CHAPTER 9

SportsStore: Navigation

In this chapter, I continue to build out the SportsStore example app. In this chapter, I add support for navigating around the application and start building a shopping cart.

Adding Navigation Controls

The SportsStore application will be more useful 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
- 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 project in the previous 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 9-1 shows the changes I made to the `ProductsListViewModel.cs` file in the `Models/ViewModels` folder.

Listing 9-1. Adding a Property in the ProductsListViewModel .cs File in the Models/ViewModels Folder

```
using System.Collections.Generic;
using SportsStore.Models;

namespace SportsStore.Models.ViewModels {

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

A. Freeman, *Pro ASP.NET Core MVC 2*, https://doi.org/10.1007/978-1-4842-3150-0_9

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. Listing 9-2 shows the changes.

Listing 9-2. Adding Category Support to the List Action in the ProductController.cs File in the Controllers

```
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;
using System.Linq;
using SportsStore.Models.ViewModels;

namespace SportsStore.Controllers {

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

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

        public ViewResult List(string category, int productPage = 1)
            => View(new ProductsListViewModel {
                Products = repository.Products
                    .Where(p => category == null || p.Category == category)
                    .OrderBy(p => p.ProductID)
                    .Skip((productPage - 1) * PageSize)
                    .Take(PageSize),
                PagingInfo = new PagingInfo {
                    CurrentPage = productPage,
                    ItemsPerPage = PageSize,
                    TotalItems = repository.Products.Count()
                },
                CurrentCategory = category
            });
    }
}
```

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 because it doesn't take the category filter into account. 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 in the `ProductControllerTests.cs` file, the action section of the unit test becomes as follows:

```
...
[Fact]
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"}
    }).AsQueryable<Product>());

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

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

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

By using `null` for the `category` argument, I receive all 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 change to the `Can_Send_Pagination_View_Model` test.

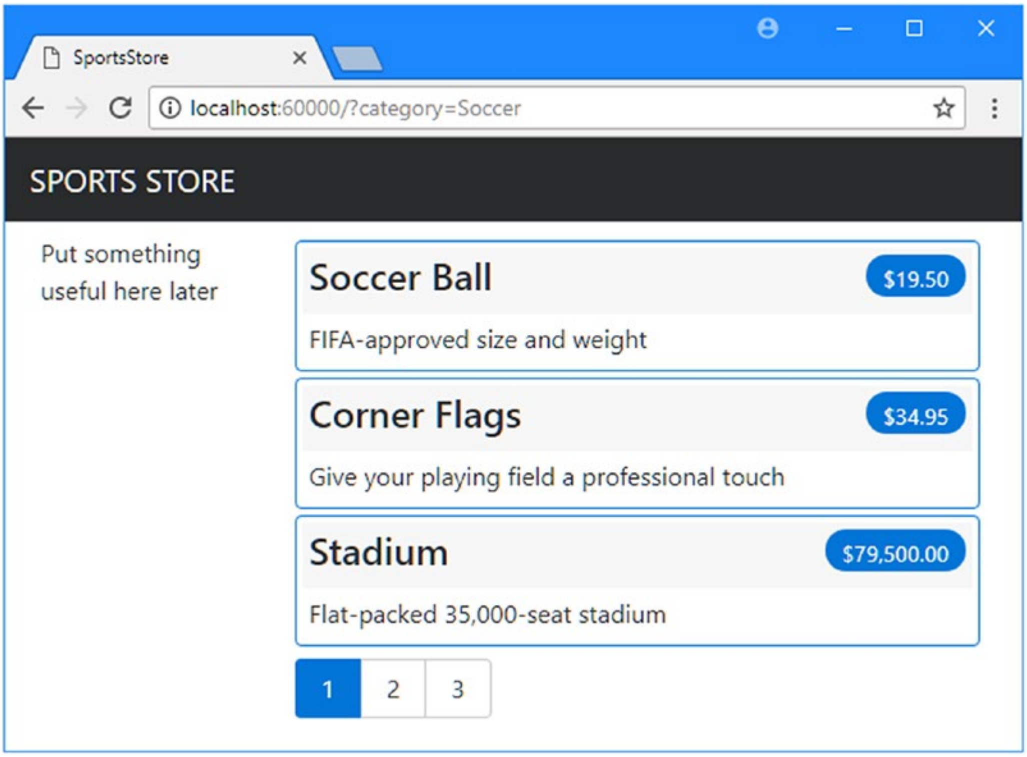
```
...
[Fact]
public void Can_Send_Pagination_View_Model() {
        // Arrange
        Mock<IProductRepository> mock = new Mock<IProductRepository>();
        mock.Setup(m => m.Products).Returns((new Product[] {
            new Product {ProductID = 1, Name = "P1"},
            new Product {ProductID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

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

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

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

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

To see the effect of the category filtering, start the application and select a category using the following query string, changing the port to match the one that Visual Studio assigned for your project (and taking care to use an uppercase S for Soccer):

```
http://localhost:60000/?category=Soccer
```

You will see only the products in the Soccer category, as shown in Figure 9-1.


SportsStore
localhost:60000/?category=Soccer
SPORTS STORE
Put something useful here later
1
2
3

Soccer Ball $19.50
FIFA-approved size and weight

Corner Flags $34.95
Give your playing field a professional touch

Stadium $79,500.00
Flat-packed 35,000-seat stadium

Figure 9-1. *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 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 method I added to the `ProductControllerTests` class:

```
...
[Fact]
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"}
    }).AsQueryable<Product>());

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

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

    // Assert
    Assert.Equal(2, result.Length);
    Assert.True(result[0].Name == "P2" && result[0].Category == "Cat2");
    Assert.True(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 change the routing configuration in the `Configure` method of the `Startup` class to create a more useful set of URLs, as shown in Listing 9-3.

■ **Caution** It is important to add the new routes in Listing 9-3 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.

Listing 9-3. Changing the Routing Schema in the Startup.cs File in the SportsStore Folder

```
...
public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
    app.UseDeveloperExceptionPage();
    app.UseStatusCodePages();
    app.UseStaticFiles();
    app.UseMvc(routes => {

        routes.MapRoute(
            name: null,
            template: "{category}/Page{productPage:int}",
            defaults: new { controller = "Product", action = "List" }
        );
```

```
            routes.MapRoute(
                name: null,
                template: "Page{productPage:int}",
                defaults: new { controller = "Product",
                    action = "List", productPage = 1 }
            );

            routes.MapRoute(
                name: null,
                template: "{category}",
                defaults: new { controller = "Product",
                    action = "List", productPage = 1 }
            );

            routes.MapRoute(
                name: null,
                template: "",
                defaults: new { controller = "Product", action = "List",
                    productPage = 1 });

            routes.MapRoute(name: null, template: "{controller}/{action}/{id?}");
        });
        SeedData.EnsurePopulated(app);
    }
...
```

Table 9-1 describes the URL scheme that these routes represent. I explain the routing system in detail in Chapters 15 and 16.

Table 9-1. *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 Core 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 both to handle incoming requests and to generate outgoing URLs, I can ensure that all the URLs in the application are consistent.

The IUrlHelper interface provides access to the URL-generating functionality. I used this interface and the Action method it defines in the tag helper I created in the previous chapter. Now that I want to start generating more complex URLs, I need a way to receive additional information from the view without having to add extra properties to the tag helper class. Fortunately, tag helpers have a nice feature that allows properties with a common prefix to be received all together in a single collection, as shown in Listing 9-4.

Listing 9-4. Receiving Prefixed Values in the PageLinkTagHelper.cs File in the Infrastructure Folder

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.AspNetCore.Mvc.Routing;
using Microsoft.AspNetCore.Mvc.ViewFeatures;
using Microsoft.AspNetCore.Razor.TagHelpers;
using SportsStore.Models.ViewModels;
using System.Collections.Generic;

namespace SportsStore.Infrastructure {

    [HtmlTargetElement("div", Attributes = "page-model")]
    public class PageLinkTagHelper : TagHelper {
        private IUrlHelperFactory urlHelperFactory;

        public PageLinkTagHelper(IUrlHelperFactory helperFactory) {
            urlHelperFactory = helperFactory;
        }

        [ViewContext]
        [HtmlAttributeNotBound]
        public ViewContext ViewContext { get; set; }

        public PagingInfo PageModel { get; set; }

        public string PageAction { get; set; }

        [HtmlAttributeName(DictionaryAttributePrefix = "page-url-")]
        public Dictionary<string, object> PageUrlValues { get; set; }
            = new Dictionary<string, object>();

        public bool PageClassesEnabled { get; set; } = false;
        public string PageClass { get; set; }
        public string PageClassNormal { get; set; }
        public string PageClassSelected { get; set; }

        public override void Process(TagHelperContext context,
                TagHelperOutput output) {
            IUrlHelper urlHelper = urlHelperFactory.GetUrlHelper(ViewContext);
            TagBuilder result = new TagBuilder("div");
            for (int i = 1; i <= PageModel.TotalPages; i++) {
                TagBuilder tag = new TagBuilder("a");
                PageUrlValues["productPage"] = i;
                tag.Attributes["href"] = urlHelper.Action(PageAction, PageUrlValues);
                if (PageClassesEnabled) {
                    tag.AddCssClass(PageClass);
                    tag.AddCssClass(i == PageModel.CurrentPage
                        ? PageClassSelected : PageClassNormal);
                }
                tag.InnerHtml.Append(i.ToString());
                result.InnerHtml.AppendHtml(tag);
            }
            output.Content.AppendHtml(result.InnerHtml);
        }
    }
}
```

Decorating a tag helper property with the HtmlAttributeName attribute allows me to specify a prefix for attribute names on the element, which in this case will be page-url-. The value of any attribute whose name begins with this prefix will be added to the dictionary that is assigned to the PageUrlValues property, which is then passed to the IUrlHelper.Action method to generate the URL for the href attribute of the a elements that the tag helper produces.

In Listing 9-5, I have added a new attribute to the div element that is processed by the tag helper, specifying the category that will be used to generate the URL. I have added only one new attribute to the view, but any attribute with the same prefix would be added to the dictionary.

Listing 9-5. Adding a New Attribute in the List.cshtml File in the Views/Home Folder

```
@model ProductsListViewModel

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

<div page-model="@Model.PagingInfo" page-action="List" page-classes-enabled="true"
     page-class="btn" page-class-normal="btn-secondary"
     page-class-selected="btn-primary" page-url-category="@Model.CurrentCategory"
     class="btn-group pull-right m-1">
</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 would be lost, and the application would present 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 links 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.

ASP.NET Core MVC has the concept of *view components*, which are perfect for creating items such as a reusable navigation control. A view component is a C# class that provides a small amount of reusable application logic with the ability to select and display Razor partial views. I describe view components in detail in Chapter 22.

In this case, I will create a view component that renders the navigation menu and integrates it into the application by invoking the component from the shared layout. This approach gives me a regular C# class that can contain whatever application logic I need and that can be unit tested like any other class. It is a nice way of creating smaller segments of an application while preserving the overall MVC approach.

Creating the Navigation View Component

I created a folder called `Components`, which is the conventional home of view components, and added to it a class called `NavigationMenuViewComponent.cs`, which I used to define the class shown in Listing 9-6.

Listing 9-6. The Contents of the NavigationMenuViewComponent.cs File in the Components Folder

```
using Microsoft.AspNetCore.Mvc;

namespace SportsStore.Components {

    public class NavigationMenuViewComponent : ViewComponent {

        public string Invoke() {
            return "Hello from the Nav View Component";
        }
    }
}
```

The view component's `Invoke` method is called when the component is used in a Razor view, and the result of the `Invoke` method is inserted into the HTML sent to the browser. I have started with a simple view component that returns a string, but I'll replace this with dynamic HTML content shortly.

I want the category list to appear on all pages, so I am going to use the view component in the shared layout, rather than in a specific view. Within a view, view components are used through the `@await Component.InvokeAsync` expression, as shown in Listing 9-7.

Listing 9-7. Using View Component in the _Layout.cshtml File in the Views/Shared Folder

```
<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <link rel="stylesheet"
          asp-href-include="/lib/bootstrap/dist/**/*.min.css"
```

```
            asp-href-exclude="**/*-reboot*,**/*-grid*" />
    <title>SportsStore</title>
</head>
<body>
    <div class="navbar navbar-inverse bg-inverse" role="navigation">
        <a class="navbar-brand" href="#">SPORTS STORE</a>
    </div>
    <div class="row m-1 p-1">
        <div id="categories" class="col-3">
            @await Component.InvokeAsync("NavigationMenu")
        </div>
        <div class="col-9">
            @RenderBody()
        </div>
    </div>
</body>
</html>
```

I removed the placeholder text and replaced it with a call to the `Component.InvokeAsync` method. The argument to this method is the name of the component class, omitting the `ViewComponent` part of the class name, such that `NavigationMenu` specifies the `NavigationMenuViewComponent` class. If you run the application, you will see that the output from the `Invoke` method is included in the HTML sent to the browser, as shown in Figure 9-2.

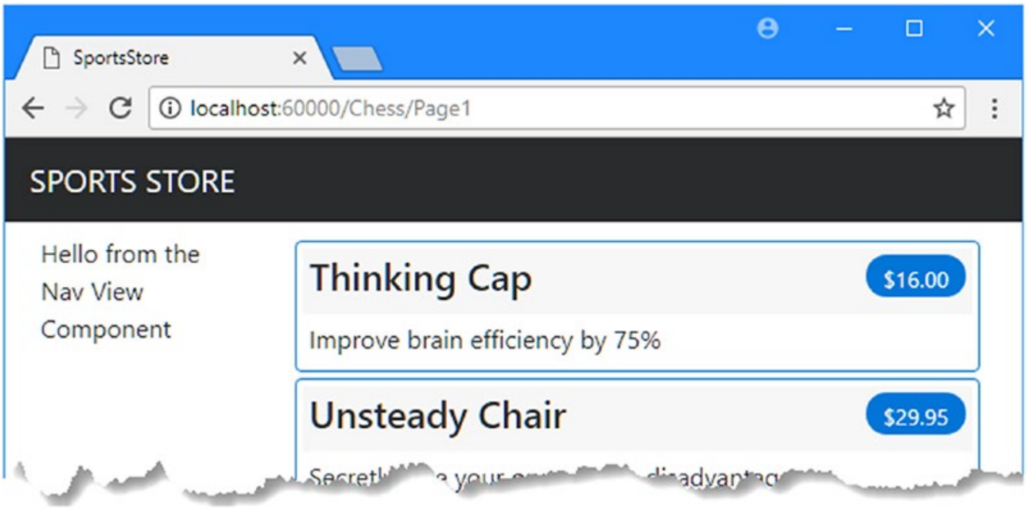

SportsStore
localhost:60000/Chess/Page1
SPORTS STORE
Hello from the Nav View Component
Thinking Cap
$16.00
Improve brain efficiency by 75%
Unsteady Chair
$29.95

Figure 9-2. *Using a view component*

Generating Category Lists

I can now return to the navigation view controller and generate a real set of categories. I could build the HTML for the categories programmatically, as I did for the page tag helper, but one of the benefits of working with view components is they can render Razor partial views. That means I can use the view component to generate the list of components and then use the more expressive Razor syntax to render the HTML that will display them. The first step is to update the view component, as shown in Listing 9-8.

Listing 9-8. Adding Categories in the NavigationMenuViewComponent.cs File in the Components Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Linq;
using SportsStore.Models;

namespace SportsStore.Components {

    public class NavigationMenuViewComponent : ViewComponent {
        private IProductRepository repository;

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

        public IViewComponentResult Invoke() {
            return View(repository.Products
                .Select(x => x.Category)
                .Distinct()
                .OrderBy(x => x));
        }
    }
}
```

The constructor defined in Listing 9-8 defines an `IProductRepository` argument. When MVC needs to create an instance of the view component class, it will note the need to provide this argument and inspect the configuration in the `Startup` class to determine which implementation object should be used. This is the same dependency injection feature that I used in the controller in Chapter 8, and it has the same effect, which is to allow the view component to access data without knowing which repository implementation will be used, as described in Chapter 18.

In the `Invoke` method, I use LINQ to select and order the set of categories in the repository and pass them as the argument to the `View` method, which renders the default Razor partial view, details of which are returned from the method using an `IViewComponentResult` object, a process I describe in more detail in Chapter 22.

UNIT TEST: GENERATING THE CATEGORY LIST

The unit test for my ability to produce a category list is relatively simple. The goal is to create a list that is sorted in alphabetical order and contains no duplicates, and 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 tag helper class, and assert that the data has been properly cleaned up. Here is the unit test, which I defined in a new class file called `NavigationMenuViewComponentTests.cs` in the `SportsStore.Tests` project:

```
using System.Collections.Generic;
using System.Linq;
using Microsoft.AspNetCore.Mvc.ViewComponents;
using Moq;
using SportsStore.Components;
using SportsStore.Models;
using Xunit;
```

```
namespace SportsStore.Tests {

    public class NavigationMenuViewComponentTests {

        [Fact]
        public void Can_Select_Categories() {
            // Arrange
            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"},
            }).AsQueryable<Product>());

            NavigationMenuViewComponent target =
                new NavigationMenuViewComponent(mock.Object);

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

            // Assert
            Assert.True(Enumerable.SequenceEqual(new string[] { "Apples",
                "Oranges", "Plums" }, results));
        }
    }
}
```

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

Razor uses different conventions for dealing with views that are selected by view components. Both the default name of the view and the locations that are searched for the view are different from those used for controllers. To that end, I created the `Views/Shared/Components/NavigationMenu` folder and added to it a view file called `Default.cshtml`, to which I added the content shown in Listing 9-9.

Listing 9-9. Contents of the Default.cshtml File in the Views/Shared/Components/NavigationMenu Folder

```
@model IEnumerable<string>

<a class="btn btn-block btn-secondary"
   asp-action="List"
   asp-controller="Product"
   asp-route-category="">
    Home
</a>
```

```
@foreach (string category in Model) {
    <a class="btn btn-block btn-secondary"
       asp-action="List"
       asp-controller="Product"
       asp-route-category="@category"
       asp-route-productPage="1">
        @category
    </a>
}
```

This view uses one of the built-in tag helpers, which I describe in Chapters 24 and 25, to create a elements whose `href` attribute contains a URL that selects a different product category.

You can see the category links if you run the application, as shown in Figure 9-3. If you click a category, the list of items is updated to show only items from the selected category.

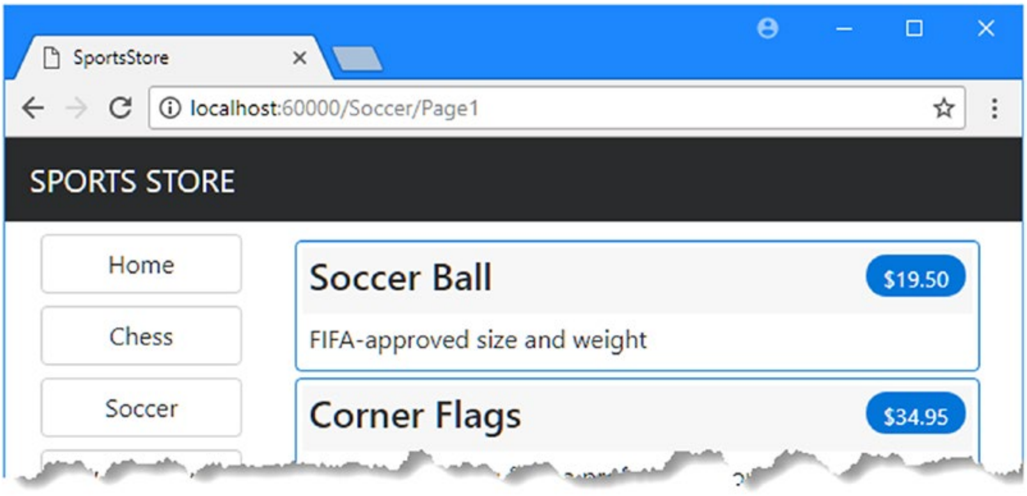


Figure 9-3. *Generating category links with a view component*

Highlighting the Current Category

There is no feedback to the user to indicate which category has been selected. It might be possible to infer the category from the items in the list, but some clear visual feedback seems like a good idea. ASP.NET Core MVC components such as controllers and view components can receive information about the current request by asking for a context object. Most of the time, you can rely on the base classes that you use to create components to take care of getting the context object for you, such as when you use the `Controller` base class to create controllers.

The `ViewComponent` base class is no exception and provides access to context objects through a set of properties. One of the properties is called `RouteData`, which provides information about how the request URL was handled by the routing system.

In Listing 9-10, I use the `RouteData` property to access the request data in order to get the value for the currently selected category. I could pass the category to the view by creating another view model class (and that's what I would do in a real project), but for variety, I am going to use the view bag feature I introduced in Chapter 2.

Listing 9-10. Passing the Selected Category in the NavigationMenuViewComponent.cs File

```
using Microsoft.AspNetCore.Mvc;
using System.Linq;
using SportsStore.Models;

namespace SportsStore.Components {

    public class NavigationMenuViewComponent : ViewComponent {
        private IProductRepository repository;

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

        public IViewComponentResult Invoke() {
            ViewBag.SelectedCategory = RouteData?.Values["category"];
            return View(repository.Products
                .Select(x => x.Category)
                .Distinct()
                .OrderBy(x => x));
        }
    }
}
```

Inside the Invoke method, I have dynamically assigned a SelectedCategory property to the ViewBag object and set its value to be the current category, which is obtained through the context object returned by the RouteData property. As I explained in Chapter 2, the ViewBag is a dynamic object that allows me to define new properties simply by assigning values to them.

UNIT TEST: REPORTING THE SELECTED CATEGORY

I can test that the view component correctly adds details of the selected category by reading the value of the ViewBag property in a unit test, which is available through the ViewViewComponentResult class, described in Chapter 22. Here is the test, which I added to the NavigatioMenuViewComponentTests class:

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

    // Arrange
    string categoryToSelect = "Apples";
    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"},
    }).AsQueryable<Product>());
    NavigationMenuViewComponent target =
        new NavigationMenuViewComponent(mock.Object);
    target.ViewComponentContext = new ViewComponentContext {
        ViewContext = new ViewContext {
            RouteData = new RouteData()
        }
    };
    target.RouteData.Values["category"] = categoryToSelect;

    // Action
    string result = (string)(target.Invoke() as
        ViewViewComponentResult).ViewData["SelectedCategory"];

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

This unit test provides the view component with routing data through the `ViewComponentContext` property, which is how view components receive all of their context data. The `ViewComponentContext` property provides access to view-specific context data through its `ViewContext` property, which in turns provides access to the routing information through its `RouteData` property. Most of the code in the unit test goes into creating the context objects that will provide the selected category in the same way that it would be presented when the application is running and the context data is provided by ASP.NET Core MVC.

Now that I am providing information about which category is selected, I can update the view selected by the view component to take advantage of this and vary the CSS classes used to style the links to make the one representing the current category distinct from the others. Listing 9-11 shows the change I made to the `Default.cshtml` file.

Listing 9-11. Highlighting in the Default.cshtml File in the Views/Shared/Components/NavigationMenu Folder

```
@model IEnumerable<string>

<a class="btn btn-block btn-secondary"
   asp-action="List"
   asp-controller="Product"
   asp-route-category="">
    Home
</a>

@foreach (string category in Model) {
    <a class="btn btn-block
       @(category == ViewBag.SelectedCategory ? "btn-primary": "btn-secondary")"
       asp-action="List"
       asp-controller="Product"
       asp-route-category="@category"
       asp-route-productPage="1">
        @category
    </a>
}
```

I have used a Razor expression within the `class` attribute to apply the `btn-primary` class to the element that represents the selected category and the `btn-secondary` class otherwise. These classes apply different Bootstrap styles and make the active button obvious, as shown in Figure 9-4.

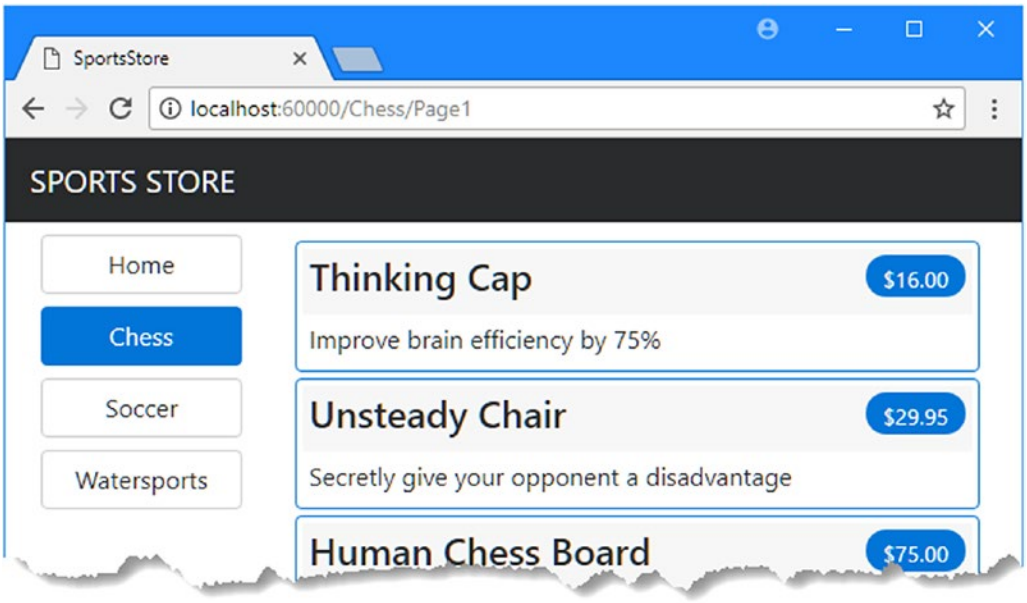


Figure 9-4. *Highlighting the selected category*

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 9-5.

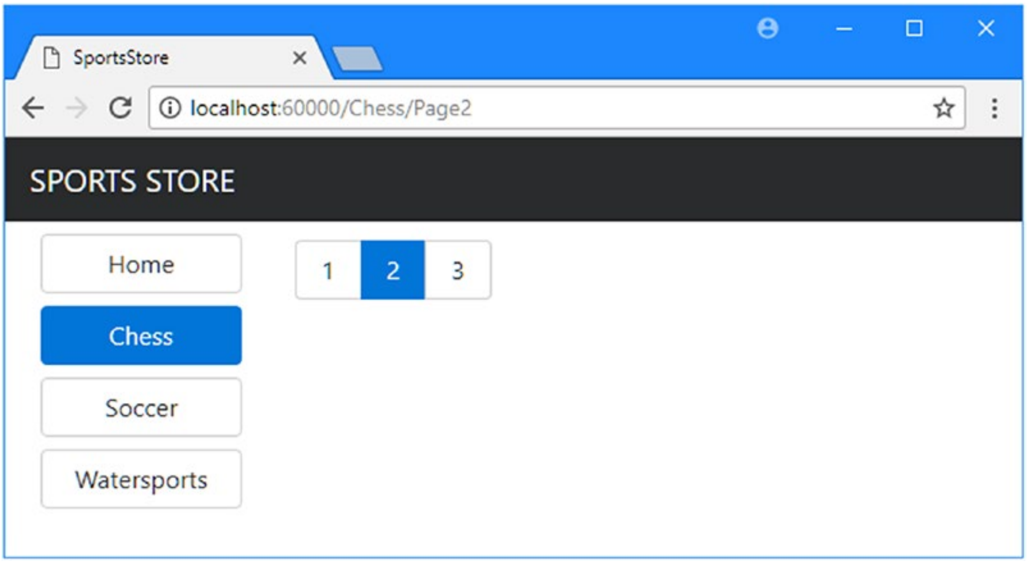


Figure 9-5. *Displaying the wrong page links when a category is selected*

I can fix this by updating the `List` action method in the `Product` controller so that the pagination information takes the categories into account, as shown in Listing 9-12.

Listing 9-12. Creating Category Pagination Data in the ProductController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;
using System.Linq;
using SportsStore.Models.ViewModels;

namespace SportsStore.Controllers {

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

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

        public ViewResult List(string category, int productPage = 1)
            => View(new ProductsListViewModel {
                Products = repository.Products
                    .Where(p => category == null || p.Category == category)
                    .OrderBy(p => p.ProductID)
                    .Skip((productPage - 1) * PageSize)
                    .Take(PageSize),
                PagingInfo = new PagingInfo {
                    CurrentPage = productPage,
                    ItemsPerPage = PageSize,
                    TotalItems = category == null ?
                        repository.Products.Count() :
                        repository.Products.Where(e =>
                            e.Category == category).Count()
                },
                CurrentCategory = category
            });
        }
    }
```

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 9-6.

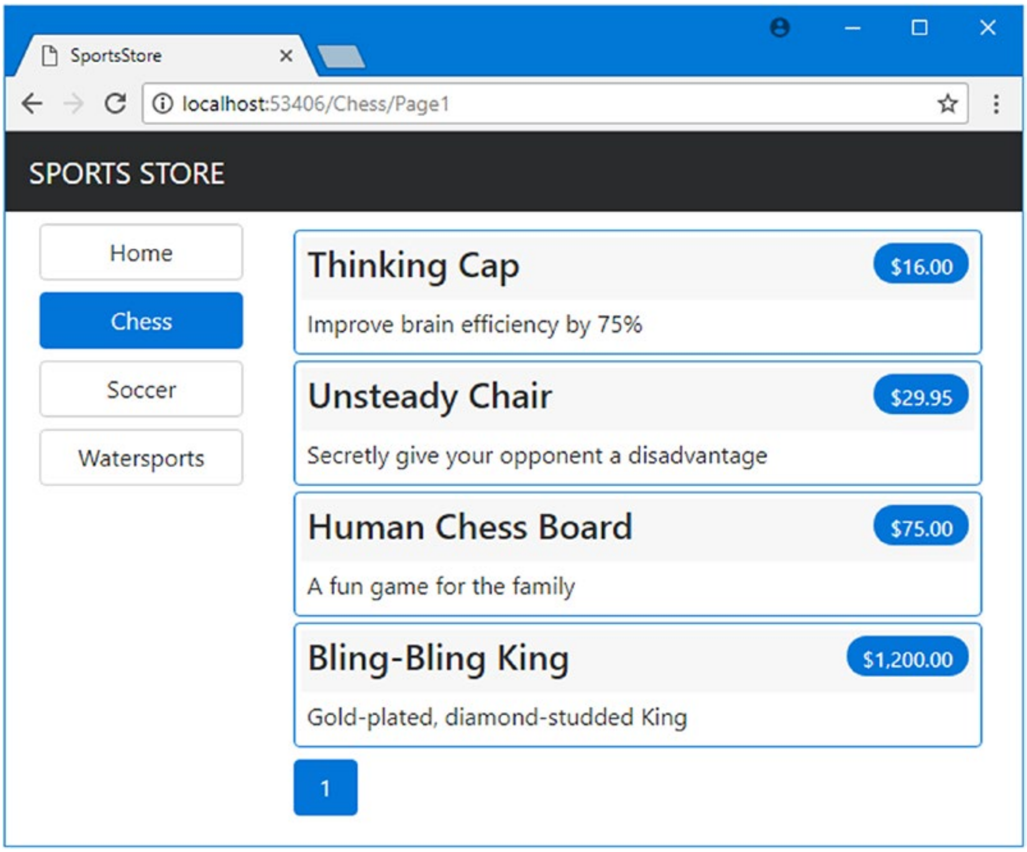

SportsStore
localhost:53406/Chess/Page1
SPORTS STORE
Home
Chess
Soccer
Watersports
Thinking Cap
$16.00
Improve brain efficiency by 75%
Unsteady Chair
$29.95
Secretly give your opponent a disadvantage
Human Chess Board
$75.00
A fun game for the family
Bling-Bling King
$1,200.00
Gold-plated, diamond-studded King
1

Figure 9-6. *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 method that I added to the `ProductControllerTests` class:

```
...
[Fact]
public void Generate_Category_Specific_Product_Count() {
    // Arrange
    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"}
    }).AsQueryable<Product>());

    ProductController target = new ProductController(mock.Object);
    target.PageSize = 3;

    Func<ViewResult, ProductsListViewModel> GetModel = result =>
        result?.ViewData?.Model as ProductsListViewModel;

    // Action
    int? res1 = GetModel(target.List("Cat1"))?.PagingInfo.TotalItems;
    int? res2 = GetModel(target.List("Cat2"))?.PagingInfo.TotalItems;
    int? res3 = GetModel(target.List("Cat3"))?.PagingInfo.TotalItems;
    int? resAll = GetModel(target.List(null))?.PagingInfo.TotalItems;

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

Notice that I also call the `List` method, specifying no category, to make sure I get the correct 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 9-7. This will be familiar to anyone who has ever made a purchase online.

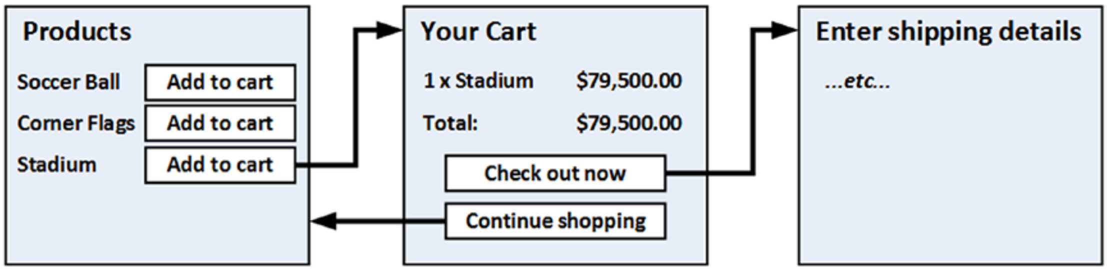


Figure 9-7. *The basic shopping cart flow*

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 Model

I started by adding a class file called `Cart.cs` to the `Models` folder in and used it to define the classes shown in Listing 9-13.

Listing 9-13. The Contents of the Cart.cs File in the Models Folder

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

namespace SportsStore.Models {

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

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

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

        public virtual void Clear() => lineCollection.Clear();

        public virtual IEnumerable<CartLine> Lines => lineCollection;
    }

    public class CartLine {
        public int CartLineID { get; set; }
        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 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 called `CartTests.cs` in the `SportsStore.Tests` project called 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 SportsStore.Models;
using Xunit;

namespace SportsStore.Tests {

    public class CartTests {
        [Fact]
        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.Equal(2, results.Length);
            Assert.Equal(p1, results[0].Product);
            Assert.Equal(p2, results[1].Product);
        }
    }
}
```

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:

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

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:

```
...
[Fact]
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.Equal(0, target.Lines.Where(c => c.Product == p2).Count());
    Assert.Equal(2, target.Lines.Count());
}
...
```

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:

```
...
[Fact]
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.Equal(450M, result);
}
...
```

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

```
...
[Fact]
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.Equal(0, target.Lines.Count());
}
...
```

Sometimes, as in this case, the code required to test the functionality of a class is longer and more complex than the class 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` partial view to add the buttons to the product listings. To prepare for this, I added a class file called `UrlExtensions.cs` to the `Infrastructure` folder and defined the extension method shown in Listing 9-14.

Listing 9-14. The Contents of the UrlExtensions.cs File in the Infrastructure Folder

```
using Microsoft.AspNetCore.Http;

namespace SportsStore.Infrastructure {

    public static class UrlExtensions {

        public static string PathAndQuery(this HttpRequest request) =>
            request.QueryString.HasValue
                ? $"{request.Path}{request.QueryString}"
                : request.Path.ToString();
    }
}
```

The PathAndQuery extension method operates on the HttpRequest class, which ASP.NET uses to describe an HTTP request. The extension method generates a URL that the browser will be returned to after the cart has been updated, taking into account the query string if there is one. In Listing 9-15, I have added the namespace that contains the extension method to the view imports file so that I can use it in the partial view.

Listing 9-15. Adding a Namespace in the _ViewImports.cshtml File in the Views Folder

```
@using SportsStore.Models
@using SportsStore.Models.ViewModels
@using SportsStore.Infrastructure
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@addTagHelper SportsStore.Infrastructure.*, SportsStore
```

In Listing 9-16, I have updated the partial view that describes each product to contain an Add To Cart button.

Listing 9-16. Adding the Buttons to the ProductSummary.cshtml File View in the Views/Shared Folder

```
@model Product

<div class="card card-outline-primary m-1 p-1">
    <div class="bg-faded p-1">
        <h4>
            @Model.Name
            <span class="badge badge-pill badge-primary" style="float:right">
                <small>@Model.Price.ToString("c")</small>
            </span>
        </h4>
    </div>
    <form id="@Model.ProductID" asp-action="AddToCart"
            asp-controller="Cart" method="post">
        <input type="hidden" asp-for="ProductID" />
        <input type="hidden" name="returnUrl"
               value="@ViewContext.HttpContext.Request.PathAndQuery()" />
        <span class="card-text p-1">
            @Model.Description
            <button type="submit"
                class="btn btn-success btn-sm pull-right" style="float:right">
                Add To Cart
            </button>
        </span>
    </form>
</div>
```

I have added a form element that contains hidden input elements specifying the ProductID value from the view model and the URL that the browser should be returned to after the cart has been updated. The form element and one of the input elements are configured using built-in tag helpers, which are a useful way of generating forms that contain model values and that target controllers and actions in the application, as described in Chapter 24. The other input element uses the extension method I created to set the return URL. I also added a button element that will submit the form to the application.

■ **Note** Notice that I have set the `method` attribute on the form element to `post`, which instructs the browser to submit the form data using an HTTP `POST` request. 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.

Enabling Sessions

I am going to store details of a user's cart using session state, which is data that is stored at the server and associated with a series of requests made by a user. ASP.NET provides a range of different ways to store session state, including storing it in memory, which is the approach that I am going to use. This has the advantage of simplicity, but it means that the session data is lost when the application is stopped or restarted. Enabling sessions requires adding services and middleware in the `Startup` class, as shown in Listing 9-17.

Listing 9-17. Enabling Sessions in the Startup.cs File in the SportsStore Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using SportsStore.Models;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;

namespace SportsStore {

    public class Startup {

        public Startup(IConfiguration configuration) =>
            Configuration = configuration;

        public IConfiguration Configuration { get; }

        public void ConfigureServices(IServiceCollection services) {
            services.AddDbContext<ApplicationDbContext>(options =>
                options.UseSqlServer(
                    Configuration["Data:SportStoreProducts:ConnectionString"]));
            services.AddTransient<IProductRepository, EFProductRepository>();
            services.AddMvc();
            services.AddMemoryCache();
            services.AddSession();
        }
        public void Configure(IApplicationBuilder app, IHostingEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseSession();
            app.UseMvc(routes => {

                // ...routing configuration omitted for brevity...

            });
            SeedData.EnsurePopulated(app);
        }
    }
}
```

The AddMemoryCache method call sets up the in-memory data store. The AddSession method registers the services used to access session data, and the UseSession method allows the session system to automatically associate requests with sessions when they arrive from the client.

Implementing the Cart Controller

I need a controller to handle the Add to Cart button presses. I added a new class file called CartController.cs to the Controllers folder and used it to define the class shown in Listing 9-18.

Listing 9-18. The Contents of the CartController.cs File in the Controllers Folder

```
using System.Linq;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using SportsStore.Infrastructure;
using SportsStore.Models;

namespace SportsStore.Controllers {

    public class CartController : Controller {
        private IProductRepository repository;

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

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

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

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

            if (product != null) {
                Cart cart = GetCart();
                cart.RemoveLine(product);
                SaveCart(cart);
            }
            return RedirectToAction("Index", new { returnUrl });
        }

        private Cart GetCart() {
            Cart cart = HttpContext.Session.GetJson<Cart>("Cart") ?? new Cart();
            return cart;
        }

        private void SaveCart(Cart cart) {
            HttpContext.Session.SetJson("Cart", 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, which is the purpose of the GetCart method. The middleware that I registered in the previous section 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.

For the AddToCart and RemoveFromCart action methods, I have used parameter names that match the input elements in the HTML forms created in the ProductSummary.cshtml view. This allows MVC to associate incoming form POST variables with those parameters, meaning I do not need to process the form myself. This is known as *model binding* and is a powerful tool for simplifying controller classes, as I explain in Chapter 26.

Defining Session State Extension Methods

The session state feature in ASP.NET Core stores only int, string, and byte[] values. Since I want to store a Cart object, I need to define extension methods to the ISession interface, which provides access to the session state data to serialize Cart objects into JSON and convert them back. I added a class file called SessionExtensions.cs to the Infrastructure folder and defined the extension methods shown in Listing 9-19.

Listing 9-19. The Contents of the SessionExtensions.cs File in the Infrastructure Folder

```
using Microsoft.AspNetCore.Http;
using Newtonsoft.Json;

namespace SportsStore.Infrastructure {

    public static class SessionExtensions {

        public static void SetJson(this ISession session, string key, object value) {
            session.SetString(key, JsonConvert.SerializeObject(value));
        }

        public static T GetJson<T>(this ISession session, string key) {
            var sessionData = session.GetString(key);
            return sessionData == null
            ? default(T) : JsonConvert.DeserializeObject<T>(sessionData);
        }
    }
}
```

These methods rely on the Json.Net package to serialize objects into the JavaScript Object Notation format, which you will encounter again in Chapter 20. The Json.Net package doesn't have to be added to the project because it is already used behind the scenes by MVC to provide the JSON helper feature, as described in Chapter 21. (See `www.newtonsoft.com/json` for information on working directly with Json.Net.)

The extension methods make it easy to store and retrieve `Cart` objects. To add a `Cart` to the session state in the controller, I make an assignment like this:

```
...
HttpContext.Session.SetJson("Cart", cart);
...
```

The `HttpContext` property is provided the `Controller` base class from which controllers are usually derived and returns an `HttpContext` object that provides context data about the request that has been received and the response that is being prepared. The `HttpContext.Session` property returns an object that implements the `ISession` interface, which is the type on which I defined the `SetJson` method, which accepts arguments that specify a key and an object that will be added to the session state. The extension method serializes the object and adds it to the session state using the underlying functionality provided by the `ISession` interface.

To retrieve the `Cart` again, I use the other extension method, specifying the same key, like this:

```
...
Cart cart = HttpContext.Session.GetJson<Cart>("Cart");
...
```

The type parameter lets me specify the type that I expecting to be retrieved, which is used in the deserialization process.

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 9-7, 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/ViewModels folder of the SportsStore project and used it to define the class shown in Listing 9-20.

Listing 9-20. The Contents of the CartIndexViewModel.cs File in the Models/ViewModels Folder

```
using SportsStore.Models;

namespace SportsStore.Models.ViewModels {

    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 9-21.

Listing 9-21. Implementing the Index Action Method in the CartController.cs File in the Controllers Folder

```
using System.Linq;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using SportsStore.Infrastructure;
using SportsStore.Models;
using SportsStore.Models.ViewModels;

namespace SportsStore.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 methods omitted for brevity...
    }
}
```

The Index action retrieves the Cart object from the session state and uses it to create a CartIndexViewModel object, which is then passed to the View method to be used as the view model.

The last step to display the contents of the cart is to create the view that the Index action will render. I created the Views/Cart folder and added to it a Razor view file called Index.cshtml with the markup shown in Listing 9-22.

Listing 9-22. The Contents of the Index.cshtml File in the Views/Cart Folder

```
@model CartIndexViewModel

<h2>Your cart</h2>
<table class="table table-bordered table-striped">
    <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 to which I assigned the elements correspond to Bootstrap styles for tables and text alignment.

The result is that the basic functions of the shopping cart are in place. First, products are listed along with a button to add them to the cart, as shown in Figure 9-8.

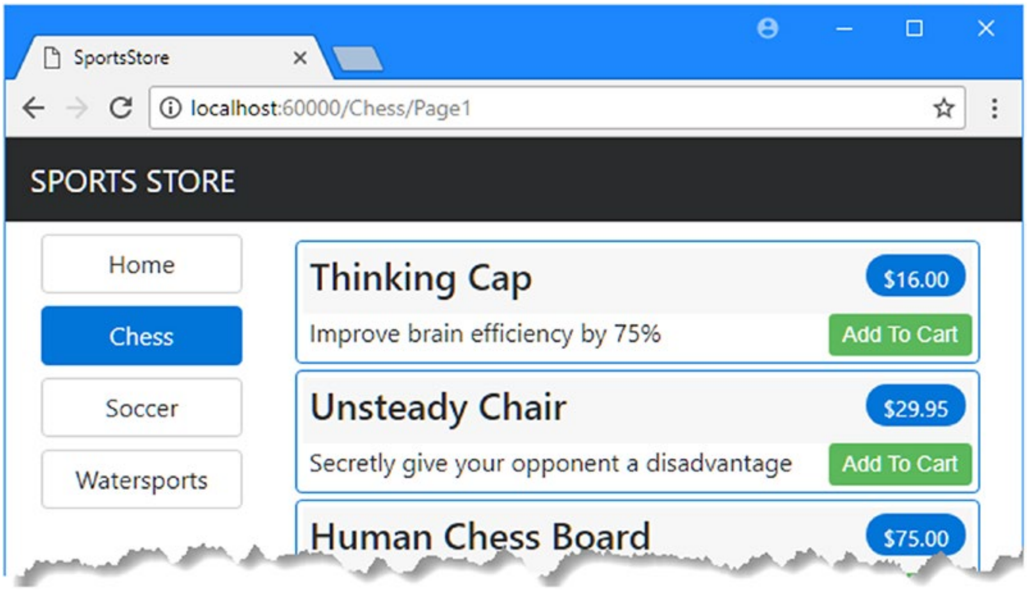
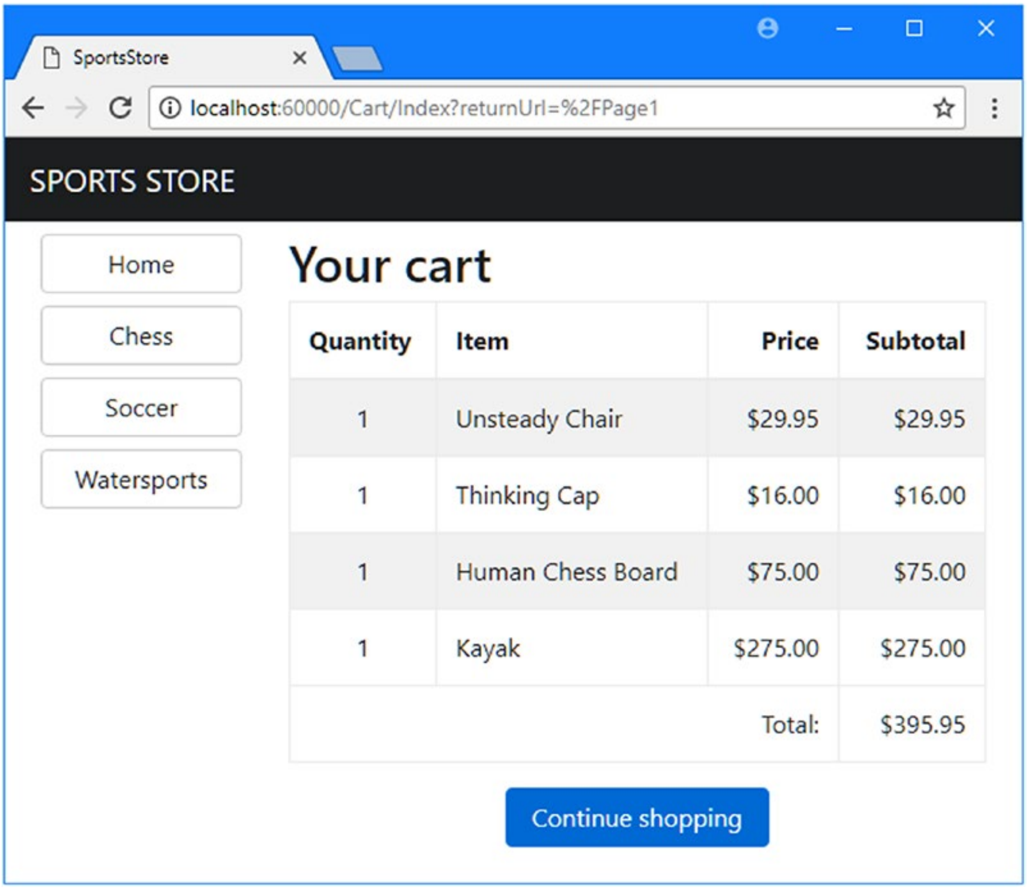


Figure 9-8. *The Add to Cart button*

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 9-9. Clicking the Continue Shopping button returns the user to the product page they came from.



Your cart

Quantity	Item	Price	Subtotal
1	Unsteady Chair	$29.95	$29.95
1	Thinking Cap	$16.00	$16.00
1	Human Chess Board	$75.00	$75.00
1	Kayak	$275.00	$275.00
		Total:	$395.95

Figure 9-9. *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.