CHAPTER 2

# SportsStore: Navigation

In the previous chapter, I set up the core infrastructure of the SportsStore application. Now I will use that infrastructure to add features to the application and you will start to see how the investment in the basic plumbing pays off. I will be able to add important customer-facing features simply and easily and, along the way, you will see some additional functionality that the MVC Framework provides.

## Adding Navigation Controls

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

- Enhance the `List` action model in the `ProductController` 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

I am going to start by enhancing the view model class, `ProductsListViewModel`, which I added to the `SportsStore.WebUI` project in the last chapter. I 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. Listing 8-1 shows the changes I made to the `ProductsListView.cs` file.

***Listing 8-1.*** Enhancing the ProductsListView .cs File

```
using System.Collections.Generic;
using SportsStore.Domain.Entities;

namespace SportsStore.WebUI.Models {
    public class ProductsListViewModel {

        public IEnumerable<Product> Products { get; set; }
        public PagingInfo PagingInfo { get; set; }
        public string CurrentCategory { get; set; }
    }
}
```

I added a property called CurrentCategory. The next step is to update the Product controller so that the List action method will filter Product objects by category and use the new property I added to the view model to indicate which category has been selected. The changes are shown in Listing 8-2.

***Listing 8-2.*** Adding Category Support to the List Action Method in the ProductController.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;
using SportsStore.WebUI.Models;

namespace SportsStore.WebUI.Controllers {

    public class ProductController : Controller {
        private IProductRepository repository;
        public int PageSize = 4;

        public ProductController(IProductRepository productRepository) {
            this.repository = productRepository;
        }

        public ViewResult List(string category, int page = 1) {
            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);
        }
    }
}
```

I made three changes to the action method. First, I added a parameter called category. This category parameter is used by the second change in the listing, 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 I added to the ProductsListViewModel class. However, these changes mean that the value of PagingInfo.TotalItems is incorrectly calculated. I will fix this in a while.

## UNIT TEST: UPDATING EXISTING UNIT TESTS

I changed the signature of the `List` action method, which will prevent some of the existing unit test methods from compiling. To address this, I need to pass `null` as the first parameter to the `List` method in those unit tests that work with the controller. For example, in the `Can_Paginate` test, the action section of the unit test becomes as follows:

```
...
[TestMethod]
public void Can_Paginate() {

    // Arrange
    Mock<IProductRepository> mock = new Mock<IProductRepository>();
    mock.Setup(m => m.Products).Returns(new Product[] {
        new Product {ProductID = 1, Name = "P1"},
        new Product {ProductID = 2, Name = "P2"},
        new Product {ProductID = 3, Name = "P3"},
        new Product {ProductID = 4, Name = "P4"},
        new Product {ProductID = 5, Name = "P5"}
    });

    // create a controller and make the page size 3 items
    ProductController controller = new ProductController(mock.Object);
    controller.PageSize = 3;

    // Act
    ProductsListViewModel result
        = (ProductsListViewModel)controller.List(null, 2).Model;

    // Assert
    Product[] prodArray = result.Products.ToArray();
    Assert.IsTrue(prodArray.Length == 2);
    Assert.AreEqual(prodArray[0].Name, "P4");
    Assert.AreEqual(prodArray[1].Name, "P5");
}
...
```

By using `null`, I receive all of the `Product` objects that the controller gets from the repository, which is the same situation I had before adding the new parameter. I need to make the same kind of change to the `Can_Send_Pagination_View_Model` test as well:

```
...
[TestMethod]
public void Can_Send_Pagination_View_Model() {
```

```
        // Arrange
        Mock<IProductRepository> mock = new Mock<IProductRepository>();
        mock.Setup(m => m.Products).Returns(new Product[] {
            new Product {ProductID = 1, Name = "P1"},
            new Product {ProductID = 2, Name = "P2"},
            new Product {ProductID = 3, Name = "P3"},
            new Product {ProductID = 4, Name = "P4"},
            new Product {ProductID = 5, Name = "P5"}
        });

        // Arrange
        ProductController controller = new ProductController(mock.Object);
        controller.PageSize = 3;

        // Act
        ProductsListViewModel result
            = (ProductsListViewModel)controller.List(null, 2).Model;

        // Assert
        PagingInfo pageInfo = result.PagingInfo;
        Assert.AreEqual(pageInfo.CurrentPage, 2);
        Assert.AreEqual(pageInfo.ItemsPerPage, 3);
        Assert.AreEqual(pageInfo.TotalItems, 5);
        Assert.AreEqual(pageInfo.TotalPages, 2);
    }
    ...
```

Keeping your unit tests synchronized with your code changes quickly becomes second nature when you get into the testing mind-set.

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/?category=Soccer
```

You will see only the products in the Soccer category, as shown in Figure 8-1.

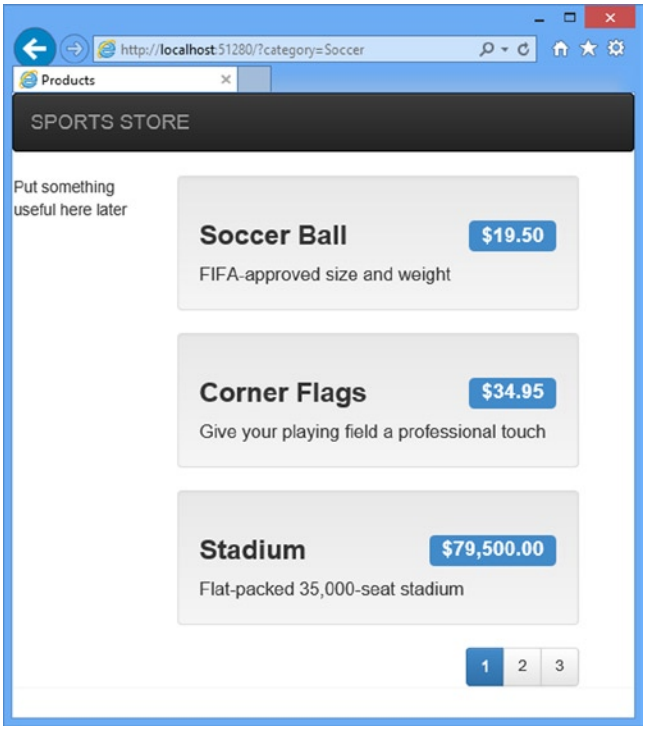


***Figure 8-1.*** *Using the query string to filter by category*

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.

## UNIT TEST: CATEGORY FILTERING

I need a unit test to properly test the category filtering function, to ensure that the filter can correctly generate products in a specified category. Here is the test:

```
...
[TestMethod]
public void Can_Filter_Products() {

    // Arrange
    // - create the mock repository
    Mock<IProductRepository> mock = new Mock<IProductRepository>();
    mock.Setup(m => m.Products).Returns(new Product[] {
        new Product {ProductID = 1, Name = "P1", Category = "Cat1"},
        new Product {ProductID = 2, Name = "P2", Category = "Cat2"},
        new Product {ProductID = 3, Name = "P3", Category = "Cat1"},
        new Product {ProductID = 4, Name = "P4", Category = "Cat2"},
        new Product {ProductID = 5, Name = "P5", Category = "Cat3"}
    });
```

```
        // Arrange - create a controller and make the page size 3 items
        ProductController controller = new ProductController(mock.Object);
        controller.PageSize = 3;

        // Action
        Product[] result = ((ProductsListViewModel)controller.List("Cat2", 1).Model)
            .Products.ToArray();

        // Assert
        Assert.AreEqual(result.Length, 2);
        Assert.IsTrue(result[0].Name == "P2" && result[0].Category == "Cat2");
        Assert.IsTrue(result[1].Name == "P4" && result[1].Category == "Cat2");
    }
    ...
```

This test creates a mock repository containing `Product` objects that belong to a range of categories. One specific category is requested using the `Action` method, and the results are checked to ensure that the results are the right objects in the right order.

## Refining the URL Scheme

No one wants to see or use ugly URLs such as `/?category=Soccer`. To address this, I am going to revisit the routing scheme to create an approach to URLs that better suits me and my customers. To implement the new scheme, I changed the `RegisterRoutes` method in the `App_Start/RouteConfig.cs` file, as shown in Listing 8-3.

***Listing 8-3.*** The New URL Scheme in the RouteConfig.cs File

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;
using System.Web.Routing;

namespace SportsStore.WebUI {

    public class RouteConfig {

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

            routes.MapRoute(null,
                "",
                new {
                    controller = "Product", action = "List",
                    category = (string)null, page = 1
                }
            );
```

```
            routes.MapRoute(null,
                "Page{page}",
                new { controller = "Product", action = "List", category = (string)null },
                new { page = @"\d+" }
            );

            routes.MapRoute(null,
                "{category}",
                new { controller = "Product", action = "List", page = 1 }
            );

            routes.MapRoute(null,
                "{category}/Page{page}",
                new { controller = "Product", action = "List" },
                new { page = @"\d+" }
            );

            routes.MapRoute(null, "{controller}/{action}");
        }
    }
}
```

■ **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.

Table 8-1 describes the URL scheme that these routes represent. I explain the routing system in detail in Chapters 15 and 16.

***Table 8-1.*** *Route Summary*

| URL | Leads To |
|---|---|
| `/` | Lists the first page of products from all categories |
| `/Page2` | Lists the specified page (in this case, page 2), showing items from all categories |
| `/Soccer` | Shows the first page of items from a specific category (in this case, the `Soccer` category) |
| `/Soccer/Page2` | Shows the specified page (in this case, page 2) of items from the specified category (in this case, `Soccer`) |

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.

■ **Note** I show you how to unit test routing configurations in Chapter 15.

The `Url.Action` method is the most convenient way of generating outgoing links. In the previous chapter, I used this helper method in the `List` view in order to display the page links. Now that I have added support for category filtering, I need to go back and pass this information to the helper method, as shown in Listing 8-4.

***Listing 8-4.*** Adding Category Information to the Pagination Links in the List.cshtml File

```
@model SportsStore.WebUI.Models.ProductsListViewModel

@{
    ViewBag.Title = "Products";
}

@foreach (var p in Model.Products) {
    @Html.Partial("ProductSummary", p)
}

<div class="btn-group pull-right">
    @Html.PageLinks(Model.PagingInfo, x => Url.Action("List",
        new { page = x, category = Model.CurrentCategory }))
</div>
```

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

```
http://<myserver>:<port>/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, I generate URLs like this instead:

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

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

## Building a Category Navigation Menu

I 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 I build out the application, I will use this list of categories in more than one controller, so I 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 (I will call it `NavController`) with an action method (which I will call `Menu`) that renders a navigation menu. I will then use the `Html.Action` helper method to inject the output from that method into the layout.

This approach gives me 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 SportsStore.WebUI 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 in Listing 8-5.

***Listing 8-5.*** Adding The Menu Action Method to the NavController.cs File

```
using System.Web.Mvc;

namespace SportsStore.WebUI.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 me started while I integrate the child action into the rest of the application. I want the category list to appear on all pages, so I am 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 in Listing 8-6.

***Listing 8-6.*** Adding the RenderAction Call to the _Layout.cshtml File

```
<!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>
    <div class="navbar navbar-inverse" role="navigation">
        <a class="navbar-brand" href="#">SPORTS STORE</a>
    </div>
    <div class="row panel">
        <div id="categories" class="col-xs-3">
            @Html.Action("Menu", "Nav")
        </div>
        <div class="col-xs-8">
            @RenderBody()
        </div>
    </div>
</body>
</html>
```

I 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 I want to call (`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 Figure 8-2.

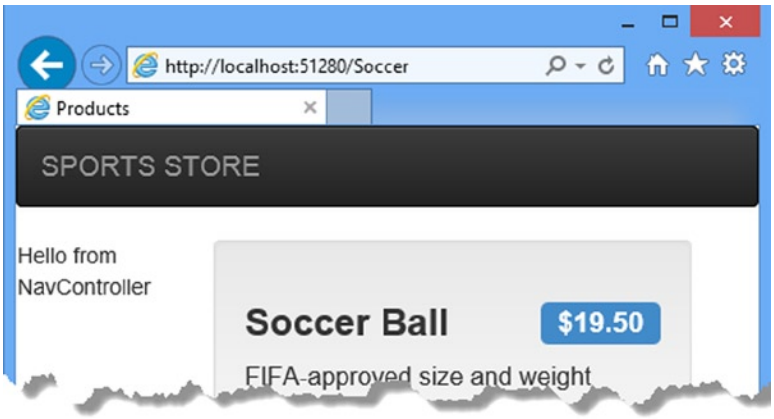


***Figure 8-2.*** *Displaying the output from the Menu action method*

## Generating Category Lists

I can now return to the `Nav` controller and generate a real set of categories. I do not want to generate the category URLs in the controller. Instead, I am going to use a helper method in the view to do that. All I am going to do in the `Menu` action method is create the list of categories, which I have done in Listing 8-7.

***Listing 8-7.*** Implementing the Menu Method in the NavController.cs File

```
using System.Collections.Generic;
using System.Web.Mvc;
using SportsStore.Domain.Abstract;
using System.Linq;

namespace SportsStore.WebUI.Controllers {

    public class NavController : Controller {
        private IProductRepository repository;

        public NavController(IProductRepository repo) {
            repository = repo;
        }

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

```
            return PartialView(categories);
        }
    }
}
```

The first change is to add a constructor that accepts an `IProductRepository` implementation as its argument. This has the effect of declaring a dependency that Ninject will resolve when it creates instances of the `NavController` class. The second change is to the `Menu` action method, which 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.

## UNIT TEST: GENERATING THE CATEGORY LIST

The unit test for my ability to produce a category list is relatively simple. My goal is to create a list that is sorted in alphabetical order and contains no duplicates. The simplest way to do this is to supply some test data that *does* have duplicate categories and that is *not* in order, pass this to the `NavController`, and assert that the data has been properly cleaned up. Here is the unit test:

```
...
[TestMethod]
public void Can_Create_Categories() {

    // Arrange
    // - create the mock repository
    Mock<IProductRepository> mock = new Mock<IProductRepository>();
    mock.Setup(m => m.Products).Returns(new Product[] {
        new Product {ProductID = 1, Name = "P1", Category = "Apples"},
        new Product {ProductID = 2, Name = "P2", Category = "Apples"},
        new Product {ProductID = 3, Name = "P3", Category = "Plums"},
        new Product {ProductID = 4, Name = "P4", Category = "Oranges"},
    });

    // Arrange - create the controller
    NavController target = new NavController(mock.Object);

    // Act = get the set of categories
    string[] results = ((IEnumerable<string>)target.Menu().Model).ToArray();

    // Assert
    Assert.AreEqual(results.Length, 3);
    Assert.AreEqual(results[0], "Apples");
    Assert.AreEqual(results[1], "Oranges");
    Assert.AreEqual(results[2], "Plums");
}
...
```

I created a mock repository implementation that contains repeating categories and categories that are not in order. I assert that the duplicates are removed and that alphabetical ordering is imposed.

## 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. Set the name to Menu and click the OK button to create the Menu.cshtml file. Remove the contents that Visual Studio adds to new views and set the content to match Listing 8-8.

***Listing 8-8.*** The Contents of the Menu.cshtml File

```
@model IEnumerable<string>

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

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

I 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. I did this using the ActionLink helper method, which generates an HTML anchor element using the routing information configured earlier.

I then enumerated the category names and created links for each of them using the RouteLink method. This is similar to ActionLink, but it lets me supply a set of name/value pairs that are taken into account when generating the URL from the routing configuration. Do not worry if all this talk of routing does not make sense yet. I explain everything in depth in Chapters 15 and 16.

The links I generate will look pretty ugly by default, so I 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 I 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 in Figure 8-3. If you click a category, the list of items is updated to show only items from the selected category.

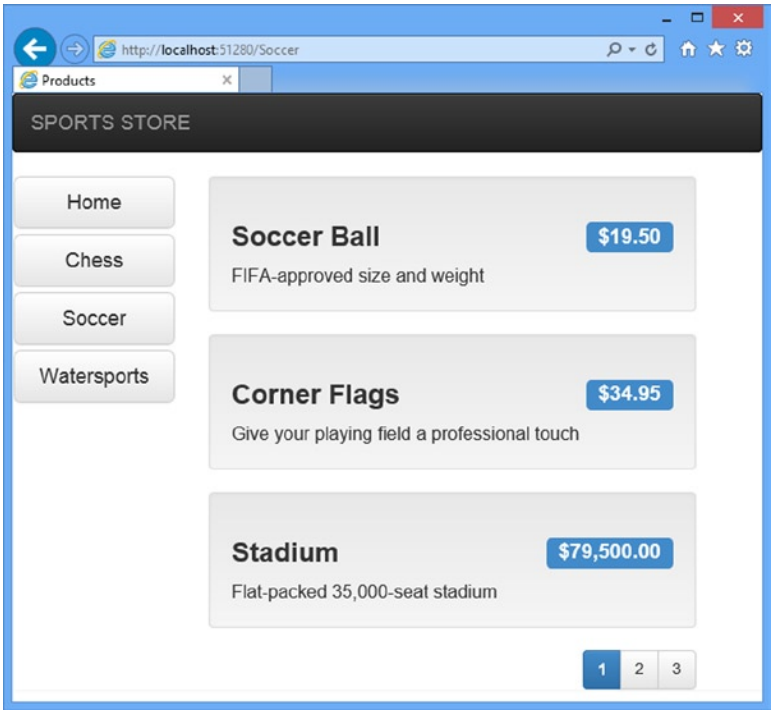


***Figure 8-3.*** *The category links*

## 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. I 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 I would usually do. But for variety I am going to use the view bag feature I introduced in Chapter 2. This feature allows me to pass data from the controller to the view without using a view model. Listing 8-9 shows the changes to the `Menu` action method in the `Nav` controller.

***Listing 8-9.*** Using the View Bag Feature in the NavController.cs File

```
using System.Collections.Generic;
using System.Web.Mvc;
using SportsStore.Domain.Abstract;
using System.Linq;

namespace SportsStore.WebUI.Controllers {

    public class NavController : Controller {
        private IProductRepository repository;

        public NavController(IProductRepository repo) {
            repository = repo;
        }
```

```
        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);
        }
    }
}
```

I 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 I explained in Chapter 2, the `ViewBag` is a dynamic object and I create new properties simply by setting values for them.

## UNIT TEST: REPORTING THE SELECTED CATEGORY

I can test that the `Menu` action method correctly adds details of the selected category by reading the value of the `ViewBag` property in a unit test, which is available through the `ViewResult` class. Here is the test:

```
...
[TestMethod]
public void Indicates_Selected_Category() {

    // Arrange
    // - create the mock repository
    Mock<IProductRepository> mock = new Mock<IProductRepository>();
    mock.Setup(m => m.Products).Returns(new Product[] {
        new Product {ProductID = 1, Name = "P1", Category = "Apples"},
        new Product {ProductID = 4, Name = "P2", Category = "Oranges"},
    });

    // Arrange - create the controller
    NavController target = new NavController(mock.Object);

    // Arrange - define the category to selected
    string categoryToSelect = "Apples";

    // Action
    string result = target.Menu(categoryToSelect).ViewBag.SelectedCategory;

    // Assert
    Assert.AreEqual(categoryToSelect, result);
}
...
```

This unit test will not compile unless you add a reference to the `Microsoft.CSharp` assembly, as described in the previous chapter.

Now that I am providing information about which category is selected, I can update the view to take advantage of this, and add a CSS class to the HTML anchor element that represents the selected category. Listing 8-10 shows the changes to the `Menu.cshtml` file.

***Listing 8-10.*** Highlighting the Selected Category in the Menu.cshtml File

```
@model IEnumerable<string>

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

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

The change is simple. If the current `link` value matches the `SelectedCategory` value, then I add the element I am creating to another Bootstrap class, which will cause the button to be highlighted. Running the application shows the effect of the category highlighting, which you can also see in Figure 8-4.

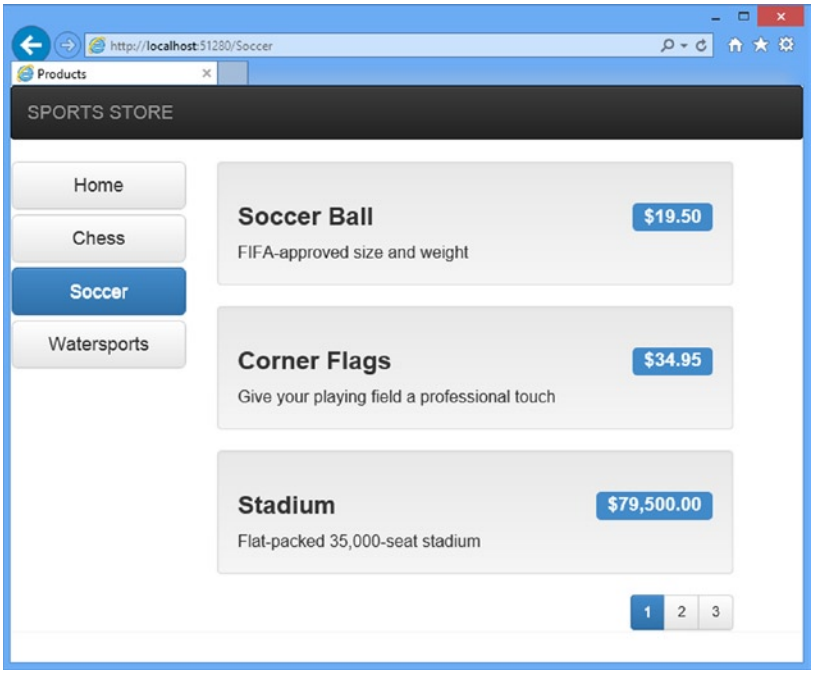


***Figure 8-4.*** *Highlighting the selected category*

## Correcting the Page Count

I need to 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 `Chess` category and end up with an empty page because there are not enough chess products to fill two pages. You can see the problem in Figure 8-5.

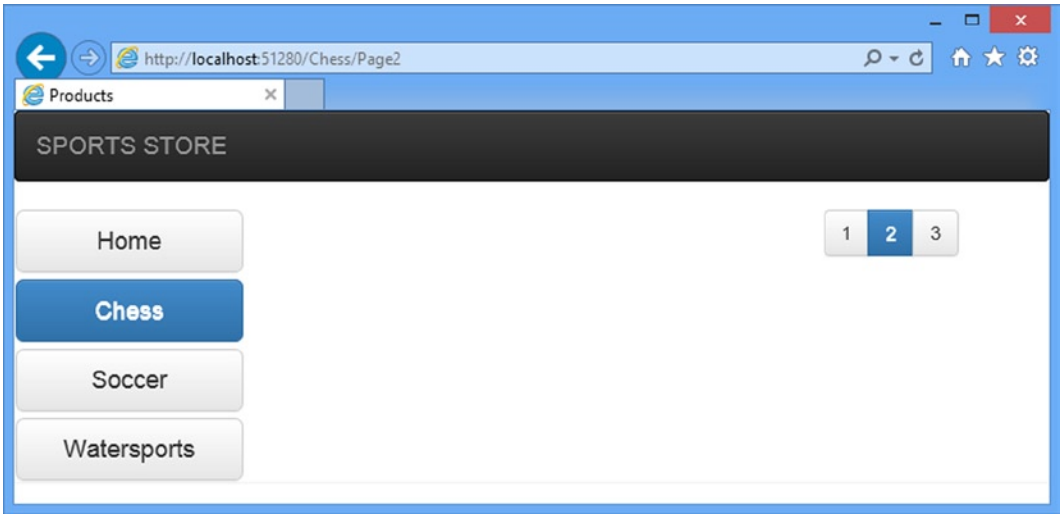


***Figure 8-5.*** *Displaying the wrong page links when a category is selected*

I can fix this by updating the `List` action method in the `Product` controller so that the pagination information takes the categories into account. You can see the required changes in Listing 8-11.

***Listing 8-11.*** Creating Category-Aware Pagination Data in the ProductController.cs File

```
...
public ViewResult List(string category, int page = 1) {

    ProductsListViewModel viewModel = 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(viewModel);
}
...
```

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 I view a category, the links at the bottom of the page correctly reflect the number of products in the category, as shown in Figure 8-6.

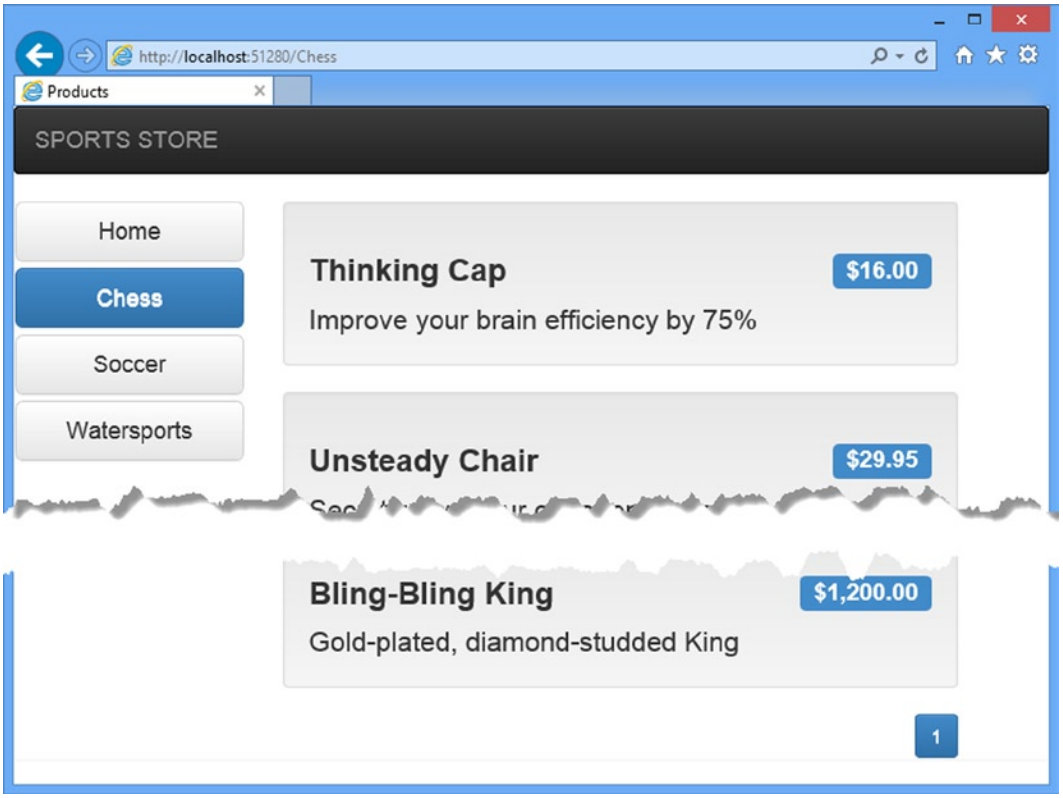


***Figure 8-6.*** *Displaying category-specific page counts*

## UNIT TEST: CATEGORY-SPECIFIC PRODUCT COUNTS

Testing that I am able to generate the current product count for different categories is simple. I create a mock repository that contains known data in a range of categories and then call the `List` action method requesting each category in turn. Here is the unit test:

```
...
[TestMethod]
public void Generate_Category_Specific_Product_Count() {
    // Arrange
    // - create the mock repository
    Mock<IProductRepository> mock = new Mock<IProductRepository>();
```

```
        mock.Setup(m => m.Products).Returns(new Product[] {
            new Product {ProductID = 1, Name = "P1", Category = "Cat1"},
            new Product {ProductID = 2, Name = "P2", Category = "Cat2"},
            new Product {ProductID = 3, Name = "P3", Category = "Cat1"},
            new Product {ProductID = 4, Name = "P4", Category = "Cat2"},
            new Product {ProductID = 5, Name = "P5", Category = "Cat3"}
        });

        // Arrange - create a controller and make the page size 3 items
        ProductController target = new ProductController(mock.Object);
        target.PageSize = 3;

        // Action - test the product counts for different categories
        int res1 = ((ProductsListViewModel)target
            .List("Cat1").Model).PagingInfo.TotalItems;
        int res2 = ((ProductsListViewModel)target
            .List("Cat2").Model).PagingInfo.TotalItems;
        int res3 = ((ProductsListViewModel)target
            .List("Cat3").Model).PagingInfo.TotalItems;
        int resAll = ((ProductsListViewModel)target
            .List(null).Model).PagingInfo.TotalItems;

        // Assert
        Assert.AreEqual(res1, 2);
        Assert.AreEqual(res2, 2);
        Assert.AreEqual(res3, 1);
        Assert.AreEqual(resAll, 5);
    }
    ...
```

Notice that I also call the `List` method, specifying no category, to make sure I get the right total count as well.

# Building the Shopping Cart

The application is progressing nicely, but I cannot sell any products until I implement a shopping cart. In this section, I will create the shopping cart experience shown in Figure 8-7. This will be familiar to anyone who has ever made a purchase online.


***Figure 8-7.*** *The basic shopping cart flow*

An Add to Cart button will be displayed alongside each of the products in the catalog. Clicking this button will show a summary of the products the customer has selected so far, including the total cost. At this point, the user can click the Continue Shopping button to return to the product catalog, or click the Checkout Now button to complete the order and finish the shopping session.

## Defining the Cart Entity

A shopping cart is part of the business domain, so it makes sense to represent a cart by creating an entity in the domain model. Add a class file called Cart.cs to the Entities folder in the SportsStore.Domain project and use it to define the classes shown in Listing 8-12.

***Listing 8-12.*** The Cart and CartLine Classes in the Cart.cs File

```
using System.Collections.Generic;
using System.Linq;

namespace SportsStore.Domain.Entities {

    public class Cart {
        private List<CartLine> lineCollection = new List<CartLine>();

        public void AddItem(Product product, int quantity) {
            CartLine line = lineCollection
                .Where(p => p.Product.ProductID == product.ProductID)
                .FirstOrDefault();

            if (line == null) {
                lineCollection.Add(new CartLine { Product = product,
                    Quantity = quantity });
            } else {
                line.Quantity += quantity;
            }
        }

        public void RemoveLine(Product product) {
            lineCollection.RemoveAll(l => l.Product.ProductID == product.ProductID);
        }

        public decimal ComputeTotalValue() {
            return lineCollection.Sum(e => e.Product.Price * e.Quantity);

        }
        public void Clear() {
            lineCollection.Clear();
        }

        public IEnumerable<CartLine> Lines {
            get { return lineCollection; }
        }
    }
```

```
    public class CartLine {
        public Product Product { get; set; }
        public int Quantity { get; set; }
    }
}
```

The `Cart` class uses the `CartLine` class, defined in the same file, to represent a product selected by the customer and the quantity the user wants to buy. I defined methods to add an item to the cart, remove a previously added item from the cart, calculate the total cost of the items in the cart, and reset the cart by removing all of the items. I also provided a property that gives access to the contents of the cart using an `IEnumerable<CartLine>`. This is all straightforward stuff, easily implemented in C# with the help of a little LINQ.

## UNIT TEST: TESTING THE CART

The `Cart` class is relatively simple, but it has a range of important behaviors that must work properly. A poorly functioning cart would undermine the entire SportsStore application. I have broken down the features and tested them individually. I created a new unit test file in the `SportsStore.UnitTests` project called `CartTests.cs` to contain these tests.

The first behavior relates to when I add an item to the cart. If this is the first time that a given `Product` has been added to the cart, I want a new `CartLine` to be added. Here is the test, including the unit test class definition:

```
using System. Linq;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using SportsStore.Domain.Entities;

namespace SportsStore.UnitTests {
    [TestClass]
    public class CartTests {

        [TestMethod]
        public void Can_Add_New_Lines() {

            // Arrange - create some test products
            Product p1 = new Product { ProductID = 1, Name = "P1" };
            Product p2 = new Product { ProductID = 2, Name = "P2" };

            // Arrange - create a new cart
            Cart target = new Cart();

            // Act
            target.AddItem(p1, 1);
            target.AddItem(p2, 1);
            CartLine[] results = target.Lines.ToArray();

            // Assert
            Assert.AreEqual(results.Length, 2);
            Assert.AreEqual(results[0].Product, p1);
            Assert.AreEqual(results[1].Product, p2);
        }

    }
}
```

However, if the customer has already added a `Product` to the cart, I want to increment the quantity of the corresponding `CartLine` and not create a new one. Here is the test:

```
..
[TestMethod]
public void Can_Add_Quantity_For_Existing_Lines() {

    // Arrange - create some test products
    Product p1 = new Product { ProductID = 1, Name = "P1" };
    Product p2 = new Product { ProductID = 2, Name = "P2" };

    // Arrange - create a new cart
    Cart target = new Cart();

    // Act
    target.AddItem(p1, 1);
    target.AddItem(p2, 1);
    target.AddItem(p1, 10);
    CartLine[] results = target.Lines.OrderBy(c => c.Product.ProductID).ToArray();

    // Assert
    Assert.AreEqual(results.Length, 2);
    Assert.AreEqual(results[0].Quantity, 11);
    Assert.AreEqual(results[1].Quantity, 1);
}
...
```

I also need to check that users can change their mind and remove products from the cart. This feature is implemented by the `RemoveLine` method. Here is the test:

```
...
[TestMethod]
public void Can_Remove_Line() {

    // Arrange - create some test products
    Product p1 = new Product { ProductID = 1, Name = "P1" };
    Product p2 = new Product { ProductID = 2, Name = "P2" };
    Product p3 = new Product { ProductID = 3, Name = "P3" };

    // Arrange - create a new cart
    Cart target = new Cart();
    // Arrange - add some products to the cart
    target.AddItem(p1, 1);
    target.AddItem(p2, 3);
    target.AddItem(p3, 5);
    target.AddItem(p2, 1);
```

```
        // Act
        target.RemoveLine(p2);

        // Assert
        Assert.AreEqual(target.Lines.Where(c => c.Product == p2).Count(), 0);
        Assert.AreEqual(target.Lines.Count(), 2);
    }
    ...
```

The next behavior I want to test is the ability to calculate the total cost of the items in the cart. Here's the test for this behavior:

```
    ...
    [TestMethod]
    public void Calculate_Cart_Total() {

        // Arrange - create some test products
        Product p1 = new Product { ProductID = 1, Name = "P1", Price = 100M};
        Product p2 = new Product { ProductID = 2, Name = "P2" , Price = 50M};

        // Arrange - create a new cart
        Cart target = new Cart();

        // Act
        target.AddItem(p1, 1);
        target.AddItem(p2, 1);
        target.AddItem(p1, 3);
        decimal result = target.ComputeTotalValue();

        // Assert
        Assert.AreEqual(result, 450M);
    }
    ...
```

The final test is simple. I want to ensure that the contents of the cart are properly removed when reset. Here is the test:

```
    ...
    [TestMethod]
    public void Can_Clear_Contents() {

        // Arrange - create some test products
        Product p1 = new Product { ProductID = 1, Name = "P1", Price = 100M };
        Product p2 = new Product { ProductID = 2, Name = "P2", Price = 50M };

        // Arrange - create a new cart
        Cart target = new Cart();

        // Arrange - add some items
        target.AddItem(p1, 1);
        target.AddItem(p2, 1);
```

```
        // Act - reset the cart
        target.Clear();

        // Assert
        Assert.AreEqual(target.Lines.Count(), 0);
    }
    ...
```

Sometimes, as in this case, the code required to test the functionality of a type is longer and more complex than the type itself. Do not let that put you off writing the unit tests. Defects in simple classes can have huge impacts, especially ones that play such an important role as `Cart` does in the example application.

## Adding the Add to Cart Buttons

I need to edit the `Views/Shared/ProductSummary.cshtml` view to add the buttons to the product listings. The changes are shown in Listing 8-13.

***Listing 8-13.*** Adding the Buttons to the Product Summary.cshtml File View

```
@model SportsStore.Domain.Entities.Product

<div class="well">
    <h3>
        <strong>@Model.Name</strong>
        <span class="pull-right label label-primary">@Model.Price.ToString("c")</span>
    </h3>

    @using (Html.BeginForm("AddToCart", "Cart")) {
        <div class="pull-right">
            @Html.HiddenFor(x => x.ProductID)
            @Html.Hidden("returnUrl", Request.Url.PathAndQuery)
            <input type="submit" class="btn btn-success" value="Add to cart" />
        </div>
    }

    <span class="lead"> @Model.Description</span>
</div>
```

I added a Razor block that creates a small HTML form for each product in the listing. When this form is submitted, it will invoke the `AddToCart` action method in the `Cart` controller (which I will implement in just a moment).

■ **Note** By default, the `BeginForm` helper method creates a form that uses the HTTP `POST` method. You can change this so that forms use the `GET` method, but you should think carefully about doing so. The HTTP specification requires that `GET` requests must be *idempotent*, meaning that they must not cause changes, and adding a product to a cart is definitely a change. I have more to say on this topic in Chapter 16, including an explanation of what can happen if you ignore the need for idempotent `GET` requests.

## CREATING MULTIPLE HTML FORMS IN A PAGE

Using the `Html.BeginForm` helper in each product listing means that every `Add to cart` button is rendered in its own separate HTML `form` element. This may be surprising if you have been developing with ASP.NET Web Forms, which imposes a limit of one form per page if you want to use the view state feature or complex controls (which tend to rely on view state). Since ASP.NET MVC does not use view state, there is no limit the number of forms you can create.

Equally, there is no requirement to create a form for each button. However, since each form will post back to the same controller method, but with a different set of parameter values, it is a nice and simple way to deal with the button presses.

# Implementing the Cart Controller

I need a controller to handle the `Add to cart` button presses. Create a new controller called `CartController` in the `SportsStore.WebUI` project and edit the content so that it matches Listing 8-14.

***Listing 8-14.*** The Contents of the CartController.cs File

```
using System.Linq;
using System.Web.Mvc;
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;

namespace SportsStore.WebUI.Controllers {

    public class CartController : Controller {
        private IProductRepository repository;

        public CartController(IProductRepository repo) {
            repository = repo;
        }

        public RedirectToRouteResult AddToCart(int productId, string returnUrl) {
            Product product = repository.Products
                .FirstOrDefault(p => p.ProductID == productId);

            if (product != null) {
                GetCart().AddItem(product, 1);
            }
            return RedirectToAction("Index", new { returnUrl });
        }

        public RedirectToRouteResult RemoveFromCart(int productId, string returnUrl) {
            Product product = repository.Products
                .FirstOrDefault(p => p.ProductID == productId);

            if (product != null) {
                GetCart().RemoveLine(product);
            }
```

```
            return RedirectToAction("Index", new { returnUrl });
        }

        private Cart GetCart() {

            Cart cart = (Cart)Session["Cart"];
            if (cart == null) {
                cart = new Cart();
                Session["Cart"] = cart;
            }
            return cart;
        }
    }
}
```

There are a few points to note about this controller. The first is that I use the ASP.NET session state feature to store and retrieve `Cart` objects. This is the purpose of the `GetCart` method. ASP.NET has a nice session feature that uses cookies or URL rewriting to associate multiple requests from a user together to form a single browsing session. A related feature is session state, which associates data with a session. This is an ideal fit for the `Cart` class. I want each user to have their own cart, and I want the cart to be persistent between requests. Data associated with a session is deleted when a session expires (typically because a user has not made a request for a while), which means that I do not need to manage the storage or life cycle of the `Cart` objects. To add an object to the session state, I set the value for a key on the `Session` object, like this:

```
...
Session["Cart"] = cart;
...
```

To retrieve an object again, I simply read the same key, like this:

```
...
Cart cart = (Cart)Session["Cart"];
...
```

---

■ **Tip** Session state objects are stored in the memory of the ASP.NET server by default, but you can configure a range of different storage approaches, including using a SQL database. See my book, *Pro ASP.NET MVC 5 Platform*, published by Apress in 2014, for details.

---

For the `AddToCart` and `RemoveFromCart` methods, I have used parameter names that match the `input` elements in the HTML forms created in the `ProductSummary.cshtml` view. This allows the MVC Framework to associate incoming form `POST` variables with those parameters, meaning I do not need to process the form myself.

## Displaying the Contents of the Cart

The final point to note about the Cart controller is that both the AddToCart and RemoveFromCart methods call the RedirectToAction method. This has the effect of sending an HTTP redirect instruction to the client browser, asking the browser to request a new URL. In this case, I have asked the browser to request a URL that will call the Index action method of the Cart controller.

I am going to implement the Index method and use it to display the contents of the Cart. If you refer back to Figure 8-7, you will see that this is the workflow when the user clicks the Add to cart button.

I need to pass two pieces of information to the view that will display the contents of the cart: the Cart object and the URL to display if the user clicks the Continue shopping button. I created a new class file called CartIndexViewModel.cs in the Models folder of the SportsStore.WebUI project. The contents of this file are shown in Listing 8-15.

***Listing 8-15.*** The Contents of the CartIndexViewModel.cs File

```
using SportsStore.Domain.Entities;

namespace SportsStore.WebUI.Models {
    public class CartIndexViewModel {
        public Cart Cart { get; set; }
        public string ReturnUrl { get; set; }
    }
}
```

Now that I have the view model, I can implement the Index action method in the Cart controller class, as shown in Listing 8-16.

***Listing 8-16.*** The Index Action Method in the CartController.cs File

```
using System.Linq;
using System.Web.Mvc;
using SportsStore.Domain.Abstract;
using SportsStore.Domain.Entities;
using SportsStore.WebUI.Models;

namespace SportsStore.WebUI.Controllers {

    public class CartController : Controller {
        private IProductRepository repository;

        public CartController(IProductRepository repo) {
            repository = repo;
        }

        public ViewResult Index(string returnUrl) {
            return View(new CartIndexViewModel {
                Cart = GetCart(),
                ReturnUrl = returnUrl
            });
        }
```

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

}
```

The last step to display the contents of the cart is to create the new view. Right-click on the `Index` action method and select `Add View` from the pop-up menu. Set the name to `Index` and click the `OK` button to create the `Index.cshtml` view file. Edit the view to match the contents shown in Listing 8-17.

***Listing 8-17.*** The Contents of the Index.cshtml File

```
@model SportsStore.WebUI.Models.CartIndexViewModel

@{
    ViewBag.Title = "Sports Store: Your Cart";
}

<h2>Your cart</h2>
<table class="table">
    <thead>
        <tr>
            <th>Quantity</th>
            <th>Item</th>
            <th class="text-right">Price</th>
            <th class="text-right">Subtotal</th>
        </tr>
    </thead>
    <tbody>
        @foreach (var line in Model.Cart.Lines) {
            <tr>
                <td class="text-center">@line.Quantity</td>
                <td class="text-left">@line.Product.Name</td>
                <td class="text-right">@line.Product.Price.ToString("c")</td>
                <td class="text-right">
                    @((line.Quantity * line.Product.Price).ToString("c"))
                </td>
            </tr>
        }
    </tbody>
    <tfoot>
        <tr>
            <td colspan="3" class="text-right">Total:</td>
            <td class="text-right">
                @Model.Cart.ComputeTotalValue().ToString("c")
            </td>
        </tr>
    </tfoot>
</table>

<div class="text-center">
    <a class="btn btn-primary" href="@Model.ReturnUrl">Continue shopping</a>
</div>
```

The view enumerates the lines in the cart and adds rows for each of them to an HTML table, along with the total cost per line and the total cost for the cart. The classes I have assigned the elements to correspond to Bootstrap styles for tables and text alignment. I now have the basic functions of the shopping cart in place. First, products are listed along with a button to add them to the cart, as shown in Figure 8-8.

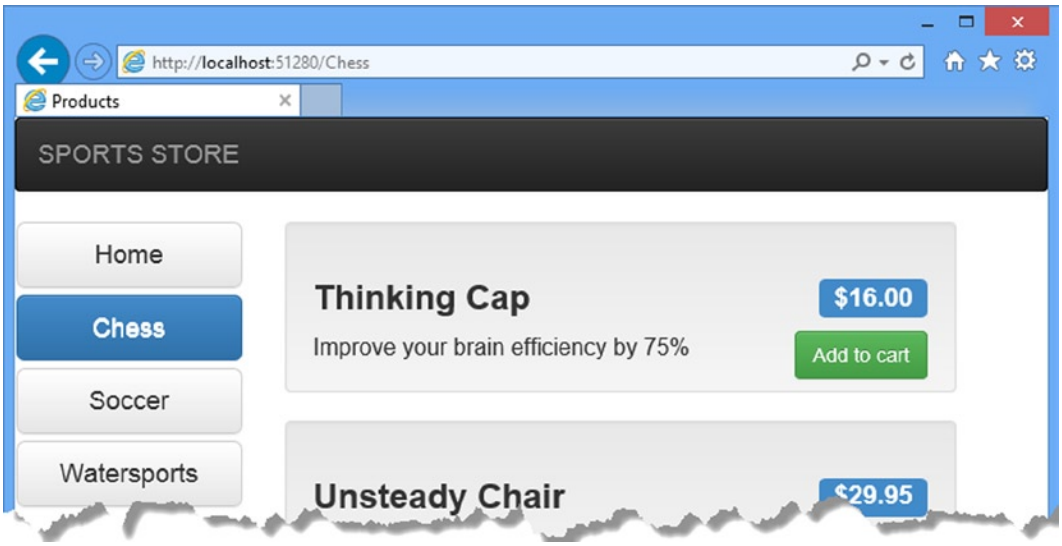


***Figure 8-8.*** *The Add to cart button*

And second, when the user clicks the `Add to cart button`, the appropriate product is added to their cart and a summary of the cart is displayed, as shown in Figure 8-9. And clicking the `Continue shopping` button returns the user to the product page they came from.

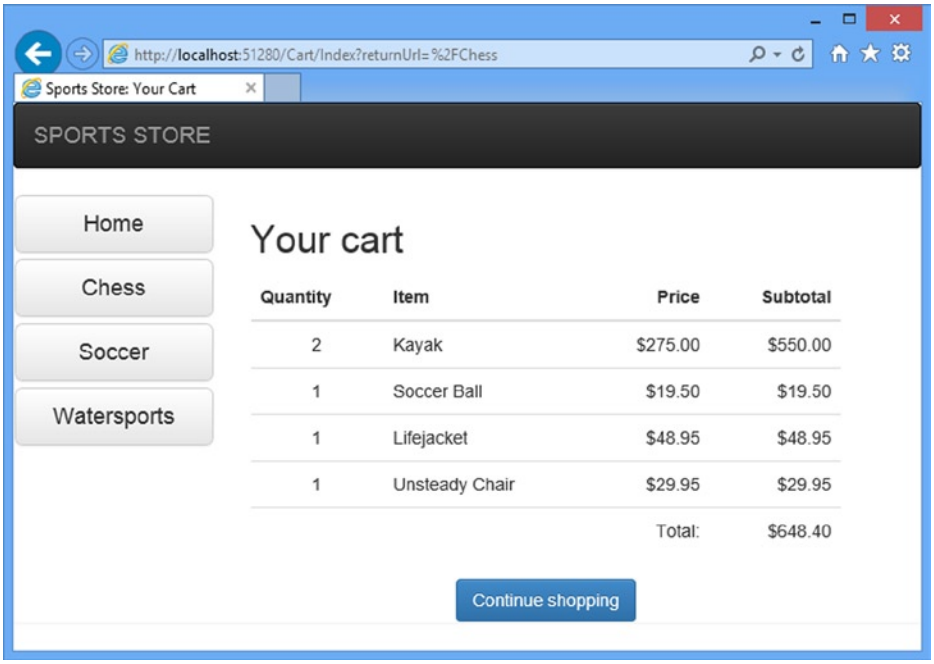


***Figure 8-9.*** *Displaying the contents of the shopping cart*

# Summary

In this chapter, I started to flesh out the customer-facing parts of the SportsStore app. I provided the means by which the user can navigate by category and put the basic building blocks in place for adding items to a shopping cart. I have more work to do and I continue the development of the application in the next chapter.