### Exercise 5: Creating a View Model

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.

In this exercise, you will learn how to create a ViewModel class and add the required properties: the number of genres in the store and a list of those genres. You will also 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**

In this task, you will create a ViewModel class that will implement the Store genre listing scenario.

1. If not already open, start **VS Express for Web**.
2. In the **File** menu, choose **Open Project**. In the Open Project dialog, browse to **Source\Ex05-CreatingAViewModel\Begin**, select**Begin.sln** and click **Open**. Alternatively, you may continue with the solution that you obtained after completing the previous exercise.
   1. If you opened the provided **Begin** solution, you will need to download some missing NuGet packages before continue. To do this, click the **Project** menu and select **Manage NuGet Packages**.
   2. In the **Manage NuGet Packages** dialog, click **Restore** in order to download missing packages.
   3. Finally, build the solution by clicking **Build** | **Build Solution**.

**Note:** One of the advantages of using NuGet is that you don't have to ship all the libraries in your project, reducing the project size. With NuGet Power Tools, by specifying the package versions in the Packages.config file, you will be able to download all the required libraries the first time you run the project. This is why you will have to run these steps after you open an existing solution from this lab.

1. Create a **ViewModels** folder to hold the ViewModel. To do this, right-click the top-level **MvcMusicStore** project, select **Add** and then**New Folder**.

Adding a new folder

1. Name the folder ViewModels.

ViewModels folder in Solution Explorer

1. Create a **ViewModel** class. To do this, right-click on the **ViewModels** folder recently created, select **Add** and then **New Item**. Under**Code**, choose the **Class** item and name the file StoreIndexViewModel.cs, then click **Add**.

Adding a new Class

Creating StoreIndexViewModel class

#### **Task 2 - Adding Properties to the ViewModel class**

There are two parameters to be passed from the StoreController to the View template in order to generate the expected HTML response: the number of genres in the store and a list of those genres.

In this task, you will add those 2 properties to the **StoreIndexViewModel** class: **NumberOfGenres** (an integer) and **Genres** (a list of strings).

1. Add **NumberOfGenres** and **Genres** properties to the **StoreIndexViewModel** class. To do this, add the following 2 lines to the class definition:

(Code Snippet - ASP.NET MVC 4 Fundamentals - Ex5 StoreIndexViewModel properties)

C#

public class StoreIndexViewModel

{

**public int NumberOfGenres { get; set; }**

**public List<string> Genres { get; set; }**

}

**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**

The **StoreIndexViewModel** class encapsulates the information needed to pass from **StoreController**'s **Index** method to a View template in order to generate a response.

In this task, you will update the **StoreController** to use the **StoreIndexViewModel**.

1. Open **StoreController** class.

Opening StoreController class

1. In order to use the **StoreIndexViewModel** class from the **StoreController**, add the following namespace at the top of the**StoreController** code:

(Code Snippet - ASP.NET MVC 4 Fundamentals - Ex5 StoreIndexViewModel using ViewModels)

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.

**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**.

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.

1. Replace the **Index** method with the following code:

(Code Snippet - ASP.NET MVC 4 Fundamentals - Ex5 StoreController Index method)

C#

**public ActionResult Index()**

**{**

**// Create a list of genres**

**var genres = new List<string> {"Rock", "Jazz", "Country", "Pop", "Disco"};**

**// Create our view model**

**var viewModel = new StoreIndexViewModel {**

**NumberOfGenres = genres.Count(),**

**Genres = genres**

**};**

**return this.View(viewModel);**

**}**

**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**

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.

1. 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 **Build** menu item and then **Build MvcMusicStore**.

Building the project

1. Create a new View template. To do that, right-click inside the **Index** method and select **Add View**.

Adding a View

1. Because the **Add View Dialog** was invoked from the **StoreController**, it will add the View template by default in a**\Views\Store\Index.cshtml** file. Check the **Create a strongly-typed-view** checkbox and then select **StoreIndexViewModel** as the**Model class**. Also, make sure that the View engine selected is **Razor**. Click **Add**.

Add View Dialog

The **\Views\Store\Index.cshtml** 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 the template inherits a **ViewPage** in C#.

#### **Task 5 - Updating the View Template**

In this task, you will update the View template created in the last task to retrieve the number of genres and their names within the page.

**Note:** You will use @ syntax (often referred to as "code nuggets") to execute code within the View template.

1. In the **Index.cshtml** file, within the **Store** folder, replace its code with the following:

HTML

**@model MvcMusicStore.ViewModels.StoreIndexViewModel**

**@{**

**ViewBag.Title = "Browse Genres";**

**}**

**<h2>Browse Genres</h2>**

**<p>Select from @Model.NumberOfGenres genres</p>**

**Note:** As soon as you finish typing the period after the word **Model**, Visual Studio's Intellisense will show a list of possible properties and methods to choose from.

Getting Model properties and methods with Visual Studio's IntelliSense

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 from the Controller to the View template via the **Model** property, and format it into an appropriate HTML response within the View template.

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 **StoreIndexViewModel** and create an HTML **<ul>** list using a **foreach** loop.

HTML(C#)

@model MvcMusicStore.ViewModels.StoreIndexViewModel

@{

ViewBag.Title = "Browse Genres";

}

<h2>Browse Genres</h2>

<p>Select from @Model.NumberOfGenres genres</p>

**<ul>**

**@foreach (string genreName in Model.Genres)**

**{**

**<li>**

**@genreName**

**</li>**

**}**

**</ul>**

1. 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.

View displaying a list of genres

1. Close the browser.

### Exercise 6: Using Parameters in View

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**

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**.

1. If not already open, start **VS Express for Web**
2. In the **File** menu, choose **Open Project**. In the Open Project dialog, browse to **Source\Ex06-UsingParametersInView\Begin**, select**Begin.sln** and click **Open**. Alternatively, you may continue with the solution that you obtained after completing the previous exercise.
   1. If you opened the provided **Begin** solution, you will need to download some missing NuGet packages before continue. To do this, click the **Project** menu and select **Manage NuGet Packages**.
   2. In the **Manage NuGet Packages** dialog, click **Restore** in order to download missing packages.
   3. Finally, build the solution by clicking **Build** | **Build Solution**.

**Note:** One of the advantages of using NuGet is that you don't have to ship all the libraries in your project, reducing the project size. With NuGet Power Tools, by specifying the package versions in the Packages.config file, you will be able to download all the required libraries the first time you run the project. This is why you will have to run these steps after you open an existing solution from this lab.

1. Add a **Genre** Model class. To do this, right-click the **Models** folder in the **Solution Explorer**, select **Add** and then the **New Item** option. Under **Code**, choose the **Class** item and name the file Genre.cs, then click **Add**.

Adding a new item

Add Genre Model Class

1. Add a **Name** property to the Genre class. To do this, add the following code:

(Code Snippet - ASP.NET MVC 4 Fundamentals - Ex6 Genre)

C#

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

namespace MvcMusicStore.Models

{

public class Genre

{

**public string Name { get; set; }**

}

}

1. 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 **New Item** option. Under **Code**, choose the **Class** item and name the file Album.cs, then click **Add**.
2. Add two properties to the Album class: **Genre** and **Title**. To do this, add the following code:

(Code Snippet - ASP.NET MVC 4 Fundamentals - Ex6 Album)

C#

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

namespace MvcMusicStore.Models

{

public class Album

{

**public string Title { get; set; }**

**public Genre Genre { get; set; }**

}

}

#### **Task 2 - Adding a StoreBrowseViewModel**

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.

1. Add a **StoreBrowseViewModel** class. To do this, right-click the **ViewModels** folder in the **Solution Explorer**, select **Add** and then the**New Item** option. Under **Code**, choose the **Class** item and name the file StoreBrowseViewModel.cs, then click **Add**.
2. Add a reference to the Models in **StoreBrowseViewModel** class. To do this, add the following using namespace:

(Code Snippet - ASP.NET MVC 4 Fundamentals - Ex6 UsingModel)

C#

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

**using MvcMusicStore.Models;**

namespace MvcMusicStore.ViewModels

{

public class StoreBrowseViewModel

{

}

}

1. Add two properties to **StoreBrowseViewModel** class: **Genre** and **Albums**. To do this, add the following code:

(Code Snippet - ASP.NET MVC 4 Fundamentals - Ex6 ModelProperties)

C#

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using MvcMusicStore.Models;

namespace MvcMusicStore.ViewModels

{

public class StoreBrowseViewModel

{

**public Genre Genre { get; set; }**

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

}

}

**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).

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**.

**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**

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:

(Code Snippet - ASP.NET MVC 4 Fundamentals - Ex6 UsingModelInController)

C#

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

using MvcMusicStore.ViewModels;

**using MvcMusicStore.Models;**

1. 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:

(Code Snippet - ASP.NET MVC 4 Fundamentals - Ex6 BrowseMethod)

C#

**//**

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

**public ActionResult Browse(string genre)**

**{**

**var genreModel = new Genre()**

**{**

**Name = genre**

**};**

**var albums = new List<Album>()**

**{**

**new Album() { Title = genre + " Album 1" },**

**new Album() { Title = genre + " Album 2" }**

**};**

**var viewModel = new StoreBrowseViewModel()**

**{**

**Genre = genreModel,**

**Albums = albums**

**};**

**return this.View(viewModel);**

**}**

1. 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:

(Code Snippet - ASP.NET MVC 4 Fundamentals - Ex6 DetailsMethod)

C#

**//**

**// GET: /Store/Details/5**

**public ActionResult Details(int id)**

**{**

**var album = new Album { Title = "Sample Album" };**

**return this.View(album);**

**}**

#### **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 **Build** menu item and then **Build MvcMusicStore**.
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**.

Adding a Browse View

1. Modify the **Browse.cshtml** to display the Genre's information, accessing the **StoreBrowseViewModel** object that is passed to the view template. To do this, replace the content with the following:

HTML(C#)

**@model MvcMusicStore.ViewModels.StoreBrowseViewModel**

**@{**

**ViewBag.Title = "Browse Albums";**

**}**

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

**<ul>**

**@foreach (var album in Model.Albums)**

**{**

**<li>@album.Title</li>**

**}**

**</ul>**

#### **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.

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, 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.cshtml** file.

Adding a Details View

1. Modify the **Details.cshtml** file to display the Album's information, accessing the **Album** object that is passed to the view template. To do this, replace the content with the following:

HTML(C#)

**@model MvcMusicStore.Models.Album**

**@{**

**ViewBag.Title = "Details";**

**}**

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

#### **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.

Browsing Album's Detail

#### **Task 8 - Adding Links Between Pages**

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.

1. 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.cshtml** page.
2. Add a link to the Browse view indicating the genre selected. To do this, replace the following highlighted code within the **<li>** tags:

HTML(C#)

@model MvcMusicStore.ViewModels.StoreIndexViewModel

@{

ViewBag.Title = "Browse Genres";

}

<h2>Browse Genres</h2>

<p> Select from @Model.NumberOfGenres genres</p>

<ul>

@foreach (string genreName in Model.Genres) {

<li>

**@Html.ActionLink(genreName, "Browse", new { genre = genreName }, null)**

</li>

}

</ul>

**Note**: another approach would be linking directly to the page, with a code like the following:

<a href="/Store/Browse?genre=@genreName">@genreName</a>

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.

Htlm.ActionLink has several overloads. In this exercise you will use one that takes three parameters:

* 1. Link text, which will display the Genre name
  2. Controller action name (**Browse**)
  3. Route parameter values, specifying both the name (**Genre**) and the value (**Genre name**)

#### **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.

Browsing Genres with links to Browse page

#### **Task 10 - Using Dynamic ViewModel Collection to Pass Values**

In this task, you will learn a simple and powerful method to pass values between the Controller and the View without making any changes in the Model. ASP.NET MVC 4 provides the collection "ViewModel", which can be assigned to any dynamic value and accessed inside controllers and views as well.

You will now use the ViewBag dynamic collection to pass a list of "**Starred genres**" from the controller to the view. The Store Index view will access to **ViewModel** and display the information.

1. Close the browser if needed, to return to the Visual Studio window. Open **StoreController.cs** and modify **Index** method to create a list of starred genres into ViewModel collection :

C#

public ActionResult Index()

{

// Create list of genres

var genres = new List<string> { "Rock", "Jazz", "Country", "Pop", "Disco" };

// Create your view model

var viewModel = new StoreIndexViewModel

{

NumberOfGenres = genres.Count,

Genres = genres

};

**ViewBag.Starred = new List<string> { "Rock", "Jazz" };**

return this.View(viewModel);

}

**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 it to the application, drag their content from a **Windows Explorer** window into the **Solution Explorer** in Visual Web Developer Express, as shown below:

Adding star image to the solution

1. Open the view **Store/Index.cshtml** 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.

HTML(C#)

@model MvcMusicStore.ViewModels.StoreIndexViewModel

@{

ViewBag.Title = "Browse Genres";

}

<h2>Browse Genres</h2>

<p>Select from @Model.NumberOfGenres genres</p>

<ul>

@foreach (string genreName in Model.Genres)

{

<li>

@Html.ActionLink(genreName, "Browse", new { genre = genreName }, null)

**@if (ViewBag.Starred.Contains(genreName))**

**{**

**<img src="../../Content/Images/starred.png" alt="Starred element" />**

**}**

</li>

}

</ul>

**<br />**

**<h5><img src="../../Content/Images/starred.png" alt="Starred element" />Starred genres 20% Off!</h5>**

#### **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:

Browsing Genres with starred elements