

## Lesson 7

The exercises in this lesson are based on Adam Freeman. *Pro ASP.NET MVC 5*, pp. 178-214 and adapted for MbmStore.

### Exercise 1, styling the website, mandatory

We have already built a great deal of infrastructure and the application is starting to come together, but we have not paid much attention to its appearance. For the catalogue page we will implement a classic two-column layout with a header, as shown in this wireframe:

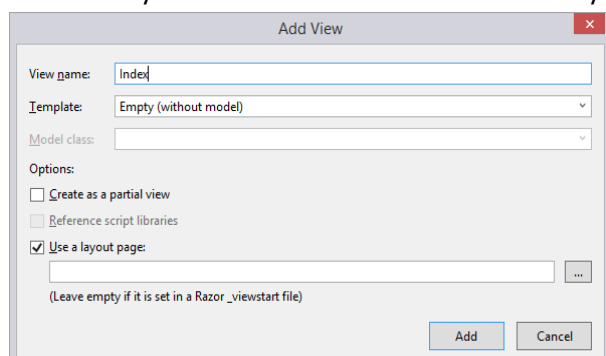
MusicBookMovieStore (header)	
<b>Home</b> <ul style="list-style-type: none"><li>• <b>Books</b></li><li>• <b>Music</b></li><li>• <b>Movies</b></li></ul>	<ul style="list-style-type: none"><li>• Product 1</li><li>• Product 2</li><li>• Product 3</li><li>• ...</li></ul>

### Installing the Bootstrap package and applying Bootstrap styles to the layout

We are going to use the Bootstrap package to provide the CSS styles we will apply to the application. To install the Bootstrap package, select Library Package Manager ➤ Package Manager Console from the Visual Studio Tools menu. Visual Studio will open the NuGet command line. Enter the following command and hit return:

```
Install-Package bootstrap
```

Whenever you create a view and check “Use a layout Page:



The screenshot shows the 'Add View' dialog box with the following details:

- View name:** Index
- Template:** Empty (without model)
- Model class:** (empty)
- Options:**
  - ☐ Create as a partial view
  - ☐ Reference script libraries
  - ☒ Use a layout page:
- Buttons:** Add, Cancel

two files are created in the `Views` folder. A `_ViewStart.cshtml` file and a `Shared/_Layout.cshtml` file. You probably already have these files in your project, but if not you can easily create them manually. The contents of the `_ViewStart.cshtml` file is this simple:

```
@{  
    Layout = "~/Views/Shared/_Layout.cshtml";  
}
```

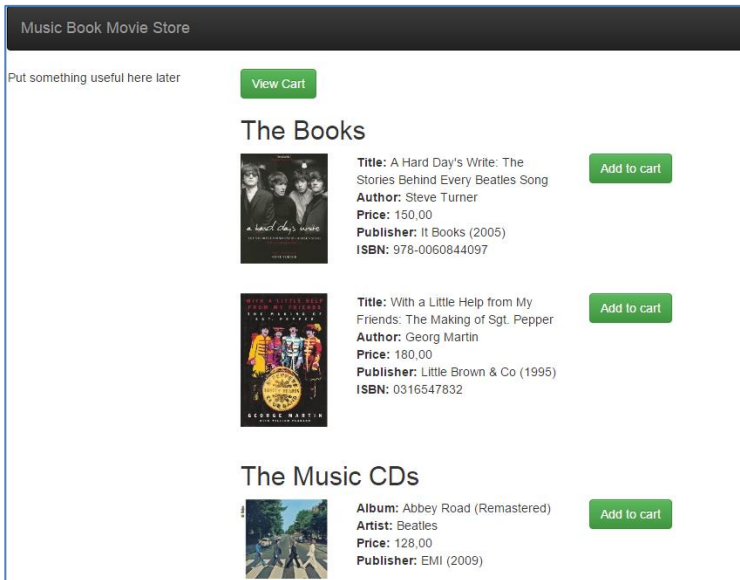
The value of the `Layout` property specifies that views will use the `Views/Shared/_Layout.cshtml` file as a layout, unless they explicitly specify an alternative.

The layout file `Views/Shared/_Layout.cshtml` looks like this:

```
<!DOCTYPE html>  
<html>  
<head>  
    <meta charset="utf-8" />  
    <meta name="viewport" content="width=device-width, initial-scale=1.0">  
    <link href="~/Content/bootstrap.css" rel="stylesheet" />  
    <link href="~/Content/bootstrap-theme.css" rel="stylesheet" />  
    <title>@ViewBag.Title</title>  
</head>  
<body class="container">  
    <div class="navbar navbar-inverse" role="navigation">  
        <a class="navbar-brand" href="#">Music Book Movie Store</a>  
    </div>  
    <div class="row panel">  
        <div id="categories" class="col-xs-3">  
            Put something useful here later  
        </div>  
        <div class="col-xs-9">  
            @RenderBody()  
        </div>  
    </div>  
</body>  
</html>
```

As you see, Bootstrap CSS files are included and some of the CSS styles it defines are applied as CSS classes.

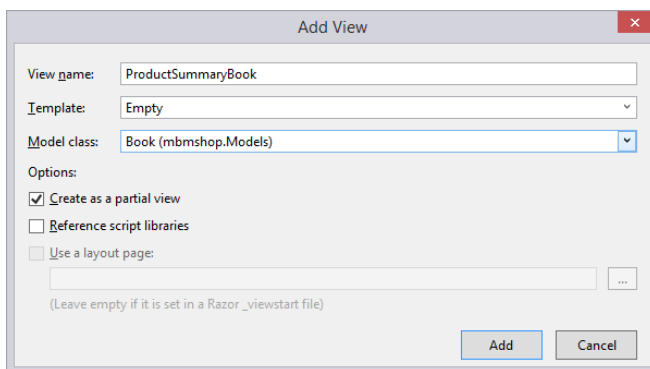
If you run the application, you will see that the improved the appearance as illustrated here:



## Exercise 2, creating Partial Views, mandatory

As a finishing trick for this chapter, we are going to refactor the application to simplify the `List.cshtml` view. You will create a *partial view*, which is a fragment of content that you can embed into another view, rather like a template. Partial views are contained within their own files and are reusable across multiple views, which can help reduce duplication if you need to render the same kind of data in several places in your application.

To add the partial view, right-click the `/Views/Shared` folder and select **Add ➤ View** from the pop-up menu. Set View Name to `ProductSummaryBook`, set Template to `Empty`, select `Book` from the Model Class drop-down list and check the **Create As A Partial View** box, as shown here:



Click the **Add** button, and Visual Studio will create a partial view file called `Views/Shared/ProductSummaryBook.cshtml`. A partial view is similar to a regular view, except that it produces a fragment of HTML, rather than a full HTML document. If you open the

`ProductSummaryBook` view, you will see that it contains only the model view directive, which is set to the `Book` domain model class. Apply the changes shown in this code sample:

```
@model MbmStore.Models.Book
```

```
<div class="row">
  <div class="col-md-1">
    
  </div>
  <div class="col-md-4">
    <strong>Title:</strong> @Model.Title<br />
    <strong>Author:</strong> @Model.Author<br />
    <strong>Price:</strong> @Model.Price<br />
    <strong>Publisher:</strong> @Model.Publisher (@Model.Published)<br />
    <strong>ISBN:</strong> @Model.ISBN<br /> <br />
  </div>

  <div class="col-md-7">
    @using (Html.BeginForm("AddToCart", "Cart"))
    {
      @Html.Hidden("ProductId", Model.ProductId)
      @Html.Hidden("returnUrl", Request.Url.PathAndQuery)
      <input type="submit" class="btn btn-success" value="Add to cart" />
    }
  </div>
</div>
```

Now we need to update `Views/Catalogue/Index.cshtml` so that it uses the partial view. You can see the change here:

```
@foreach (Book book in Model.OfType<Book>().ToList())
{
  @Html.Partial("ProductSummaryBook", book);
  <br />
}
```

I have taken the markup that was previously in the `foreach` loop in the `Catalogue.Index.cshtml` view (or maybe in an html helper funtion) and moved it to the new partial view. I call the partial view using the `Html.Partial` helper method. The parameters are the name of the view and the view model object. This partial view renders this output:

[View Cart](#)

## The Books



**Title:** A Hard Day's Write: The Stories Behind Every Beatles Song  
**Author:** Steve Turner  
**Price:** 150,00  
**Publisher:** It Books (2005)  
**ISBN:** 978-0060844097

[Add to cart](#)



**Title:** With a Little Help from My Friends: The Making of Sgt. Pepper  
**Author:** Georg Martin  
**Price:** 180,00  
**Publisher:** Little Brown & Co (1995)  
**ISBN:** 0316547832

[Add to cart](#)

Create similar partial view for `MusicCD` and `Movie`.

### Exercise 3, adding pagination, mandatory

Right now all products is displayed on a single page. In this exercise, we will add support for pagination so that the view displays a number of products on a page, and the user can move from page to page to view the overall catalogue. To do this, I am going to add a parameter to the `List` method in the `Product` controller, as shown here:

```
public class CatalogueController : Controller
{
    public int PageSize = 4;

    // GET: Catalogue
    public ActionResult Index(int page = 1)
    {
        Repository repository = new Repository();
        return View(repository.Products
            .OrderBy(p => p.ProductId)
            .Skip((page - 1) * PageSize)
            .Take(PageSize));
    }
}
```

The `PageSize` field specifies that I want four products per page. We will come back and replace this with a better mechanism later on. I have added an *optional parameter* to the `Index` method. This means that if we call the method without a parameter (`Index()`), the call is treated as though we had supplied the value specified in the parameter definition (`List(1)` – the default value is 1). The effect is that the action method displays the first page of products when the MVC Framework invokes it without an argument.

Within the body of the action method, we get the `Product` objects, order them by the primary key, skip over the products that occur before the start of the current page, and take the number of products specified by the `PageSize` field.

## Displaying Page Links

If you run the application, you will see that there are only four items shown on the page. If you want to view another page, you can append query string parameters to the end of the URL, like this:

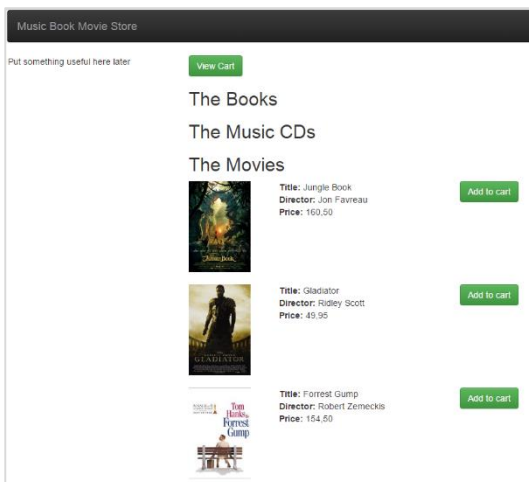
<http://localhost:51280/?page=2>

You will need to change the port part of the URL to match whatever port your ASP.NET development server is running on. Using these query strings, you can navigate through the catalogue of products. Of course, there is no way for customers to figure out that these query string parameters exist, and even if there were, they are not going to want to navigate this way. Instead, we need to render some page links at the bottom of the each list of products so that customers can navigate between pages. To do this, we are going to implement a reusable HTML helper method, similar to the `Html.TextBoxFor` and `Html.BeginForm` methods. The helper will generate the HTML markup for the navigation links we require.

Right now, when navigate to page 2,

<http://localhost:37143/Catalogue/Index/?page=2>

you'll see a page similar to this:



You can easily improve the display by excluding categories that have no objects in the `Product` list returned from the controller by adding this simple condition:

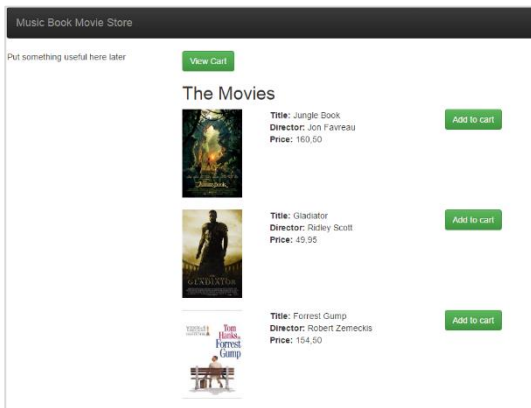
```
@if (Model.OfType<Book>().Count() > 0) {  
    <h2>The Books</h2>  
    foreach (Book book in Model.OfType<Book>().ToList())  
    {  
        @Html.Partial("ProductSummaryBook", book);  
    }  
}
```

```

    }
}

```

The `Count()` method returns the number of elements of a list. When add this condition to each category, you will only see the header if there are objects of the specified type:



## Adding the View Model

To support the HTML helper, we will pass information to the view about the number of pages available, the current page, and the total number of products in the repository. The easiest way to do this is to create a view model called `PagingInfo`, to the `ViewModels` folder.

## Adding the HTML Helper Method

Now that we have a view model, we can implement the HTML helper method, which we will call `PageLinks`. Create a new folder called `HtmlHelpers` and add a new class file called `PagingHelpers.cs`, with this content:

```

using MbmStore.ViewModels;
using System;
using System.Text;
using System.Web.Mvc;

namespace MbmStore.HtmlHelpers
{
    public static class PagingHelpers
    {
        public static MvcHtmlString PageLinks(this HtmlHelper html, PagingInfo pagingInfo,
            Func<int, string> pageUrl)
        {
            StringBuilder result = new StringBuilder();
            for (int i = 1; i <= pagingInfo.TotalPages; i++)
            {
                TagBuilder tag = new TagBuilder("a");
                tag.MergeAttribute("href", pageUrl(i));
                tag.InnerHtml = i.ToString();
                if (i == pagingInfo.CurrentPage)
                {
                    tag.AddCssClass("selected");
                    tag.AddCssClass("btn-primary");
                }
            }
        }
    }
}

```

```

        tag.AddCssClass("btn btn-default");
        result.Append(tag.ToString());
    }
    return MvcHtmlString.Create(result.ToString());
}
}
}

```

The `PageLinks` extension method for the `HtmlHelper` class that generates the HTML for a set of page links using the information provided in a `PagingInfo` object. The `Func` parameter accepts a delegate that it uses to generate the links to view other pages.

An extension method is available for use only when the namespace that contains it is in scope. In a code file, this is done with a `using` statement; but for a Razor view, you must add a configuration entry to the `Web.config` file, or add a `@using` statement to the view itself. There are, confusingly, two `Web.config` files in a Razor MVC project: the main one, which resides in the root directory of the application project, and the view-specific one, which is in the `Views` folder. The change I want to make is to the `Views/web.config` file as shown here:

```

<pages pageBaseType="System.Web.Mvc.WebViewPage">
  <namespaces>
    <add namespace="System.Web.Mvc" />
    <add namespace="System.Web.Mvc.Ajax" />
    <add namespace="System.Web.Mvc.Html" />
    <add namespace="System.Web.Routing" />
    <add namespace="MbmStore" />
    <add namespace="MbmStore.HtmlHelpers"/>
  </namespaces>
</pages>

```

Every namespace that I refer to in a Razor view needs to be used explicitly, declared in the `web.config` file or applied with a `@using` expression.

### Adding the View Model Data

We not quite ready to use the HTML helper method. We have yet to provide an instance of the `PagingInfo` view model class to the view. I could do this using the view bag feature, but I would rather wrap all of the data I am going to send from the controller to the view in a single view model class. To do this, I added a class file called `ProductsListViewModel.cs` to the `ViewModels` folder of the project:

```

using MbStore.Models;
using System.Collections.Generic;

namespace MbStore.ViewModels
{
    public class ProductsListViewModel
    {
        public IEnumerable<Product> Products { get; set; }
        public PagingInfo PagingInfo { get; set; }
    }
}

```



We can update the `Index` action method in the `ProductController` class to use the `ProductsListViewModel` class to provide the view with details of the products to display on the page and details of the pagination, as shown here:

```
using MbmStore.Infrastructure;
using MbmStore.ViewModels;
using System.Linq;
using System.Web.Mvc;

namespace MbmStore.Controllers
{
    public class CatalogueController : Controller
    {
        public int PageSize = 4;

        // GET: Catalogue
        public ActionResult Index(int page = 1)
        {
            Repository repository = new Repository();
            ProductsListViewModel model = new ProductsListViewModel
            {
                Products = repository.Products
                    .OrderBy(p => p.ProductId)
                    .Skip((page - 1) * PageSize)
                    .Take(PageSize),

                PagingInfo = new PagingInfo
                {
                    CurrentPage = page,
                    ItemsPerPage = PageSize,
                    TotalItems = repository.Products.Count()
                }
            };
            return View(model);
        }
    }
}
```

These changes pass a `ProductsListViewModel` object as the model data to the view.

### Displaying the Page Links

I have everything in place to add the page links to the List view. I created the view model that contains the paging information, updated the controller so that it passes this information to the view, and changed the `@model` directive to match the new model view type. All that remains is to call the HTML helper method from the view, which you can see here:

```
@using MbmStore.Models;
@model MbmStore.ViewModels.ProductsListViewModel
@{
    ViewBag.Title = "Index";
}

@using (Html.BeginForm("Index", "Cart"))
```

```

{
    @Html.Hidden("returnUrl", Request.Url.PathAndQuery)
    <input type="submit" class="btn btn-success" value="View Cart" />
}

@if(Model.Products.OfType<Book>().Count() > 0) {
    <h2>The Books</h2>
    foreach (Book book in Model.Products.OfType<Book>().ToList())
    {
        @Html.Partial("ProductSummaryBook", book);
        <br />
    }
}

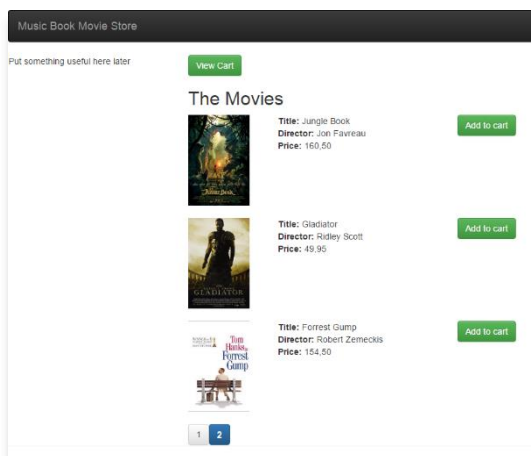
@if (Model.Products.OfType<MusicCD>().Count() > 0)
{
    <h2>The Music CDs</h2>
    foreach (MusicCD musicCD in Model.Products.OfType<MusicCD>().ToList())
    {
        @Html.Partial("ProductSummaryMusicCd", musicCD);
        <br />
    }
}

@if (Model.Products.OfType<Movie>().Count() > 0)
{
    <h2>The Movies</h2>
    foreach (Movie movie in Model.Products.OfType<Movie>().ToList())
    {
        @Html.Partial("ProductSummaryMovie", movie);
        <br />
    }
}

<div>
    @Html.PageLinks(Model.PagingInfo, x => Url.Action("Index", new { page = x }))
</div>

```

If you run the application, you will see the new page links, as illustrated below:



## Improving the URLs

We have the page links working, but they still use the query string to pass page information to the server, like this:

<http://localhost/Catalogue/?page=2>

We can create URLs that are more appealing by creating a scheme that follows the pattern of *composable URLs*.

A composable URL is one that makes sense to the user, like this one:

<http://localhost/Catalogue/Page2>

MVC makes it easy to change the URL scheme in application because it uses the ASP.NET *routing* feature. All I need do is add a new route to the `RegisterRoutes` method in the `RouteConfig.cs` file, which you will find in the `App_Start` folder of the project. You can see the change you need to make here:

```
using System.Web.Mvc;
using System.Web.Routing;

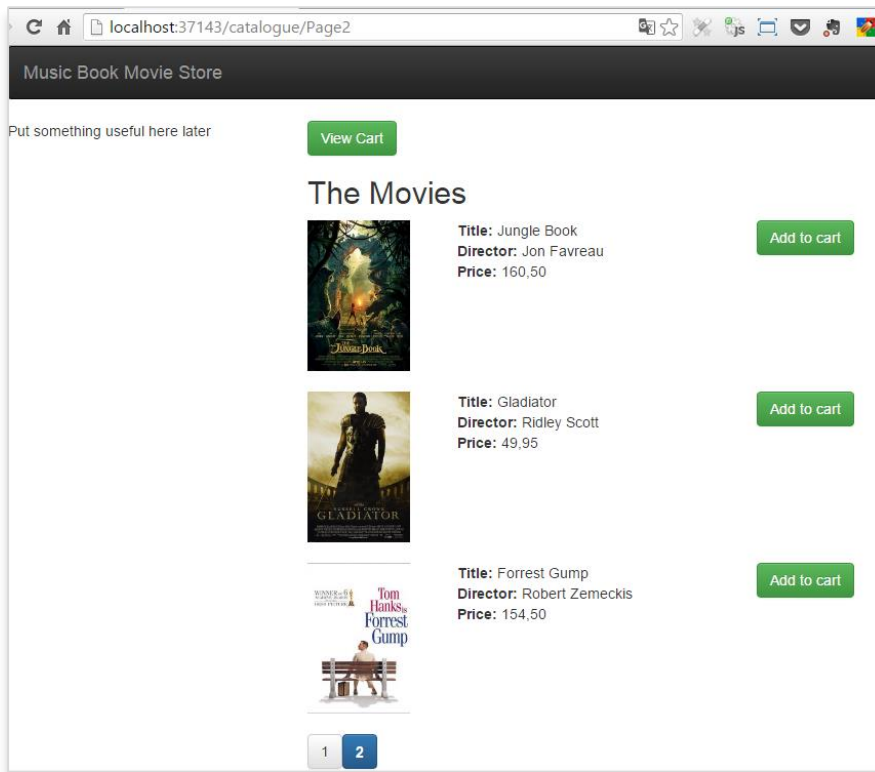
namespace MbmStore
{
    public class RouteConfig
    {
        public static void RegisterRoutes(RouteCollection routes)
        {
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

            routes.MapRoute(
                name: null,
                url: "{controller}/Page{page}",
                defaults: new { controller = "Catalogue", action = "Index" }
            );
            routes.MapRoute(
                name: "Default",
                url: "{controller}/{action}/{id}",
                defaults: new { controller = "Catalogue", action = "Index", id =
UrlParameter.Optional }
            );
        }
    }
}
```

It is important that you add this route before the `Default` one that is already in the file as and you need the new route to take precedence over the existing one. This is the only alteration required to change the URL scheme for product pagination. The MVC Framework and the routing function are tightly integrated, and so the application automatically reflects a change like this in the result produced by the `Url.Action` method (which is what we used in the `Catalogue/Index.cshtml` view to generate the page links).

If you run the application and navigate to a page, you will see the new URL scheme in action, as illustrated

here:



### Exercise 4, navigate by category, mandatory

The MusicBookMovieStore application will be more usable if customers can navigate products by category. We will do this in three phases:

- Enhance the List action model in the `CatalogueController` class so that it is able to filter the `Product` objects in the repository.
- Revisit and enhance the URL scheme and revise the routing strategy.
- Create a category list that will go into the sidebar of the site, highlighting the current category and linking to others.

### Filtering the Product List

In this exercise, we will start by adding a `Category` property to the `Product` class:

```
namespace MbmStore.Models {  
  
    public class Product {  
        public int ProductId { get; set; }  
        public string Title { get; set; }  
        public decimal Price { get; set; }  
        public string ImageUrl { get; set; }  
        public string Category { get; set; }  
    }  
}
```

```

        // constructor
        public Product() { }

        // constructor
        public Product(string title, decimal price) {
            Title = title;
            Price = price;
        }
    }
}

```

After that, you can open the `Infrastructure/Repository.cs` file and add a category to each `Product` object, like

```

// create a new Movie objects
Movie jungleBook = new Movie("Jungle Book", 160.50M, "junglebook.jpg", "Jon Favreau");
jungleBook.ProductId = 5;
jungleBook.Category = "Movie";

```

The next step is to enhance the view model class, `ProductsListViewModel`. We need to communicate the current category to the view in order to render the sidebar, and this is as good a place to start as any. The code below shows the changes we need to make to the `ProductsListView.cs` file.

```

using MbmStore.Models;
using System.Collections.Generic;

namespace MbmStore.ViewModels
{
    public class ProductsListViewModel
    {
        public IEnumerable<Product> Products { get; set; }
        public PagingInfo PagingInfo { get; set; }
        public string CurrentCategory { get; set; }
    }
}

```

With the new `CurrentCategory`, we can now update the `Catalogue` controller so that the `Index` action method will filter `Product` objects by category and use the new property we added to the view model to indicate which category has been selected:

```

public class CatalogueController : Controller
{
    public int PageSize = 4;

    // GET: Catalogue
    public ActionResult Index(string category, int page = 1)
    {
        Repository repository = new Repository();
        ProductsListViewModel model = new ProductsListViewModel
        {
            Products = repository.Products

```

```

        .Where(p => category == null || p.Category == category)
        .OrderBy(p => p.ProductId)
        .Skip((page - 1) * PageSize)
        .Take(PageSize),

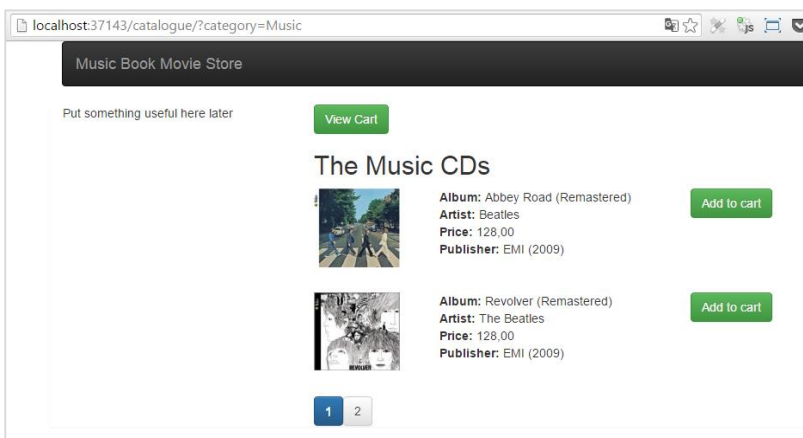
        PagingInfo = new PagingInfo
        {
            CurrentPage = page,
            ItemsPerPage = PageSize,
            TotalItems = repository.Products.Count()
        },
        CurrentCategory = category
    };
    return View(model);
}
}

```

We made three changes to the action method. First, we added a parameter called `category`. This `category` parameter is used by the second change, which is an enhancement to the LINQ query. If `category` is not null, only those `Product` objects with a matching `Category` property are selected. The last change is to set the value of the `CurrentCategory` property we added to the `ProductsListViewModel` class. However, these changes mean that the value of `PagingInfo.TotalItems` is incorrectly calculated. We will fix this in a while.

The effect of the category filtering is evident, even with these small changes. Start the application and select a category using the follow query string, changing the port to match the one that Visual Studio assigned for your project:

<http://localhost:51280/Catalogue/?category=Music>



You will see only the products in the Music category. Obviously, users won't want to navigate to categories using URLs, but you can see how small changes can have a big impact in an MVC Framework application once the basic structure is in place.

## Refining the URL Scheme

No one wants to see or use ugly URLs such as `/?category=Music`. To address this, we are going to revisit the routing scheme to create an approach to URLs that better suits us and our customers. To implement the new scheme, we will use the `RegisterRoutes` method in the `App_Start/RouteConfig.cs` file, as shown here:

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

        routes.MapRoute(
            name: null,
            url: "{controller}",
            defaults: new { controller = "Catalogue", action = "Index", category =
(string)null, page = 1 }
        );

        routes.MapRoute(
            name: null,
            url: "{controller}/Page{page}",
            defaults: new { controller = "Catalogue", action = "Index", category =
(string)null },
            constraints: new { page = @"\d+" }
        );

        routes.MapRoute(
            name: null,
            url: "Catalogue/{category}",
            defaults: new { controller = "Catalogue", action = "Index", page = 1 }
        );

        routes.MapRoute(
            name: null,
            url: "{controller}/{category}/Page{page}",
            defaults: new { controller = "Catalogue", action = "Index" },
            constraints: new { page = @"\d+" }
        );

        routes.MapRoute(
            name: "Default",
            url: "{controller}/{action}/{id}",
            defaults: new { controller = "Catalogue", action = "Index", id =
UrlParameter.Optional }
        );
    }
}
```

---

**Caution** It is important to add the new routes in Listing 8-3 in the order they are shown. Routes are applied in the order in which they are defined, and you will get some odd effects if you change the order.

---

The table describes the URL scheme that these routes represent.

URL	Leads to
/Catalogue	Lists the first page of products from all categories
/Catalogue/Page2	Lists the specified page (in this case, page 2), showing items from all categories
/Catalogue/Music	Shows the first page of items from a specific category (in this case, the Music category)
/Catalogue/Music/Page2	Shows the specified page (in this case, page 2) of items from the specified category (in this case, Music)

The ASP.NET routing system is used by MVC to handle *incoming* requests from clients, but it also generates *outgoing* URLs that conform to the URL scheme and that can be embedded in Web pages. By using the routing system to handle incoming requests and generate outgoing URLs, I can ensure that all of the URLs in the application are consistent.

The `Url.Action` method is the most convenient way of generating outgoing links. We have already used this helper method in the `Catalogue/Index.cshtml` view in order to display the page links. Now that we have added support for category filtering, we need to go back and pass this information to the helper method, as shown here:

```
@if (Model.Products.OfType<Movie>().Count() > 0)
{
    <h2>The Movies</h2>
    foreach (Movie movie in Model.Products.OfType<Movie>().ToList())
    {
        @Html.Partial("ProductSummaryMovie", movie);
        <br />
    }
}

<div>
    @Html.PagedListLinks(Model.PagingInfo, x => Url.Action("Index", new { page = x, category =
Model.CurrentCategory }))
</div>
```

Prior to this change, the links generated for the pagination links were like this:

```
http://<myserver>:<port>/Catalogue/Page1
```

If the user clicked a page link like this, the category filter he applied would be lost, and he would be presented with a page containing products from all categories. By adding the current category, taken from the view model, we generate URLs like this instead:

```
http://<myserver>:<port>/Catalogue/Music/Page1
```



When the user clicks this kind of link, the current category will be passed to the `Index` action method, and the filtering will be preserved. After you have made this change, you can visit a URL such as `/Music` or `/Movie`, and you will see that the page link at the bottom of the page correctly includes the category.

## Building a Category Navigation Menu

We need to provide customers with a way to select a category that does not involve typing in URLs. This means presenting them with a list of the categories available and indicating which, if any, is currently selected. As we build out the application, we will use this list of categories in more than one controller, so we need something that is self-contained and reusable.

The ASP.NET MVC Framework has the concept of *child actions*, which are perfect for creating items such as a reusable navigation control. A child action relies on the HTML helper method called `Html.Action`, which lets you include the output from an arbitrary action method in the current view. In this case, I can create a new controller (we will call it `NavController`) with an action method (which we will call `Menu`) that renders a navigation menu. We will then use the `Html.Action` helper method to inject the output from that method into the layout.

This approach gives us a real controller that can contain whatever application logic I need and that can be unit tested like any other controller. It is a nice way of creating smaller segments of an application while preserving the overall MVC Framework approach.

## Creating the Navigation Controller

Right-click the `Controllers` folder in the project and select `Add -> Controller` from the pop-up menu. Select `MVC 5 Controller – Empty` from the list, click the `Add` button, set the controller name to `NavController` and click the `Add` button to create the `NavController.cs` class file. Remove the `Index` method that Visual Studio adds to new controllers by default and add a new action method called `Menu`, as shown here:

```
namespace MbmStore.Controllers
{
    public class NavController : Controller
    {
        public string Menu()
        {
            return "Hello from NavController";
        }
    }
}
```

This method returns a static message string but it is enough to get us started while we integrate the child action into the rest of the application. We want the category list to appear on all pages, so we are going to render the child action in the layout, rather than in a specific view. Edit the `Views/Shared/_Layout.cshtml` file so that it calls the `Html.Action` helper method, as shown here:

```
<!DOCTYPE html>
<html>
<head>
```

```

<meta charset="utf-8" />
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<link href="~/Content/bootstrap.css" rel="stylesheet" />
<link href="~/Content/bootstrap-theme.css" rel="stylesheet" />
<title>@ViewBag.Title</title>
</head>
<body class="container">
  <div class="navbar navbar-inverse" role="navigation">
    <a class="navbar-brand" href="#">Music Book Movie Store</a>
  </div>
  <div class="row panel">
    <div id="categories" class="col-xs-3">
      @Html.Action("Menu", "Nav")
    </div>
    <div class="col-xs-9">
      @RenderBody()
    </div>
  </div>
</body>
</html>

```

We removed the placeholder text and replaced it with a call to the `Html.Action` method. The parameters to this method are the name of the action method (`Menu`) and the controller that contains it (`Nav`). If you run the application, you will see that the output of the `Menu` action method is included in the response sent to the browser, as shown in this screen shot:



## Generating Category Lists

We can now return to the `Nav` controller and generate a real set of categories. We do not want to generate the category URLs in the controller. Instead, we want the View to handle that and we will use a helper method in the view to do it. All we are going to do in the `Menu` action method is create the list of categories, as you see here:

```

using MbmStore.Infrastructure;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace MbmStore.Controllers
{
    public class NavController : Controller
    {

```

```

    Repository repository = new Repository();

    public PartialViewResult Menu()
    {
        IEnumerable<string> categories = repository.Products
            .Select(x => x.Category)
            .Distinct()
            .OrderBy(x => x);
        return PartialView(categories);
    }
}

```

The Menu action method now uses a LINQ query to obtain a list of categories from the `repository` and passes them to the view. Notice that, since I am working with a partial view in this controller, I call the `PartialView` method in the action method and that the result is a `PartialViewResult` object.

## Creating the View

To create the view for the Menu action method, right-click on the `Views/Nav` folder and select **Add ➤ MVC 5 View Page (Razor)** from the pop-up menu. Create the view as a partial view and set the name to `Menu` and click the **Add** button to create the `Menu.cshtml` file. Remove the contents – if any – that Visual Studio adds to new views and set the content to match this:

```

@model IEnumerable<string>

@Html.ActionLink("Home", "Index", "Catalogue", null,
    new { @class = "btn btn-block btn-default btn-lg" })

@foreach (var link in Model)
{
    @Html.RouteLink(link, new
    {
        controller = "Catalogue",
        action = "Index",
        category = link,
        page = 1
    }, new
    {
        @class = "btn btn-block btn-default btn-lg"
    })
}

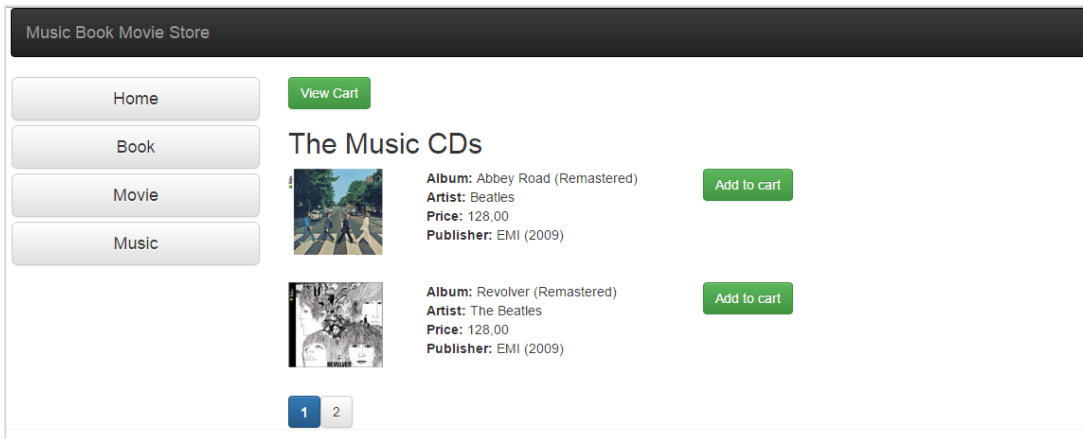
```

We added a link called `Home` that will appear at the top of the category list and will list all of the products with no category filter. We did this using the `ActionLink` helper method, which generates an HTML anchor element using the routing information configured earlier.

We then enumerated the category names and created links for each of them using the `RouteLink` method. This is similar to `ActionLink`, but it let us supply a set of name/value pairs that are taken into account when generating the URL from the routing configuration. The links we generate will look pretty ugly by default, so we have supplied an object to both the `ActionLink` and `RouteLink` helper

methods that specifies values for attributes on the elements that are created. The objects we created define the class attribute (prefixed with a `@` because `class` is a reserved C# keyword) and apply Bootstrap classes to style the links as large buttons.

You can see the category links if you run the application, as shown here:



If you click a category, the list of items is updated to show only items from the selected category.

### Highlighting the Current Category

At present, I do not indicate to users which category they are viewing. It might be something that the customer could infer from the items in the list, but I would prefer to provide solid visual feedback. We could do this by creating a view model that contains the list of categories and the selected category, and in fact, this is exactly what we would usually do. But for now we are going to use the view bag, which allow us to pass data from the controller to the view without using a view model. You must implement these changes in the `Menu` action method in the `Nav` controller:

```
public class NavController : Controller
{
    Repository repository = new Repository();

    public PartialViewResult Menu(string category = null)
    {
        ViewBag.SelectedCategory = category;

        IEnumerable<string> categories = repository.Products
            .Select(x => x.Category)
            .Distinct()
            .OrderBy(x => x);
        return PartialView(categories);
    }
}
```

We added a parameter to the `Menu` action method called `category`. The value for this parameter will be provided automatically by the routing configuration. Inside the method body, I have dynamically assigned a `SelectedCategory` property to the `ViewBag` object and set its value to be the current category. As

you already know, ViewBag is a dynamic object and I create new properties simply by setting values for them.

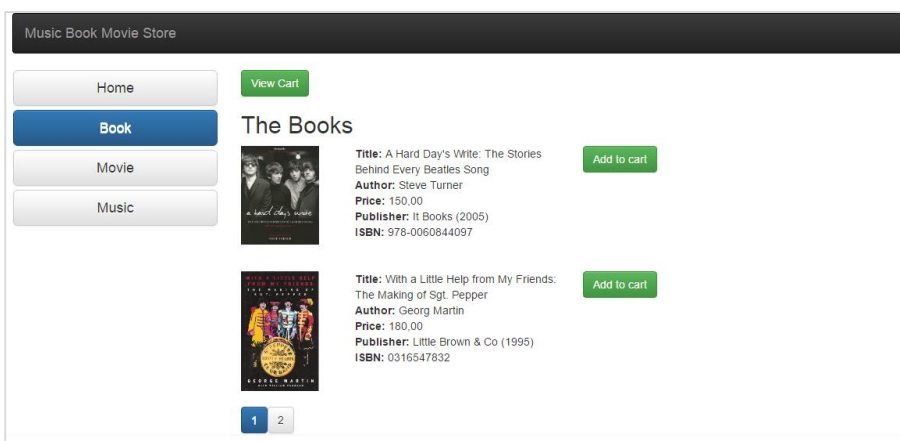
Now we have provided information about which category is selected, we can update the view to take advantage of this, and add a CSS class to the HTML anchor element that represents the selected category. This code shows the changes to the Menu.cshtml file:

```
@model IEnumerable<string>

@Html.ActionLink("Home", "Index", "Catalogue", null, new { @class = "btn btn-block btn-default btn-lg" + (ViewBag.SelectedCategory == null ? " btn-primary" : "") })

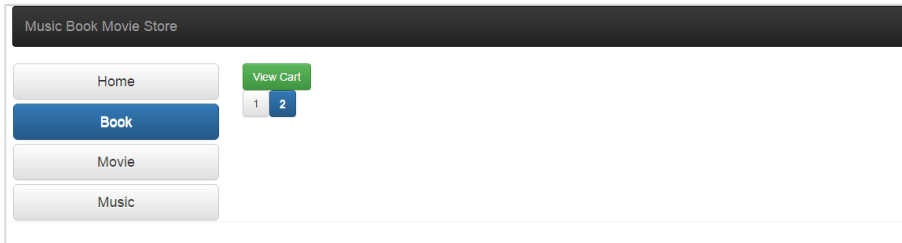
@foreach (var link in Model)
{
    @Html.RouteLink(link, new
    {
        controller = "Catalogue",
        action = "Index",
        category = link,
        page = 1
    }, new
    {
        @class = "btn btn-block btn-default btn-lg"
        + (link == ViewBag.SelectedCategory ? " btn-primary" : "")
    })
}
```

The changes is simple. If the current link value matches the SelectedCategory value, then I add the element were are creating to another Bootstrap class, which will cause the button to be highlighted. If the the SelectedCategory value is null that class is added to the Home ActionLink. Running the application shows the effect of the category highlighting, which you can also see in this screen shot:



## Correcting the Page Count

As the last step, we can now correct the page links so that they work correctly when a category is selected. Currently, the number of page links is determined by the total number of products in the repository and not the number of products in the selected category. This means that the customer can click the link for page 2 of the `Book` category and end up with an empty page because there are not enough chess products to fill two pages. You can see the problem here:

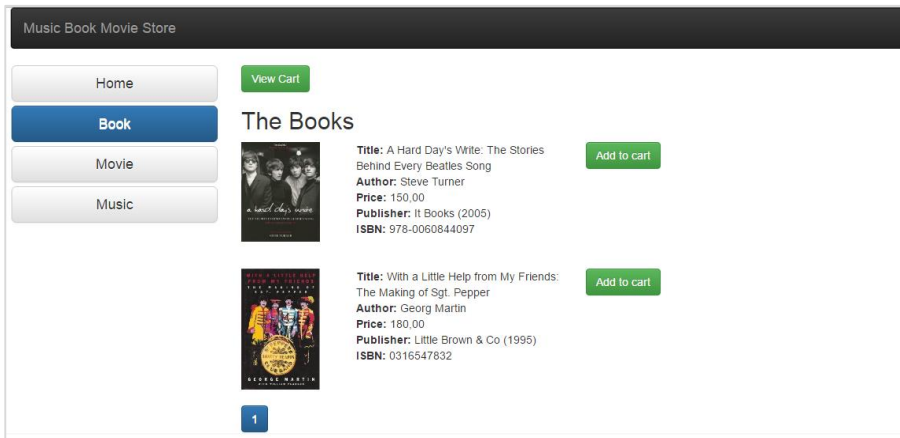


We can fix this by updating the `Index` action method in the `Catalogue` controller so that the pagination information takes the categories into account. You can see the required changes here:

```
public ActionResult Index(string category, int page = 1)
{
    Repository repository = new Repository();
    ProductsListViewModel model = new ProductsListViewModel
    {
        Products = repository.Products
            .Where(p => category == null || p.Category == category)
            .OrderBy(p => p.ProductId)
            .Skip((page - 1) * PageSize)
            .Take(PageSize),

        PagingInfo = new PagingInfo
        {
            CurrentPage = page,
            ItemsPerPage = PageSize,
            TotalItems = category == null ?
                repository.Products.Count() :
                repository.Products.Where(e => e.Category == category).Count()
        },
        CurrentCategory = category
    };
    return View(model);
}
```

If a category has been selected, I return the number of items in that category; if not, I return the total number of products. Now, when you view a category, the links at the bottom of the page correctly reflect the number of products in the category, as shown here:



## Exercise 5, Completing the Cart, Mandatory

We may have a functioning cart, but there is an issue with the way it is integrated into the interface. Customers can tell what is in their cart only by viewing the cart summary screen. And they can view the cart summary screen only by clicking the **View Cart** button.

### Adding the Cart Summary

To solve this problem, we are going to add a widget that summarizes the contents of the cart and that can be clicked to display the cart contents throughout the application. We will do this in much the same way that we added the navigation widget—as an action whose output we will inject into the Razor layout. To start, we need to add the simple method, shown here below to the `CartController` class.

```
using MbmStore.Infrastructure;
using MbmStore.Models;
using MbmStore.ViewModels;
using System.Linq;
using System.Web.Mvc;

namespace MbmStore.Controllers
{
    public class CartController : Controller
    {
        private Repository repository;

        // constructor
        // instantiate a new repository object
        public CartController()
        {
            repository = new Repository();
        }

        // ...other action methods omitted for brevity...

        public PartialViewResult Summary(Cart cart)
        {
            return PartialView(cart);
        }
    }
}
```

```

    }
}
}

```

This simple method needs to render a partial view, supplying the current `Cart` (which will be obtained using the custom model binder) as view data. To create the view, right-click the `Summary` action method and select `Add View` from the pop-up menu. Set the name of the view to `Summary` and click the `OK` button to create the `Views/Cart/Summary.cshtml` file. Edit the view so that it matches:

```

@model MbmStore.ViewModels.Cart

<div class="navbar-right">
    @Html.ActionLink("Checkout", "Index", "Cart",
new { returnUrl = Request.Url.PathAndQuery },
new { @class = "btn btn-default navbar-btn" })
</div>
<div class="navbar-text navbar-right">
    <b>Your cart:</b>
    @Model.Lines.Sum(x => x.Quantity) item(s),
    @Model.TotalPrice.ToString("n")
</div>

```

The view displays the number of items in the cart, the total cost of those items, and a link that shows the details of the cart to the user (and, as you will have expected by now, I have assigned the elements that the view contains to classes defined by Bootstrap). Now that we have created the partial view that is returned by the `Summary` action method, we can call the `Summary` action method in the `_Layout.cshtml` file to display the cart summary, as shown:

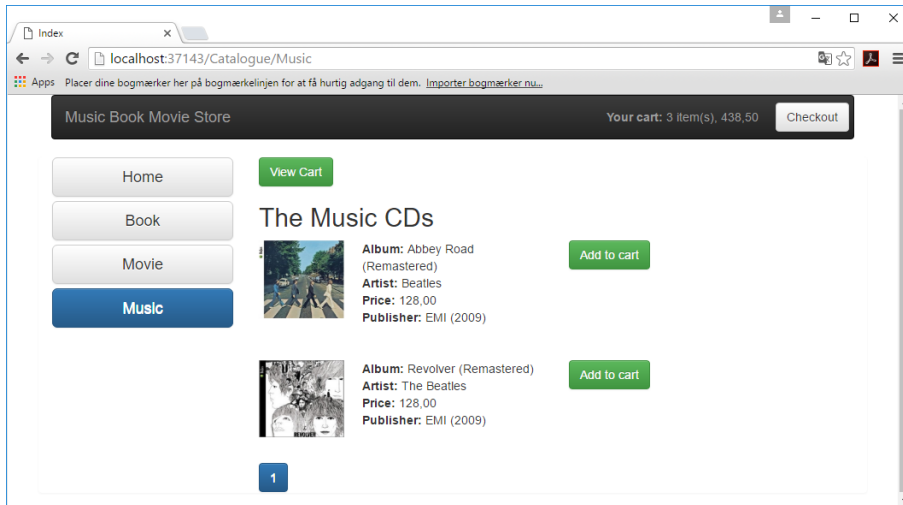
```

<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link href="~/Content/bootstrap.css" rel="stylesheet" />
    <link href="~/Content/bootstrap-theme.css" rel="stylesheet" />
    <title>@ViewBag.Title</title>
</head>
<body class="container">
    <div class="navbar navbar-inverse" role="navigation">
        <a class="navbar-brand" href="#">Music Book Movie Store</a>
        @Html.Action("Summary", "Cart")
    </div>
    <div class="row panel">
        <div id="categories" class="col-xs-3">
            @Html.Action("Menu", "Nav")
        </div>
        <div class="col-xs-9">
            @RenderBody()
        </div>
    </div>
</body>
</html>

```



You can see the cart summary by running the application. The item count and total increase as you add items to the cart, as shown here:



## Exercise 6, Submitting Orders, Optional

We have now reached the final customer feature in MbmStore: the ability to check out and complete an order. In the following sections, we will extend the domain model to provide support for capturing the shipping details from a user and add the application support to process those details.

### Extending the Domain Model

Add a class file called `ShippingDetails.cs` to the `ViewModels` folder and edit it to match the contents shown below. This is the class we will use to represent the shipping details for a customer.

```
namespace MbmStore.ViewModels
{
    public class ShippingDetails
    {
        public string Firstname { get; set; }
        public string Lastname { get; set; }
        public string Address { get; set; }
        public string Zip { get; set; }
        public string City { get; set; }
        public string Country { get; set; }
        public string Email { get; set; }
        public bool GiftWrap { get; set; }
    }
}
```

### Adding the Checkout Process

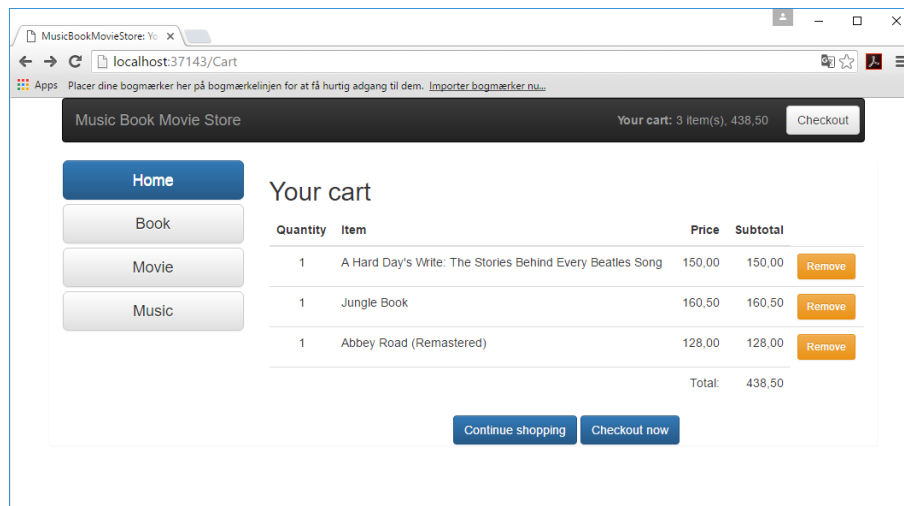
The goal is to reach the point where users are able to enter their shipping details and submit their order. To start this off, we need to add a `Checkout` button to the cart summary view. The code below shows the change we need to apply to the `Views/Cart/Index.cshtml` file:

```

...
<div class="text-center">
  <a class="btn btn-primary" href="@Model.ReturnUrl">Continue shopping</a>
  @Html.ActionLink("Checkout now", "Checkout", null, new { @class = "btn btn-primary" })
</div>
...

```

This change generates a link that I have styled as a button and that, when clicked, calls the `Checkout` action method of the `Cart` controller. You can see how this button appears:



As you might expect, we now need to define the `Checkout` method in the `CartController` class, as shown here:

```

using MbmStore.Infrastructure;
using MbmStore.Models;
using MbmStore.ViewModels;
using System.Linq;
using System.Web.Mvc;

namespace MbmStore.Controllers
{
    public class CartController : Controller
    {
        private Repository repository;

        // constructor
        // instantiate a new repository object
        public CartController()
        {
            repository = new Repository();
        }

        // ... other action methods omitted for brevity ...

        public ViewResult Checkout()
        {
            return View(new ShippingDetails());
        }
    }
}

```

```

    }
}

```

The `Checkout` method returns the default view and passes a new `ShippingDetails` object as the view model. To create the partial view for the action method, right-click on the `Checkout` action method and select **Add View** from the pop-up menu. Set the name to `Checkout` and click the OK button. Visual Studio will create the `Views/Cart/Checkout.cshtml` file, which you should edit to match this:

```

@model MbmStore.ViewModels.ShippingDetails
@{
    ViewBag.Title = "SportStore: Checkout";
}
<h2>Check out now</h2>
<p>Please enter your details, and we'll ship your goods right away!</p>
@using (Html.BeginForm())
{
    <h3>Ship to</h3>
    <div class="form-group">
        <label>Firstname:</label>
        @Html.TextBoxFor(x => x.Firstname, new { @class = "form-control" })
    </div>
    <div class="form-group">
        <label>Lastname:</label>
        @Html.TextBoxFor(x => x.Lastname, new { @class = "form-control" })
    </div>
    <div class="form-group">
        <label>Address:</label>
        @Html.TextBoxFor(x => x.Address, new { @class = "form-control" })
    </div>
    <div class="form-group">
        <label>Zip:</label>
        @Html.TextBoxFor(x => x.Zip, new { @class = "form-control" })
    </div>
    <div class="form-group">
        <label>Country:</label>
        @Html.TextBoxFor(x => x.Country, new { @class = "form-control" })
    </div>
    <div class="form-group">
        <label>Email:</label>
        @Html.TextBoxFor(x => x.Email, new { @class = "form-control" })
    </div>

    <h3>Options</h3>
    <div class="checkbox">
        <label>
            @Html.EditorFor(x => x.GiftWrap)
            Gift wrap these items
        </label>
    </div>
    <div class="text-center">
        <input class="btn btn-primary" type="submit" value="Complete order" />
    </div>
}

```

For each of the properties in the model, we have created a label and input element formatted with Bootstrap to capture the user input. You can see the effect by starting the application and clicking the `Checkout` button at the top of the page and then clicking `Checkout now`, as shown here below. (You can also reach this view by navigating to the `/Cart/Checkout` URL).

The problem with this view is that it contains a lot of repeated markup. There are MVC Framework HTML helpers that could reduce the duplication, but they make it hard to structure and style the content in the way that we want. Instead, we're going to use a handy feature to get metadata about the view model object and combine it with a mix of C# and Razor expressions like this:

```
@model MbmStore.ViewModels.ShippingDetails
@{
    ViewBag.Title = "SportStore: Checkout";
}
<h2>Check out now</h2>
<p>Please enter your details, and we'll ship your goods right away!</p>
@using (Html.BeginForm())
{
    <h3>Ship to</h3>

    foreach (var property in ViewData.ModelMetadata.Properties)
    {
        if (property.PropertyName != "GiftWrap")
        {
            <div class="form-group">
                <label>@{property.DisplayName ?? property.PropertyName}</label>
                @Html.TextBox(property.PropertyName, null, new { @class = "form-control" })
            </div>
        }
    }

    <h3>Options</h3>
    <div class="checkbox">
        <label>
```

```

        @Html.EditorFor(x => x.GiftWrap)
        Gift wrap these items
    </label>
</div>
<div class="text-center">
    <input class="btn btn-primary" type="submit" value="Complete order" />
</div>
}

```

The static `ViewData.ModelMetadata` property returns a `System.Web.Mvc.ModelMetadata` object that provides information about the model type for the view. The `Properties` property used in the `foreach` loop returns a collection of `ModelMetadata` objects, each of which represents a property defined by the model type. We use the `PropertyName` property to ensure that I don't generate content for the `GiftWrap` property (which are displayed elsewhere in the view) and generate a set of elements, complete with Bootstrap classes, for all of the other properties.

---

**Tip** The `for` and `if` keywords that we have used are within the scope of a Razor expression (the `@using` expression that creates the form) and so we don't need to prefix them with the `@` character. In fact, if we choose to do so, Razor would report an error. It can take a little while to get used to when the `@` character is required with Razor, but it becomes second nature for most programmers. For those that can't quite get it right first time (which includes me), the Razor error message displayed in the browser provides specific instructions to correct any mistakes.

---

## Processing the order

The last step is to process the order. We will not add the processing logic yet because it involves storing order information in a database. For now, we'll prepare that step by adding a `Checkout` method to the `Cart` controller:

```

[HttpPost]
public ActionResult Checkout(Cart cart, ShippingDetails shippingDetails)
{
    if (cart.Lines.Count() == 0)
    {
        ModelState.AddModelError("", "Sorry, your cart is empty!");
    }

    if (ModelState.IsValid)
    {
        // order processing logic
        cart.Clear();
        return View("Completed");
    }
    else
    {
        return View(shippingDetails);
    }
}

```

You can see that the `Checkout` action method I added is decorated with the `HttpPost` attribute, which means that it will be invoked for a `POST` request—in this case, when the user submits the form. The form

data will be processed – by writing some extra code – if the model is valid (`ModelState` is valid), otherwise the form with shipping details are displayed. Right now we are not doing any validation – besides from checking if the cart is empty – and we are not displaying any error messages. You'll learn more about validation and how to display custom error messages in lesson 10.

### Displaying a Summary Page

To complete the checkout process, I will show customers a page that confirms the order has been processed and thanks them for their business. Create a new view called `Completed.cshtml` in the `Views/Cart` folder and edit the content to match this:

```
@{  
    ViewBag.Title = "MusicBookMovie Store: Order Submitted";  
}  
<h2>Thanks!</h2>  
Thanks for placing your order. We'll ship your goods as soon as possible.
```

We do not need to make any code changes to integrate this view into the application because we already added the required statements when we defined the `Checkout` action method in the `Cart` controller. Now customers can go through the entire process, from selecting products to checking out. If they have items in their cart, they will see the summary page when they click the `Complete order` button, as shown here:

