* 1. 

Hands-On Lab

ASP.NET MVC Helpers, Forms and Validation

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

Contents

[Overview 4](#_Toc282517403)

[Exercise 1: Creating the Store Manager controller and its Index view 6](#_Toc282517404)

[Task 1 – Creating the StoreManagerController 6](#_Toc282517405)

[Task 2 – Customizing the StoreManager Index 7](#_Toc282517406)

[Task 3 – Creating the Index View 7](#_Toc282517407)

[Task 4 – Customizing the scaffold of the Index View 9](#_Toc282517408)

[Task 5 – Running the Application 12](#_Toc282517409)

[Exercise 2: Adding an HTML Helper 14](#_Toc282517410)

[Task 1 – Extending the HTML Helper 14](#_Toc282517411)

[Task 2 – Registering the HTML Helper 17](#_Toc282517412)

[Task 3 – Truncating Text in the Page 18](#_Toc282517413)

[Task 4 – Running the Application 18](#_Toc282517414)

[Exercise 3: Creating the Edit View 19](#_Toc282517415)

[Task 1 – Creating the StoreManagerViewModel 20](#_Toc282517416)

[Task 2 – Implementing the HTTP-GET Edit Action Method 21](#_Toc282517417)

[Task 3 – Creating the Edit View 22](#_Toc282517418)

[Task 4 – Customizing the Edit View 25](#_Toc282517419)

[Task 5 – Running the Application 25](#_Toc282517420)

[Task 6 – Creating a Shared Album Editor Template 26](#_Toc282517421)

[Task 7 – Implementing drop-downs on the Album Editor Template 28](#_Toc282517422)

[Task 8 – Running the Application 29](#_Toc282517423)

[Task 9 – Implementing the HTTP-POST Edit action method 30](#_Toc282517424)

[Task 10 – Running the Application 32](#_Toc282517425)

[Exercise 4: Adding a Create View 34](#_Toc282517426)

[Task 1 – Implementing the HTTP-GET Create action method 34](#_Toc282517427)

[Task 2 – Adding the Create View 35](#_Toc282517428)

[Task 3 – Running the Application 37](#_Toc282517429)

[Task 4 – Implementing the HTTP-POST Create Action Method 37](#_Toc282517430)

[Task 5 – Running the Application 38](#_Toc282517431)

[Exercise 5: Handling Deletion 40](#_Toc282517432)

[Task 1 – Implementing the HTTP-GET Delete Action Method 40](#_Toc282517433)

[Task 2 – Running the Application 43](#_Toc282517434)

[Task 3– Implementing the HTTP-POST Delete Action Method 44](#_Toc282517435)

[Task 4 – Deleting on Cascade 44](#_Toc282517436)

[Task 5 – Running the Application 46](#_Toc282517437)

[Exercise 6: Adding Validation 47](#_Toc282517438)

[Task 1 – Adding Data Annotations 47](#_Toc282517439)

[Task 2 – Running the Application 50](#_Toc282517440)

[Task 3 – Adding Client-Side (AJAX) validation 52](#_Toc282517441)

[Task 4 – Running the Application 53](#_Toc282517442)

[Summary 55](#_Toc282517443)

Overview

* 1. **Note:** This Hands-on Lab assumes you have basic knowledge of **ASP.NET MVC**. If you have not used **ASP.NET MVC** before, we recommend you to go over **ASP.NET MVC Fundamentals** Hand-on Lab.
  2. In **ASP.NET MVC Models and Data Access** Hand-on Lab, you have been loading and displaying data from the database. In this Hands-on Lab, you will add to the **Music Store** application the ability to edit that data.
  3. With that goal in mind, you will first create the controller that will support the Create, Read, Update and Delete (CRUD) actions of albums. You will generate an Index View template taking advantage of ASP.NET MVC’s scaffolding feature to display the albums’ properties in an HTML table. To enhance that view, you will add a custom HTML helper that will truncate long descriptions.
  4. Afterwards, you will add the Edit and Create Views that will let you alter the albums in the database, with the help of form elements like dropdowns.
  5. Lastly, you will let users delete an album and also you will prevent them from entering wrong data by validating their input.

# Objectives

* 1. In this Hands-On Lab, you will learn how to:
  + Create a controller to support CRUD operations
  + Generate an Index View to display entity properties in an HTML table
  + Add a custom HTML helper
  + Create and customize an Edit View
  + Differentiate between action methods that react to either HTTP-GET or HTTP-POST calls
  + Add and customize a Create View
  + Handle the deletion of an entity
  + Validate user input

# System Requirements

* 1. You must have the following items to complete this lab:
  + ASP.NET and ASP.NET MVC 3
  + Visual Studio 2010 Express
  + SQL Server Database (Express edition or above)
    1. **Note:** You can install the previous system requirements by using the Web Platform Installer 3.0: <http://go.microsoft.com/fwlink/?LinkID=194638>.

# 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.

# Exercises

* 1. This Hands-On Lab is comprised by the following exercises:
  2. Exercise 1: Creating the Store Manager controller and its Index view
  3. Exercise 2: Adding an HTML Helper
  4. Exercise 3: Creating the Edit View
  5. Exercise 4: Adding a Create View
  6. Exercise 5: Handling Deletion
  7. Exercise 6: Adding Validation
  8. Estimated time to complete this lab: **35 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.

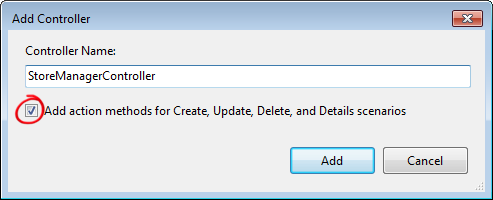
# Next Step

Exercise 1: Creating the Store Manager controller and its Index view

Exercise 1: Creating the Store Manager controller and its Index view

* 1. In this exercise, you will learn how to create a new controller to support CRUD operations, customize its Index action method to return a list of albums from the database and finally generating an Index View template taking advantage of ASP.NET MVC’s scaffolding feature to display the albums’ properties in an HTML table.

Task 1 – Creating the StoreManagerController

* 1. In this task, you will create a new controller called **StoreManagerController** to support CRUD operations.
  2. 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\Ex01-CreatingTheStoreManagerController\Begin, select **MvcMusicStore.sln** and click **Open**.
  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 **StoreManagerController** and make sure that the checkbox **Add action methods for Create, Update, Delete, and Details scenarios** is checked. Click **Add**.
     1. 
     2. Figure 1
     3. Add Controller Dialog
  5. A new Controller class is generated. Since you indicated to **add action methods for Create, Update, Delete, and Details scenarios**, stub methods for those, common CRUD actions are created with TODO comments filled in, prompting to include the application specific logic.
  6. Since you don’t need the Details Controller action, delete that method.

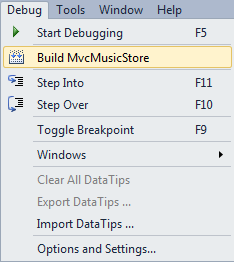
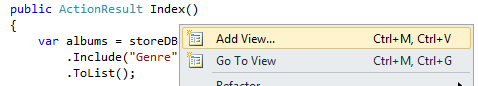
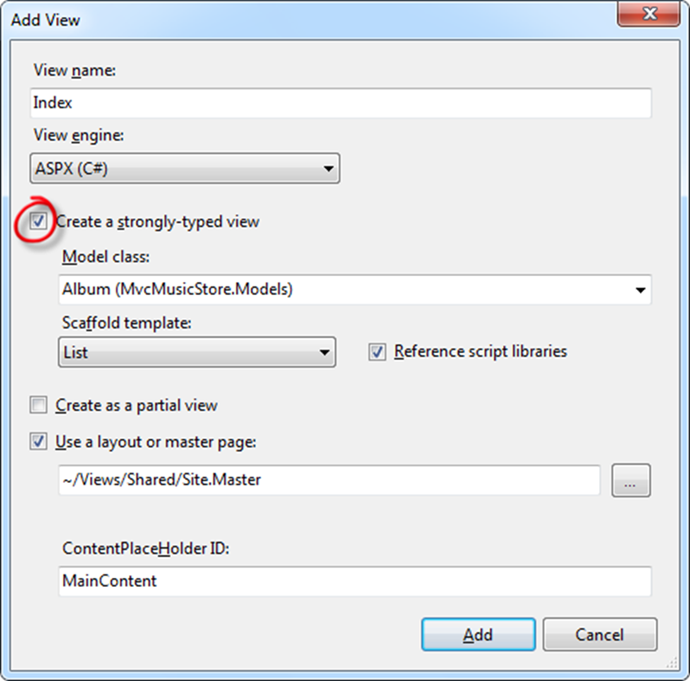
Task 2 – Customizing the StoreManager Index

* 1. In this task, you will customize the StoreManager Index action method to return a View with the list of albums from the database.
  2. Add a using directive to **MvcMusicStore.Models**:
     1. (Code Snippet – ASP.NET MVC 3 Helpers and Forms and Validation – Ex1 using MvcMusicStore – CSharp)
     2. C#
     3. using System.Web.Mvc;
     4. **using MvcMusicStore.Models;**
  3. Add a field to the **StoreManagerController** to hold an instance of **MusicStoreEntities.**
     1. (Code Snippet – ASP.NET MVC 3 Helpers and Forms and Validation – Ex1 MusicStoreEntities – CSharp)
     2. C#
     3. public class StoreManagerController : Controller
     4. {
     5. **MusicStoreEntities storeDB = new MusicStoreEntities();**
  4. Implement the **StoreManagerController** **Index** action to return a View with the list of albums. The Controller action logic will be very similar to the StoreController’s Index action written earlier. Use LINQ to retrieve all albums, including Genre and Artist information for display.

(Code Snippet – ASP.NET MVC 3 Helpers and Forms and Validation – Ex1 StoreManagerController Index – CSharp)

* + 1. C#
    2. //
    3. // GET: /StoreManager/
    4. public ActionResult Index()
    5. {
    6. **var albums = storeDB.Albums**
    7. **.Include("Genre").Include("Artist")**
    8. **.ToList();**
    9. **return View(albums);**
    10. }

Task 3 – Creating the Index View

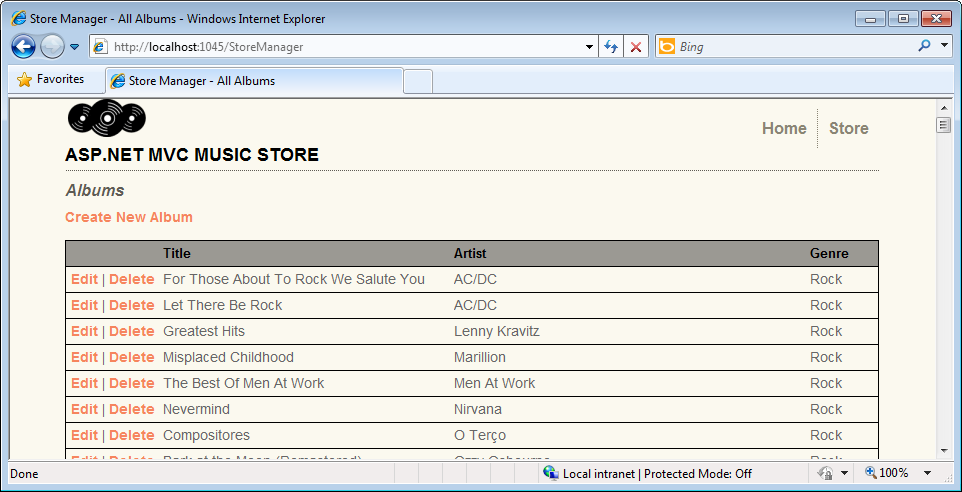
* 1. In this task, you will create the Index View template to display the list of albums returned by the **StoreManager** Controller.
  2. Before creating the new View template, you should build the project so that the **Add View Dialog** knows about the **Album** class to use. Build the project by selecting the **Debug** menu item and then **Build MvcMusicStore**.
     1. 
     2. Figure 2
     3. Building the project
  3. Right-click inside the **Index** action method and select **Add View**. This will bring up the Add View dialog.
     1. 
     2. Figure 3
     3. Adding a View from within the Index method
  4. In the Add View dialog, verify that the View Name is **Index**. Check the **Create a strongly-typed view** checkbox and select **Album (MvcMusicStore.Models)** from the **View data class** drop-down. Select **List** from the **View content** drop-down. Leave the **View Engine** to **ASPX (C#)**  and the other fields with their default value and then click **Add**.
     1. 
     2. Figure 4
     3. Adding an Index View

Task 4 – Customizing the scaffold of the Index View

* 1. In this task, you will adjust the simple View template created with ASP.NET MVC scaffolding feature to have it display the fields you want.
  2. **Note:** The **scaffolding** support within ASP.NET MVC generates a simple View template which lists all fields in the Album model. **Scaffolding** provides a quick way to get started on a strongly typed view: rather than having to write the View template manually, scaffolding quickly generates a default template and then you can modify the generated code.
  3. Review the code created. The generated list of fields will be part of the following HTML table that **Scaffolding** is using for displaying tabular data.
     1. HTML
     2. <table>
     3. <tr>
     4. <th></th>
     5. <th>
     6. GenreId
     7. </th>
     8. <th>
     9. ArtistId
     10. </th>
     11. <th>
     12. Title
     13. </th>
     14. <th>
     15. Price
     16. </th>
     17. <th>
     18. AlbumArtUrl
     19. </th>
     20. </tr>
     21. <% foreach (var item in Model) { %>
     22. <tr>
     23. <td>
     24. <%: Html.ActionLink("Edit", "Edit", new { id=item.AlbumId }) %> |
     25. <%: Html.ActionLink("Details", "Details", new { id=item.AlbumId }) %> |
     26. <%: Html.ActionLink("Delete", "Delete", new { id=item.AlbumId }) %>
     27. </td>
     28. <td>
     29. <%: item.GenreId %>
     30. </td>
     31. <td>
     32. <%: item.ArtistId %>
     33. </td>
     34. <td>
     35. <%: item.Title %>
     36. </td>
     37. <td>
     38. <%: String.Format("{0:F}", item.Price) %>
     39. </td>
     40. <td>
     41. <%: item.AlbumArtUrl %>
     42. </td>
     43. </tr>
     44. <% } %>
     45. </table>
     46. **Note:** the generated template is following the same coding practices you have seen at ASP.NET MVC Fundamentals Hands-on lab: **<%: %>** code nuggets to HTML encode values, and **Html.ActionLink()** helper method to generate links to other controller actions.
  4. Display only the **Album Title**, **Artist**, and **Genre** fields. To do this, delete the **AlbumId**, **Price** and **Album Art URL** columns. Also, change GenreId and ArtistId columns to display their linked class properties of **Artist.Name** and **Genre.Name**. Finally, remove the **Details** link.
     1. To do this, replace the **<table>** code with the following:
     2. HTML
     3. **<table>**
     4. **<tr>**
     5. **<th></th>**
     6. **<th>Title</th>**
     7. **<th>Artist</th>**
     8. **<th>Genre</th>**
     9. **</tr>**
     10. **<% foreach (var item in Model) { %>**
     11. **<tr>**
     12. **<td>**
     13. **<%: Html.ActionLink("Edit", "Edit", new { id=item.AlbumId }) %> |**
     14. **<%: Html.ActionLink("Delete", "Delete", new { id=item.AlbumId })%>**
     15. **</td>**
     16. **<td><%: item.Title %></td>**
     17. **<td><%: item.Artist.Name %></td>**
     18. **<td><%: item.Genre.Name %></td>**
     19. **</tr>**
     20. **<% } %>**
     21. **</table>**
  5. Change the following descriptions.
     1. HTML
     2. <asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
     3. **Store Manager - All Albums**
     4. </asp:Content>
     5. <asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
     6. **<h2>Albums</h2>**
  6. Also update the link to create a new Album changing its text.
     1. HTML
     2. <asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
     3. Store Manager - All Albums
     4. </asp:Content>
     5. <asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
     6. <h2>Albums</h2>
     7. <p>
     8. <%: Html.ActionLink("Create New Album", "Create") %>
     9. </p>

Task 5 – Running the Application

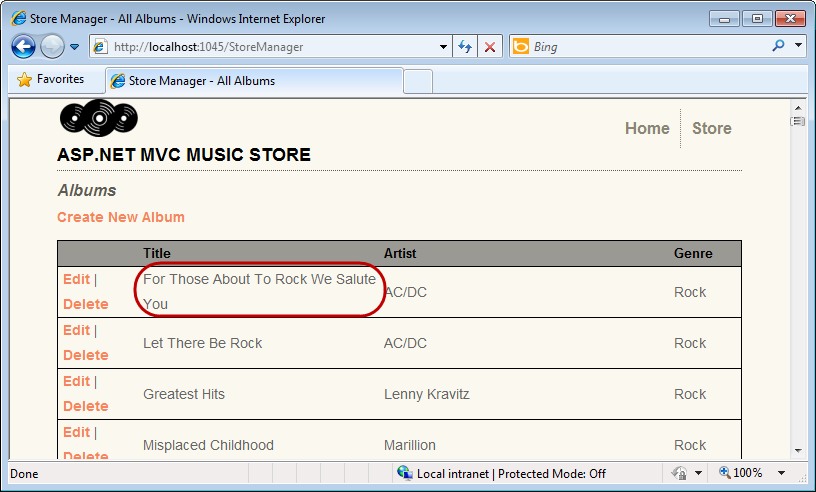
In this task, you will test that the **StoreManager** **Index** View template displays a list of albums according to the design of the previous steps.

* 1. Press **F5** to run the Application.
  2. The project starts in the Home page. Change the URL to **/StoreManager** to verify that a list of albums is displayed, showing their **Title**, **Artist** and **Genre**.
     1. ****
     2. Figure 5
     3. Browsing the list of Albums

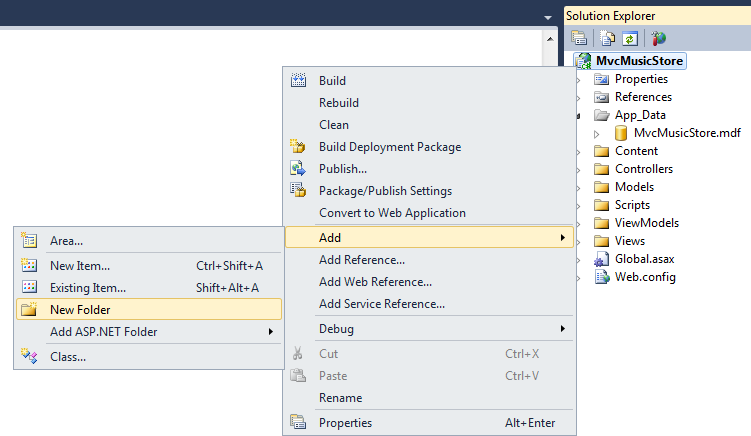
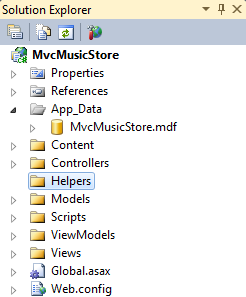
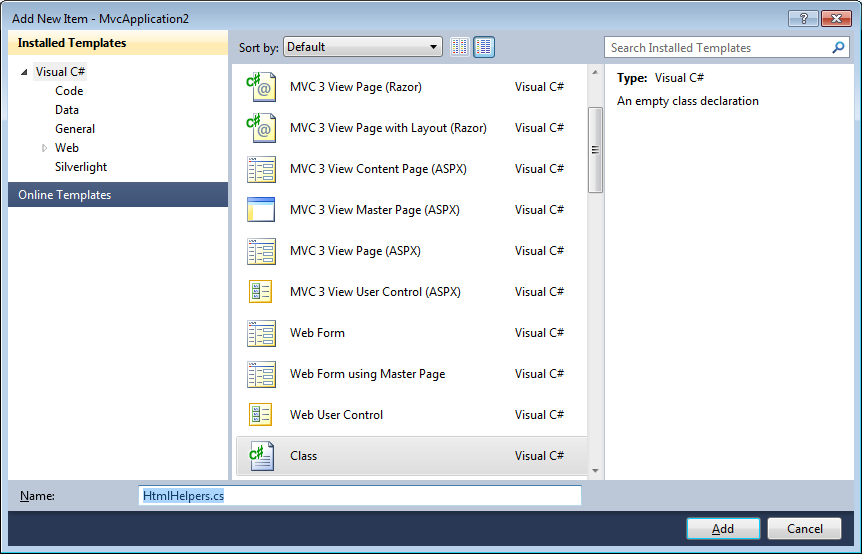
# Next Step

* 1. Exercise 2: Adding an HTML Helper

Exercise 2: Adding an HTML Helper

* 1. The StoreManager Index page has one potential issue: Title and Artist Name properties can both be long enough to throw off the table formatting. In this exercise you will learn how to add a custom HTML helper to truncate that text.
  2. In the following figure, you can see how the format is modified because of the length of the text.
  3. 
  4. Figure 6
  5. Browsing the list of Albums with not truncated text

Task 1 – Extending the HTML Helper

* 1. In this task, you will add a new method **Truncate** to the **Html** object exposed within ASP.NET MVC Views. To do this, you will implement an **extension method** to the built-in **System.Web.Mvc.HtmlHelper** class provided by ASP.NET MVC.
  2. **Note:** To read more about **extension methods**, please visit this [msdn article.](http://msdn.microsoft.com/en-us/library/bb383977.aspx)
  3. Start Microsoft Visual Web Developer 2010 Express from **Start** | **All Programs** | **Microsoft Visual Studio 2010 Express** | **Microsoft Visual Web Developer 2010 Express**.
  4. In the **File** menu, choose **Open Project**. In the Open Project dialog, browse to Source\Ex02-AddingAnHTMLHelper\Begin, select **MvcMusicStore.sln** and click **Open**.
  5. Add a new directory **Helpers** to place the new extension method. To do this, in the Solution Explorer, right-click the **MvcMusicStore** project, then **Add**, and **New Folder.**
     1. 
     2. Figure 7
     3. Adding Helpers folder
  6. Rename the folder to **Helpers**.
     1. 
     2. Figure 8
     3. Helpers folder
  7. Add a new class **HtmlHelpers** in the **Helpers** folder. To do this, right-click the **Helpers** folder, select **Add** and then **Class**. In the **Add New Item** dialog, change the name to **HtmlHelpers.cs** and click **Add**.
     1. 
     2. Figure 9
     3. Adding HtmlHelpers.cs class
  8. Add a using directive to **System.Web.Mvc** and a new method to the class. Because of extension methods definition, HtmlHelpers and the new method need to be static. Replace with the following code:
     1. (Code Snippet – ASP.NET MVC 3 Helpers and Forms and Validation – Ex2 TruncateMethod – 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. **namespace MvcMusicStore.Helpers**
     9. **{**
     10. **public static class HtmlHelpers**
     11. **{**
     12. **public static string Truncate(this HtmlHelper helper, string input, int length)**
     13. **{**
     14. **if (input.Length <= length)**
     15. **{**
     16. **return input;**
     17. **}**
     18. **else**
     19. **{**
     20. **return input.Substring(0, length) + "...";**
     21. **}**
     22. **}**
     23. **}**
     24. **}**
     25. **Note:** This **Truncate** method will truncate a text if it is longer than the **length** parameter, adding a **…** for the remainder. If the text is shorter, the text will remain as is.
     26. You may notice that the first parameter in the **Truncate** methodincludes a **this**. This is the way of defining the method as an extension method to the **HtmlHelpers** class. This will allow all **HtmlHelpers** class instances to use the **Truncate** method as if it was a class method. For example the implementation you will see in Task 3: **Html.Truncate(item.Title, 25)**.

Task 2 – Registering the HTML Helper

* 1. In this task, you will register the HTML Helper with the application by modifying the application’s **Web.config** file.
  2. Open the **Web.config** file from the Solution Explorer.
  3. Add a reference to the **MvcMusicStore.Helpers** namespace in the **pages** section.
     1. XML
     2. <pages>
     3. <namespaces>
     4. <add namespace="System.Web.Mvc" />
     5. <add namespace="System.Web.Mvc.Ajax" />
     6. <add namespace="System.Web.Mvc.Html" />
     7. <add namespace="System.Web.Routing" />
     8. **<add namespace="MvcMusicStore.Helpers" />**
     9. </namespaces>

</pages>

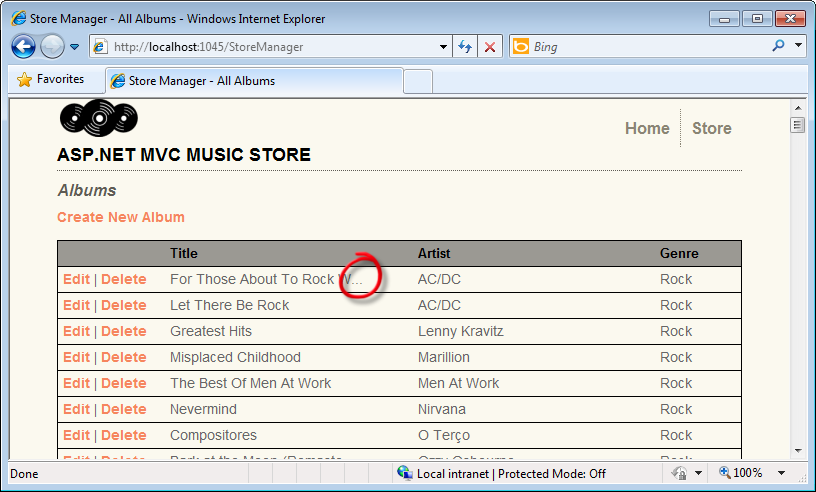
* + 1. **Note:**  Another aproach would be to add an import directive in all the pages that need it:
    2. <%@ Import Namespace="MvcMusicStore.Helpers" %>.
    3. Although, by adding the reference in the Web.config file, it will be available for all pages.
    4. You will find more information about import directive in this [msdn article](http://msdn.microsoft.com/en-us/library/eb44kack%28VS.71%29.aspx).

Task 3 – Truncating Text in the Page

* 1. In this task, you will use the **Truncate** method to truncate the text in the View template.
  2. Open StoreManager’s Index View. To do this, in the Solution Explorer expand the **Views** folder, then the **StoreManager** and open the **Index.aspx** file.
  3. Replace the lines that show the Album’s **Title** and **Artist Name**. To do this, replace the following lines:
     1. HTML
     2. <% foreach (var item in Model) { %>
     4. <tr>
     5. <td>
     6. <%: Html.ActionLink("Edit", "Edit", new { id=item.AlbumId }) %> |
     7. <%: Html.ActionLink("Delete", "Delete", new { id=item.AlbumId })%>
     8. </td>
     9. **<td><%: Html.Truncate(item.Title, 25) %></td>**
     10. **<td><%: Html.Truncate(item.Artist.Name, 25) %></td>**
     11. <td><%: item.Genre.Name %></td>
     12. </tr>
     14. <% } %>

Task 4 – Running the Application

In this task, you will test that the **StoreManager** **Index** View template truncates the Album’s Title and Artist Name.

* 1. Press **F5** to run the Application.
  2. The project starts in the Home page. Change the URL to **/StoreManager** to verify that long texts in the **Title** and **Artist** columnare truncated.
     1. 
     2. Figure 10
     3. Truncated Titles and Artist Names

# Next Step

* 1. Exercise 3: Creating the Edit View

Exercise 3: Creating the Edit View

In this exercise, you will learn how to create a form to allow store managers to edit an Album. They will browse the **/StoreManager/Edit/id** URL (**id** being the unique id of the album to edit), thus making an HTTP-GET call to the server.

The Controller Edit action method will retrieve the appropriate Album from the database, create a **StoreManagerViewModel** object to encapsulate it (along with a list of Artists and Genres), and then pass it off to a View template to render the HTML page back to the user. This page will contain a **<form>** element with textboxes and dropdowns for editing the Album properties.

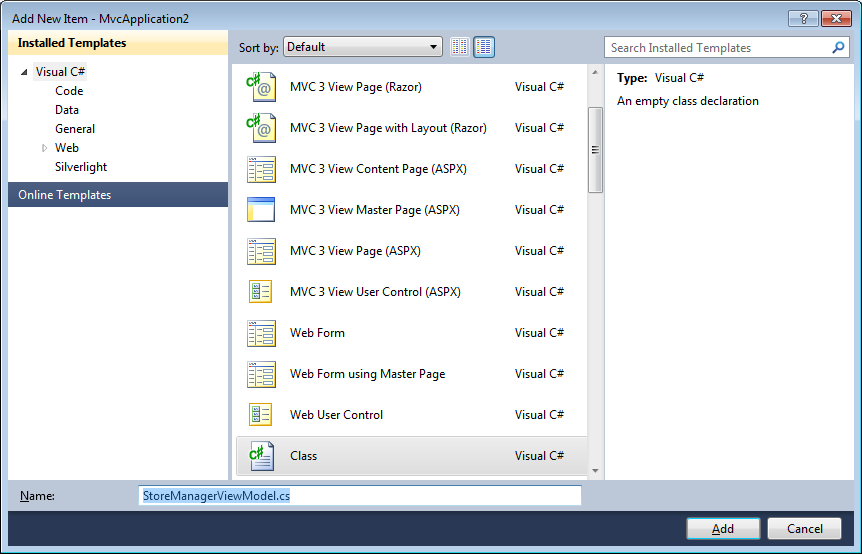
Once the user updates the Album form values and clicks the **Save** button, the changes are submitted via an HTTP-POST call back to **/StoreManager/Edit/id**. Although the URL remains the same as in the last call, ASP.NET MVC identifies that this time it is an HTTP-POST and therefore executes a different Edit action method (one decorated with **[HttpPost]**).

Task 1 – Creating the StoreManagerViewModel

In order to build and manage the intended Edit form, you will first need to pass to the View template an object with the following:

* 1. An Album object that represents the current Album being edited
  2. A List of all Genres to populate the Genre dropdown list
  3. A List of all Artists to populate the Artist dropdown list

In this task, you will create a new StoreManagerViewModel class to help manage all of the above data. This class will be used within both Edit and Create action methods.

* 1. Start Microsoft Visual Web Developer 2010 Express from **Start** | **All Programs** | **Microsoft Visual Studio 2010 Express** | **Microsoft Visual Web Developer 2010 Express**.
  2. In the **File** menu, choose **Open Project**. In the Open Project dialog, browse to Source\Ex03-CreatingTheEditView\Begin, select **MvcMusicStore.sln** and click **Open**.
  3. Create the **StoreManagerViewModel** class. To do this, right-click on the **ViewModels** folder, select **Add** and then **Class**.
  4. Change the name to **StoreManagerViewModel.cs** and click **Add.**
     1. 
     2. Figure 11
     3. Creating StoreManagerViewModel class
  5. Add the following using directives:
     1. (Code Snippet – ASP.NET MVC 3 Helpers and Forms and Validation – Ex3 using System.Web.Mvc and MvcMusicStore.Models – 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.Models;**
  6. Add the **Album**, **Artists** and **Genres** properties.
     1. (Code Snippet – ASP.NET MVC 3 Helpers and Forms and Validation – Ex3 StoreManagerViewModel properties– CSharp)
     2. C#
     3. public class StoreManagerViewModel
     4. {
     5. **public Album Album { get; set; }**
     6. **public SelectList Artists { get; set; }**
     7. **public SelectList Genres { get; set; }**
     8. }
     9. **Note:** You are using **System.Web.Mvc** **SelectList** for Artists and Genres instead of the **System.Collections.Generic** List.
     10. **SelectList** is a cleaner way to populate HTML dropdowns and manage things like current selection. Instantiating and later setting up these ViewModel objects in the controller action will make the Edit form scenario cleaner.

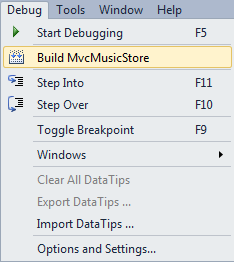
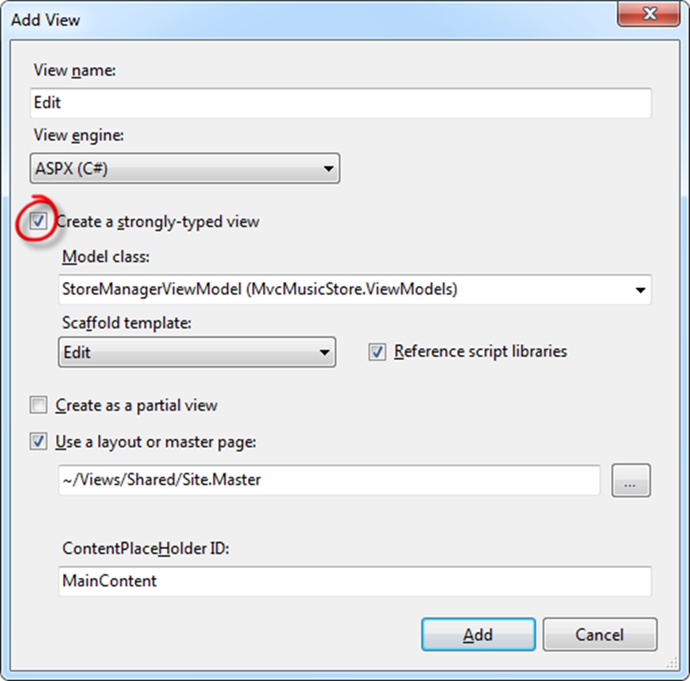
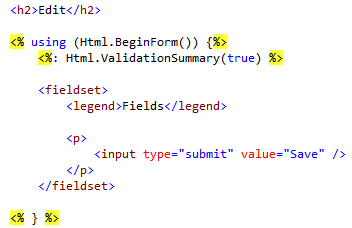
Task 2 – Implementing the HTTP-GET Edit Action Method

In this task, you will implement the HTTP-GET version of the Edit action method to retrieve the appropriate Album from the database, as well as a list of all Genres and Artists. It will package this data up into the StoreManagerViewModel object defined in the last step, which will then be passed to a View template to render the response with.

* 1. Open **StoreManagerController** class. To do this, expand the **Controllers** folder and double-click **StoreManagerController.cs**.
  2. Add a using directive to **MvcMusicStore.ViewModels**. In order for the code to compile correctly, add the following **using** directive:
     1. (Code Snippet – ASP.NET MVC 3 Helpers and Forms and Validation – Ex3 using MvcMusicStore.ViewModels – CSharp)
     2. C#
     3. using MvcMusicStore.Models;
     4. **using MvcMusicStore.ViewModels;**
  3. Replace the **HTTP-GET Edit** action method with the following code to retrieve the appropriate **Album** as well as the **Genres** and **Artists** lists.
     1. (Code Snippet – ASP.NET MVC 3 Helpers and Forms and Validation – Ex3 StoreManagerController HTTP-GET Edit action – CSharp)
     2. C#
     3. //
     4. // GET: /StoreManager/Edit/5
     6. public ActionResult Edit(int id)
     7. {
     8. **Album album = storeDB.Albums.Single(a => a.AlbumId == id);**
     9. **var viewModel = new StoreManagerViewModel()**
     10. **{**
     11. **Album = album,**
     12. **Genres = new SelectList(storeDB.Genres.ToList(), "GenreId", "Name", album.GenreId),**
     13. **Artists = new SelectList(storeDB.Artists.ToList(), "ArtistId", "Name", album.ArtistId)**
     14. **};**
     15. **return View(viewModel);**
     16. }

Task 3 – Creating the Edit View

In this task, you will create an Edit View template that will later display the album properties.

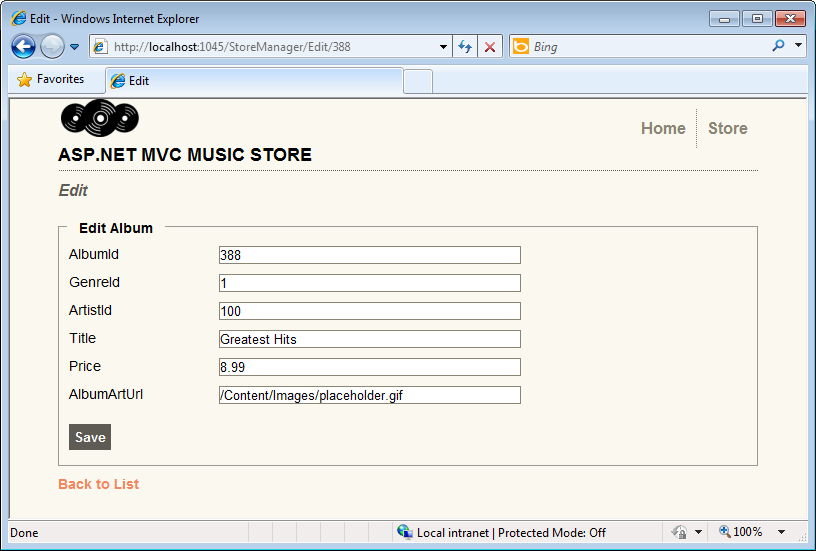
* 1. Before creating the new View template, you should build the project so that the **Add View Dialog** knows about the class to use. Build the project by selecting the **Debug** menu item and then **Build MvcMusicStore**.
     1. 
     2. Figure 12
     3. Building the project
  2. Create the Edit View. To do this, right-click inside the **Edit** action method and select **Add View**.
  3. In the Add View dialog, verify that the View Name is **Edit**. Check the **Create a strongly-typed view** checkbox and select **StoreManagerViewModel (MvcMusicStore.ViewModels)** from the **View data class** drop-down. Select **Edit** from the **View content** drop-down. Leave the other fields with their default value and then click **Add**.
     1. 
     2. Figure 13
     3. Adding an Edit View
  4. The generated Edit View template doesn’t include any fields because none of the properties in the StoreManagerViewModel are simple types like strings and integers.
     1. 
     2. Figure 14
     3. Edit View without fields

Task 4 – Customizing the Edit View

* 1. In this task, you will change the default View template to display the Album properties.
  2. Add an **Html.EditorFor()** helper to render a default HTML editing form for the Album.
     1. HTML
     2. **<legend>Edit Album</legend>**
     3. **<%: Html.EditorFor(model => model.Album,**
     4. **new { Artists = Model.Artists, Genres = Model.Genres })%>**
     6. <p>
     7. <input type="submit" value="Save" />
     8. </p>
     9. **Note:** **Html.EditorFor()** helper will create a form that will enable the edition of its first parameter, in this case an **Album**. Additionally, you are passing a second parameter which is optional, that contains the list of **Artists** and **Genres**. This will be used later on.

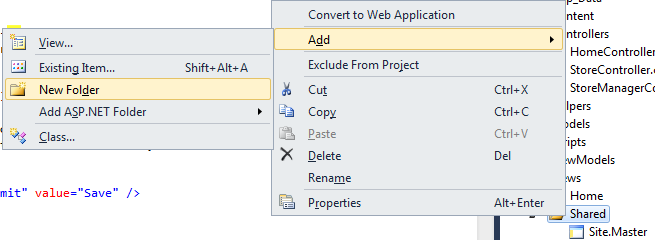
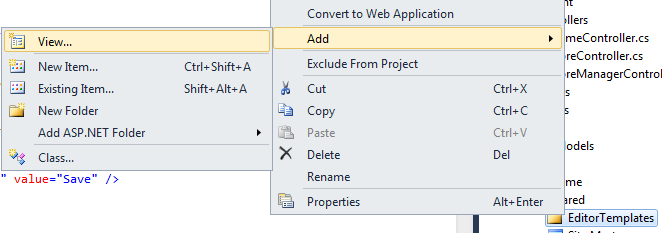
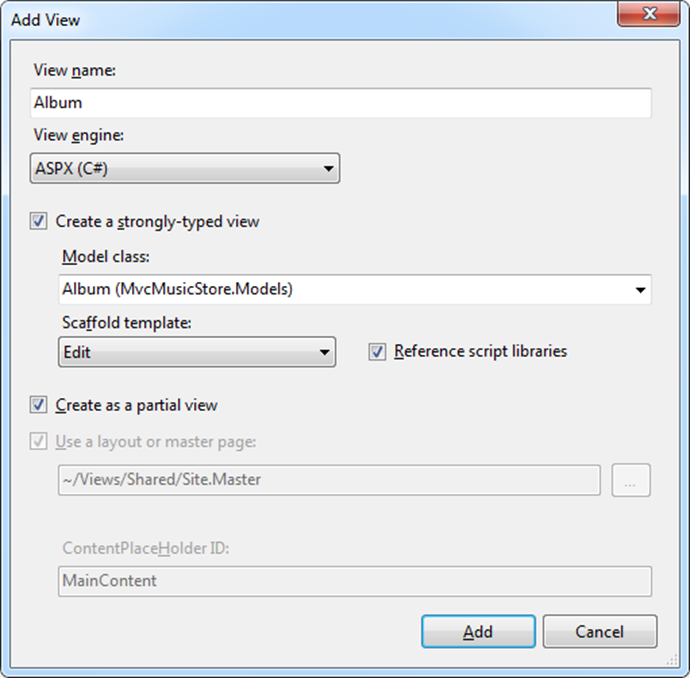
Task 5 – Running the Application

In this task, you will test that the **StoreManager** **Edit** View page displays the properties’ values for the album passed as parameter.

* 1. Press **F5** to run the Application.
  2. The project starts in the Home page. Change the URL to **/StoreManager/Edit/388** to verify that the properties’ values for the album passed are displayed.
     1. ****
     2. Figure 15
     3. Browsing Album’s Edit view

Task 6 – Creating a Shared Album Editor Template

Since the exact same form fields for Album Edit will be needed to handle the Album Create case, in this task you will create a Shared Album Editor template that will take care of both cases with the same code.

* 1. Close the browser if needed, to return to the Visual Studio window. Create a folder inside **/Views/Shared** folder and name it **EditorTemplates**.
     1. 
     2. Figure 16
     3. Adding a new folder
  2. Right-click the **EditorTemplates** folder, select **Add** and then **View**.
     1. 
     2. Figure 17
     3. Adding a View template
  3. This will be a Partial View, meaning that it is intended to be displayed inside another view. In the Add View dialog, change the Name to **Album**. Check the **Create a partial view (.ascx)** checkbox and the **Create a strongly-typed view** checkbox. Select **Album (MvcMusicStore. Models)** from the **View data class** drop-down and select **Edit** from the **View content** drop-down. Leave the other fields with their default value and then click **Add**.
     1. 
     2. Figure 18
     3. Adding a Partial View

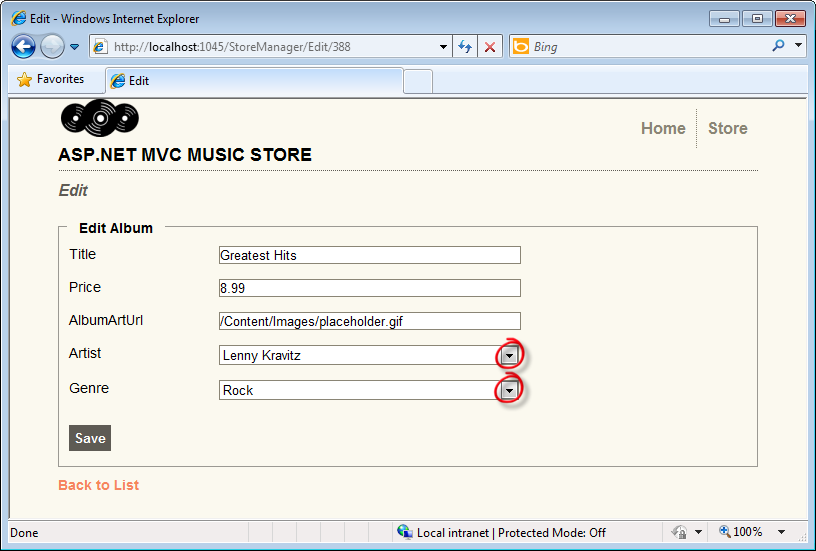
Task 7 – Implementing drop-downs on the Album Editor Template

In this task, you will add drop-downs to the View template created in the last task, so that the user can select from a list of Artists and Genres.

* 1. Replace **Album.ascx Partial View** code with the following:
     1. HTML
     2. **<%@ Control Language="C#" Inherits="System.Web.Mvc.ViewUserControl<MvcMusicStore.Models.Album>" %>**
     3. **<p>**
     4. **<%: Html.LabelFor(model => model.Title) %>**
     5. **<%: Html.TextBoxFor(model => model.Title) %>**
     6. **<%: Html.ValidationMessageFor(model => model.Title) %>**
     7. **</p>**
     8. **<p>**
     9. **<%: Html.LabelFor(model => model.Price) %>**
     10. **<%: Html.TextBoxFor(model => model.Price) %>**
     11. **<%: Html.ValidationMessageFor(model => model.Price) %>**
     12. **</p>**
     13. **<p>**
     14. **<%: Html.LabelFor(model => model.AlbumArtUrl) %>**
     15. **<%: Html.TextBoxFor(model => model.AlbumArtUrl) %>**
     16. **<%: Html.ValidationMessageFor(model => model.AlbumArtUrl) %>**
     17. **</p>**
     18. **<p>**
     19. **<%: Html.LabelFor(model => model.Artist) %>**
     20. **<%: Html.DropDownList("ArtistId", (SelectList) ViewData["Artists"]) %>**
     21. **</p>**
     22. **<p>**
     23. **<%: Html.LabelFor(model => model.Genre) %>**
     24. **<%: Html.DropDownList("GenreId", (SelectList) ViewData["Genres"]) %>**
     25. **</p>**
     26. **Note:** An **Html.DropDownList** helper has been added to render drop-downs for choosing Artists and Genres. The parameters passed to **Html.DropDownList** are:
     27. 1. The name of the form field (**“ArtistId”**)
     28. 2. The **SelectList** of values for the drop-down. **ViewData** is actually the object taken from the **Html.EditorFor** second parameter you included in task 4:
     29. **<%: Html.EditorFor(model => model.Album,**
     30. **new { Artists = Model.Artists, Genres = Model.Genres })%>**
     31. **Html.EditorFor()** helper will create a form that will enable the edition of its first parameter, in this case an **Album**. Additionally, you are passing a second parameter which is optional, that contains the list of **Artists** and **Genres**. This will be used later on.

Task 8 – Running the Application

In this task, you will test that the **StoreManager** **Edit** View page displays drop-downs instead of Artist and Genre ID text fields.

* 1. Press **F5** to run the Application.
  2. The project starts in the Home page. Change the URL to **/StoreManager/Edit/388** to verify that it displays drop-downs instead of Artist and Genre ID text fields.
     1. ****
     2. Figure 19
     3. Browsing Album’s Edit view, this time with dropdowns

Task 9 – Implementing the HTTP-POST Edit action method

* 1. Now that the Edit View displays as expected, you need to implement the HTTP-POST Edit Action method to save the changes made to the Album.
  2. Close the browser if needed, to return to the Visual Studio window. Open **StoreManagerController** from the **Controllers** folder.
  3. Replace **HTTP-POST Edit** action method code with the following (note that the method that must be replaced is overloaded version that receives two parameters):
     1. (Code Snippet – ASP.NET MVC 3 Helpers and Forms and Validation – Ex3 StoreManagerController HTTP-POST Edit action – CSharp)
     2. C#
     3. //
     4. // POST: /StoreManager/Edit/5
     5. [HttpPost]
     6. public ActionResult Edit(int id, FormCollection collection)
     7. {
     8. **var album = storeDB.Albums.Single(a => a.AlbumId == id);**
     9. **try**
     10. **{**
     11. **// Save Album**
     12. **UpdateModel(album, "Album");**
     13. **storeDB.SaveChanges();**
     15. **return RedirectToAction("Index");**
     16. **}**
     17. **catch**
     18. **{**
     19. **// Error occurred - so redisplay the form**
     20. **var viewModel = new StoreManagerViewModel()**
     21. **{**
     22. **Album = album,**
     23. **Genres = new SelectList(storeDB.Genres.ToList(), "GenreId", "Name", album.GenreId),**
     24. **Artists = new SelectList(storeDB.Artists.ToList(), "ArtistId", "Name", album.ArtistId)**
     25. **};**
     26. **return View(viewModel);**
     27. **}**

}

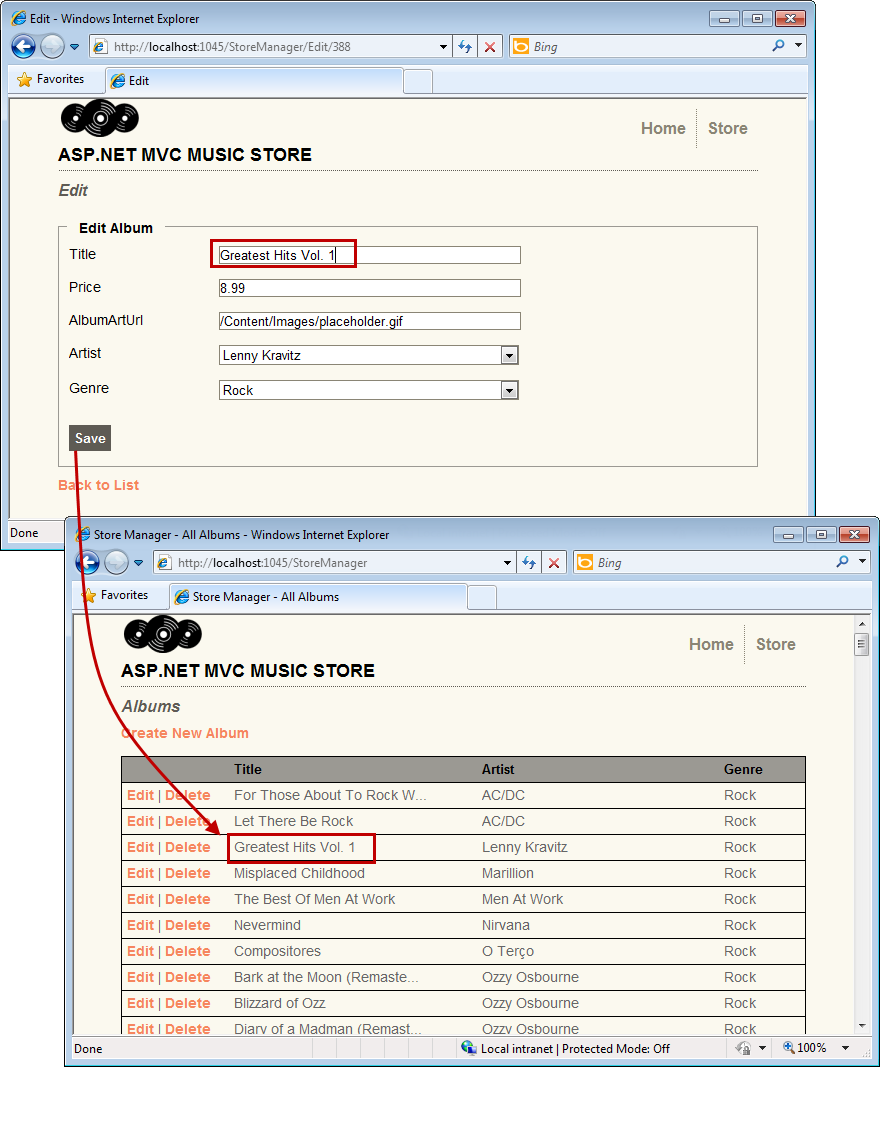
* + 1. **Note:** This method will be executed when the user clicks the **Save** button of the View and performs an HTTP-POST of the form values back to the server to persist them in the database. The decorator **[HttpPost]** indicates that the method should be used for those HTTP-POST scenarios.
    2. The method takes an **id** (read from the route parameter values) and a **FormCollection** (read from the HTML Form).
    3. The method will perform three steps:
    4. 1. Load the existing album object from the database with the **id** passed as parameter

2. Try to update the album using the values posted from the client, using the Controller’s built-in **UpdateModel** method. The **UpdateModel** method actually relies on the **ModelBinder** (MVC 2.0) to identify that you are actually updating a **StoreManagerViewModel** instance. Since that **ViewModel** holds an **Album** object and also 2 **SelectList**s for **Artists** and **Genres**, the key **“Album”** you are including as **UpdateModel**’s second parameter, refers to the need of updating the **“Album”** property of the Model.

* + 1. 3. Display results back to the user – either by redisplaying the form in case of an error, or by redirecting back to the list of albums in case of a successful update

Task 10 – Running the Application

In this task, you will test that the **StoreManager** **Edit** View page actually saves the updated Album data in the database.

* 1. Press **F5** to run the Application.
  2. The project starts in the Home page. Change the URL to **/StoreManager/Edit/388**. Change the Album title to **Greatest Hits Vol. 1** and click on **Save**. Verify that album’s title actually changed in the list of albums.
     1. 
     2. Figure 20
     3. Updating an Album

# Next Step

* 1. Exercise 4: Adding a Create View

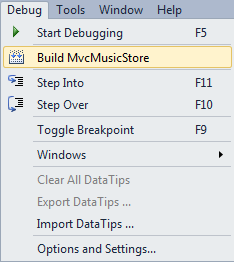
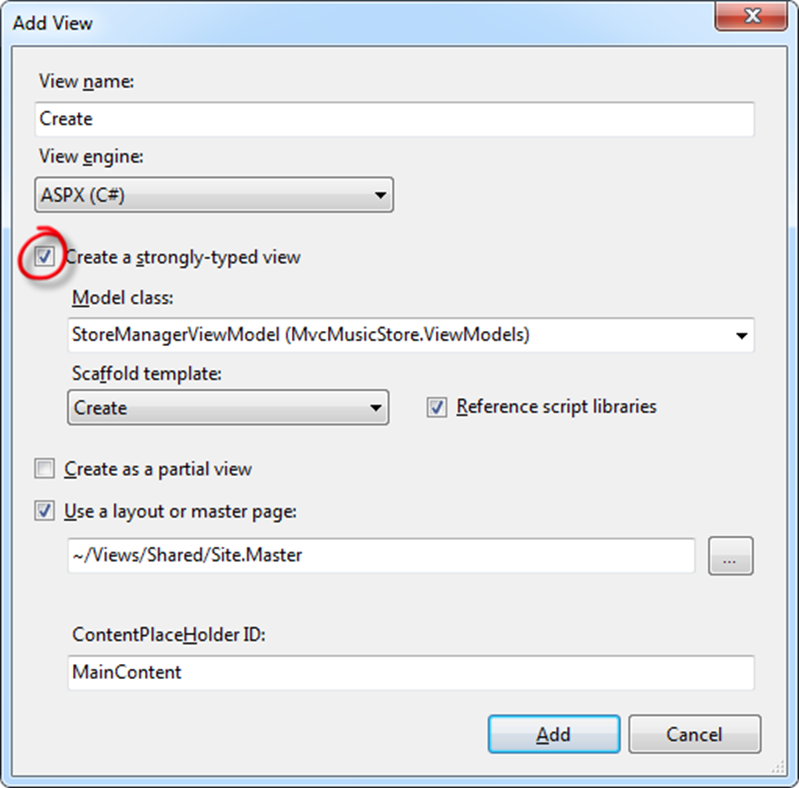
Exercise 4: Adding a Create View

* 1. Now that the **StoreManagerController** supports the **Edit** ability, in this exercise you will learn how to add a Create View template to let store managers to add new Albums to the application.
  2. Like you did with the Edit functionality, you will implement the Create scenario using two separate methods within the **StoreManagerController** class:
  3. One action method will display an empty form when store managers first visit the **/StoreManager/Create** URL.
  4. A second action method will handle the scenario where the store manager clicks the **Save** button within the form and submits the values back to the **/StoreManager/Create** URL as an HTTP-POST.

Task 1 – Implementing the HTTP-GET Create action method

* 1. In this task, you will implement the HTTP-GET version of the Create action method to retrieve a list of all Genres and Artists, package this data up into a StoreManagerViewModel object, which will then be passed to a View 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. In the **File** menu, choose **Open Project**. In the Open Project dialog, browse to Source\Ex04-AddingACreateView\Begin, select **MvcMusicStore.sln** and click **Open**.
  4. Open **StoreManagerController** class. To do this, expand the **Controllers** folder and double-click **StoreManagerController.cs**.
  5. Replace the **Create** action method code with the following:
     1. (Code Snippet – ASP.NET MVC 3 Helpers and Forms and Validation – Ex4 StoreManagerController HTTP-GET Create action – CSharp)
     2. C#
     3. //
     4. // GET: /StoreManager/Create
     5. public ActionResult Create()
     6. {
     7. **var viewModel = new StoreManagerViewModel()**
     8. **{**
     9. **Album = new Album(),**
     10. **Genres = new SelectList(storeDB.Genres.ToList(), "GenreId", "Name"),**
     11. **Artists = new SelectList(storeDB.Artists.ToList(), "ArtistId", "Name")**
     12. **};**
     14. **return View(viewModel);**
     15. }

Task 2 – Adding the Create View

* 1. In this task, you will add the Create View template that will display a new (empty) Album form.
  2. Before creating the new View template, you should build the project so that the **Add View Dialog** knows about the class to use. Build the project by selecting the **Debug** menu item and then **Build MvcMusicStore**.
     1. 
     2. Figure 21
     3. Building the project
  3. Right-click inside the **Create** action method and select **Add View**. This will bring up the Add View dialog.
  4. In the Add View dialog, verify that the View Name is **Create**. Check the **Create a strongly-typed view** checkbox and select **StoreManagerViewModel (MvcMusicStore.ViewModels)** from the **View data class** drop-down. Select **Create** from the **View content** drop-down. Leave the other fields with their default value and then click **Add**.
     1. 
     2. Figure 22
     3. Adding an Edit View
  5. Update the generated View template to invoke the **Html.EditorFor()** helper method:
     1. HTML
     2. <% using (Html.BeginForm()) {%>
     3. <%: Html.ValidationSummary(true) %>
     4. <fieldset>
     5. **<legend>Create Album</legend>**
     6. **<%: Html.EditorFor(model => model.Album,**
     7. **new { Artists = Model.Artists, Genres = Model.Genres })%>**
     8. **<p>**
     9. **<input type="submit" value="Save" />**
     10. **</p>**
     11. </fieldset>
     12. <% } %>

Task 3 – Running the Application

In this task, you will test that the **StoreManager** **Create** View page displays an empty Album form.

* 1. Press **F5** to run the Application.
  2. The project starts in the Home page. Change the URL to **/StoreManager/Create**. Verify that an empty form is displayed for filling the new Album properties.
     1. ****
     2. Figure 23
     3. Create View with an empty form

Task 4 – Implementing the HTTP-POST Create Action Method

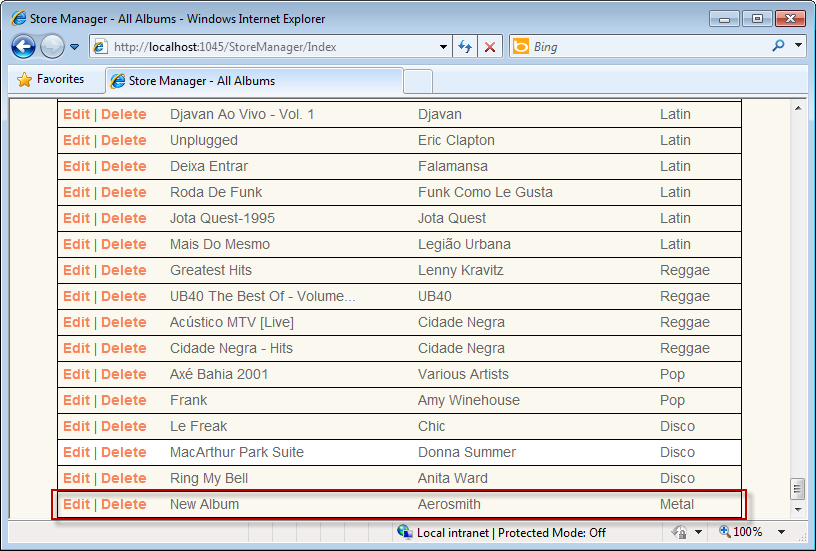
* 1. In this task, you will implement the HTTP-POST version of the Create action method that will be invoked when a user clicks the **Save** button. The method should save the new album in the database.
  2. Close the browser if needed, to return to the Visual Studio window. Open **StoreManagerController** class. To do this, expand the **Controllers** folder and double-click **StoreManagerController.cs**.
  3. Replace **HTTP-POST Create** action method code with the following:
     1. (Code Snippet – ASP.NET MVC 3 Helpers and Forms and Validation – Ex4 StoreManagerController HTTP- POST Create action – CSharp)
     2. C#
     3. //
     4. // POST: /StoreManager/Create
     5. [HttpPost]
     6. **public ActionResult Create(Album album)**
     7. **{**
     8. **try**
     9. **{**
     10. **if (ModelState.IsValid)**
     11. **{**
     12. **// Save Album**
     13. **storeDB.AddToAlbums(album);**
     14. **storeDB.SaveChanges();**
     15. **return Redirect("Index");**
     16. **}**
     17. **}**
     18. **catch (Exception ex)**
     19. **{**
     20. **ModelState.AddModelError(String.Empty, ex);**
     21. **}**
     22. **// Invalid - redisplay with errors**
     23. **var viewModel = new StoreManagerViewModel()**
     24. **{**
     25. **Album = album,**
     26. **Genres = new SelectList(storeDB.Genres.ToList(), "GenreId", "Name", album.GenreId),**
     27. **Artists = new SelectList(storeDB.Artists.ToList(), "ArtistId", "Name", album.ArtistId)**
     28. **};**
     29. **return View(viewModel);**
     30. **}**

**Note:** One difference from the previous Edit action method is that instead of loading an existing Album and calling UpdateModel, the action method is receiving an Album as a parameter. ASP.NET MVC will automatically create this Album object from the posted <form> values.

The method checks that the submitted values are valid, and if they are, it saves the new Album in the database and then redirects to the StoreManager Index View.

Task 5 – Running the Application

In this task, you will test that the **StoreManager** **Create** View page lets you create a new Album and then redirects to the StoreManager Index View.

* 1. Press **F5** to run the Application.
  2. The project starts in the Home page. Change the URL to **/StoreManager/Create**. Fill the empty form with data for a new Album, like the one in the following figure:
     1. ****
     2. Figure 24
     3. Creating an Album
  3. Verify that you get redirected to the StoreManager Index View that includes the new Album just created.
     1. ****
     2. Figure 25
     3. New Album created

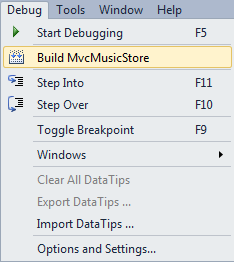
# Next Step

* 1. Exercise 5: Handling Deletion

Exercise 5: Handling Deletion

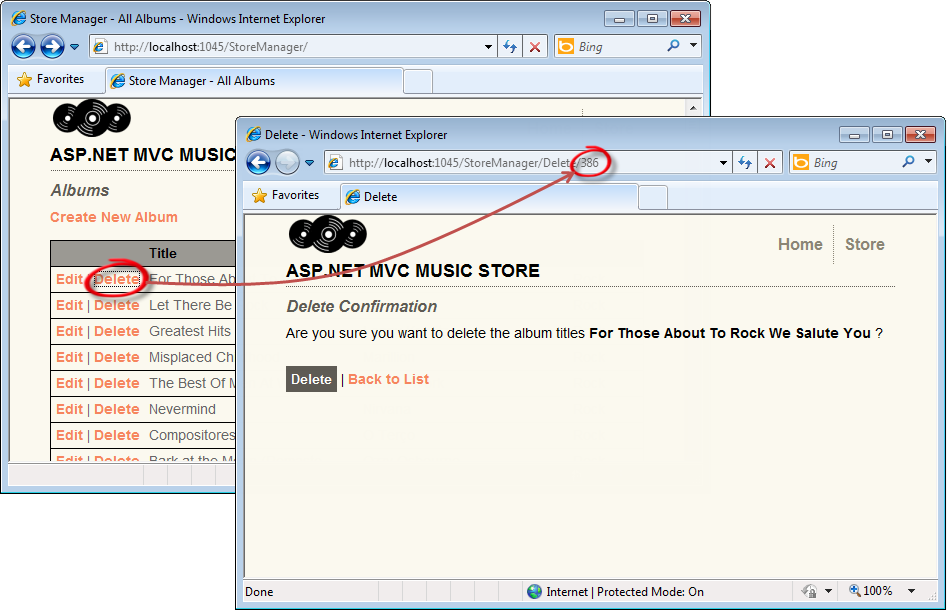
* 1. The ability to delete albums is not yet implemented. This is what this exercise will be about. Like before, you will implement the Delete scenario using two separate methods within the **StoreManagerController** class:
  2. One action method will display a confirmation form
  3. A second action method will handle the form submission

Task 1 – Implementing the HTTP-GET Delete Action Method

* 1. In this task, you will implement the HTTP-GET version of the Delete action method to retrieve the album’s information.
  2. 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-HandlingDeletion\Begin, select **MvcMusicStore.sln** and click **Open**.
  4. Open **StoreManagerController** class. To do this, expand the **Controllers** folder and double-click **StoreManagerController.cs**.
  5. The Delete controller action is exactly the same as the previous Store Details controller action: it queries the **album** object from the database using the **id** provided in the URL and returns the appropiate **View**. To do this, replace the HTTP-GET **Delete** action method code with the following:
     1. (Code Snippet – ASP.NET MVC 3 Helpers and Forms and Validation – Ex5 Handling Deletion HTTP-GET Delete action – CSharp)
     2. C#
     3. //
     4. // GET: /StoreManager/Delete/5
     5. **public ActionResult Delete(int id)**
     6. **{**
     7. **var album = storeDB.Albums.Single(a => a.AlbumId == id);**
     8. **return View(album);**
     9. **}**
  6. Before creating the new View template, you should build the project so that the **Add View Dialog** knows about the class to use. Build the project by selecting the **Debug** menu item and then **Build MvcMusicStore**.
     1. 
     2. Figure 26
     3. Building the project
  7. Right-click inside the **Delete** action method and select **Add View**. This will bring up the Add View dialog.
  8. In the Add View dialog, verify that the View name is **Delete**. Check the **Create a strongly-typed view** checkbox and select **Album (MvcMusicStore.Models)** from the **View data class** drop-down. Select **Delete** from the **View content** drop-down. Leave the other fields with their default value and then click **Add**.
     1. 
     2. Figure 27
     3. Adding a Delete View
  9. The Delete template shows all the fields from the model. You will show only the album’s title. To do this, replace the **asp:Content** with the following code:
     1. HTML
     2. <asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
     3. **<h2>Delete Confirmation</h2>**
     4. **<p>**
     5. **Are you sure you want to delete the album titles**
     6. **<strong><%: Model.Title %></strong> ?**
     7. **</p>**
     8. **<div>**
     9. **<% using (Html.BeginForm()) { %>**
     10. **<input type="submit" value="Delete" /> |**
     11. **<%: Html.ActionLink("Back to List", "Index") %>**
     12. **<% } %>**
     13. **</div>**
     14. </asp:Content>

Task 2 – Running the Application

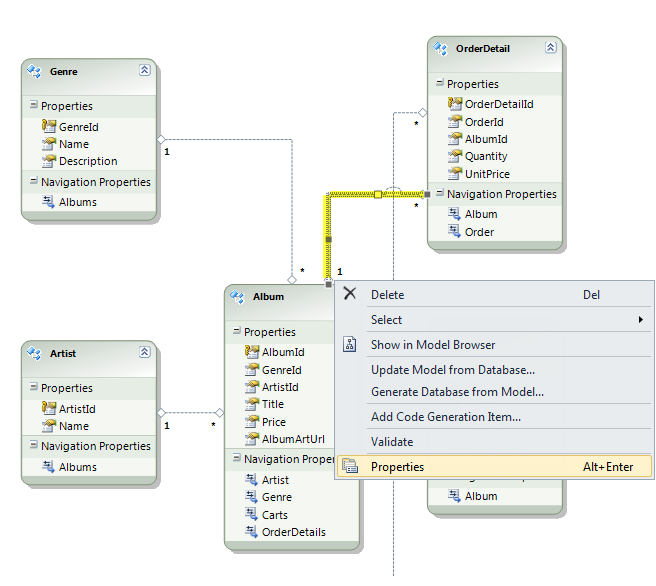
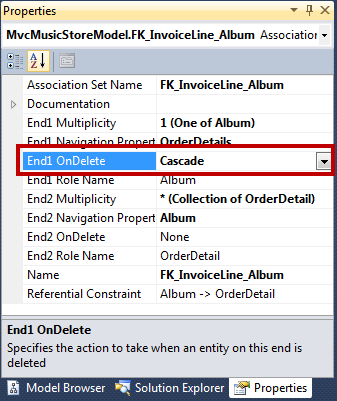
In this task, you will test that the **StoreManager** **Delete** View page displays a confirmation deletion form.

* 1. Press **F5** to run the Application.
  2. The project starts in the Home page. Change the URL to **/StoreManager**. Select one album to delete by clicking **Delete** and verify that the new view is uploaded.
     1. 
     2. Figure 28
     3. Deleting an Album

Task 3– Implementing the HTTP-POST Delete Action Method

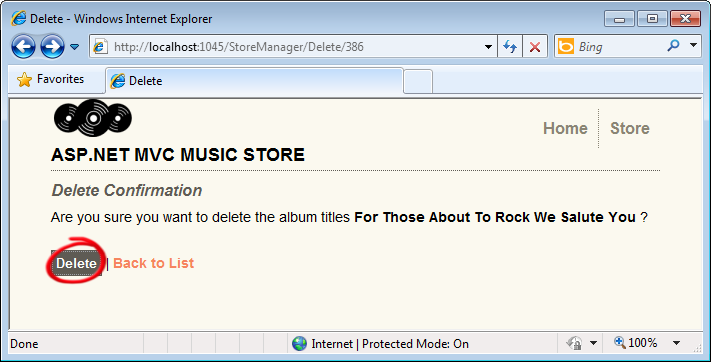
* 1. In this task, you will implement the HTTP-POST version of the Delete action method that will be invoked when a user clicks the **Delete** button. The method should delete the album in the database.
  2. Close the browser if needed, to return to the Visual Studio window. Open **StoreManagerController** class. To do this, expand the **Controllers** folder and double-click **StoreManagerController.cs**.
  3. Replace **HTTP-POST Delete** action method code with the following:
     1. (Code Snippet – ASP.NET MVC 3 Helpers and Forms and Validation – Handling Deletion HTTP-POST Delete action – CSharp)
     2. C#
     3. //
     4. // POST: /StoreManager/Delete/5
     5. [HttpPost]
     6. public ActionResult Delete(int id, FormCollection collection)
     7. {
     8. **var album = storeDB.Albums**
     9. **.Include("OrderDetails").Include("Carts")**
     10. **.Single(a => a.AlbumId == id);**
     11. **storeDB.DeleteObject(album);**
     12. **storeDB.SaveChanges();**
     13. **return RedirectToAction("Index");**
     14. }

Task 4 – Deleting on Cascade

* 1. Because of Referential Integrity, a deletion of an **Album** could raise an exception if it has **OrderDetails** entries. To solve this, you should allow cascade deletes. So, when you delete an Album, it will delete all the OrderDetails entries for that album. In this task, you will activate deletion on cascade.
  2. **Note:** In this scenario you will delete an Album no matter if it has an order associated. Consider that in other application scenarios this could not be the correct action.
  3. On the Solution Explorer expand the **Models** folder and then double-click **StoreDB.edmx**. This opens the Entity Data Model designer.
  4. Open the **Models/StoreDB.edmx** entity diagram. Right-click the relation between **Album** and **OrderDetail** and select **Properties**.
     1. 
     2. Figure 29
     3. Editing Properties of a Relation
  5. In the **Properties** window, set the **End1 OnDelete** value to **Cascade**.
     1. 
     2. Figure 30
     3. OnDelete property

Task 5 – Running the Application

In this task, you will test that the **StoreManager** **Delete** View page lets you delete an Album and then redirects to the StoreManager Index View.

* 1. Press **F5** to run the Application.
  2. The project starts in the Home page. Change the URL to **/StoreManager**. Select one album to delete by clicking **Delete.** Confirm the deletion by clicking **Delete** button:
     1. 
     2. Figure 31
     3. Deleting an Album
  3. Verify that the album was deleted since it does not appear in the **Index.aspx** page.

# Next Step

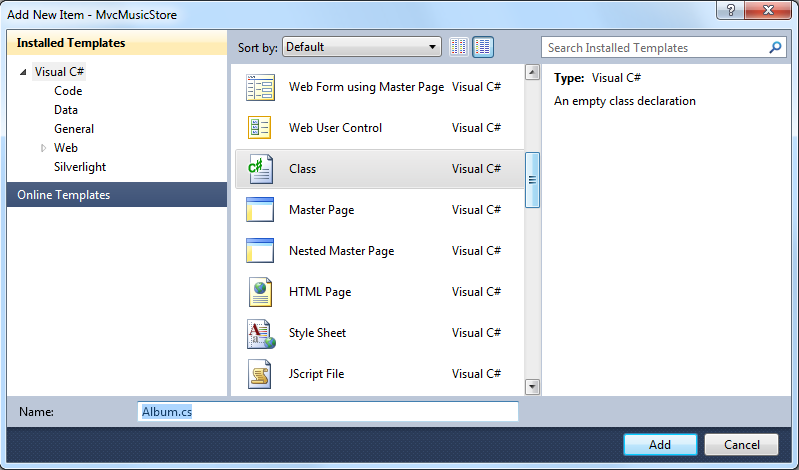
* 1. Exercise 6: Adding Validation

Exercise 6: Adding Validation

* 1. Currently, the Create and Edit forms you have in place do not perform any kind of validation. If the user leaves a required field blank or type letters in the price field, the first error you will get will be from the database.
  2. You can add validation to the application by adding Data Annotations to your model class. Data Annotations allow describing the rules you want applied to your model properties, and ASP.NET MVC will take care of enforcing and displaying appropriate message to users.

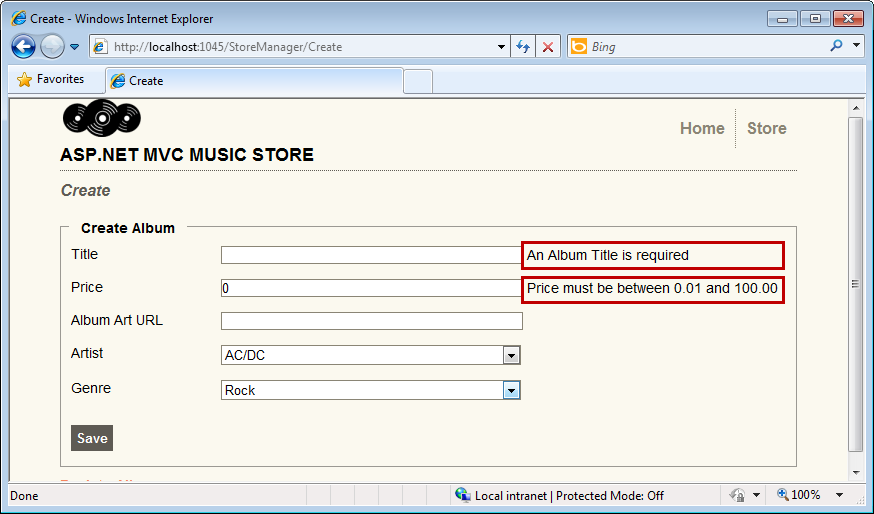
Additionally, you will add client-side (AJAX) validation that will check all fields in the browser before being submitted to the controller actions.

Task 1 – Adding Data Annotations

* 1. In this task, you will add Data Annotations to the Album Model that will make the Create and Edit page display validation messages when appropriate.
  2. 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-AddingValidation\Begin**, select **MvcMusicStore.sln** and click **Open**.
     1. **Note:** For a simple Model class, adding a Data Annotation is just handled by adding a **using** statement for **System.ComponentModel.DataAnnotation**, then placing a **[Required]** attribute on the appropriate properties.
     2. The following example would make the **Name** property a required field in the View.
     3. **using System.ComponentModel.DataAnnotations;**
     4. **namespace SuperheroSample.Models**
     5. **{**
     6. **public class Superhero**
     7. **{**
     8. **[Required]**
     9. **public string Name { get; set; }**
     10. **public bool WearsCape { get; set; }**
     11. **}**
     12. **}**
     13. This is a little more complex in cases like this application where the Entity Data Model is generated. If you added Data Annotations directly to the model classes, they would be overwritten if you update the model from the database.
     14. Instead, you can make use of metadata partial classes which will exist to hold the annotations and are associated with the model classes using the **[MetadataType]** attribute.
  4. Add a new metadata partial class. To do this, right-click the **Models** folder within the Solution Explorer, select **Add** and then **New Item**.
  5. In the **Add New Item** dialog, select the **Class** template, located under **Visual C# -> Web** template list. Change the name to **Album.cs** and click **Add**.
     1. 
     2. Figure 32
     3. Adding Album.cs
  6. Replace **Album.cs** content with the highlighted code, so that it looks like the following:
     1. (Code Snippet – ASP.NET MVC 3 Helpers and Forms and Validation – Ex6 Album metadata partial class – CSharp)
     2. C#
     3. using System;
     4. using System.Collections.Generic;
     5. using System.Linq;
     6. using System.Web;
     7. **using System.ComponentModel;**
     8. **using System.ComponentModel.DataAnnotations;**
     9. **using System.Web.Mvc;**
     10. **namespace MvcMusicStore.Models**
     11. **{**
     12. **[MetadataType(typeof(AlbumMetaData))]**
     13. **public partial class Album**
     14. **{**
     15. **// Validation rules for the Album class**
     16. **[Bind(Exclude = "AlbumId")]**
     17. **public class AlbumMetaData**
     18. **{**
     19. **[ScaffoldColumn(false)]**
     20. **public object AlbumId { get; set; }**
     21. **[DisplayName("Genre")]**
     22. **public object GenreId { get; set; }**
     23. **[DisplayName("Artist")]**
     24. **public object ArtistId { get; set; }**
     25. **[Required(ErrorMessage = "An Album Title is required")]**
     26. **[DisplayFormat(ConvertEmptyStringToNull = false)]**
     27. **[StringLength(160)]**
     28. **public object Title { get; set; }**
     29. **[DisplayName("Album Art URL")]**
     30. **[StringLength(1024)]**
     31. **public object AlbumArtUrl { get; set; }**
     32. **[Required(ErrorMessage = "Price is required")]**
     33. **[Range(0.01, 100.00, ErrorMessage = "Price must be between 0.01 and 100.00")]**
     34. **public object Price { get; set; }**
     35. **}**
     36. **}**
     37. **}**
     38. **Note:** This **Album** partial class has a **MetadataType** attribute which points to the **AlbumMetaData** class for the Data Annotations. These are some of the Data Annotation attributes you are using to annotate the Album model:
     39. • Required – Indicates that the property is a required field
     40. • DisplayName – Defines the text to be used on form fields and validation messages
     41. • DisplayFormat – Specifies how data fields are displayed and formatted.
     42. • StringLength – Defines a maximum length for a string field
     43. • Range – Gives a maximum and minimum value for a numeric field
     44. • Bind – Lists fields to exclude or include when binding parameter or form values to model properties
     45. • ScaffoldColumn – Allows hiding fields from editor forms
     46. **Note:** The line **[DisplayFormat(ConvertEmptyStringToNull=false)]** indicates that empty strings from the model won’t be converted to null when the data field is updated in the data source.   
         This setting will avoid an exception when the Entity Framework assigns null values to the model before Data Annotation validates the fields.

Task 2 – Running the Application

In this task, you will test that the Create and Edit pages validate fields, using the display names chosen in the last task.

* 1. Press **F5** to run the Application.
  2. The project starts in the Home page. Change the URL to **/StoreManager/Create**. Verify that the display names match the ones in the partial class (like **Album Art URL** instead of **AlbumArtUrl**)
  3. Click **Save**, without filling the form. Verify that you get the corresponding validation messages.
     1. ****
     2. Figure 33
     3. Validated fields in Create page
  4. You can verify that the same occurs with the **Edit** page. Change the URL to **/StoreManager/Edit/388** and verify that the display names match the ones in the partial class (like **Album Art URL** instead of **AlbumArtUrl**). Empty the **Title** and **Price** fields and click **Save**. Verify that you get the corresponding validation messages.
     1. 
     2. Figure 34
     3. Validated fields in Edit page

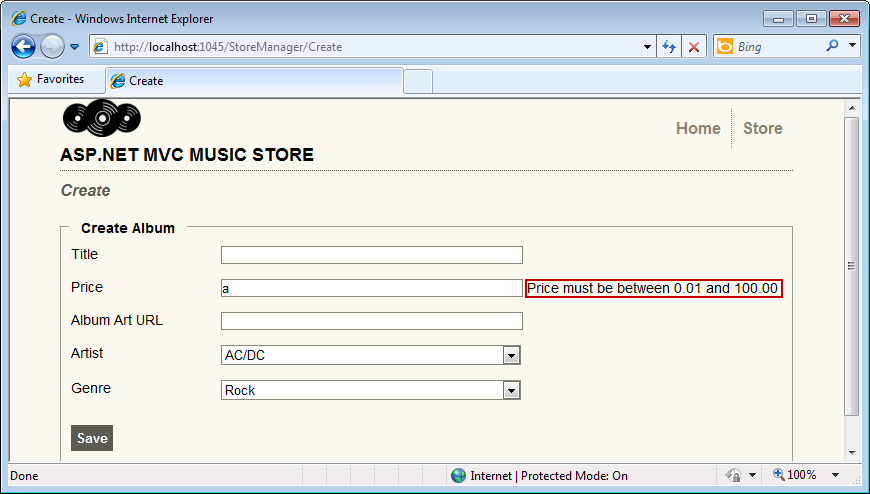
Task 3 – Adding Client-Side (AJAX) validation

In this task, you will add client-side (AJAX) validation to the forms to check all fields in the browser before being submitted to the controller actions.

* 1. Close the browser if needed, to return to the Visual Studio window. Open **/Views/Shared/EditorTemplates/Album.ascx**
  2. Include the needed references to the libraries that will take care of the client-side validation:
     1. HTML
     2. <%@ Control Language="C#" Inherits="System.Web.Mvc.ViewUserControl<MvcMusicStore.Models.Album>" %>
     3. **<script src="/Scripts/MicrosoftAjax.js" type="text/javascript"></script>**
     4. **<script src="/Scripts/MicrosoftMvcAjax.js" type="text/javascript"></script>**
     5. **<script src="/Scripts/MicrosoftMvcValidation.js" type="text/javascript"></script>**
  3. Use the **Html.EnableClientValidation** helper method to turn on client-side validation. For both the **Create** and **Edit** view templates, add that call directly above the **Html.BeginForm** call. In order to do that, open **/Views/StoreManager/Create.aspx** and **/Views/StoreManager/Edit.aspx** and add the following:
     1. (Code Snippet – ASP.NET MVC 3 Helpers and Forms and Validation – Ex6 Client-Side validation – CSharp)
     2. C#
     3. **<% Html.EnableClientValidation(); %>**
     4. <% using (Html.BeginForm()) {%>
     5. <%: Html.ValidationSummary(true) %>
     6. <fieldset>
  4. Edit the **Web.config** in the project root to disable UnobtrusiveJavaScript. To do this, replace the appSettings entry with the following one:
     1. XML
     2. <configuration>
     3. <appSettings>
     4. <add key="ClientValidationEnabled" value="true"/>
     5. <add key="UnobtrusiveJavaScriptEnabled" value="**false**"/>
     6. </appSettings>

Task 4 – Running the Application

In this task, you will test that the Create and Edit pages validate fields in the browser before being submitted to the controller actions.

* 1. Press **F5** to run the Application.
  2. The project starts in the Home page. Change the URL to **/StoreManager/Create**. Enter a letter in the **Price** field and click anywhere else in the form, taking focus out of the text box. Verify that without submitting the form, the corresponding validation message appears.
     1. 
     2. Figure 35
     3. Validated fields at client-side
  3. Delete the letter entered in the **Price** field and verify that the appropriate validation message appears without submitting the form.
     1. 
     2. Figure 36
     3. Validated fields at client-side

# Next Step

* 1. Summary

Summary

* 1. By completing this Hands-On Lab you have learned how to enable users to change the data stored in the database with the use of the following:
  + Controller actions like Index, Create, Edit, Delete
  + ASP.NET MVC’s scaffolding feature for displaying properties in an HTML table
  + Custom HTML helpers to improve user experience
  + Action methods that react to either HTTP-GET or HTTP-POST calls
  + A shared editor template for similar View templates like Create and Edit
  + Form elements like drop-downs
  + Data annotations for Model validation
  + Client-side AJAX validation