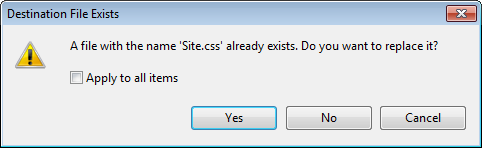
* 1. ASP.NET MasterPages are layout files that allow you to setup a template for common HTML to use across the entire website. In this task you will add a MasterPage with a common header with links to the Home page and Store area.
  2. If not already open, start Microsoft Visual Web Developer 2010 Express from **Start** | **All Programs** | **Microsoft Visual Studio 2010 Express** | **Microsoft Visual Web Developer 2010 Express**.
  3. In the **File** menu, choose **Open Project**. In the Open Project dialog, browse to Source\Ex04-CreatingAView\Begin, select **MvcMusicStore.sln** and click **Open**. Alternatively, you may continue with the solution that you obtained after completing the previous exercise.
  4. Add a MasterPage. Because it is a shared resource you will create it under the **/Views/Shared** folder. To do this, expand the **Views** folder and right-click the **Shared** folder. Select **Add** and then the **New Item** command.
     1. 
     2. Figure 21
     3. Adding a new item
  5. In the **Add New Item** dialog box select the **MVC 3 View Master Page (ASPX)** template, located under **Visual C#,** **Web** template list. Change the name to **Site.Master** and click **Add**.
     1. ****
     2. Figure 22
     3. Add New Item Dialog
  6. Site.Master file is added. Thistemplate contains the HTML container layout for all pages on the site. It includes the **<html>** element for the HTML response, as well as the **<head>** and **<body>** elements. **<asp:ContentPlaceholder>** tags within the HTML content identify regions that view templates will be able to fill in with dynamic content.
     1. HTML
     2. <%@ Master Language="C#" Inherits="System.Web.Mvc.ViewMasterPage" %>
     3. <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
     4. <html xmlns="http://www.w3.org/1999/xhtml" >
     5. <head runat="server">
     6. <title><asp:ContentPlaceHolder ID="TitleContent" runat="server" /></title>
     7. </head>
     8. <body>
     9. <div>
     10. <asp:ContentPlaceHolder ID="MainContent" runat="server">
     12. </asp:ContentPlaceHolder>
     13. </div>
     14. </body>

</html>

* 1. Add a common header with links to the Home page and Store area on all pages in the site. In order to do that, add the following code inside the **<div>** statement.
     1. HTML
     2. <%@ Master Language="C#" Inherits="System.Web.Mvc.ViewMasterPage" %>
     3. <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
     4. <html xmlns="http://www.w3.org/1999/xhtml" >
     5. <head runat="server">
     6. <title><asp:ContentPlaceHolder ID="TitleContent" runat="server" /></title>
     7. </head>
     8. <body>
     9. <div>
     10. **<div id="header">**
     11. **<h1>ASP.NET MVC MUSIC STORE</h1>**
     12. **<ul id="navlist">**
     13. **<li class="first"><a href="/" id="current">Home</a></li>**
     14. **<li><a href="/Store/">Store</a></li>**
     15. **</ul>**
     16. **</div>**
     17. <asp:ContentPlaceHolder ID="MainContent" runat="server">
     19. </asp:ContentPlaceHolder>
     20. </div>
     21. </body>
     22. </html>
  2. **Note:** Did you know? Visual Studio 2010 has snippets that make it easy to add commonly used code in HTML, code files and more! Try it out by typing **<div + tab (twice)** to insert a complete **div** tag.

Task 2 – Adding CSS Stylesheet

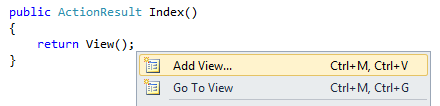
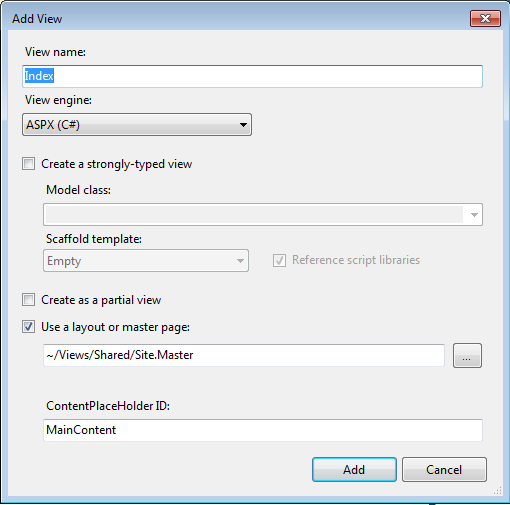
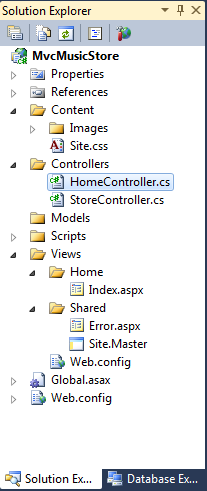
* 1. The empty project template includes a very streamlined CSS file which just includes styles used to display validation messages. You will use additional CSS and images (potentially provided by a designer) in order to enhance the look and feel of the site.
  2. In this task, you will add a CSS stylesheet to **Site.Master** to define the styles of the site.
  3. The CSS file and images are included in the Source\Assets\Content of this Lab. In order to add them to the application, drag their content from a **Windows Explorer** window into the **Solution Explorer** in Visual Web Developer Express, as shown below:
     1. 
     2. Figure 23
     3. Dragging style contents
  4. A warning dialog will appear, asking for confirmation to replace Site.css file. Click **Yes**.



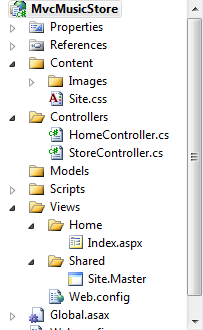
* + 1. Figure 24
    2. Warning Dialog
  1. Add a **<link>** element into the **<head>** tag of **Site.Master** file, with a reference to the css file you just added:
     1. HTML
     2. <head runat="server">
     3. **<link href="/Content/Site.css" rel="Stylesheet" type="text/css" />**
     4. <title><asp:ContentPlaceHolder ID="TitleContent" runat="server" /></title>

</head>

Task 3 – Adding a View Template

* 1. In this task, you will add a View template to generate the HTML response that will use the Master Page and CSS added in this exercise.
  2. To use a View template when browsing the site’s home page, first you need to indicate that instead of returning a string, the **HomeController Index** method should return a **View**. Open **HomeController** classand change its **Index** method to return an **ActionResult,** and have it return **View()**.
     1. (Code Snippet – ASP.NET MVC 3.0 Fundamentals – Ex4 HomeController Index - CSharp)
     2. C#
     3. public class HomeController : Controller
     4. {
     5. //
     6. // GET: /Home/
     7. **public ActionResult Index()**
     8. **{**
     9. **return View();**
     10. **}**
     11. }
  3. Now, you need to add an appropriate View template. To do this, **right-click** inside the **Index** action method and select **Add View**. This will bring up the Add View dialog.
     1. 
     2. Figure 25
     3. Adding a View from within the Index method
  4. The **Add View** Dialog appears. It allows generating View template files. By default this dialog pre-populates the name of the View template so that it matches the action method that will use it. Because you used the **Add View** context menu within the **Index** action method of the HomeController, the **Add View** dialog has **Index** as the default view name. Because there is a **Site.Master** MasterPage template within the project, the dialog also pre-fills that name as the master page template the View should be based on. Click **Add**.
     1. 
     2. Figure 26
     3. Add View Dialog
  5. Visual Studio generates an **Index.aspx** view template inside the Views\Home folder and then opens it.
     1. 
     2. Figure 27
     3. Add View Dialog
     4. **Note:** name and location of the **Index.aspx** file is relevant and follows the default ASP.NET MVC naming conventions.
     5. The folder (\Views\**Home**) matches the controller name (**Home**Controller). The View template name (**Index**), matches the controller action method which will be displaying the View.
     6. This way, ASP.NET MVC avoids having to explicitly specify the name or location of a View template when using this naming convention to return a View.
  6. The generated View template is based on the **Site.master** template earlier defined, and contains two **<asp:content>** sections that enables you to override to fill with the page content. Update the Title to **Home**, and change the main content to **This is the Home Page**, as shown in the code below:
     1. HTML
     2. <asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
     3. **Home**
     4. </asp:Content>
     5. <asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">

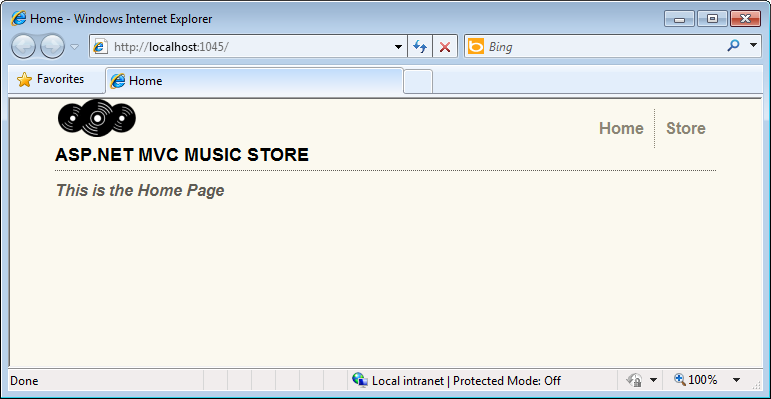
**<h2>This is the Home Page</h2>**

* + 1. </asp:Content>
  1. Select **MvcMusicStore** project in the Solution Explorer and Press **F5** to run the Application.
     1. 
     2. Figure 28
     3. MvcMusicStore Project selected in the Solution Explorer

# Next Step

* 1. Exercise 4: Verification

### Exercise 4: Verification

* 1. In order to verify that you have correctly performed all the steps in the previous exercise, proceed as follows:
  2. With the application open in a browser, you should note that:
     1. The HomeController’s Index action method found and displayed the **\Views\Home\Index.aspx** View template, even though the code called **return View()**, because the View template followed the standard naming convention.
     2. The Home Page displays the welcome message defined within the **\Views\Home\Index.aspx** view template.
     3. The Home Page is using the **MasterPage** template, and so the welcome message is contained within the standard site HTML layout.
        1. 
        2. Figure 29
        3. Home Index View using the MasterPage and style defined

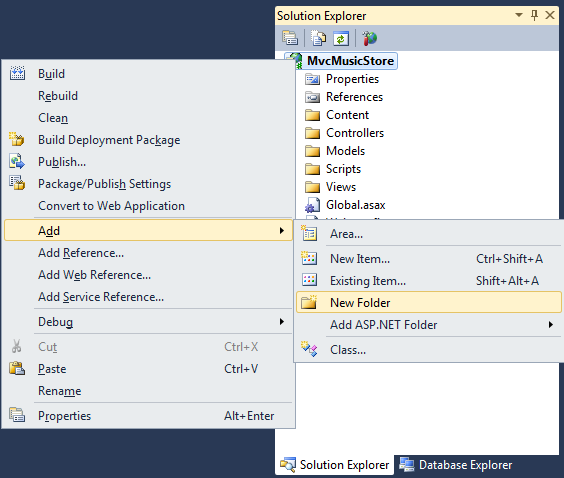
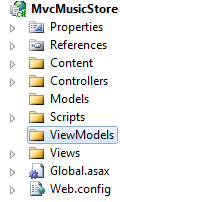
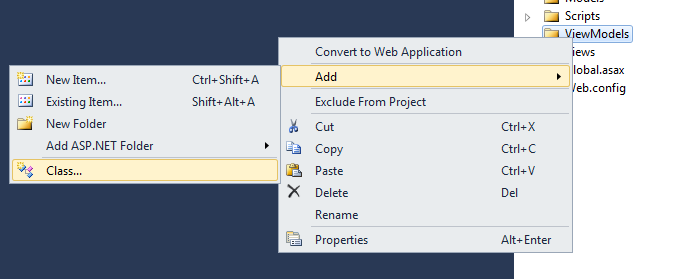
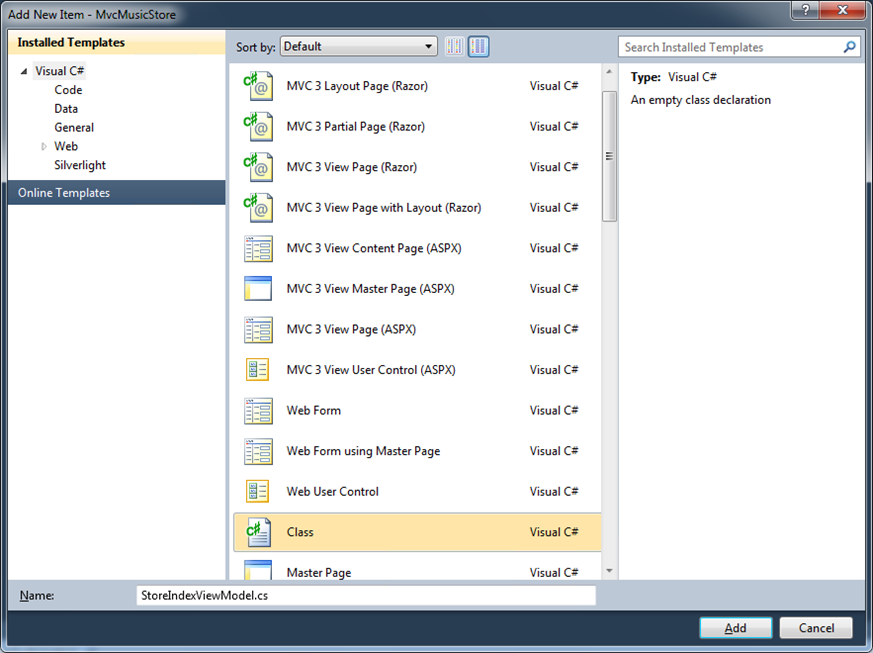
# Next Step

Exercise 5: Creating a View Model

Exercise 5: Creating a View Model

* 1. So far you made your Views display hardcoded HTML, but in order to create dynamic web applications, the View template should receive information from the Controller. One common technique to be used for that purpose is the **ViewModel** pattern, which allows a Controller to package up all the information needed to generate the appropriate HTML response.
  2. In this exercise, you will learn how to create a ViewModel class and add the needed properties: in this case the number of genres in the store and a list of those genres. Also, you will update the StoreController to use the created ViewModel and finally you will create a new View template that will display the mentioned properties in the page.

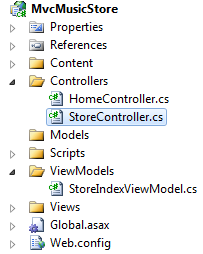
Task 1 – Creating a ViewModel Class

* 1. In this task, you will create a ViewModel class that will implement the Store genre listing scenario.
  2. If not already open, start Microsoft Visual Web Developer 2010 Express from **Start** | **All Programs** | **Microsoft Visual Studio 2010 Express** | **Microsoft Visual Web Developer 2010 Express**.
  3. In the **File** menu, choose **Open Project**. In the Open Project dialog, browse to Source\Ex05-CreatingAViewModel\Begin, select **MvcMusicStore.sln** and click **Open**. Alternatively, you may continue with the solution that you obtained after completing the previous exercise.
  4. Create a **ViewModels** folder to hold the ViewModel. To do this, right-click the top-level **MvcMusicStore** project, select **Add** and then **New Folder**.
     1. 
     2. Figure 30
     3. Adding a new folder
  5. Name the folder **ViewModels**.
     1. 
     2. Figure 31
     3. ViewModels folder in Solution Explorer
  6. Create a **ViewModel** class. To do this, right-click on the **ViewModels** folder recently created, select **Add** and then **Class**.
     1. 
     2. Figure 32
     3. Adding a new Class
  7. Name the class **StoreIndexViewModel** and click **Add**.
     1. 
     2. Figure 33
     3. Creating StoreIndexViewModel class

Task 2 – Adding Properties to the ViewModel class

* 1. There are two pieces of information to be passed from the StoreController to the View template in order to generate the HTML response wanted: the number of genres in the store and a list of those genres.
  2. In this task, you will add those 2 properties to the **StoreIndexViewModel** class: **NumberOfGenres** (an integer) and **Genres** (a list of strings).
  3. Add **NumberOfGenres** and **Genres** properties to the **StoreIndexViewModel** class. To do this, add the following 2 lines to the class definition:
     1. (Code Snippet – ASP.NET MVC 3.0 Fundamentals – Ex5 StoreIndexViewModel properties – CSharp)
     2. C#
     3. public class StoreIndexViewModel
     4. {
     5. **public int NumberOfGenres { get; set; }**
     6. **public List<string> Genres { get; set; }**
     7. }
     8. **Note**: The **{ get; set; }** notation makes use of C#’s auto-implemented properties feature. It provides the benefits of a property without requiring us to declare a backing field.

Task 3 – Updating StoreController to use the StoreIndexViewModel

* 1. The **StoreIndexViewModel** class encapsulates the information needed to pass from **StoreController**’s **Index** method to a View template in order to generate a response.
  2. In this task, you will update the **StoreController** to use the **StoreIndexViewModel**.
  3. Open **StoreController** class.
     1. 
     2. Figure 34
     3. Opening StoreController class
  4. In order to use the **StoreIndexViewModel** class from the **StoreController**, add the following namespace to the top of the **StoreController** code:
     1. (Code Snippet – ASP.NET MVC 3.0 Fundamentals – Ex5 StoreIndexViewModel using ViewModels – CSharp)
     2. C#

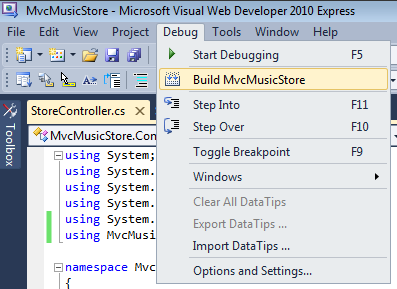
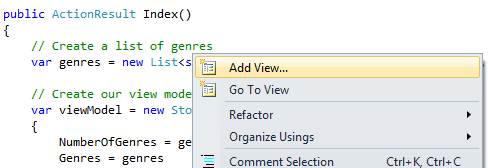
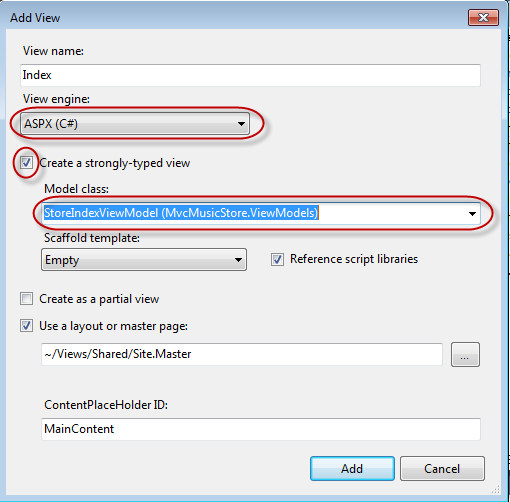
**using MvcMusicStore.ViewModels;**

* 1. Change the **StoreController**’s **Index** action method so that it creates and populates a **StoreIndexViewModel** object and then passes it off to a View template to generate an HTML response with it.
     1. **Note:** In Lab “ASP.NET MVC Models and Data Access” you will write code that retrieves the list of store genres from a database. In the following code, you will create a **List** of dummy data genres that will populate the **StoreIndexViewModel**.
     2. After creating and setting up the **StoreIndexViewModel** object, it will be passed as an argument to the **View** method. This indicates that the View template will use that object to generate an HTML response with it.
  2. Replace the **Index** method with the following code:
     1. (Code Snippet – ASP.NET MVC 3.0 Fundamentals – Ex5 StoreController Index method – CSharp)
     2. C#
     3. **public ActionResult Index()**
     4. **{**
     5. **// Create a list of genres**
     6. **var genres = new List<string> {"Rock", "Jazz", "Country", "Pop", "Disco"};**
     7. **// Create our view model**
     8. **var viewModel = new StoreIndexViewModel {**
     9. **NumberOfGenres = genres.Count(),**
     10. **Genres = genres**
     11. **};**
     12. **return View(viewModel);**

**}**

* + 1. **Note:** If you’re unfamiliar with C#, you may assume that using **var** means that the **viewModel** variable is late-bound. That’s not correct – the C# compiler is using type-inference based on what you assign to the variable to determine that **viewModel** is of type **StoreIndexViewModel**. Also, by compiling the local **viewModel** variable as a **StoreIndexViewModel** type you get compile-time checking and Visual Studio code-editor support.

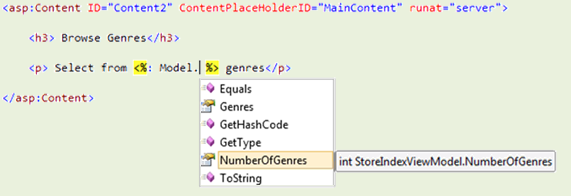
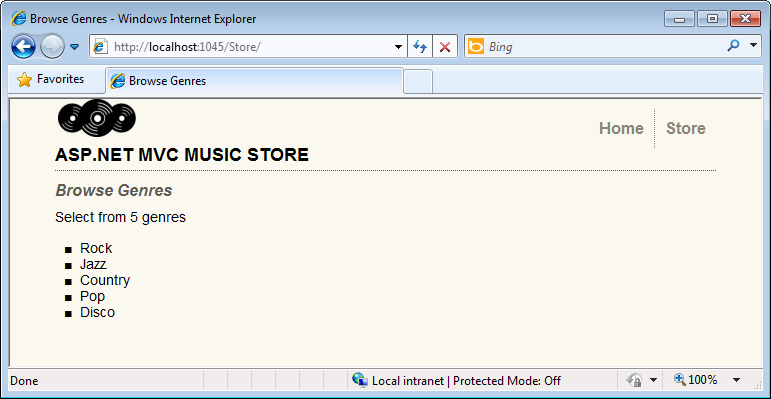
Task 4 – Creating a View Template that Uses StoreIndexViewModel

* 1. In this task, you will create a View template that will use a StoreIndexViewModel object passed from the Controller to display a list of genres.
  2. Before creating the new View template, let’s build the project so that the **Add View Dialog** knows about the **StoreIndexViewModel** class. Build the project by selecting the **Debug** menu item and then **Build MvcMusicStore**.
     1. 
     2. Figure 35
     3. Building the project
  3. Create a new View template. To do that, right-click inside the **Index** method and select **Add View.**
     1. 
     2. Figure 36
     3. Adding a View
  4. Because the **Add View Dialog** was invoked from the **StoreController**, it will add the View template by default in a \Views\Store\Index.aspx file. The view will be based on the Site.Master MasterPage template. Check the **Create a strongly-typed-view** checkbox and then select the **StoreIndexViewModel** as the **Model class**. Also, make sure that the View engine selected is **ASPX (C#)**. Click **Add**.
     1. 
     2. Figure 37
     3. Add View Dialog
     4. The \Views\Store\Index.aspx View template file is created and opened. Based on the information provided to the **Add View** dialog in the last step, the View template will expect a **StoreIndexViewModel** instance as the data to use to generate an HTML response. You will notice that by seeing that the template inherits a **ViewPage<MusicStore.ViewModels.StoreIndexViewModel>**.

Task 5 – Updating the View Template

* 1. In this task, you will update the View template created in the last task to output the number of genres and their names within the page.
  2. **Note:** You will use <%: %> syntax (often referred to as “code nuggets”) to execute code within the View template. There are two main ways you will see this used:
  3. 1. Code enclosed within <% %> will be executed
  4. 2. Code enclosed within <%: %> will be executed, and the result will be output to the page
  5. In the Index.aspx file, replace the code inside the **Content2** ASP.NET Content control with the following:
     1. HTML
     2. <asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
     3. **<h3> Browse Genres</h3>**
     4. **<p> Select from <%: Model.NumberOfGenres %> genres</p>**

</asp:Content>

* + 1. **Note:** As soon as you finish typing the period after the word **Model**, Visual Studio’s Intellisense feature supplies you with a list of possible properties and methods to choose from.
    2. 
    3. **Figure 38**
    4. *Getting Model properties and methods with Visual Studio’s IntelliSense*
    5. The **Model** property references the **StoreIndexViewModel** object that the Controller passed to the View template. This means that you can access all of the data passed the Controller to the View template via the **Model** property, and format it into an appropriate HTML response within the View template.
    6. You can just select the **NumberOfGenres** property from the Intellisense list rather than typing it in and then it will auto-complete it by pressing the **tab key**.
  1. Loop over the genre list in the **StoreIndexViewModel** and create an HTML **<ul>** list using a **foreach** loop. Also change the code inside **Content1** ASP.NET Content control with a simple text: **Store Genres.**
     1. HTML
     2. <%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master" Inherits="System.Web.Mvc.ViewPage<MvcMusicStore.ViewModels.StoreIndexViewModel>" %>
     3. <asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
     4. **Store Genres**
     5. </asp:Content>
     6. <asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
     7. <h3>Browse Genres</h3>
     8. <p>Select from <%: Model.NumberOfGenres %> genres</p>
     9. **<ul>**
     10. **<% foreach (string genreName in Model.Genres) { %>**
     11. **<li>**
     12. **<%: genreName %>**
     13. **</li>**
     14. **<% } %>**
     15. **</ul>**
     16. </asp:Content>
  2. Press **F5** to run the Application and browse **/Store**. You will see the list of genres passed in the **StoreIndexViewModel** object from the **StoreController** to the View template.
     1. 
     2. Figure 39
     3. View displaying a list of genres
  3. Close the browser.

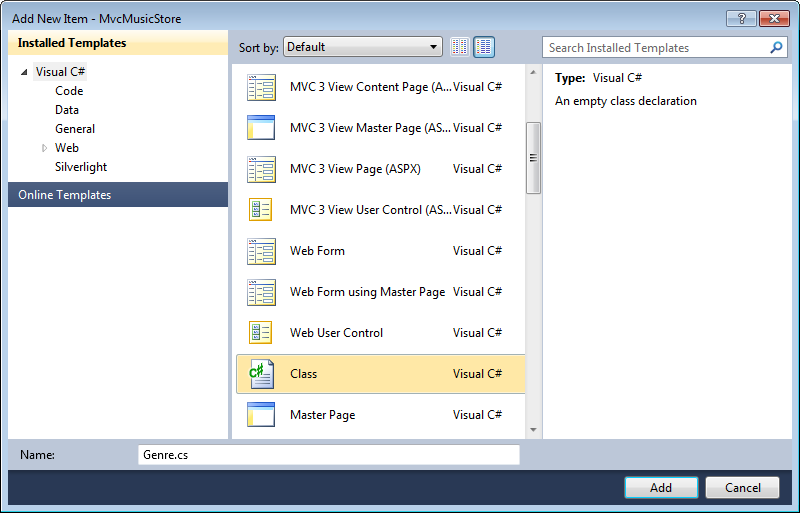
# Next Step

* 1. Exercise 6: Using Parameters in View

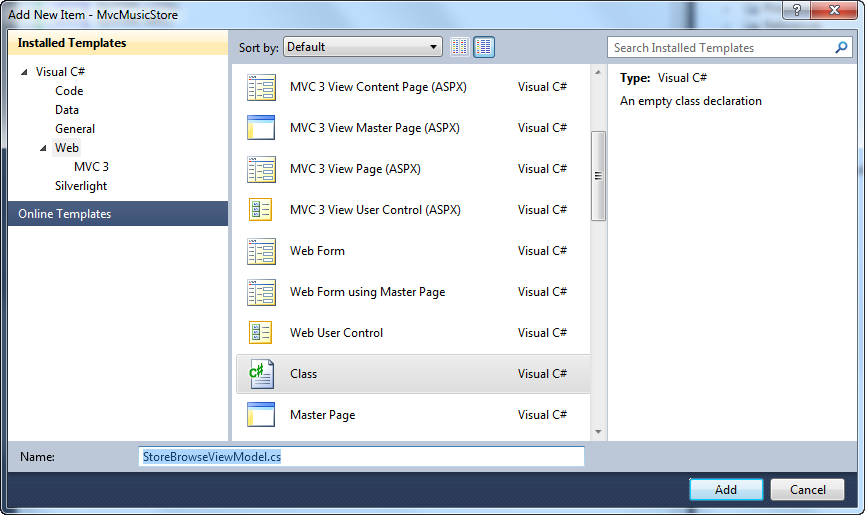
Exercise 6: Using Parameters in View

* 1. In Exercise 3 you learned how to pass parameters to the Controller. In this exercise, you will learn how to use those parameters in the View template. For that purpose, you will be introduced first to Model classes that will help you to manage your data and domain logic. Additionally, you will learn how to create links to pages inside the ASP.NET MVC application without worrying of things like URL paths encoding.

Task 1 – Adding Model Classes

* 1. Unlike ViewModels, which are created just to pass information from the Controller to the View, Model classes are built to contain and manage data and domain logic. In this task you will add two model classes to represent these concepts: **Genre** and **Album**. In the next Hands-on Lab, these classes will be hooked to a database, mapping each one of them to a table.
  2. If not already open, start Microsoft Visual Web Developer 2010 Express from **Start | All Programs | Microsoft Visual Studio 2010 Express | Microsoft Visual Web Developer 2010 Express**.
  3. In the **File** menu, choose **Open Project**. In the Open Project dialog, browse to Source\Ex06-UsingParametersInView\Begin, select **MvcMusicStore.sln** and click **Open**. Alternatively, you may continue with the solution that you obtained after completing the previous exercise.
  4. Add a **Genre** Model class. To do this, right-click the **Models** folder in the **Solution Explorer**, select **Add** and then the **Class** option. Name the file **Genre.cs** and click **Add**.
     1. 
     2. Figure 40
     3. Add Genre Model Class
  5. Add a **Name** property to the Genre class. To do this, add the following code:
     1. (Code Snippet – ASP.NET MVC 3.0 Fundamentals – Ex6 Genre – CSharp)
     2. C#
     3. using System;
     4. using System.Collections.Generic;
     5. using System.Linq;
     6. using System.Web;
     7. namespace MvcMusicStore.Models
     8. {
     9. public class Genre
     10. {
     11. **public string Name { get; set; }**
     12. }
     13. }
  6. Following the same procedure as before, add an **Album** class. To do this, right-click the **Models** folder in the **Solution Explorer**, select **Add** and then the **Class** option. Name the file **Album.cs** and click **Add**.
  7. Add two properties to the Album class**: Genre** and **Title**. To do this, add the following code:
     1. (Code Snippet – ASP.NET MVC 3.0 Fundamentals – Ex6 Album – CSharp)
     2. C#
     3. using System;
     4. using System.Collections.Generic;
     5. using System.Linq;
     6. using System.Web;
     7. namespace MvcMusicStore.Models
     8. {
     9. public class Album
     10. {
     11. **public string Title { get; set; }**
     12. **public Genre Genre { get; set; }**
     13. }
     14. }

Task 2 – Adding a StoreBrowseViewModel

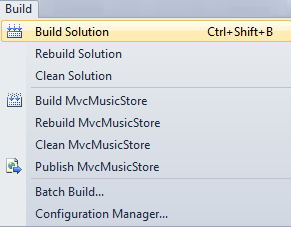
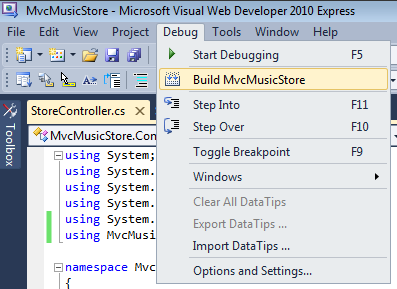
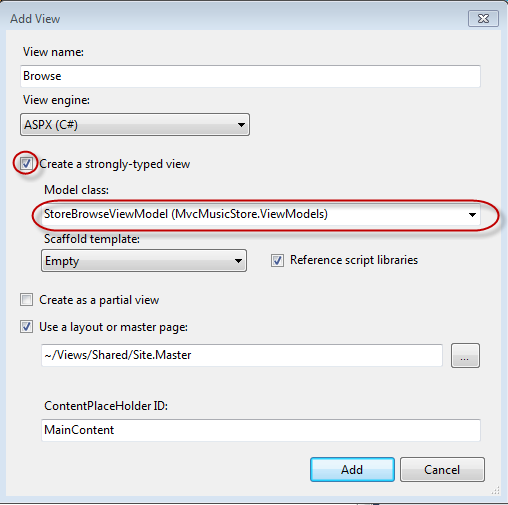
* 1. A **StoreBrowseViewModel** will be used in this task to show the Albums that match a selected Genre. In this task, you will create this class and then add two properties to handle the **Genre** and its **Album**’s List.
  2. Add a **StoreBrowseViewModel** class. To do this, right-click the **ViewModels** folder in the **Solution Explorer**, choose **Add** and then **Class**. Name the file **StoreBrowseViewModel.cs**.
     1. 
     2. Figure 41
     3. Adding StoreBrowseViewModel Class
  3. Add a reference to the Models in **StoreBrowseViewModel** class. To do this, add the following using namespace:
     1. (Code Snippet – ASP.NET MVC 3.0 Fundamentals – Ex6 UsingModel – CSharp)
     2. C#
     3. using System;
     4. using System.Collections.Generic;
     5. using System.Linq;
     6. using System.Web;
     7. **using MvcMusicStore.Models;**
     8. namespace MvcMusicStore.ViewModels
     9. {
     10. public class StoreBrowseViewModel
     11. {
     12. }
     13. }
  4. Add two properties to **StoreBrowseViewModel** class: **Genre** and **Albums.** To do this, add the following code:
     1. (Code Snippet – ASP.NET MVC 3.0 Fundamentals – Ex6 ModelProperties – CSharp)
     2. C#
     3. using System;
     4. using System.Collections.Generic;
     5. using System.Linq;
     6. using System.Web;
     7. using MvcMusicStore.Models;
     8. namespace MvcMusicStore.ViewModels
     9. {
     10. public class StoreBrowseViewModel
     11. {
     12. **public Genre Genre { get; set; }**
     13. **public List<Album> Albums { get; set; }**
     14. }
     15. }
     16. **Note:** What is **List<Album>** ?: This definition is using the **List<T>** type, where **T** constrains the type to which elements of this **List** belong to, in this case **Album** (or any of its descendants).
     17. This ability to design classes and methods that defer the specification of one or more types until the class or method is declared and instantiated by client code is a feature of the C# language called **Generics**.
     18. **List<T>** is the generic equivalent of the **ArrayList** type and is available in the **System.Collections.Generic** namespace**.** One of the benefits of using **generics** is that since the type is specified, you do not need to take care of type checking operations such as casting the elements into **Album** as you would do with an **ArrayList**.

Task 3 – Using the New ViewModel in the StoreController

1. In this task, you will modify the **StoreController**’s **Browse** and **Details** action methods to use the new **StoreBrowseViewModel**.
   1. Add a reference to the Models folder in **StoreController** class. To do this, expand the **Controllers** folder in the **Solution Explorer** and open the **StoreController** class. Then add the following code:
      1. (Code Snippet – ASP.NET MVC 3.0 Fundamentals – Ex6 UsingModelInController – CSharp)
      2. C#
      3. using System;
      4. using System.Collections.Generic;
      5. using System.Linq;
      6. using System.Web;
      7. using System.Web.Mvc;
      8. using MvcMusicStore.ViewModels;
      9. **using MvcMusicStore.Models;**
   2. Replace the **Browse** action method to use the **StoreViewBrowseController** class**.** You will create a Genre and two new Albums objects with dummy data (in the next Hands-on Lab you will consume real data from a database). To do this, replace the **Browse** method with the following code:
      1. (Code Snippet – ASP.NET MVC 3.0 Fundamentals – Ex6 BrowseMethod – CSharp)
      2. C#
      3. **//**
      4. **// GET: /Store/Browse?genre=Disco**
      5. **public ActionResult Browse(string genre)**
      6. **{**
      7. **var genreModel = new Genre()**
      8. **{**
      9. **Name = genre**
      10. **};**
      11. **var albums = new List<Album>()**
      12. **{**
      13. **new Album() { Title = genre + " Album 1" },**
      14. **new Album() { Title = genre + " Album 2" }**
      15. **};**
      16. **var viewModel = new StoreBrowseViewModel()**
      17. **{**
      18. **Genre = genreModel,**
      19. **Albums = albums**
      20. **};**
      21. **return View(viewModel);**
      22. **}**
   3. Replace the **Details** action method to use the **StoreViewBrowseController** class**.** You will create a new **Album** object to be returned to the **View**. To do this, replace the **Details** method with the following code:
      1. (Code Snippet – ASP.NET MVC 3.0 Fundamentals – Ex6 DetailsMethod – CSharp)
      2. C#
      3. **//**
      4. **// GET: /Store/Details/5**
      5. **public ActionResult Details(int id)**
      6. **{**
      7. **var album = new Album { Title = "Sample Album" };**
      8. **return View(album);**
      9. **}**

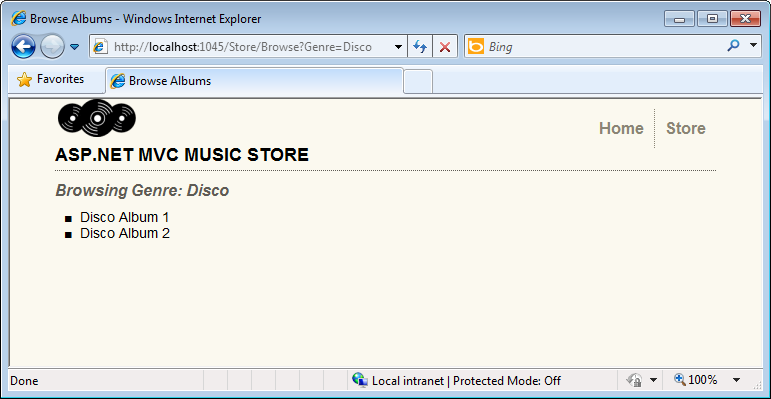
Task 4 – Adding a Browse View Template

In this task, you will add a **Browse** View to show the Albums found for a specific Genre.

* 1. Before creating the new View template, you should build the project so that the **Add View Dialog** knows about the **ViewModel** class to use. Build the project by selecting the **Debug** menu item and then **Build MvcMusicStore**.
     1. 
     2. Figure 42
     3. Building the project in Visual Web Developer 2010
     4. 
     5. Figure 43
     6. Building the project in Visual Studio 2010
  2. Add a **Browse** View. To do this, right-click in the **Browse** action method of the **StoreController** and click **Add View**.
  3. In the **Add View** dialog box, verify that the View Name is **Browse**. Check the **Create a strongly-typed view** checkbox and select **StoreBrowseViewModel** from the **Model class** dropdown. Leave the other fields with their default value. Then click **Add**.
     1. 
     2. Figure 44
     3. Adding a Browse View
  4. Modify the **Browse.aspx** to display the Genre’s information, accessing the **StoreBrowseViewModel** object that is passed to the view template. To do this, replace the **Title Content** and the **Main Content** with the following content:
     1. HTML
     2. <%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master" Inherits="System.Web.Mvc.ViewPage<MvcMusicStore.ViewModels.StoreBrowseViewModel>" %>
     3. <asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
     4. **Browse Albums**
     5. </asp:Content>
     6. <asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
     7. **<h2>Browsing Genre: <%: Model.Genre.Name %></h2>**
     8. **<ul>**
     9. **<% foreach (var album in Model.Albums)**
     10. **{ %>**
     11. **<li><%: album.Title %></li>**
     12. **<% } %>**
     13. **</ul>**
     14. </asp:Content>

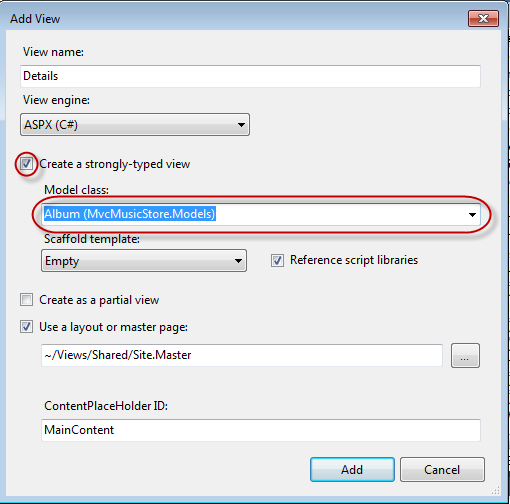
Task 5 – Running the Application

In this task, you will test that the **Browse** method retrieves Albums from the **Browse** method action.

* 1. Press **F5** to run the Application.
  2. The project starts in the Home page. Change the URL to **/Store/Browse?Genre=Disco** to verify that the action returns two Albums.
     1. 
     2. Figure 45
     3. Browsing Store Disco Albums

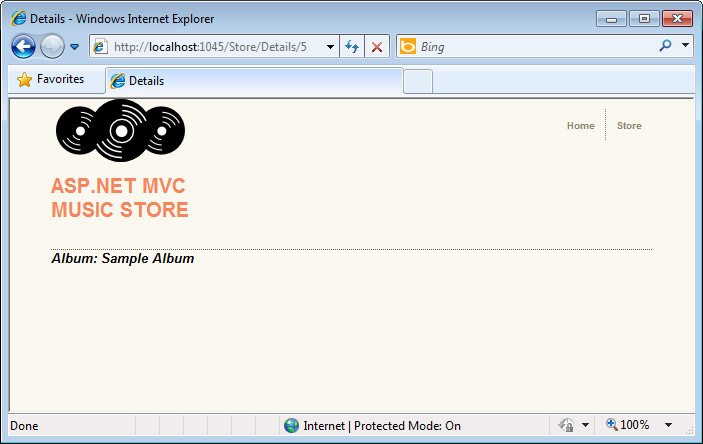
Task 6 – Displaying information About a Specific Album

In this task, you will implement the **Store/Details** view to display information about a specific album. In this Hands-on Lab, everything you will display about the album is already contained in the View template. So, instead of creating a **StoreDetailsViewModel** class you will use the current **StoreBrowseViewModel** template passing the Album to it.

* 1. Close the browser if needed, to return to the Visual Studio window. Add a new **Details** view for the **StoreController**’s **Details** action method. To do this, right-click the **Details** method in the **StoreController** class and click **Add View**.
  2. In the **Add View** dialog view, verify that the **View Name** is **Details**. Check the **Create a strongly-typed view** checkbox and select **Album** from the **Model class** drop-down. Leave the other fields with their default value. Then click **Add**. This will create and open a **\Views\Store\Details.aspx** file.
     1. 
     2. Figure 46
     3. Adding a Details View
  3. Modify the **Details.aspx** fileto display the Album’s information, accessing the **Album** object that is passed to the view template. To do this, replace the **Main Content** with the following content:
     1. HTML
     2. <%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master" Inherits="System.Web.Mvc.ViewPage<MvcMusicStore.Models.Album>" %>
     3. <asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
     4. Details
     5. </asp:Content>
     6. <asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
     7. <h2>Album: <%: Model.Title %></h2>
     8. </asp:Content>

Task 7 – Running the Application

In this task, you will test that the **Details** View retrieves Album’s information from the **Details action** method.

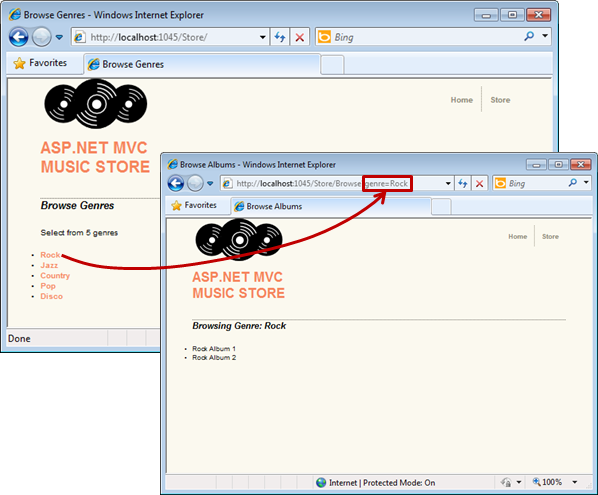
* 1. Press **F5** to run the Application.
  2. The project starts in the Home page. Change the URL to **/Store/Details/5** to verify the album’s information.
     1. 
     2. Figure 47
     3. Browsing Album’s Detail

Task 8 – Adding Links Between Pages

* 1. In this task, you will add a link in the Store View to have a link in every Genre name to the appropriate **/Store/Browse** URL. This way, when you click on a Genre, for instance **Disco**, it will navigate to **/Store/Browse?genre=Disco** URL.
  2. Close the browser if needed, to return to the Visual Studio window. Update the **Index** page to add a link to the **Browse** page. To do this, in the **Solution Explorer** expand the **Views** folder, then the **Store** folder and double-click the **Index.aspx** page.
  3. Add a link to the Browse view indicating the genre selected. To do this, replace the following code in the **Index.aspx**:
     1. HTML
     2. <%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master" Inherits="System.Web.Mvc.ViewPage<MvcMusicStore.ViewModels.StoreIndexViewModel>" %>
     3. <asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
     4. Browse Genres
     5. </asp:Content>
     6. <asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
     7. <h2>Browse Genres</h2>
     9. <p>Select from **<%: Model.NumberOfGenres** %></p>
     10. <ul>
     11. <% foreach (string genreName in Model.Genres)
     12. { %>
     13. <li>
     14. **<%: Html.ActionLink(genreName, "Browse", new { genre = genreName }, null) %>**
     15. </li>
     16. <% } %>
     17. </ul>
     18. </asp:Content>
     19. **Note**: another approach would be linking directly to the page, with a code like the following:
     20. <a href="/Store/Browse?genre=<%: genreName %>"><%: genreName %></a>
     21. Although this approach works, it depends on a hardcoded string. If you later rename the Controller, you will have to change this instruction manually. A better alternative is to use an **HTML Helper** method. ASP.NET MVC includes an HTML Helper method which is available for such tasks. The **Html.ActionLink()** helper method makes it easy to build HTML **<a>** links, making sure URL paths are properly URL encoded.
     22. Htlm.ActionLink has several overloads. In this exercise you will use one that takes three parameters:
     23. 1. Link text, which will display the Genre name
     24. 2. Controller action name (**Browse**)
     25. 3. Route parameter values, specifying both the name (**Genre**) and the value (**Genre name**)

Task 9 – Running the Application

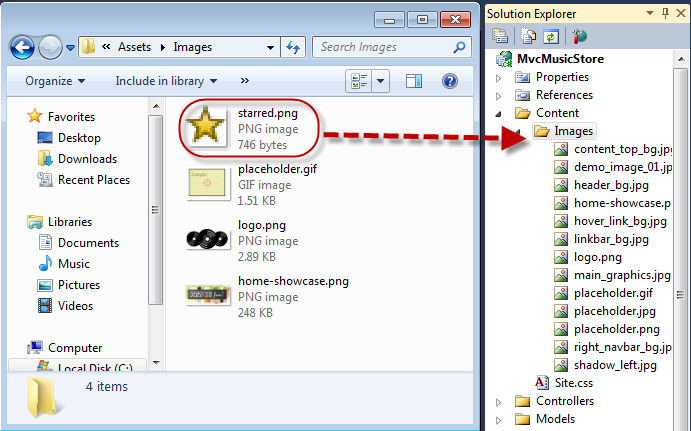
In this task, you will test that each Genre is displayed with a link to the appropriate **/Store/Browse** URL.

* 1. Press **F5** to run the Application.
  2. The project starts in the Home page. Change the URL to **/Store** to verify that each Genre links to the appropriate **/Store/Browse** URL.
     1. 
     2. Figure 48
     3. Browsing Genres with links to Browse page

Task 10 – Using Dynamic ViewModel Collection to Pass Values

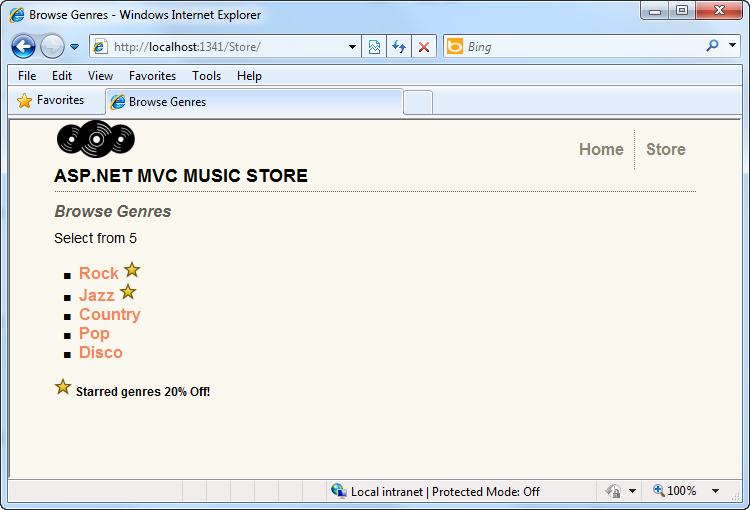
* 1. In this task, you will learn a simple and powerfull method to pass values between the Controller and the View without making any changes in the Model. MVC 3 provides the collection “ViewModel” than can be assigned to any dynamic value and accessed inside controllers and views as well.
  2. You will now use ViewBag dynamic collection to pass from the controller to the view a list of “**Starred genres**”. The Store Index view will access to **ViewModel** and display the information.
  3. Close the browser if needed, to return to the Visual Studio window. Open **StoreController.cs** and modify **Index** methodto create a list of starred genres into ViewModel collection :
     1. C#
     2. public ActionResult Index()
     3. {
     4. // Create list of genres
     5. var genres = new List<string> { "Rock", "Jazz", "Country", "Pop", "Disco" };
     6. // Create your view model
     7. var viewModel = new StoreIndexViewModel
     8. {
     9. NumberOfGenres = genres.Count,
     10. Genres = genres
     11. };
     12. **ViewBag.Starred = new List<string> { "Rock", "Jazz" };**
     14. return View(viewModel);

}

* + 1. **Note:** You could also use the syntax **ViewBag[“Starred”]** to access the properties.
  1. The star icon **“starred.png”** is included in the **Source\Assets\Images** folder of this lab. In order to add them to the application, drag their content from a **Windows Explorer** window into the **Solution Explorer** in Visual Web Developer Express, as shown below:
     1. 
     2. Figure 49
     3. Adding star image to the solution
  2. Open the view **Store/Index.aspx** and modify the content. You will read the “starred” property in the **ViewBag** collection, and ask if the current genre name is in the list. In that case you will show a star icon right to the genre link.
     1. HTML
     2. <%@ Page Title="" Language="C#" MasterPageFile="~/Views/Shared/Site.Master" Inherits="System.Web.Mvc.ViewPage<MvcMusicStore.ViewModels.StoreIndexViewModel>" %>
     3. <asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
     4. Browse Genres
     5. </asp:Content>
     6. <asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
     7. <h2>Browse Genres</h2>
     9. <p>Select from <%: Model.NumberOfGenres %></p>
     10. <ul>
     11. <% foreach (string genreName in Model.Genres)
     12. { %>
     13. <li>
     14. <%: Html.ActionLink(genreName, "Browse", new { genre = genreName }, null) %>
     16. **<% if (ViewBag.Starred.Contains(genreName)) { %>**
     17. **<img src="../../Content/Images/starred.png" alt="Starred element" />**
     18. **<% } %>**
     19. </li>
     20. <% } %>
     21. </ul>
     22. **<br/>**
     23. **<h5> <img src="../../Content/Images/starred.png" alt="Starred element" /> Starred genres 20% Off! </h5>**
     24. </asp:Content>

Task 11 – Running the Application

In this task, you will test that the starred genres display a star icon.

* 1. Press **F5** to run the Application.
  2. The project starts in the Home page. Change the URL to **/Store** to verify that each featured genre has the respecting label:
     1. 
     2. Figure 50
     3. Browsing Genres with starred elements

# Next Step

* 1. Summary

Summary

* 1. By completing this Hands-On Lab you have learned the fundamentals of ASP.NET MVC:
  + The core elements of an MVC application and how they interact
  + How to create an ASP.NET MVC Application
  + How to add and configure Controllers to handle parameters passed through the URL and querystring
  + How to add a MasterPage to setup a template for common HTML content, a StyleSheet to enhance the look and feel and a View template to display HTML content
  + How to use the ViewModel pattern for passing properties to the View template for them to display dynamic information
  + How to use parameters passed to Controllers in the View template
  + How to add links to pages inside the ASP.NET MVC application
  + How to add and use dynamic properties in a View