Detecting Device Capabilities

Just a few years ago, the world of web clients consisted of browsers running on desktops and browsers running on mobile devices. The desktop browsers offered the best support for HTML, CSS, and JavaScript and made their requests over fast and reliable network connections. The mobile browsers had limited support for the web standards, made requests over slow and unreliable cellular networks, and displayed their content on small screens running on underpowered hardware. In those days, it was important for web applications to be able to work out what kind of client had made a request because mobile devices could support only the simplest of applications.

The situation is different today. Smartphones and tablets run the same browsers as desktops, have high-resolution and high-density displays, and support a range of touch interactions. And functionality has started to migrate from the smartphone to the desktop: The latest versions of Windows support touch on the desktop, and more desktop monitors are being sold with touch sensors.

The term mobile client is still shorthand for describing a broad classification of devices, but any complex web application has to take a more nuanced view of what each client is capable of and respond appropriately.

Web applications can deal with device capabilities in a range of ways. The simplest approach is to ignore the differences and let the user figure it out. This isn’t as bad as it sounds because smartphone and tablet browsers have become adept at presenting all kinds of content to users and users have become adept at navigating content that isn’t optimized for their devices. A better approach is to use responsive design, which relies on features in CSS version 3 to adapt content based on the device, a technique that is usually supplemented by JavaScript code that adds support for different kinds of interaction when they are supported, such as touch and orientation sensors.

In this chapter, I show a different approach, which is to adapt the application at the server. I show you the facilities that the ASP.NET platform provides for classifying requests based on device capabilities and demonstrate how you can use these in your MVC framework applications to differentiate your content to create the best user experience. Table 7-1 summarizes this chapter.
Table 7-1. Chapter Summary

<table>
<thead>
<tr>
<th>Problem</th>
<th>Solution</th>
<th>Listing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Determine the capabilities of the browser used to make a request.</td>
<td>Read the properties of the <code>HttpBrowserCapabilities</code> object accessible through the <code>HttpRequest.Browser</code> property.</td>
<td>1–5</td>
</tr>
<tr>
<td>Define custom capabilities data.</td>
<td>Define new browser files or create a capabilities provider class.</td>
<td>6–8</td>
</tr>
<tr>
<td>Replace the built-in capabilities data.</td>
<td>Install data from a third-party provider.</td>
<td>9–11</td>
</tr>
<tr>
<td>Alter the content generated by the application based on the browser capabilities.</td>
<td>Use a Razor conditional statement in the view or select partial views through the <code>Html.Partial</code> helper.</td>
<td>12, 14–17</td>
</tr>
<tr>
<td>Adapt content to device screen size and orientation.</td>
<td>Use responsive design.</td>
<td>13</td>
</tr>
<tr>
<td>Automate the selection of partial views based on browser capabilities.</td>
<td>Use display modes.</td>
<td>18, 19</td>
</tr>
</tbody>
</table>

Preparing the Example Project

I am going to create a new project called Mobile for chapter, following the same approach that I have used for earlier examples. I used the Visual Studio ASP.NET Web Application template, selected the Empty option, and added the core MVC references. You should be familiar with the process by now, but see Chapter 6 if you want step-by-step instructions. I'll be using Bootstrap again in this chapter, so enter the following command into the Package Manager Console:

```
Install-Package -version 3.0.3 bootstrap
```

Listing 7-1 shows the content of the `Programmer.cs` file, which I created in the `Models` folder.

Listing 7-1. The Contents of the Programmer.cs File

```csharp
namespace Mobile.Models {

    public class Programmer {

        public Programmer(string firstName, string lastName, string title, string city, string country, string language) {
            FirstName = firstName; LastName = lastName; Title = title;
            City = city; Country = country; Language = language;
        }

        public string FirstName { get; set; }
        public string LastName { get; set; }
        public string Title { get; set; }
        public string City { get; set; }
        public string Country { get; set; }
        public string Language { get; set; }
    }
}
```
This will be the model class for the application, and I’ll be using it to demonstrate how to adapt content to different kinds of devices. Listing 7-2 shows the contents of the HomeController.cs file, which I used to define the default controller for the project in the Controllers folder.

Listing 7-2. The Contents of the HomeController.cs File

```csharp
using System.Web.Mvc;
using Mobile.Models;

namespace Mobile.Controllers {
    public class HomeController : Controller {
        private Programmer[] progs = {
            new Programmer("Joe", "Dunston", "Developer", "London", "UK", "Java"),
            new Programmer("Peter", "Jones", "Developer", "Chicago", "USA", "C#"),
            new Programmer("Murray", "Woods", "Jnr Developer", "Boston", "USA", "C#")
        };

        public ActionResult Index() {
            return View(progs);
        }
    }
}
```

The controller creates an array of Programmer objects and passes them to the View method. I created a view by right-clicking the action method in the code editor and selecting Add View from the pop-up menu. I called the view Index.cshtml, selected the Empty (without model) template, and unchecked all of the view option boxes. You can see the content I defined in the view in Listing 7-3.

Listing 7-3. The Contents of the Index.cshtml File

```html
@using Mobile.Models
@model Programmer[]

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Mobile Devices</title>
    <link href="~/Content/bootstrap.min.css" rel="stylesheet" />
    <link href="~/Content/bootstrap-theme.min.css" rel="stylesheet" />
    <style>
        body { padding-top: 10px; }
    </style>
</head>

<body>
    <div class="alert alert-success">
        This is the /Views/Home/Index.cshtml view
    </div>
```
The layout generates a table display of the Programmer objects, styled using CSS classes. You can see how the view is rendered by starting the application, as illustrated by Figure 7-1.

![Image of the HTML code and a table displaying programmer information](image-url)

**Figure 7-1. Testing the example application**
As the figure shows, I have switched from Internet Explorer to Google Chrome to display the output from the example application in this chapter. Recent versions of Chrome have added support for emulating a range of devices, which I’ll be using to demonstrate techniques in this chapter. Chrome can be downloaded from http://google.com/chrome.

To enable device simulation, press the F12 key to open the developer tools window, click the Show Drawer button (the one with the > character and three horizontal lines), and select the Emulation tab.

**Tip** If you don’t see the Emulation tab, then open the Settings window by clicking the cog icon in the developer tools window and enable the Show Emulation View in the Console Drawer option.

Select Apple iPhone 5 from the list of devices and click the Emulate button. Figure 7-2 shows the way that the output from the application is displayed when Chrome simulates an iPhone 5. Smartphone and tablet browsers are pretty good at laying out HTML content automatically. I stacked this example in my favor by using a table because they present a difficult layout problem. Scaling the content so that the table is entirely visible makes the content unreadable, and reformatting the table so that each row spans multiple lines is often confusing to the user.

![Table screenshot](image.png)

**Figure 7-2. Emulating an iPhone 5 using Google Chrome**

I have cropped the figure because tall thin screenshots take up a lot of space on the page, but even so, you can see that the table doesn’t display well on the narrow iPhone screen. Throughout this chapter, I’ll show you different ways that you can adapt the content to the capabilities of the device that it is being displayed on.
I have used Google Chrome to emulate mobile devices in this chapter because it is simple and free and gives a pretty good idea of how different capabilities affect the way content will be rendered by the target browsers. Since Google introduced the emulation features, I find myself using them at the start of projects as I create the main building blocks of functionality. Having Chrome installed locally means that the emulator performs well and is always available for quick tests without delay. The limitation of this approach is that the Chrome rendering engine is always used, which means you can get a sense of how content is affected by screen size, for example, but not how different implementations of CSS properties or JavaScript APIs affect an application.

As I get further into the detail of a project, I switch to using a remote browser test suite. I use browserstack.com, but there are others available. These test suites allow you to run your application using emulators for popular mobile devices. This isn’t perfect, of course, but it starts to give you a sense of where browser implementation issues are going to be a problem. (To be clear: I don’t have any relationship with browserstack.com other than as a customer. I pay the standard fees and don’t receive any special support or features.) The drawback of using a testing service is that the emulators are running remotely and it takes a while to create a session, start the emulator, and load the application. This can be a tedious process when you want to make rapid changes, which is why I start with Chrome for the major functionality areas and don’t switch to the emulators until I have a solid foundation in place.

I start testing with real devices when all of the major functionality is complete. No emulator can re-create the feel of interacting with an application through a touch screen, and I spend some time making sure that gestures feel natural and that the application gives the user a sense of context about where they are in the application. I adjust the fit and finish of the application until I produce something that works well. I don’t have hardware for every device, but I keep a small pile of the most popular and representative devices, most of which I purchased used or refurbished. Testing with real devices is a laborious process, which is why I wait until the end of the development process before using them.

### Detecting Device Capabilities

The starting point for adapting to different devices is to access their capabilities. In this section, I show you how ASP.NET provides information about different devices and how you can customize and improve this information that is available. Table 7-2 puts the process of detecting device capabilities into context.

<table>
<thead>
<tr>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>What is it?</td>
<td>Device capabilities provide you with information about the browser and device from which a request originates.</td>
</tr>
<tr>
<td>Why should I care?</td>
<td>Not all devices can cope with complex content or implement the latest HTML and CSS features. Assessing capabilities allows you to tailor the content and behavior of your application to reach the widest possible audience.</td>
</tr>
<tr>
<td>How is it used by the MVC framework?</td>
<td>The MVC framework implements the display modes feature, which is commonly used in conjunction with capabilities data. See the “Using Display Modes” section later in the chapter for details.</td>
</tr>
</tbody>
</table>
Getting Browser Capabilities

The ASP.NET platform focuses on the browser, rather than the underlying device, although the two are usually one in the same when it comes to smartphones and tablets. The HttpRequest.Browser property returns a System.Web.HttpBrowserCapabilities object that describes the capabilities of the device that has made the request. The HttpBrowserCapabilities class defines a great many properties, but only a few are truly useful, and I have described them in Table 7-3. You can see a complete list of the properties defined by the HttpBrowserCapabilities class at http://msdn.microsoft.com/en-us/library/system.web.httpbrowsercapabilities(v=vs.110).aspx. Later in the chapter, I’ll explain how you can extend the set of properties using freely available third-party data.

Table 7-3. The Most Useful Properties Defined by the HttpBrowserCapabilities Class

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Browser</td>
<td>Returns the browser name.</td>
</tr>
<tr>
<td>IsMobileDevice</td>
<td>Returns true if the device is mobile. There is no fixed definition of what constitutes a mobile device, and it is the responsibility of the provider of capability data to make the assessment. As a general rule, you can expect this property to return true if the device is handheld, is battery powered, and connects via a wireless or cellular network.</td>
</tr>
<tr>
<td>MobileDeviceManufacturer</td>
<td>Returns the name of the device manufacturer.</td>
</tr>
<tr>
<td>MobileDeviceModel</td>
<td>Returns the name of the device.</td>
</tr>
<tr>
<td>ScreenPixelsHeight</td>
<td>Returns the size of the screen in pixels.</td>
</tr>
<tr>
<td>ScreenPixelsWidth</td>
<td>Returns the width of the screen in pixels.</td>
</tr>
<tr>
<td>Version</td>
<td>Returns the version number of the browser.</td>
</tr>
</tbody>
</table>

**Note** I have included two properties defined by the HttpBrowserCapabilities class that look more useful than they really are: ScreenPixelsHeight and ScreenPixelsWidth. I listed them because they are so widely used and so that I can highlight the problems they cause. The root issue is that the quality of information about screen size is patchy and doesn’t always take into account the pixel density of the display. Making decisions about the content sent to a client based on the value of these properties can cause a lot of issues, especially for clients that support resizable browser windows that don’t correlate directly to the size of the screen (commonly the case for desktop clients). The short version is that you should not categorize clients based on the ScreenPixelsHeight and ScreenPixelsWidth properties.

To demonstrate the basic use of the HttpBrowserCapabilities class, I added a new action method to the Home controller, as shown in Listing 7-4.

Listing 7-4. Adding a New Action Method to the HomeController.cs File

```csharp
using System.Web.Mvc;
using Mobile.Models;

namespace Mobile.Controllers {

    public class HomeController : Controller {
```


private Programmer[] progs = {
    new Programmer("Joe", "Dunston", "Developer", "London", "UK", "Java"),
    new Programmer("Peter", "Jones", "Developer", "Chicago", "USA", "C#"),
    new Programmer("Murray", "Woods", "Jnr Developer", "Boston", "USA", "C#")
};

public ActionResult Index() {
    return View(progs);
}

public ActionResult Browser() {
    return View();
}
}

The action, called Browser, simply asks the MVC framework to render the default view, which I created by right-clicking the action method in the code editor and selecting Add View from the pop-up menu. You can see the contents of the view file I created in Listing 7-5.

Listing 7-5. The Contents of the Browser.cshtml File

@model IEnumerable<Tuple<string, string>>
@{
    Layout = null;
}
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width"/>
    <title>Device Capabilities</title>
    <link href="~/Content/bootstrap.min.css" rel="stylesheet" />
    <link href="~/Content/bootstrap-theme.min.css" rel="stylesheet" />
</head>
<body class="container">
    <div class="panel panel-primary">
        <div class="panel-heading">Capabilities</div>
        <table class="table table-striped table-bordered">
            <tr><th>Property</th><th>Value</th></tr>
            <tr><td>Browser</td><td>@Request.Browser.Browser</td></tr>
            <tr><td>IsMobileDevice</td><td>@Request.Browser.IsMobileDevice</td></tr>
            <tr><td>MobileDeviceManufacturer</td><td>@Request.Browser.MobileDeviceManufacturer</td></tr>
            <tr><td>MobileDeviceModel</td><td>@Request.Browser.MobileDeviceModel</td></tr>
        </table>
    </div>
</body>
</html>
The view populates a table with rows that contain the property names and values from the `HttpBrowserCapabilities` object. You can see the data generated for the iPhone in Figure 7-3.

![Figure 7-3. The ASP.NET browser capabilities properties for the iPhone](image)

**Caution** Notice that the values of the `ScreenPixelsHeight` and `ScreenPixelsWidth` properties are wrong. ASP.NET will default to reporting a screen size of 640 by 480 pixels when there is no information available. This is why you should not use these properties to adapt the content you sent to the device: You can’t tell whether the information is accurate or just not available.
Improving Capability Data

ASP.NET uses the user-agent string sent as part of the HTTP request to identify a client. As an example, here is the user-agent string that Chrome sends when it is emulating an iPhone 5:

Mozilla/5.0 (iPhone; CPU iPhone OS 7.0 like Mac OS X; en-us) AppleWebKit/537.51.1 (KHTML, like Gecko) Version/7.0 Mobile/11A465 Safari/9537.53

You may see a slightly different string because the version number of the browser or the operating system may change. To populate the HttpBrowserCapabilities properties, ASP.NET processes the user-agent string using a set of browser files, which are contained in the following location:

%SystemRoot%\Microsoft.NET\Framework\<version>\CONFIG\Browsers

For me, this means that the files are in the C:\Windows\Microsoft.NET\Framework\v4.0.30319\Config\Browsers folder. I’ll explain the format of these files in the “Creating a Custom Browser File” section later in the chapter, but for now it is enough to understand that the browser doesn’t send details of its capabilities to the application. Instead, the ASP.NET platform has to be able to translate user-agent strings into meaningful capabilities and present them to the application. There is some useful information in a user-agent string, such as the version of the browser or operating system being used, but most of the useful information, such as whether a request has originated from a mobile device, has to be obtained by the browser files.

Microsoft includes the browser files in the .NET Framework because there has to be some initial reference point from which translating between user-agent strings and capabilities can begin. But .NET isn’t updated all that often, and the information in the browser files is rudimentary and gets stale quickly given the vibrant market for smartphones and tablets. If you rely on just the built-in browser files, then you’ll find that new devices can mislead an application because of the way that ASP.NET describes the characteristics of devices that it doesn’t recognize. As an example, Figure 7-4 shows the HttpBrowserCapabilities values displayed for a request from a second-generation Google Nexus 7 tablet, which sends a user-agent string that the built-in browser files don’t contain information for.

![Figure 7-4. Misleading data for an unrecognized device](image)
There are obviously some problems with this data. The browser has been correctly recognized as Chrome, but the manufacturer and model are unknown, the screen size is incorrect, and the IsMobileDevice property returns false, even though tablets are generally considered to be mobile. In the sections that follow, I’ll show you different ways to improve the accuracy of the ASP.NET capabilities data.

Creating a Custom Browser File

The first technique for improving the ASP.NET capabilities data is to create custom browser files that supplement the built-in ones. A browser file describes one or more new browsers. To create a new browser file, right-click the project in the Solution Explorer and select Add ➤ Add ASP.NET Folder ➤ App_Browsers from the pop-up menu. This is the location that ASP.NET looks in for custom browser files. To create a new file, right-click the App_Browsers folder and select Add ➤ Browser File from the pop-up menu. Set the name of the new file to Nexus and click the OK button to create the App_Browsers/Nexus.browser file. In Listing 7-6, you can see how I used the browser file to define capabilities for the Nexus 7 tablet.

Listing 7-6. The Contents of the Nexus.browser File

```xml
<browsers>
  <browser id="Nexus" parentID="Chrome">
    <identification>
      <userAgent match="Nexus" />
    </identification>

    <capture>
      <userAgent match="Nexus (?'model'\d+)" />
    </capture>

    <capabilities>
      <capability name="MobileDeviceManufacturer" value="Google" />
      <capability name="MobileDeviceModel" value="Nexus ${model}" />
      <capability name="isMobileDevice" value="true" />
    </capabilities>
  </browser>

  <browser id="Nexus7" parentID="Nexus">
    <identification>
      <userAgent match="Nexus 7" />
    </identification>

    <capabilities>
      <capability name="ScreenPixelsHeight" value="1900" />
      <capability name="ScreenPixelsWidth" value="1200" />
    </capabilities>
  </browser>
</browsers>
```

Tip  I only describe the XML elements that I use in this example, but there is a complete description of the browser file schema at [http://msdn.microsoft.com/en-us/library/ms228122(v=vs.85).aspx](http://msdn.microsoft.com/en-us/library/ms228122(v=vs.85).aspx).
Browser files are XML. The top-level element is `browsers`, and individual browser definitions are denoted by the `browser` element. New browser definitions can build on existing ones. In the example, I used the `id` attribute to define a new browser called Nexus that builds on the built-in definition for the Chrome browser, which is specified by the `parentID` attribute. (The built-in browser files contain definitions for all the mainstream browsers.) The `identification` attribute tells ASP.NET how to determine that a request originates from the browser. I have used the most common option, which is to perform a regular expression match on the user-agent string, specified with the `userAgent` element and the `match` attribute. My browser definition matches any request that contains Nexus. You can also identify browsers using headers, but the Nexus products include the information I need in the user-agent string, like this:

```xml
Mozilla/5.0 (Linux; Android 4.3; Nexus 7 Build/JSS15Q) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/29.0.1547.72 Safari/537.36
```

The `capture` element allows me to pull out information from the request that I will use to set the value for capability properties later. I want to be able to accurately report the model of a Nexus device, so I use the `userAgent` element to match the digits that follow Nexus in the user-agent string and assign them to a temporary variable called `model`:

```xml
...<capture>
    <userAgent match="Nexus (?\model\d+)" />
</capture>
...
```

The capabilities element contains one or more `capability` elements that generate values for the `HttpBrowserCapabilities` object. I use literal values to set `MobileDeviceManufacturer` and `isMobileDevice` but include the `model` variable from the capture section to set the `MobileDeviceModel` property, as follows:

```xml
...<capabilities>
    <capability name="MobileDeviceManufacturer" value="Google" />
    <capability name="MobileDeviceModel" value="Nexus $\{model\}" />
    <capability name="isMobileDevice" value="true" />
</capabilities>
...
```

The result is that all requests that have a user-agent string that contains Nexus will report the built-in capabilities defined for the Chrome browser, with the exception of the three properties I redefined using capability elements. The second `browser` element further refines the capabilities for the Nexus 7 device. If the user-agent string contains Nexus 7, then set the value of the `ScreenPixelsHeight` and `ScreenPixelsWidth` properties. Figure 7-5 shows the capabilities reported when Google Chrome is used to emulate the Nexus 5 phone and Nexus 7 tablet.
Note I am using the screen size properties to demonstrate another problem they represent. You can create custom browser definitions to override the default values from the built-in files, but it is still hard to provide useful data. In this case, the values I have set for the properties are accurate for the second-generation Nexus 7, but the first generation used the same user-agent string and has a smaller screen, meaning that inaccurate capabilities will be reported for requests that come from the earlier devices.

Creating a Capability Provider

Creating individual browser files works well, but it can quickly get fiddly if you have to maintain a lot of capabilities data. A more flexible approach is to create a capability provider, which is a class that is derived from the System.Web.Configuration.HttpCapabilitiesProvider class and provides ASP.NET with capability information about requests. A custom provider allows you to use C# code to define capabilities, rather than XML elements.

To demonstrate creating a custom capabilities provider, I created a folder called Infrastructure in the example project and created a new class file called KindleCapabilities.cs. Listing 7-7 shows how I used the class file to define a provider for Amazon Kindle Fire tablets, which I selected because Google Chrome will emulate them and because there is no definition for them in the browser files.

Listing 7-7. The Contents of the KindleCapabilities.cs File

```csharp
using System.Web;
using System.Web.Configuration;

namespace Mobile.Infrastructure {
    public class KindleCapabilities : HttpCapabilitiesProvider {
        public override HttpBrowserCapabilities GetBrowserCapabilities(HttpRequest request) {
```
HttpCapabilitiesDefaultProvider defaults =
    new HttpCapabilitiesDefaultProvider();
HttpBrowserCapabilities caps = defaults.GetBrowserCapabilities(request);

if (request.UserAgent.Contains("Kindle Fire")) {
    caps.Capabilities["Browser"] = "Silk";
    caps.Capabilities["IsMobileDevice"] = "true";
    caps.Capabilities["MobileDeviceManufacturer"] = "Amazon";
    caps.Capabilities["MobileDeviceModel"] = "Kindle Fire";
    if (request.UserAgent.Contains("Kindle Fire HD")) {
        caps.Capabilities["MobileDeviceModel"] = "Kindle Fire HD";
    }
}

return caps;

The HttpCapabilitiesProvider class requires subclasses to implement the GetBrowserCapabilities method, which receives an HttpRequest object and returns the HttpBrowserCapabilities object that describes the browser. There can be only one instance of the HttpCapabilitiesProvider class for an application and so the most common approach is to implement the GetBrowserCapabilities method so that it supplements the data produced by the HttpCapabilitiesDefaultProvider class, which is the default capabilities provider and is responsible for processing the browser files. In the listing, you can see how I get the capabilities of the browser using the default provider and add to them only for the Kindle devices. The overall effect is that my capabilities are drawn from a combination of the built-in browser files, the custom browser file I created for the Nexus devices, and the code in the KindleCapabilities provider class.

The provider must be registered with ASP.NET during application initialization, and in Listing 7-8 you can see how I have used the Application_Start method in the global application class to tell ASP.NET that I want to use the KindleCapabilities class as the browser capabilities provider. (I described the role that the Application_Start method plays in the ASP.NET life cycle in Chapter 3.)

Listing 7-8. Registering the Capabilities Provider in the Global.asax.cs File

using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Configuration;
using System.Web.Mvc;
using System.Web.Routing;
using Mobile.Infrastructure;

namespace Mobile {
    public class MvcApplication : System.Web.HttpApplication {
        protected void Application_Start() {
            AreaRegistration.RegisterAllAreas();
            RouteConfig.RegisterRoutes(RouteTable.Routes);
            HttpCapabilitiesBase.BrowserCapabilitiesProvider = new KindleCapabilities();
        }
    }
}
The static HttpCapabilitiesBase.BrowserCapabilitiesProvider property sets the capabilities provider for the application, and in the listing I have applied an instance of my KindleCapabilities class. Figure 7-6 shows the effect of requesting the /Home/Browser URL using Chrome while it is emulating one of the Kindle Fire tablets before and after the addition of the custom capabilities provider.

![Capabilities table]

Figure 7-6. The effect of creating a custom capabilities provider

Using Third-Party Capabilities Data

Using a custom capabilities provider can be more flexible than using XML files, but you still have to provide all of the capabilities data for the devices that you want to support. Keeping track of all the devices that are released can be a lot of work, which is why you may choose to use a third-party source for the device data. There are three main suppliers of capabilities data, and two of them provide no-cost options for using their data. I have listed all three companies in Table 7-4.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>51degrees.mobi</td>
<td>Offers freely available data that can be used in most projects. The free data contains a subset of the capability properties in the commercial offering and delays adding new device data for three months. See the next section for details of use.</td>
</tr>
<tr>
<td>Scientiamobile</td>
<td>Freely available data from <a href="http://wurfl.sourceforge.net">http://wurfl.sourceforge.net</a> and a commercial offering that includes cloud access to data (which has a free option for up to 5,000 requests per month).</td>
</tr>
<tr>
<td>Device Atlas</td>
<td>Commercial-only offering of on-site data and cloud service.</td>
</tr>
</tbody>
</table>

The 51degrees.mobi data is the most popular because it is easy to integrate into an ASP.NET project and because the data is pretty good. The free data is well-maintained and extends the core set of capabilities defined by the built-in browser files, but it doesn’t offer the depth of additional capabilities that the paid-for option has, which differentiates
between smartphones and tablets, for example. (The other limitation is that new devices are not added to the free
data for three months, which can present a problem when requests from popular new devices start to arrive before the
capabilities data has been released.)

**Caution** You must keep third-party data up-to-date, which generally means downloading a new data file and
publishing an update of the application. An alternative is to use one of the cloud service offerings, which have the
benefit of always being current but are outside of your control and require a commercial contract.

### Installing the Module and Data File

The 51degrees data is most easily installed through the NuGet package called 51Degrees.mobi, but I don’t use this
option because the package installs extra features that go beyond device capabilities and get in the way of features
such as ASP.NET display modes (which I detail later in this chapter). Instead, I prefer to download a .NET assembly

To add the 51degrees data to the example application, go to the URL just mentioned and download the
51Degrees.mobi DLL Website Enhancer file. This is a zip file from which you will need to copy the
FiftyOne.Foundation.dll file from the NET4\bin into the application’s bin folder.

The DLL file contains a module that adds the capabilities data to HttpRequest objects and that is registered using
the PreApplicationStartMethod attribute, which I described in Chapter 4.

The DLL also contains device data, but it isn’t updated as frequently as the separate data file listed on the same
CodePlex web page, so download the latest version of the binary data file, rename it to be 51Degrees.mobi.dat, and
copy it into the App_Data folder.

**Tip** You will have to use the Windows File Explorer to copy the files. Neither file will show up in the Solution Explorer
window by default. The bin folder isn’t usually shown, but you can show the data file by right-clicking the App_Data
directory, selecting Add ➤ Existing Item from the pop-up menu, and locating the 51Degrees.mobi.dat file. The module and
data file will work even if you don’t perform this step.

### Configuring the Module

The next step is to create a configuration file that tells the module to use the separate data file. Right-click the
Mobile project item in the Solution Explorer and select Add ➤ New Item from the pop-up menu. Select the Web
Configuration File template item from the Web category, set the name of the file to be 51degrees.mobi.config, and
click the Add button to create the file. Edit the file that Visual Studio creates to match Listing 7-9.

**Listing 7-9. The Contents of the 51degrees.mobi.config File**

```xml
<?xml version="1.0"?>
<configuration>
  <configSections>
    <sectionGroup name="fiftyOne">
      <section name="detection"
          type="FiftyOne.Foundation.Mobile.Detection.Configuration.DetectionSection,
            FiftyOne.Foundation"
          requirePermission="false" allowDefinition="Everywhere"
```
This configuration file specifies the location of the device data file. The 51degrees.mobi developers have extended the standard configuration file schema to define their own configuration sections, a technique that I describe in Chapter 9.

**Disabling the Custom Capabilities Provider**

Setting a custom capabilities provider overrides the 51degrees.mobi data, so I have to comment out the statement in the global application class that sets up the KindleCapabilities object from the previous section. Listing 7-10 shows the commented-out statement.

**Listing 7-10.** Disabling the Custom Capabilities Provider in the Global.asax.cs File

```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Configuration;
using System.Web.Mvc;
using System.Web.Routing;
using Mobile.Infrastructure;

namespace Mobile {
    public class MvcApplication : System.Web.HttpApplication {
        protected void Application_Start() {
            AreaRegistration.RegisterAllAreas();
            RouteConfig.RegisterRoutes(RouteTable.Routes);
            // HttpCapabilitiesBase.BrowserCapabilitiesProvider
            //      = new KindleCapabilities();
        }
    }
}
```

**Displaying Additional Properties**

The final step is to demonstrate that the third-party capabilities data is being used. The 51degrees.mobi data defines a number of additional properties, and in Listing 7-11 you can see how I have extended the markup in the Browser.cshtml view to display two of them.
Listing 7-11. Adding Capability Properties to the Browser.cshtml File

```csharp
@model IEnumerable<Tuple<string, string>>
@

Layout = null;

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Device Capabilities</title>
    <link href="/Content/bootstrap.min.css" rel="stylesheet" />
    <link href="/Content/bootstrap-theme.min.css" rel="stylesheet" />
</head>
<body class="container">
    <div class="panel panel-primary">
        <div class="panel-heading">Capabilities</div>
        <table class="table table-striped table-bordered">
            <tr><th>Property</th><th>Value</th></tr>
            <tr><td>Browser</td><td>@Request.Browser.Browser</td></tr>
            <tr><td>IsMobileDevice</td><td>@Request.Browser.IsMobileDevice</td></tr>
            <tr><td>MobileDeviceManufacturer</td><td>@Request.Browser.MobileDeviceManufacturer</td></tr>
            <tr><td>MobileDeviceModel</td><td>@Request.Browser.MobileDeviceModel</td></tr>
            <tr><td>ScreenPixelsHeight</td><td>@Request.Browser.ScreenPixelsHeight</td></tr>
            <tr><td>ScreenPixelsWidth</td><td>@Request.Browser.ScreenPixelsWidth</td></tr>
            <tr><td>Version</td><td>@Request.Browser.Version</td></tr>
            <tr><td>CssColumn</td><td>@Request.Browser["CssColumn"]</td></tr>
            <tr><td>CssFlexbox</td><td>@Request.Browser["CssFlexbox"]</td></tr>
        </table>
    </div>
</body>
</html>

Tip You can see a complete set of additional properties at http://51degrees.mobi/Products/DeviceData/PropertyDictionary.aspx. Most of the properties listed are available only with the commercial data, but there are some useful additions in the free data as well.
These two capabilities, `CssColumn` and `CssFlexbox`, indicate whether a browser supports two CSS3 layouts. The `HttpBrowserCapabilities` object doesn’t define properties that correspond to these capabilities, but you can get their values by treating it as a collection indexed by name, like this:

```html
<tr><td>CssColumn</td><td><@Request.Browser["CssColumn"]></td></tr>
```

Figure 7-7 shows the additional capabilities when Google Chrome is used to emulate an iPhone 5.

![Capabilities Table](image)

**Figure 7-7. Using third-party capabilities data**

---

**Tip** You may have to refresh the browser tab to change the browser emulation.

---

**Adapting to Capabilities**

The first part of this chapter was all about identifying devices and adding capabilities, and with that out of the way, I can turn to the different ways in which you can use the capabilities data in an application to adapt to different devices.
Avoiding the Capabilities Trap

I am going to start by showing you the most common mistake with capability data, which is to reduce the amount of content sent to mobile devices in order to make difficult layouts fit on the screen. Listing 7-12 shows how I have edited the Index.cshtml file to reduce the number of columns in the table of programmers.

Listing 7-12. Adapting Content for Mobile Devices in the Index.cshtml View

```csharp
@using Mobile.Models
@model Programmer[]
{@
    Layout = null;
}
<!DOCTYPE html>
<html>
<head>
<meta name="viewport" content="width=device-width" />
<title>Mobile Devices</title>
<link href="~/Content/bootstrap.min.css" rel="stylesheet" />
<link href="~/Content/bootstrap-theme.min.css" rel="stylesheet" />
<style>
    body { padding-top: 10px; }
</style>
</head>
<body>
<div class="alert alert-success">
    This is the /Views/Home/Index.cshtml view
</div>
<div class="panel panel-primary">
    <div class="panel-heading">Programmers</div>
    <table class="table table-striped">
        <tr>
            <th>First Name</th>
            @if (!Request.Browser.IsMobileDevice) {
                <th>Last Name</th>
                <th>Title</th>
            }
            <th>City</th>
            @if (!Request.Browser.IsMobileDevice) {
                <th>Country</th>
            }
            <th>Language</th>
        </tr>
        @foreach (Programmer prog in Model) {
            <tr>
                <td>@prog.FirstName</td>
                @if (!Request.Browser.IsMobileDevice) {
                    <td>@prog.LastName</td>
                    <td>@prog.Title</td>
                }
                <td>@prog.City</td>
                @if (!Request.Browser.IsMobileDevice) {
                    <td>@prog.Country</td>
                }
                <td>@prog.Language</td>
            </tr>
        }
    </table>
</div>
</body>
</html>
```
Razor makes it easy to build logic into views that alters the HTML sent to the browser based on the device capabilities, and it is simple and easy to get some effective results. In the listing, I have used the IsMobileDevice property to reduce the number of table columns, and you can see the result when the view is rendered for an iPhone in Figure 7-8.

Figure 7-8. Adapting content to support device capabilities

The problem with this approach is that it isn’t adapting to the device capabilities in a useful way. The IsMobileDevice property doesn’t convey any information about the device screen size or screen orientation, just that the device is likely to run on battery and use cellular networks. Equating mobility with a narrow screen doesn’t make sense for today’s smartphones and tablets, not least because they allow the user to view content in two orientations. Figure 7-9 shows the same view rendered for the Nexus 7 tablet in landscape screen orientation.
The result is that all mobile devices get the same user experience, even those that are capable of displaying all of the content. That’s not to say that the IsMobileDevice property can’t be useful when it used appropriately, but it is important to understand that the definition of the property depends on the source of the capabilities data and that it can be relied on to make assessments of display size.

### Using Responsive Design Instead of Capabilities

The browser knows the size and orientation of the device screen and is far better placed to alter the layout to suit the display capabilities of the device through responsive design. Responsive design relies on CSS media queries, which change CSS property values based on the current characteristics of the device, including the screen. And, since media queries are performed at the browser, they allow content to be adapted dynamically, such as when the user changes the orientation of the device. I get into the topic of responsive design in detail in my *Pro MVC 5 Client Development* book, but you don’t have to understand advanced CSS features when you are working with a framework such as Bootstrap because it includes convenience CSS classes that take care of the work for you. In Listing 7-13, you can see that I have removed the conditional Razor statements from the Index.cshtml view and added a CSS class to th and td elements.

### Listing 7-13. Creating a Responsive Table in the Index.cshtml File

```csharp
@using Mobile.Models
@model Programmer[]
{@
    Layout = null;
}
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Mobile Devices</title>
    <link href="/Content/bootstrap.min.css" rel="stylesheet" />
    <link href="/Content/bootstrap-theme.min.css" rel="stylesheet" />
    <style>
        body { padding-top: 10px; }
    </style>
</head>

```

![Figure 7-9. Content displayed on a tablet](image)

The result is that all mobile devices get the same user experience, even those that are capable of displaying all of the content. That’s not to say that the IsMobileDevice property can’t be useful when it used appropriately, but it is important to understand that the definition of the property depends on the source of the capabilities data and that it can be relied on to make assessments of display size.

### Using Responsive Design Instead of Capabilities

The browser knows the size and orientation of the device screen and is far better placed to alter the layout to suit the display capabilities of the device through responsive design. Responsive design relies on CSS media queries, which change CSS property values based on the current characteristics of the device, including the screen. And, since media queries are performed at the browser, they allow content to be adapted dynamically, such as when the user changes the orientation of the device. I get into the topic of responsive design in detail in my *Pro MVC 5 Client Development* book, but you don’t have to understand advanced CSS features when you are working with a framework such as Bootstrap because it includes convenience CSS classes that take care of the work for you. In Listing 7-13, you can see that I have removed the conditional Razor statements from the Index.cshtml view and added a CSS class to th and td elements.

### Listing 7-13. Creating a Responsive Table in the Index.cshtml File

```csharp
@using Mobile.Models
@model Programmer[]
{@
    Layout = null;
}
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Mobile Devices</title>
    <link href="/Content/bootstrap.min.css" rel="stylesheet" />
    <link href="/Content/bootstrap-theme.min.css" rel="stylesheet" />
    <style>
        body { padding-top: 10px; }
    </style>
</head>

```

![Figure 7-9. Content displayed on a tablet](image)
Bootstrap includes a set of responsive CSS classes that show and hide elements based on the width of the screen. The `hidden-xs` class, which I used in the listing, hides an element when the width of the screen is less than 768 pixels. The result of the changes I made in Listing 7-13 is that narrow screens show fewer columns but wider screens display the full table, as shown in Figure 7-10.

![Figure 7-10. The effect of responsive design](image-url)
Caution  If you are going to use the IsMobileDevice to offer a variation of the application to all mobile devices, then you should also provide a mechanism by which the user can elect to switch back to the full-fat version. This means you won’t alienate those users who don’t share your assessment of the capabilities of their device. I usually do this by adding a simple switch or button to the HTML layout and keeping track of a user’s choice through the session state data feature (which I describe in Chapter 10) or the user data feature (which I describe in Chapter 15).

Tailoring Content to Match Devices

Having shown you the common pitfall with capabilities data, I want to show you that there are useful ways in which to adapt your application to support different devices but, importantly, based on the actual capabilities of the device, rather than a loose proxy such as the value from the IsMobileDevice property.

Adapting views works best when you focus on the capabilities of specific devices or browsers. HTML and CSS provide many opportunities for this kind of adaptation as different generations of browsers implement aspects of the standards at different paces.

In effect, capabilities can be used to work around deficiencies of specific devices, especially older devices whose built-in browsers don’t support recent HTML and CSS features. I find this especially useful for dealing with older versions of Internet Explorer, which are notoriously difficult to support because they were produced during the dark days of Microsoft’s embrace-and-extend approach to web standards. Tools such as jQuery and Bootstrap can go a long way to helping support old browsers, but complex applications can still encounter problems, and that’s where device capabilities can be useful.

I don’t want to get into the process of setting up test rigs for old versions of Internet Explorer, so I am going to simulate the problem by identifying a particular browser that Google Chrome can emulate and handling the requests it makes differently. For simplicity, I am going to pretend that the Safari browser, which is used on iOS devices such as the iPhone, can’t support the table layout in my application and that I need to work around this problem by using an alternative approach. I’ll show you different ways of solving the problem, building up to the display modes feature, which is the most flexible and easiest to work with in complex applications.

Adapting Directly in the View

The most direct approach is to adapt the content you send to the client directly in the view. You can see how I have done this in Listing 7-14 using a Razor conditional statement.

Listing 7-14. Using Capabilities in the Index.cshtml File

```csharp
@using Mobile.Models
@model Programmer[]
@
{
    Layout = null;
}
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Mobile Devices</title>
    <link href="/Content/bootstrap.min.css" rel="stylesheet" />
    <link href="/Content/bootstrap-theme.min.css" rel="stylesheet" />
```
I check to see which browser has made the request using the `HttpBrowserCapabilities.IsBrowser` convenience method, which checks to see whether the browser matches the specified name. This method is useful because it respects the hierarchy of browser files, which simplifies the matching process and means you don’t have to remember to synchronize your capabilities data with your conditional view statements. When I receive a request from the Safari browser, I generate a simple list of programmers rather than the table. You can see the result in Figure 7-11.
Using Partial Views

The problem with putting the conditional statements for browser capabilities into the view is that they quickly become unwieldy and difficult to maintain. A more elegant approach is to put the content for different browsers into partial views and select the one you need when the view is rendered. Small and simple views are easier to work with, and breaking out the browser-specific content makes it easy to keep changes isolated when the application enters testing and deployment. For my example application, I need to create two partial views. The first, which will be the default, is the /Views/Home/Programmers.cshtml file, the contents of which you can see in Listing 7-15.

Listing 7-15. The Contents of the Programmers.cshtml File

```csharp
@using Mobile.Models
@model Programmer[]

<table class="table table-striped">
    <tr>
        <th>First Name</th>
        <th class="hidden-xs">Last Name</th>
        <th class="hidden-xs">Title</th>
        <th>City</th>
        <th class="hidden-xs">Country</th>
        <th>Language</th>
    </tr>
    @foreach (Programmer prog in Model) {
        <tr>
            <td>@prog.FirstName</td>
            <td class="hidden-xs">@prog.LastName</td>
            <td class="hidden-xs">@prog.Title</td>
            <td>@prog.City</td>
            <td class="hidden-xs">@prog.Country</td>
            <td>@prog.Language</td>
        </tr>
    }
</table>
```

Figure 7-11. Responding to device capabilities
This is the partial view that will be used for all requests that don’t come from the Safari browser, and it contains the responsive table layout. I called the second partial view Programmers.Safari.cshtml, and you can see the contents in Listing 7-16. (Visual Studio won’t let you create a view called Programmers.Safari.cshtml directly; first create a view called Safari.cshtml and then rename it in the Solution Explorer to Programmers.Safari.cshtml.)

Listing 7-16. The Contents of the Programmers.Safari.cshtml File

```csharp
@using Mobile.Models
@model Programmer[]

<div class="panel-body">
  <ul>
    @foreach (Programmer prog in Model) {
      <li>
        @prog.FirstName @prog.LastName,
        @prog.City (@prog.Language)
      </li>
    }
  </ul>
</div>
```

Tip  The naming of the partial views is slightly awkward, but it helps set the scene for the display modes feature, which I describe shortly.

This is the view that will be set to the Safari browser, and it contains the simpler list layout that I created in the previous section. In Listing 7-17, you can see how I updated the Index.cshtml file to use the new partial views.

Listing 7-17. Using the Partial Views in the Index.cshtml File

```csharp
@using Mobile.Models
@model Programmer[]
@{
    Layout = null;
}
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Mobile Devices</title>
  <link href="/Content/bootstrap.min.css" rel="stylesheet" />
  <link href="/Content/bootstrap-theme.min.css" rel="stylesheet" />
  <style>
    body { padding-top: 10px; }
  </style>
</head>
<body>
  <div class="alert alert-success">
    This is the /Views/Home/Index.cshtml view
  </div>
```
I use the `Html.Partial` helper to select one of the partial views based on the browser capability information. The result is a more maintainable approach than having the markup in a single view file.

**Using Display Modes**

The only problem with the approach in the previous section is that it requires the use of the `Html.Partial` helper method whenever an alternative view is required for the Safari browser. The final technique is the one that I have been building up in the previous steps: using *display modes*.

Display modes are not part of the ASP.NET platform, but I going to demonstrate their use here because they are usually applied in conjunction with the device capabilities feature. In short, display modes will automatically select alternative views when they exist based on a set of rules defined by the application. As a demonstration, Listing 7-18 shows how I created a display mode in the global application class.

**Listing 7-18. Defining a Display Mode**

```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Configuration;
using System.Web.Mvc;
using System.Web.Routing;
using Mobile.Infrastructure;
using System.Web.WebPages;
namespace Mobile {
    public class MvcApplication : System.Web.HttpApplication {
        protected void Application_Start() {
            AreaRegistration.RegisterAllAreas();
            RouteConfig.RegisterRoutes(RouteTable.Routes);

            //HttpCapabilitiesBase.BrowserCapabilitiesProvider
            //     = new KindleCapabilities();

            DisplayModeProvider.Instance.Modes.Insert(0,
                new DefaultDisplayMode("Safari") {
                    ContextCondition = (ctx => ctx.Request.Browser.IsBrowser("Safari"))
                });
        }
    }
}
```
The support for display modes is built into the view engine, which is why the important classes are in the `System.Web.WebPages` namespace. The static `DisplayModeProvider.Instance.Modes` property returns a collection of objects that implement the `IDisplayMode` interface. Rather than work directly with this interface, it is easier to use the `DefaultDisplayMode` class, which takes a constructor argument that will be used to look for views and defines the `ContextCondition` property that is used to match requests.

The `ContextCondition` property is set using a lambda expression that receives an `HttpContext` object and returns `true` if the request matches the display mode condition. In the example, my condition is that the `IsBrowser` method matches the Safari browser. I used Safari as the constructor argument as well, and this means that when the `ContextCondition` expression returns `true`, the display mode will append Safari to the view name specified by the action method or HTML helper. For example, when I specify the `Programmers` view and the request is from the Safari browser, the display mode will instead cause the view engine to look for a `Programmers.Safari` view. If no such view exists, then the one originally specified will be used.

The collection of `IDisplayMode` implementation objects is applied in sequence, which is why I used the `Insert` method to place my display mode at the start of the list. There is a built-in display mode that uses the `IsMobileDevice` property and locates views that contain `Mobile` in the name, such as `Programmers.Mobile.cshtml`. You can use this view without any configuration, but you should be cautious about the broad range of devices that this display mode will be applied to if you use it.

In Listing 7-19, you can see how I removed the ternary statement from the `Index.cshtml` view so that the display mode in Listing 7-17 is responsible for selecting the view automatically.

**Listing 7-19.** Updating the `Index.cshtml` File to Rely on Display Modes

```csharp
@using Mobile.Models
@model Programmer[]
@

<!DOCTYPE html>
<html>
<head>
<meta name="viewport" content="width=device-width" />
<title>Mobile Devices</title>
<link href="~/Content/bootstrap.min.css" rel="stylesheet" />
<link href="~/Content/bootstrap-theme.min.css" rel="stylesheet" />
<style>
  body { padding-top: 10px; }
</style>
</head>
<body>
<div class="alert alert-success">
  This is the /Views/Home/Index.cshtml view
</div>
<div class="panel panel-primary">
  <div class="panel-heading">Programmers</div>
  @Html.Partial("Programmers", Model)
</div>
</body>
</html>
```
The Programmers partial view is always specified, and the display mode takes care of looking for the Programmers.Safari view when it is required. Display modes make working with device capabilities easy and consistent throughout an application, especially since they will fall back to using the default views when the special view isn’t available.

Summary

In this chapter I explained the importance of device capabilities in supporting a wide range of browsers and devices. I demonstrated how to get capabilities data and how to extend the built-in data with custom browser files, providers, and third-party data. I showed you how to adapt an application to the capabilities of a client, directly using Razor and indirectly using partial views, HTML helpers, and, ultimately, display modes. In the next chapter, I show you can trace requests to get insight into how your application behaves.