CHAPTER 10

SportsStore: Mobile

There is no escaping the popularity of devices such as smartphones and tablets, and if you want to deliver your application to the widest possible audience, you will have to embrace the world of mobile web browsers. If I sound less than enthusiastic, it is because the phrase *mobile web browsers* runs the gamut from fast, capable, and modern browsers that can rival a decent desktop browser right through to the slow, inconsistent, and outdated.

The bottom line is that delivering a good experience to mobile users is *hard*—much harder than delivering content just to desktops. It takes careful planning, design and an enormous amount of testing—and even then it is easy to be caught out by a new smartphone or tablet.

Putting Mobile Web Development in Context

The MVC Framework does have some features that can help with mobile development. But the MVC Framework is a server-side framework that receives HTTP requests and generates HTML responses and there is a limited amount it can do to deal with the wide variation in capabilities that you will encounter when targeting mobile clients. The degree to which The MVC Framework can help depends on the mobile strategy that you have adopted. There are three basic mobile web strategies you can follow, which I describe the following sections.

Tip There is a fourth option, which is to create a native application, but I don't discuss that here since it doesn't directly involve the MVC Framework or, for that matter, web applications.

Doing Nothing (Or As Little As Possible)

It may seem like an odd idea to do nothing, but some mobile devices are capable of handling content that has been developed for desktop clients. Many—admittedly the most recent—have high-resolution, high-density displays with plenty of memory and browsers that can render HTML and run JavaScript quickly. If your app isn't too demanding, you may find that many mobile devices won't encounter any problems at all displaying your application content. As an example, Figure 10-1 shows how an iPad displays the SportsStore application without any modifications.

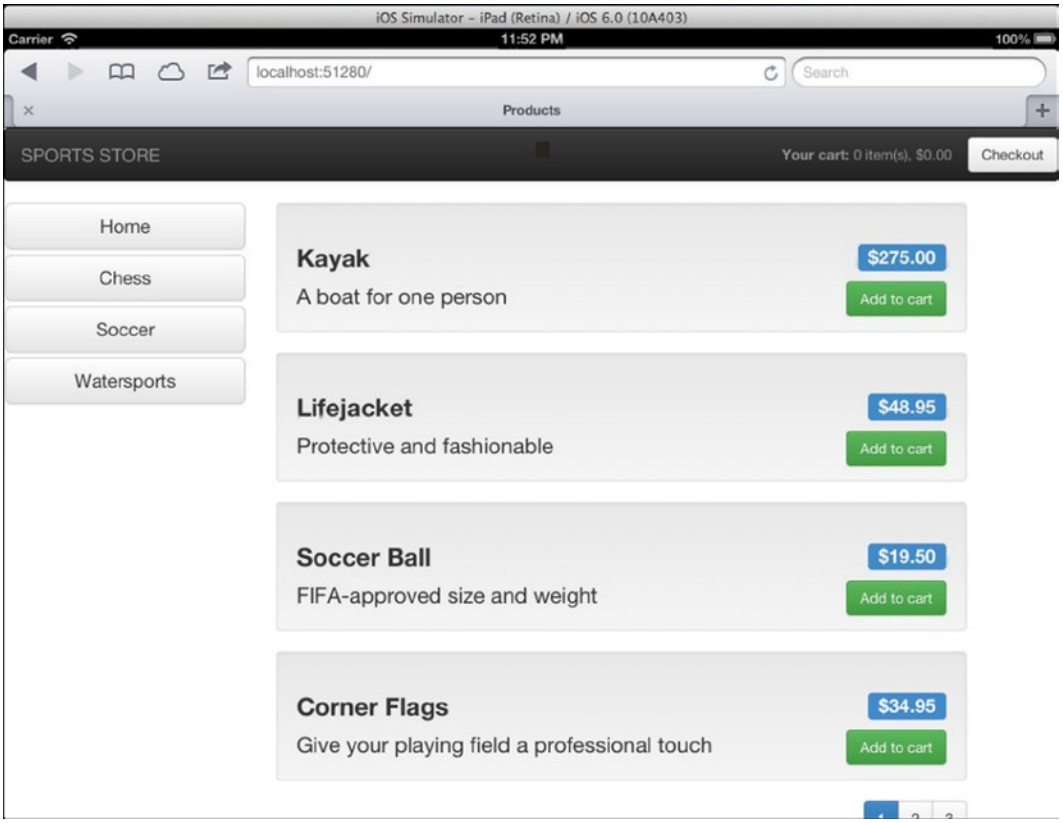

Figure 10-1. *Displaying the SportsStore application on a tablet*

It does pretty well. The only issue is that the pagination links are off the bottom of the page, which is easily adjusted by changing the page layout or changing the number of products that are displayed on a page.

■ **Note** The screenshots I show in this chapter are all obtained using `browserstack.com`, which is the cross-platform browser testing service I use for my own projects. It isn't a perfect service. It is often slow to use at peak times, can be fragile from outside of the US, and the mobile devices are emulated. I use it mainly for the desktop browser support, which is more robust, but I get decent results and I don't have to maintain my own set of emulators. You can get a free trial to follow the examples in this chapter and there are plenty of competitors out there if you want to go elsewhere. Note that I don't have any relationship with Browser Stack other than as a regular customer, for which I pay full price and receive no special treatment.

Using Responsive Design

The next strategy is to create content that adapts to the capabilities of the device on which it is being displayed, known as *responsive design*. The CSS standard has features that let you change the styling applied to elements based on the capabilities of the device, a technique that is most frequently used to alter the layout of content based on screen width.

Responsive design is something that is handled by the client using CSS and not directly managed by the MVC Framework. I go into the topic of responsive design in depth in my *Pro ASP.NET MVC 5 Client* book, but to give a demonstration how the technique can be applied (and some considerations that *do* touch on the MVC Framework), I am going to use some responsive features that are included in the Bootstrap library, which I have been using to style the SportsStore application (and which has become one of the libraries Microsoft includes in the MVC 5 project templates for Visual Studio 2013).

My goal will be to adjust the layout of the main part of the application so that it is visible on an iPhone. My "do nothing" strategy doesn't pay off for this kind of device because it has a narrow screen, as Figure 10-2 shows.

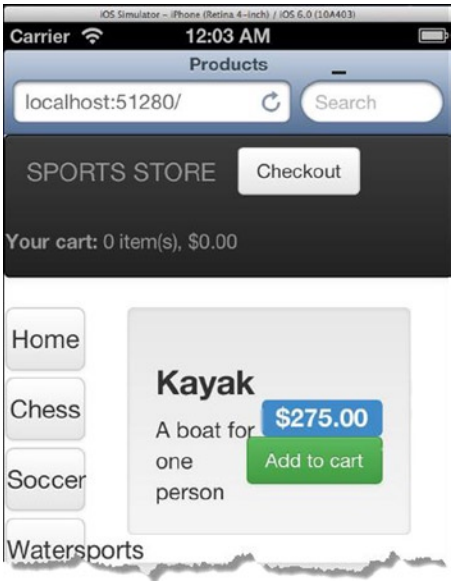

Figure 10-2. *Displaying the SportsStore application on a smartphone*

I am going to tackle this problem in sections, focusing on different aspects of the layout. My goal is to preserve all of the functionality of the application, but presented in a different way.

■ **Note** The MVC Framework isn't an active participant in the responsive design. It sends all browsers the same content and lets them figure out which bits they need to display. This means that there is no sensible way to add unit tests for responsive design to a Visual Studio project. It is a technique that requires careful testing in the client and is difficult to automate.

Creating a Responsive Header

I am going to start with the page header, which contains the SportsStore name, the cart summary, and the Checkout button. Although the simplest solution would be to remove the SportsStore name and free up enough space for the rest of the content, I am going to keep it there (see the *Accepting the Realities of Branding* sidebar) and rearrange everything across two lines.

ACCEPTING THE REALITIES OF BRANDING

One of the easiest ways to free up screen real-estate is to remove your branding from the application. I am only displaying the SportsStore name as text, but you can see how much of the screen it occupies. What was a modest degree of branding on the desktop becomes a space hog on a smart phone.

Removing branding is difficult, however. Not for technical reasons, but because most branding teams are obsessed with slathering branding on everything. It is the reason that there are company-branded pens in the boardroom, company-branded cups in the break room, and why you get business cards with new logos every 18 months. Companies rebrand so often because people who work in branding know, deep down, that they don't have real jobs and the constant emphasis on logos and color schemes creates a frenzy of activity that distracts them from the creeping existential dread that haunts their every waking moment.

My advice is to accept that a certain amount of screen space will always be given over to branding, even on the smallest and least capable device. You might try to fight against the idea, but the branding team is usually part of the marketing department, marketing usually reports to the VP of Sales, and the VP of Sales has a hot line to the CEO because revenue is all that the board cares about. Some arguments just can't be won.

In Listing 10-1, you can see how I have adjusted the content of the header in the _Layout.cshtml file in the SportsStore.WebUI project.

Listing 10-1. Adding Responsive Content 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" />
    <link href="~/Content/ErrorStyles.css" rel="stylesheet" />
    <title>@ViewBag.Title</title>
    <style>
        .navbar-right {
            float: right !important;
            margin-right: 15px; margin-left: 15px;
        }
    </style>
</head>
<body>
    <div class="navbar navbar-inverse">
        <a class="navbar-brand" href="#">
            <span class="hidden-xs">SPORTS STORE</span>
```

```
            <div class="visible-xs">SPORTS</div>
            <div class="visible-xs">STORE</div>
        </a>
        @Html.Action("Summary", "Cart")
    </div>
    <div class="row panel">
        <div id="categories" class="col-xs-3">
            @Html.Action("Menu", "Nav")
        </div>
        <div class="col-xs-8">
            @RenderBody()
        </div>
    </div>
</body>
</html>
```

Bootstrap defines a set of classes that can be used to show or hide elements based on the width of the device screen. This is something you can do manually using CSS *media queries*, but the Bootstrap classes are integrated into the other styles.

For the SportsStore branding, I have used the `visible-xs` and `hidden-xs` classes to switch to text on two lines that will be shown vertically when the window size is below 768 pixels. Bootstrap provides pairs of classes that show and hide elements at different browser window sizes, the names of which start with `visible-` or `hidden-`. The `*-xs` classes (i.e., `visible-xs` and `hidden-xs`) are the ones I used in the example. The `*-sm` classes work on windows wider than 768 pixels, the `*-md` classes work on windows wider than 992 pixels, and the `*-lg` classes work on windows wider than 1200 pixels.

■ **Caution** Responsive CSS features like the ones that Bootstrap provide are based on the size of the browser window, not the device screen. Mobile device browsers are usually displayed full-screen, which means that the window size and the screen size are the same, but you can't always rely on this being the case. As ever, you need to test against the devices you are targeting to ensure that you have not made assumptions that catch you out.

You can see the effect of these changes by starting the application and viewing the product listing in a regular desktop browser, which has the advantage of letting you change the size of the window. Make the window smaller (less than 786 pixels), and you will see the SportsStore text break into two lines, as illustrated by Figure 10-3.

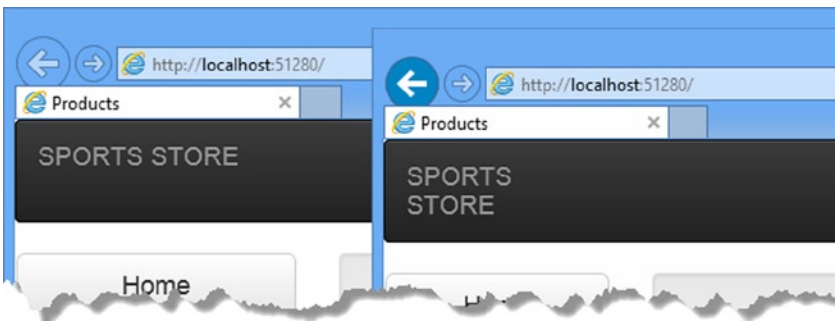

Figure 10-3. *Using Bootstrap responsive design features to adjust the application branding*

This may seem like a small change but it has a big impact on smaller screens, especially when combined with the changes I made to the Views/Cart/Summary.cshtml file, which is the view that provides the summary of the cart and its contents. You can see the changes I made in Listing 10-2.

Listing 10-2. Adding Responsive Content to the Summary.cshtml File

```
@model SportsStore.Domain.Entities.Cart

<div class="navbar-right hidden-xs">
    @Html.ActionLink("Checkout", "Index", "Cart",
    new { returnUrl = Request.Url.PathAndQuery },
    new { @class = "btn btn-default navbar-btn" })
</div>

<div class="navbar-right visible-xs">
    <a href=@Url.Action("Index", "Cart", new { returnUrl = Request.Url.PathAndQuery })
       class="btn btn-default navbar-btn">
        <span class="glyphicon glyphicon-shopping-cart"></span>
    </a>
</div>

<div class="navbar-text navbar-right">
    <b class="hidden-xs">Your cart:</b>
    @Model.Lines.Sum(x => x.Quantity) item(s),
    @Model.ComputeTotalValue().ToString("c")
</div>
```

This is the same technique that I applied to the _Layout.cshtml file, where I selectively show and hide content. In this case, however, I hide the standard Checkout now button on small screens and replace it with an icon button, using one of the icons that is included in the Bootstrap package.

The Bootstrap icons are applied through a span element, which means that I can't use the Html.ActionLink helper method because it doesn't present me with the ability to set the contents of the element it creates. Instead, I define the a element directly and use the Url.Action helper method (which I describe properly in Chapter 23) to generate a URL for the href attribute. The result is an a element with the same attributes that would be created by the Html.ActionLink method, but that contains a span element. You can see the effect of the changes I made to both files in Figure 10-4, which shows the header content displayed on the iPhone.

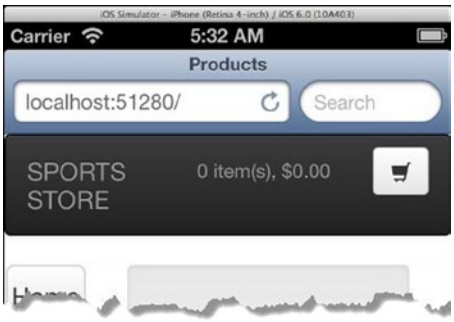

Figure 10-4. *The modified SportsStore header displayed on an iPhone simulator*

MOBILE FIRST VERSUS DESKTOP FIRST

Most web application projects start with desktop clients and then add support for mobile clients, just as I am doing in this book. This is known as *desktop first* design/development and a common problem is that the server-side development has largely finished before work on the mobile client begins, resulting in an awkward mobile experience which is hacked out of features designed for more capable desktop clients.

There is an alternative philosophy called *mobile first* design/development which, as the name suggests, starts with the mobile client as the foundation for the application and adds features to take advantage of more capable desktop browsers.

Or to put it another way, desktop first tends to start with a full set of features and degrade gracefully for less capable devices, while mobile first tends to start with a smaller set of features that are gracefully enhanced for more capable devices.

Both approaches have their merits and I tend to favor desktop first development because it is easy to get desktop browsers to load content from a local development workstation, something that can be surprisingly hard when working with real mobile hardware. I tend to work on a tight cycle of write-compile-check (which means reloading URLs into the browser frequently), and I get frustrated with the hoops one has to go through to get the same cycle going on a mobile device.

The danger in putting any group of users first is that you create a substandard experience for another group, just moving the pain around. Proponents of mobile first design often argue that this can't happen when you start with the basic features and then scale up, but that has not been my experience.

It is important to have a solid plan for what functionality and layout you are going to deliver to *all* devices *before* you start developing for *any* of them. When you have such a plan, it doesn't matter what kind of device you begin with and, critically, the server-side parts of the application will be built from the ground up to support a full range of clients.

Creating a Responsive Product List

To complete my responsive adaptations, I need a product list that will display on narrow devices. The biggest problem that I have is caused by the horizontal space that is taken up by the category buttons. I am going to move the buttons out of the way, giving individual product descriptions the whole width of the display. In Listing 10-3, you can see that I have further modified the `_Layout.cshtml` file.

Listing 10-3. Creating a Responsive Product List in 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" />
    <link href="~/Content/ErrorStyles.css" rel="stylesheet" />
    <title>@ViewBag.Title</title>
    <style>
```

```
        .navbar-right {
            float: right !important;
            margin-right: 15px; margin-left: 15px;
        }
    </style>
</head>
<body>
    <div class="navbar navbar-inverse">
        <a class="navbar-brand" href="#">
            <span class="hidden-xs">SPORTS STORE</span>
            <div class="visible-xs">SPORTS</div>
            <div class="visible-xs">STORE</div>
        </a>
        @Html.Action("Summary", "Cart")
    </div>
    <div class="row panel">
        <div class="col-sm-3 hidden-xs">
            @Html.Action("Menu", "Nav")
        </div>
        <div class="col-xs-12 col-sm-8">
            @RenderBody()
        </div>
    </div>
</body>
</html>
```

There can only be one call to the `RenderBody` method in a layout. I get into the details of layouts in Chapter 20, but the effect of this limit is that I can't have duplicate sets of elements that I show and hide, each containing a `RenderBody` call. Instead, I need to change the layout of the grid that contains the `RenderBody` method call so that the elements in the layout adapt around the content from the view.

One of the reasons that I used the Bootstrap grid to structure the content in the `_Layout.cshtml` file in Chapter 7 is that it includes some responsive design features that let me work around the `RenderBody` limitation. The Bootstrap grid layout supports 12 columns and you specify how many an element will occupy by applying a class, like this, which is how I applied the Bootstrap classes in Chapter 7:

```
...
<div class="col-xs-8">
    @RenderBody()
</div>
...
```

Much like the `hidden-*` and `visible-*` classes that I described earlier, Bootstrap provides a set of classes that set the number of columns that an element occupies in the grid based on the width of the window.

The `col-xs-*` classes are fixed and don't change based on the width of the screen. My use of the `col-xs-8` class tells Bootstrap that the `div` element should span 8 of the 12 available columns and that the visibility of the element should not change based on the width of the window. The `col-sm-*` classes set the columns when the window is 768 pixels or wider, the `col-md-*` classes work on windows that are 992 pixels or wider and, finally, the `col-lg-*`

classes work on windows that are 1200 pixels or wider. With this in mind, here are the classes that I applied to the `div` element that surrounds the `RenderBody` method call from Listing 10-3:

```
...
<div class="col-xs-12 col-sm-8">
    @RenderBody()
</div>
...
```

The effect of applying both classes is that the `div` element will occupy all 12 columns in the grid by default and 8 columns when the screen is 768 pixels or wider. The other columns in the grid contain the category buttons, as follows:

```
...
<div class="col-sm-3 hidden-xs">
    @Html.Action("Menu", "Nav")
</div>
...
```

This element will occupy 3 columns when the screen is wider than 768 pixels and be hidden otherwise. Combined with the other classes I applied, the effect is that the product descriptions fill small windows and share the available space with the category buttons for larger windows. You can see both layouts in Figure 10-5. I used a desktop browser to for this figure because I am able to easily vary the width of the window.

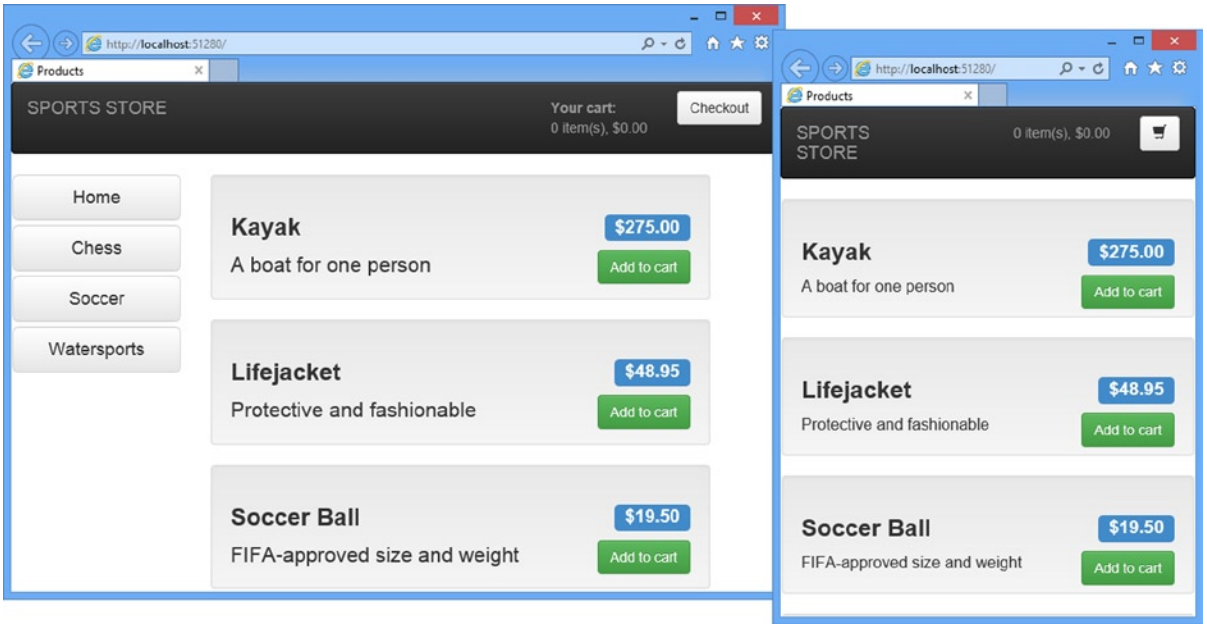

Figure 10-5. *Using a responsive grid in the product layout*

Helping the Controller Select a View

I don't want to leave mobile users without the ability to filter products, which means that I need to present the categories in a different way. To do this, I created a new view called `MenuHorizontal.cshtml` in the `Views/Nav` folder with the content shown in Listing 10-4.

Listing 10-4. The Contents of the MenuHorizontal.cshtml File

```
@model IEnumerable<string>

<div class="btn-group btn-group-sm btn-group-justified">
   @Html.ActionLink("Home", "List", "Product", new { @class = "btn btn-default btn-sm" })

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

This is a variation on the original `Menu.cshtml` layout, but with a container `div` element and some Bootstrap classes to create a horizontal layout of the buttons. The basic functionality, however, is the same. I generate a set of links which will filter the products by category.

The set of category buttons is generated through the `Menu` action method of the `Nav` controller, which I need to update so that it selects the right view file based on the required orientation of the buttons, as shown in Listing 10-5.

Listing 10-5. Updating the Menu Action 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(string category = null,
            bool horizontalLayout = false) {

            ViewBag.SelectedCategory = category;
```

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

            string viewName = horizontalLayout ? "MenuHorizontal" : "Menu";
            return PartialView(viewName, categories);
        }
    }
}
```

I have defined a new parameter for the action method that specifies the orientation, which I use to select the name of the view file passed to the `PartialView` method. To set the value of this parameter, I need to return to the `_Layout.cshtml` file, as shown in Listing 10-6.

Listing 10-6. Updating the _Layout.cshtml File to Include the Horizontal Buttons

```
...
<body>
    <div class="navbar navbar-inverse">
        <a class="navbar-brand" href="#">
            <span class="hidden-xs">SPORTS STORE</span>
            <div class="visible-xs">SPORTS</div>
            <div class="visible-xs">STORE</div>
        </a>
        @Html.Action("Summary", "Cart")
    </div>
    <div class="visible-xs">
        @Html.Action("Menu", "Nav", new { horizontalLayout = true })
    </div>
    <div class="row panel">
        <div class="col-sm-3 hidden-xs">
            @Html.Action("Menu", "Nav")
        </div>
        <div class="col-xs-12 col-sm-8">
            @RenderBody()
            </div>
    </div>
</body>
...
```

The optional third argument to the `Html.Action` method is an object that lets me set values for the routing system, which I explain in Chapters 15 and 16. I use this feature to signal which view the controller should select. The overall effect of these changes is shown in Figure 10-6.

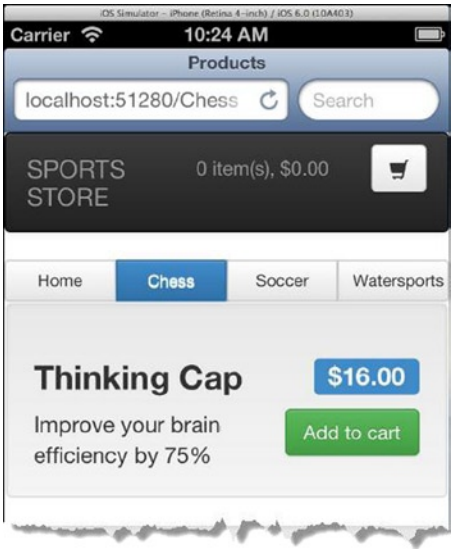

Figure 10-6. *The revised product listing for small screens*

You can see that moving the buttons to the top of the product listing creates enough space for each product to be displayed properly. I could continue improving the fit and finish of the views, but you get the idea. Aside from a brief demonstration of how responsive CSS classes can be used, I wanted to touch upon some of the limitations that the MVC Framework imposes (such as the `RenderBody` method limit) and some of the facilities it can provide to assist generating content in different ways (such as passing data from a view to a controller via the routing system and the `Html.Action` helper method).

■ **Tip** I have focused on one orientation for the iPhone, but don't forget that most mobile devices allow for multiple orientations and that you will have to cater for them all in a real project.

Removing View Duplication

In the previous example, I wanted to show you how you can have a controller select a view based on routing information passed by a call to the `Html.Action` helper method. It is an important and useful feature, but I would not have used it in a real project because it leaves me with two views, `Menu.cshtml` and `MenuHorizontal.cshtml`, that contain largely similar markup and Razor expressions. This is a maintenance risk because any changes that I require to the category filter buttons will have to be applied in two places. To resolve this I am going to consolidate the views. I created a new view file called `FlexMenu.cshtml` in the `Views/Nav` folder with the content shown in Listing 10-7.

Listing 10-7. The Contents of the FlexMenu.cshtml File

```
@model IEnumerable<string>

@{
bool horizontal = ((bool)(ViewContext.RouteData.Values["horizontalLayout"] ?? false));
string wrapperClasses = horizontal ? "btn-group btn-group-sm btn-group-justified" : null;
}
```

```
<div class="@wrapperClasses">

    @Html.ActionLink("Home", "List", "Product",
    new { @class =  horizontal ? "btn btn-default btn-sm" :
        "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 = (horizontal ? "btn btn-default btn-sm"
                : "btn btn-block btn-default btn-lg" )
                + (link == ViewBag.SelectedCategory ? " btn-primary" : "")
        })
    }

</div>
```

The cost of removing duplication is a more complex view that can generate both orientations of buttons and it is a matter of personal preference as to which approach you take. If you are like me and prefer to avoid duplication, then this listing shows several useful features you can apply to views.

The first is the ability to access routing information directly from the view. The `ViewContext` property provides information about the current state of the request that is being processed, including details of the routing information, as follows:

```
...
bool horizontal = ((bool)(ViewContext.RouteData.Values["horizontalLayout"] ?? false));
...
```

The second feature is the ability to create local variables within a view. This is possible because of the way that Razor views are compiled into classes (which I describe in Chapter 20), and I have created a local variable called `horizontal` that means I don't have to check the route data throughout the listing to figure out which orientation the view is being used for.

■ **Caution** Local variables should be used sparingly because it is the slippery slope into creating views that are hard to maintain and hard to test, but I sometimes use them in situations like this where I see them as an acceptable cost of simplifying a view.

A related feature is the way that Razor will conditionally set attributes based on variables. I defined a string of class names as a local variable in the view like this:

```
...
string wrapperClasses = horizontal ? "btn-group btn-group-sm btn-group-justified" : null;
...
```

The value of the wrapperClasses variable is either the string of class names that I used for horizontal layouts or null. I apply this variable to the class attribute like this:

```
...
<div class="@wrapperClasses">
...
```

When the variable is null, Razor is smart enough to remove the class attribute from the div element entirely, generating an element like this:

```
<div>
```

When the variable is not null, Razor will insert the value and leave the class attribute intact, producing a result like this:

```
<div class="btn-group btn-group-sm btn-group-justified">
```

This is a nice way of matching the characteristics of the C# with the semantics of HTML and is a feature that is endlessly useful when writing complex views because it won't insert null values into attributes and it won't generate empty attributes, which can cause problems with CSS selectors (and JavaScript libraries that use attributes to select elements, such as jQuery).

■ **Tip** Conditional attributes will work on any variable, not just the ones you have defined in the view. This means that you can apply this feature to model properties and the view bag.

To use my consolidate view, I need to revise the Menu action method in the Nav controller, as shown in Listing 10-8.

Listing 10-8. Updating the Menu Action in the NavController.cs File

```
...
public PartialViewResult Menu(string category = null) {

    ViewBag.SelectedCategory = category;

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

    return PartialView("FlexMenu", categories);
}
...
```

I removed the parameter that receives the orientation and changed the call to the PartialView method so that the FlexMenu view is always selected. The result of these changes doesn't alter the layout of the content or the effect of the responsive design, but it does remove the duplication in the views and means that you can delete the Menu.cshtml and MenuHorizontal.cshtml views from the Visual Studio project. Both orientations of category filter button are now produced by the FlexMenu.cshtml view.

THE LIMITATIONS OF RESPONSIVE DESIGN

There are some problems with responsive design as a way to support mobile clients. The first is that you end up duplicating a lot of content and sending it to the server so that it can be displayed in different scenarios. You saw this in the previous section when the HTML generated by the layout contained multiple sets of elements for the page header and the category filter buttons. The extra elements don't amount to much on a per-request basis, but the overall effect for a busy application is a sharp increase in the amount of bandwidth you will need to provision, with the corresponding increase in running costs.

The second problem is that responsive design can be fiddly and it requires endless testing to get right. Not all devices handle the underlying CSS features that enable responsive design (known as *media queries*) properly and, unless you are thorough and careful, you will end up with an application that delivers an adequate experience on every device without excelling on any of them, a kind of blandness that comes from averaging out all of the device quirks.

Responsive design can be useful when applied thoughtfully, but it can easily result in an application that is riddled with compromises that don't deliver a good experience for any of your target users.

Creating Mobile Specific Content

Responsive design delivers the same content to all devices and uses CSS to figure out how that content should be presented, a process that doesn't involve the server-side part of the application and which assumes that you want to treat all devices as being variations on the same basic theme. An alternative approach is to use the server to assess the capabilities of the client browser and send different HTML to different kinds of client. This works well if you want to present a completely different aspect of the application on the desktop to, say, a tablet.

■ **Tip** You don't have to choose between responsive design and mobile-specific content and, in most projects, you'll need to use both to get a good result on the devices you target. As an example, you may decide to create content specifically for tablets and use responsive design to create the horizontal and vertical orientations that most tablets support.

The MVC Framework supports a feature called *display modes*, which allows you to create different views that are delivered based on the client that has made the request, a feature provided by the ASP.NET Framework. I explain how you can create and manage display modes in depth in my *Pro ASP.NET MVC 5 Platform* book, but for the SportsStore application, I am going to use the simplest form of display modes, which is to treat all mobile devices as being the same. My goal will be to deliver an experience to mobile devices using the popular jQuery Mobile library, while keeping the existing content for desktop devices.

■ **Tip** I am not going to go into any detail about jQuery Mobile in this book, other than to demonstrate how it can be used to deliver mobile-specific content. For full details of jQuery Mobile, see my *Pro jQuery 2.0* book, published by Apress.

Creating a Mobile Layout

All I have to do to create mobile-specific content is to create views and layouts that have a `.Mobile.cshtml` suffix. I created a new layout called `_Layout.Mobile.cshtml` in the `Views/Shared` folder with the content shown in Listing 10-9.

■ **Tip** Because the name of the view contains an additional period, you will need to create the view by right-clicking the `Shared` folder and selecting `Add ➤ MVC 5 Layout Page (Razor)` from the pop-up menu.

Listing 10-9. The Contents of the _Layout.Mobile.cshtml File

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link rel="stylesheet"
          href="http://code.jquery.com/mobile/1.3.2/jquery.mobile-1.3.2.min.css" />
    <script src="http://code.jquery.com/jquery-1.9.1.min.js"></script>
    <script
          src="http://code.jquery.com/mobile/1.3.2/jquery.mobile-1.3.2.min.js"></script>
    <title>@ViewBag.Title</title>
</head>
<body>
    <div data-role="page" id="page1">
        <div data-theme="a" data-role="header" data-position="fixed">
            <h3>SportsStore</h3>
            @Html.Action("Menu", "Nav")
        </div>
        <div data-role="content">
            <ul data-role="listview" data-divider-theme="b" data-inset="false">
                @RenderBody()
            </ul>
        </div>
    </div>
</body>
</html>
```

This layout uses jQuery Mobile, which I obtain using a content delivery network (CDN) so that I don't have to install a NuGet package for the JavaScript and CSS files I need.

■ **Tip** I am just scratching the surface by creating mobile-specific views, because I am using the same controllers and action methods that are used for desktop clients. Having separate views allows you to introduce different controllers, which are particular to a set of clients, and this can be used to create totally different features and functionality for different types of client.

The MVC Framework will automatically identify mobile clients and use the `_Layout.Mobile.cshtml` file when it is rendering views, seamlessly replacing the `_Layout.cshtml` file which is used for other clients. You can see the impact of the change in Figure 10-7.

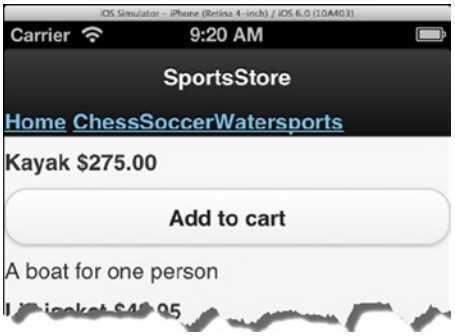

Figure 10-7. *The effect of creating a mobile layout in the SportsStore application*

You can see that the layout is different, but the overall effect is a mess, and that's because I need to create mobile versions of the main view that is handling the request and the partial view that is used for the category filtering buttons.

Creating the Mobile Views

I am going to start with the category filtering, which means creating a view called `FlexMenu.Mobile.cshtml` in the `Views/Nav` folder with the content shown in Listing 10-10.

Listing 10-10. The Contents of the FlexMenu.Mobile.html File

```
@model IEnumerable<string>

<div data-role="navbar">
    <ul>
        @foreach (var link in Model) {
            <li>
                @Html.RouteLink(link, new {
                    controller = "Product",
                    action = "List",
                    category = link,
                    page = 1
                }, new {
                    data_transition = "fade",
                    @class = (link == ViewBag.SelectedCategory
                        ? "ui-btn-active" : null)
                })
            </li>
        }
    </ul>
</div>
```

This view uses a Razor `foreach` expression to generate `li` elements for the product categories, producing elements that are organized in the way that jQuery Mobile expects for a navigation bar at the top of the page. You can see the effect in Figure 10-8.

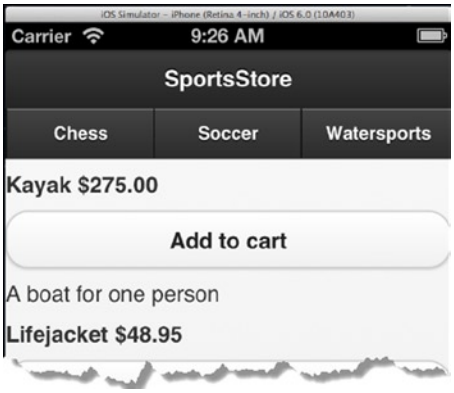

Figure 10-8. *The effect of creating a mobile-specific view*

■ **Tip** jQuery Mobile relies on the use of data attributes to format elements. Data attributes are prefixed with `data-` and were an unofficial way of defining custom attributes for years before becoming an official part of the HTML5 standard. In the listing, I needed to add a `data-transition` attribute to the `li` elements, but I can't use `data-transition` as the property name for the anonymous object because this would be a C# expression. The problem is the hyphen and Razor works around this by translating underscores in the property names to hyphens in attribute names, such that I was able to use `data_transition` in the listing and get a `data-transition` attribute on the elements I generate.

The product information is still a mess, but the category buttons are now being generated by the new mobile-specific view. It is worth taking a moment to reflect on what the MVC Framework is doing to render the content in Figure 10-8.

The HTTP request from the browser targets the `List` action method in the `Product` controller, which tells the MVC Framework to render the `List.cshtml` view file. The MVC Framework knows that the request came from a mobile browser and so it starts looking for mobile-specific views. There is no `List.Mobile.cshtml`, and so the `List.cshtml` file is processed instead. This view relies on the `_Layout.cshtml` file, but the MVC Framework notices that there is a mobile-specific version available and so it uses `_Layout.Mobile.cshtml` instead. The layout requires the `FlexMobile.cshtml` file but there is a mobile version of that as well, and so on.

The result is that the response to the browser is generated from a mix of mobile-specific and general views, with the MVC Framework using the most specific view file available, but seamlessly falling back when needed.

THE TWO PROBLEMS IN THE EXAMPLE

The example in this chapter is intended to demonstrate the way that the MVC Framework can deliver mobile-specific content, but I would be remiss if I didn't point out two potentially serious problems that this example introduces to the SportsStore application.

The first is that I have provided less functionality in my mobile views than in the desktop ones. There is no cart summary in the page header, for example. I left some features out to simplify the changes I had to make, but I recommend against delivering reduced functionality to *any* device unless there is a technical limitation that prevents the device from being able to support it. Mobile devices are increasingly capable and many users will only access your application with a mobile browser. The days when you could assume that mobile access was a supplement to desktop use have passed.

The second problem is that I have not offered the user the chance to switch back to the desktop layout. Don't underestimate the number of users that would prefer to have the desktop layout on a mobile device, even though it might be a little awkward and require some zooming and scrolling on smaller screens. Some mobile devices allow larger monitors to be connected, for example, and this is rarely detected by the mechanism that the ASP.NET Framework uses to identify mobile devices. You should always give mobile users the choice about which layout they receive.

Neither of these issues prevents me from deploying my application, but they are the kind of frustration that plagues mobile web application users. Mobile devices will be a big part of any modern web application and you should take every precaution to deliver a good user experience to this important category of users.

My last change is to create a mobile-specific version of the view that generates the product summary. I created a view file called `ProductSummary.Mobile.cshtml` in the `Views/Shared` folder with the contents shown in Listing 10-11.

Listing 10-11. The Contents of the ProductSummary.Mobile.cshtml File

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

<div data-role="collapsible" data-collapsed="false" data-content-theme="c">
    <h2>
        @Model.Name
    </h2>
    <div class="ui-grid-b">
        <div class="ui-block-a">
            @Model.Description
        </div>
        <div class="ui-block-b">
            <strong>(@Model.Price.ToString("c"))</strong>
        </div>
        <div class="ui-block-c">
            @using (Html.BeginForm("AddToCart", "Cart")) {
                @Html.HiddenFor(x => x.ProductID)
                @Html.Hidden("returnUrl", Request.Url.PathAndQuery)
                <input type="submit" data-inline="true"
                       data-mini="true" value="Add to cart" />
            }
        </div>
    </div>
</div>
```

This view uses a jQuery Mobile widget that allows users to open and collapse regions of content. It isn't an ideal way of presenting product information, but it is simple and my emphasis in this section is on mobile-specific content rather than the jQuery Mobile library. You can see the effect of this new view in Figure 10-9.

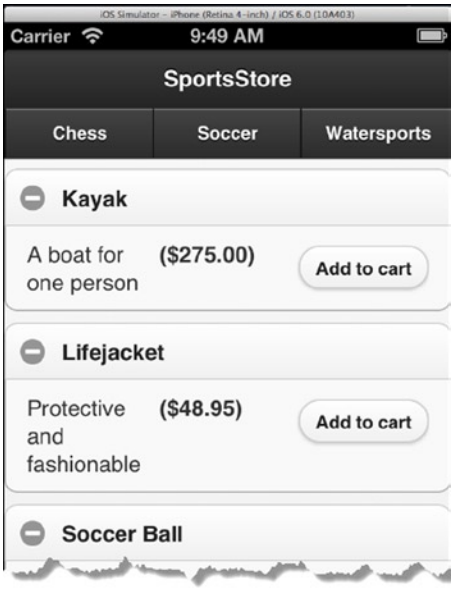

Figure 10-9. *The effect of mobile-specific views*

In a real project, I would carry on, of course, and create mobile-specific versions of the views that display the pagination links, the shopping cart and the checkout form. I am not going to because you have already seen how the MVC Framework lets you target mobile devices.

Summary

In this chapter I have shown you two techniques for handling mobile devices: responsive design and mobile-specific content. Responsive design isn't directly related to the MVC Framework, which sends the same content to all browsers and lets them figure out what to do with it. But as I demonstrated, there are some limitations in the way that views work that require careful thought and some nice Razor features that can ease the overall process.

Creating mobile-specific content is something that the MVC Framework does actively participate in by automatically applying mobile-specific views and layouts if they are available and blending them seamlessly into the process that renders HTML for the clients. In the next chapter, I add the basic features required to administer the SportsStore product catalog.