CHAPTER 26

Bundles

In this chapter, I am going to look at the *bundles* feature, which the MVC Framework provides to organize and optimize the CSS and JavaScript files that views and layouts cause the browser to request from the server. Table 26-1 provides the summary for this chapter.

Table 26-1. *Chapter Summary*

Problem	Solution	Listing
Define bundles	Create instances of the `StyleBundle` and `ScriptBundle` classes and add them to the bundle table.	1–6
Prepare an application for bundles	Ensure that the `Views/web.config` file includes a reference for the `System.Web.Optimization` namespace.	7
Add a bundle to a view or layout	Use the `Styles.Render` and `Scripts.Render` helpers.	8
Enable concatenation and minification of bundles	Set the `debug` attribute of the `compilation` attribute in the `Web.config` file to `false`.	9

Preparing the Example Application

For this chapter, I have created a new MVC project called `ClientFeatures` using the `Empty` template option, checking the option to add the core MVC folders and references.

Adding the NuGet Packages

The bundles feature that I describe in this chapter makes it easier to manage JavaScript and CSS files. To that end, I am going to install a number of NuGet packages that are commonly used for client-side development. Select `Package Manager Console` from the `Visual Studio Tools` ➤ `Library Package Manager` menu and enter the following commands:

```
Install-Package jQuery –version 1.10.2
Install-Package jQuery.Validation –version 1.11.1
Install-Package Microsoft.jQuery.Unobtrusive.Validation –version 3.0.0
Install-Package Bootstrap -version 3.0.0
Install-Package Microsoft.jQuery.Unobtrusive.Ajax –version 3.0.0
```

Creating the Model and Controller

I am going to create a variation on the application that I used in the previous chapter, so I started by creating a new class file called Appointment.cs in the Models folder. You can see the contents of this file in Listing 26-1.

Listing 26-1. The Contents of the Appointment.cs File

```
using System;
using System.ComponentModel.DataAnnotations;

namespace ClientFeatures.Models {
    public class Appointment {

        [Required]
        public string ClientName { get; set; }

        public bool TermsAccepted { get; set; }
    }
}
```

I created a Home controller that operates on the Appointment model class, as shown in Listing 26-2.

Listing 26-2. The Contents of the HomeController.cs File

```
using System;
using System.Web.Mvc;
using ClientFeatures.Models;

namespace ClientFeatures.Controllers {
    public class HomeController : Controller {

        public ViewResult MakeBooking() {
            return View(new Appointment {
                ClientName = "Adam",
                TermsAccepted = true
            });
        }

        [HttpPost]
        public JsonResult MakeBooking(Appointment appt) {
            // statements to store new Appointment in a
            // repository would go here in a real project
            return Json(appt, JsonRequestBehavior.AllowGet);
        }
    }
}
```

There are two version of the MakeBooking method in this controller. The version with no parameters creates an Appointment object and passes it to the View method to render the default view. The HttpPost version of the MakeBooking method relies on the model binder to create an Appointment object and uses the Json method to encode the Appointment and send it back to the client in the JSON format.

I am focused on an MVC Framework feature that support client-side development in this chapter, so I have taken some shortcuts in the controller that wouldn't be sensible or useful in a real project. Most importantly, I do not perform any kind of validation when I receive a POST request and just send the details of the object created by the model binder back to the browser as JSON (with no support for HTML responses).

Creating the Layout and View

I created the `Views/Shared` folder and added a view file called `_Layout.cshtml` to it, the content of which you can see in Listing 26-3. The main purpose of this the layout is to import the JavaScript and CSS files that I added via NuGet so that they can be used in views.

Listing 26-3. The Contents of the _Layout.cshtml File

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
    <style>
        .field-validation-error { color: #f00; }
        .validation-summary-errors { color: #f00; font-weight: bold; }
        .input-validation-error { border: 2px solid #f00; background-color: #fee; }
        input[type="checkbox"].input-validation-error { outline: 2px solid #f00; }
        div.hidden { display: none; }
        div.visible { display: block; }
    </style>
    <link href="~/Content/bootstrap.css" rel="stylesheet" />
    <link href="~/Content/bootstrap-theme.css" rel="stylesheet" />
    <script src="~/Scripts/jquery-1.10.2.js"></script>
    <script src="~/Scripts/jquery.validate.js"></script>
    <script src="~/Scripts/jquery.validate.unobtrusive.js"></script>
    <script src="~/Scripts/jquery.unobtrusive-ajax.js"></script>
    @RenderSection("Scripts", false)
</head>
<body>
    @RenderSection("Body")
</body>
</html>
```

I defined two Razor sections in the view. The `Scripts` section will allow views to add JavaScript code to the `head` element section of the HTML response to the server, and the `Body` section allows the view to add content to the `body` element. (I explained how Razor sections worked in Chapter 20.) I added a view file called `MakeBooking.cshtml` to the `Home/Views` folder, as shown in Listing 26-4.

Listing 26-4. The Contents of the MakeBooking.cshtml File

```
@model ClientFeatures.Models.Appointment

@{
    ViewBag.Title = "Make Appointment";
    Layout = "~/Views/Shared/_Layout.cshtml";
    AjaxOptions ajaxOpts = new AjaxOptions {
        OnSuccess = "processResponse"
    };
}

@section Scripts {

<script type="text/javascript">

    function switchViews() {
        $(".hidden, .visible").toggleClass("hidden visible");
    }

    function processResponse(appt) {
        $('#successClientName').text(appt.ClientName);
        switchViews();
    }

    $(document).ready(function () {
        $('#backButton').click(function (e) {
            switchViews();
        });
    });
</script>

}

@section Body {

<div id="formDiv" class="visible well">
    @using (Ajax.BeginForm(ajaxOpts)) {
        @Html.ValidationSummary(true)
        <div class="form-group">
            <label for="ClientName">Your name:</label>
            <p>@Html.ValidationMessageFor(m => m.ClientName)</p>
            @Html.TextBoxFor(m => m.ClientName, new  {@class = "form-control" })
        </div>
        <div class="checkbox">
            <label>
                @Html.CheckBoxFor(m => m.TermsAccepted)
                I accept the terms & conditions
            </label>
        </div>
        <input type="submit" value="Make Booking" class="btn btn-primary"/>
    }
</div>
```

```
<div id="successDiv" class="hidden well">
    <h4 class="lead">Your appointment is confirmed</h4>
    <p>Your name is: <b id="successClientName"></b></p>
    <button id="backButton" class="btn btn-primary">Back</button>
</div>

}
```

My goal in this view is just to use all of the JavaScript and CSS files that I defined in the layout. To that end, I have defined an Ajax form that uses the unobtrusive Ajax library (described in Chapter 23) and that relies on the unobtrusive client-side validation library (described in Chapter 25). Both of these libraries depend on jQuery and I have used Bootstrap CSS classes to style the content.

I have taken advantage of the `Scripts` section I defined in the layout to include some JavaScript code that responds to the JSON response from the controller and manipulates the markup to display the results using some simple jQuery. This lets me deal with a single view for the example.

I want to create a typical scenario for a complex view file without needing to create a complex application, which is why I have added lots of JavaScript and CSS files for such a simple example. The key idea is that there are lots of files to be managed. When you are writing real applications, you will be struck by just how many script and style files you have to deal with in your views.

You can see how the example application works by starting the application and navigating to the `/Home/MakeBooking` URL. The form is pre-populated with data so that you can just click the `Make Booking` button to submit the form data to the server using Ajax. When the response is received, you will see a summary of the `Appointment` object that was created by the model binder from the form data, along with a `button` element that will return you to the form, as illustrated in Figure 26-1.

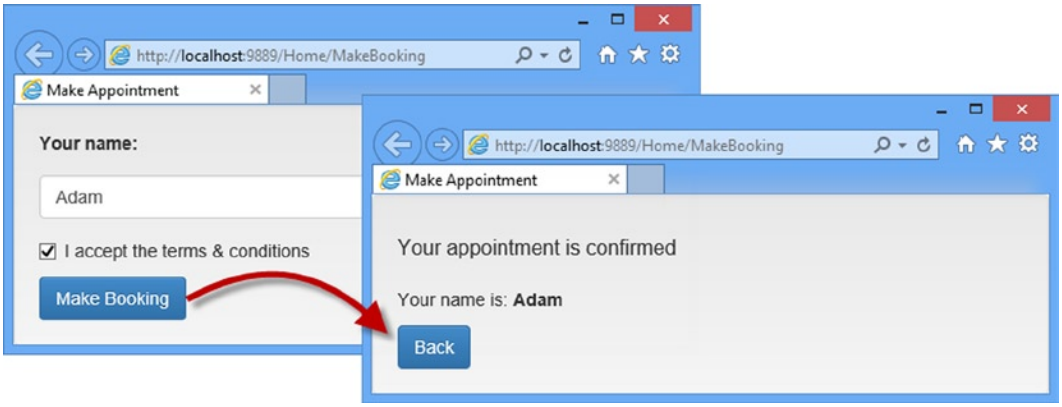

Figure 26-1. *Using the example application*

The project contains a number of JavaScript and CSS files, which are combined with inline JavaScript code and CSS styles to generate HTML for the browser. This is a typical mix that you will encounter in most MVC Framework projects.

Developers tend to write view files just as they would write HTML pages, which is fine but isn't the most effective approach. As I will show you in the sections that follow, there are some hidden problems in the `MakeBooking.cshtml` view file and I show you a number of improvements in the way that scripts and style sheets are managed.

Profiling Script and Style Sheet Loading

When considering any kind of optimization in any kind of project, you need to start by taking some measurements. I am all for efficient and optimized applications, but my experience is that people rush to optimize problems that don't have much impact and, in doing so, make design decisions that cause problems later.

For the problems that I am going to look at in this chapter, I am going to take the measurements using the Internet Explorer *F12 tools* (so called because you access them by pressing the F12 key).

I want to focus just on the HTTP requests that are made in the normal execution of the application, and that means disabling the Visual Studio Browser Link feature, which works by adding JavaScript code to the HTML sent to the browser, leading to additional HTTP requests.

Click on the small down arrow next to the Browser Link button on the Visual Studio toolbar and uncheck the `Enable Browser Link` menu item, as shown in Figure 26-2.

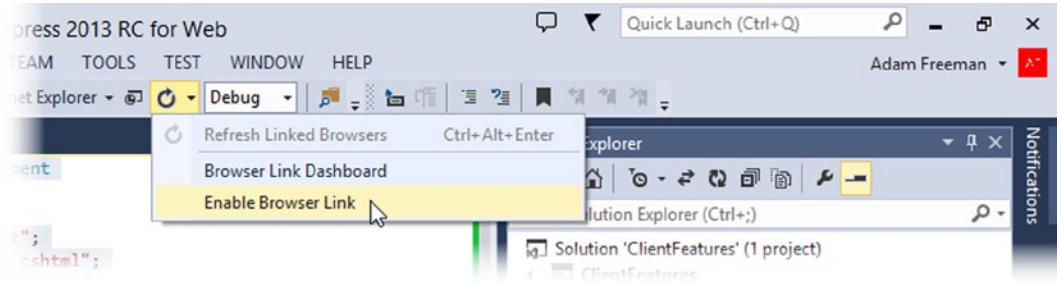

Figure 26-2. *Disabling the browser link feature in Visual Studio*

Load the application, navigate to the `/Home/MakeBooking` URL, and then press the F12 key. When the tools window opens, navigate to the `Network` tab and click the green arrow button to start capturing the HTTP requests that the browser makes. Then click the `Clear Browser Cache` button, which will ensure that the browser requests the contents of all of the JavaScript and CSS files that are referenced in the layout. Reload the contents of the browser tab (right-click in the browser window and select `Refresh`), and you will see the results shown in Figure 26-3.

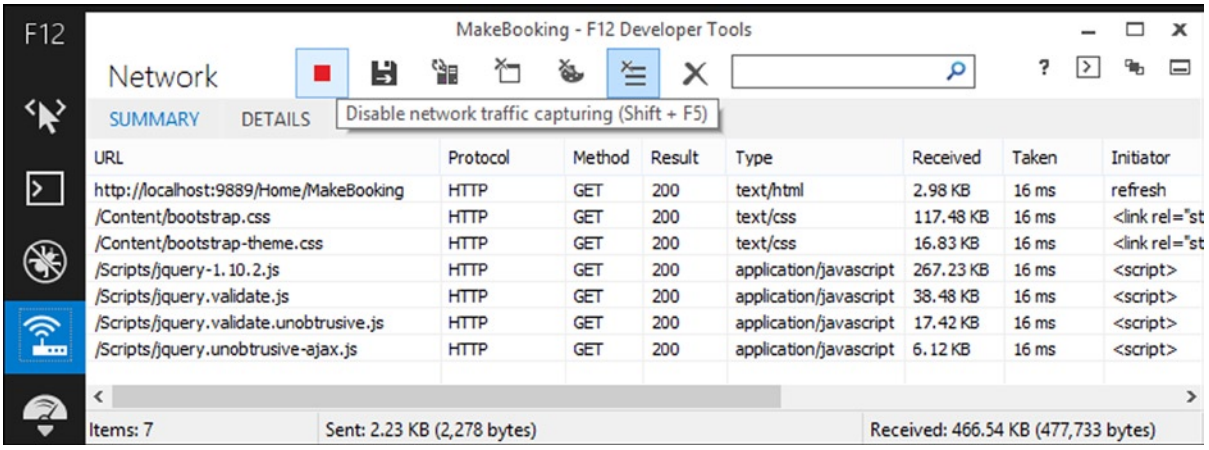

Figure 26-3. *Profiling script and style sheet loading for the example application*

The F12 tools allow you to profile the network requests that your application makes. (All of the mainstream browsers offer similar developer tools and there are other alternatives. My favorite is Fiddler, which you can get from www.fiddler2.com).

So that I can assess the optimizations that I make in this chapter, I will use the data shown in Figure 26-3 as the baseline. Here are the key figures:

- The browser made seven requests for the /Home/MakeBooking URL.
- There were two requests for CSS files.
- There were four requests for JavaScript files.
- A total of 2,278 bytes were sent from the browser to the server.
- A total of 477,733 bytes were sent from the server to the browser

This is the worst-case profile for the example application because I cleared the browser's cache before I reloaded the view. I have done this because it allows me to easily create a measurable starting point, even though I know that real-world use would be improved by the browser caching files from previous requests.

If I reload the /Home/MakeBooking URL without clearing the cache, then I get the following results:

- The browser made seven requests for the /Home/MakeBooking URL.
- There were two requests for CSS files.
- There were four requests for JavaScript files.
- A total of 2,086 bytes were sent from the browser to the server.
- A total of 5,214 bytes were sent from the server to the browser.

This is the best-case scenario, where all the requests for CSS and JavaScript files were able to be serviced using previously cached files.

■ **Note** In a real project, I would stop at this point and ask myself if I have a problem to solve or if the current state of the application is acceptable. It may seem that 473K is a lot of bandwidth for a simple Web page, but context is everything. I might be developing an application for Intranet use where bandwidth is cheap and plentiful, and optimizations of any sort are outweighed by the cost of the developer, who could be working on more important projects. Equally, I could be writing an application that operates over the Internet with high-value customers in countries with low-speed connections, in which case it is worth spending the time to optimize every aspect of the application. The point is that you shouldn't automatically assume that you have to squeeze every optimization into every application. There will often be better things you could be doing. (This is *always* the case if you are sneakily optimizing your application without telling anyone. Stealth optimization is a bad idea and will catch up with you eventually.)

Using Script and Style Bundles

My goal is to turn the JavaScript and CSS files into *bundles*, which allows me to treat several related files as a single unit. I walk through the steps required to set up and apply bundles in the sections that follow.

Adding the NuGet Package

The bundles feature requires a NuGet package that isn't included in the Visual Studio Empty template. Select Package Manager Console from the Visual Studio Tools ➤ Library Package Manager menu and enter the following command:

```
Install-Package Microsoft.AspNet.Web.Optimization -version 1.1.1
```

Defining the Bundles

The convention is to define bundles in a file called BundleConfig.cs, which is placed in the App_Start folder. In Listing 26-5, you can see the contents of the BundleConfig.cs file that I added to the example project. (You won't need to create this file yourself if you are using some of the other Visual Studio project templates because it will be added automatically.)

Listing 26-5. The Contents of the BundleConfig.cs File

```
using System.Web.Optimization;

namespace ClientFeatures {

    public class BundleConfig {

        public static void RegisterBundles(BundleCollection bundles) {

            bundles.Add(new StyleBundle("~/Content/css").Include(
                "~/Content/*.css"));

            bundles.Add(new ScriptBundle("~/bundles/clientfeaturesscripts")
                .Include("~/Scripts/jquery-{version}.js",
                    "~/Scripts/jquery.validate.js",
                    "~/Scripts/jquery.validate.unobtrusive.js",
                    "~/Scripts/jquery.unobtrusive-ajax.js"));
        }
    }
}
```

■ **Tip** Notice that I have changed the namespace in which the class is defined in this file. The convention is that the classes defined in the files in the App_Start folder are defined in the top-level namespace for the application, which is ClientFeatures for this project.

The static RegisterBundles method is called from the Application_Start method in Global.asax—which I'll set up in the next section—when the MVC Framework application first starts. The RegisterBundles method takes a BundleCollection object, which I use to register new bundles of files through the Add method.

■ **Tip** The classes that are used for creating bundles are contained in the `System.Web.Optimization` namespace and, as I write this, the MSDN API documentation for this namespace isn't easy to find. You can navigate directly to `http://msdn.microsoft.com/en-us/library/system.web.optimization.aspx` if you want to learn more about the classes in this namespace.

I can create bundles for script files and for style sheets and it is important that I keep these types of files separate because the MVC Framework optimizes the files differently. Styles are represented by the `StyleBundle` class and scripts are represented by the `ScriptBundle` class.

When you create a new bundle, you create an instance of either `StyleBundle` or `ScriptBundle`, both of which take a single constructor argument that is the path that the bundle will be referenced by. The path is used as a URL for the browser to request the contents of the bundle, so it is important to use a scheme for your paths that won't conflict with the routes your application supports. The safest way to do this is to start your paths with `~/bundles` or `~/Content`. (The importance of this will become apparent as I explain how bundles work.)

Once you have created the `StyleBundle` or `ScriptBundle` objects, you use the `Include` method to add details of the style sheets or script files that the bundle will contain. There are some nice features available for making your bundles flexible.

I started by creating a `StyleBundle` with the `~/Content/css` path. I want this bundle to include all the CSS files in the application, so I passed the value `~/Content/*.css` as the argument to the `Include` method. The asterisk (*) character is a wild card, which means that the bundle refers to all of the CSS files in the `/Content` folder of the project. This is an excellent way of ensuring that files in a directory are automatically included in a bundle and where the order in which the files are loaded isn't important.

■ **Tip** The order in which the browser loads the Bootstrap CSS files isn't important, so using a wildcard is just fine. But if you are relying on the CSS style precedence rules, then you need to list the files individually to ensure a specific order, which is what I did for the JavaScript files.

The other bundle in the `BundleConfig.cs` file is a `ScriptBundle` whose path I set to `~/bundles/clientfeaturesscripts`. You will see the paths for both bundles again when I apply them to the application shortly. I used the `Include` method for this bundle to specify individual JavaScript files, separated by commas. I could have used another wildcard, but the order in which JavaScript files are processed usually matters and so I have listed out the individual files. Notice how I specified the jQuery library file:

```
...
~/Scripts/jquery-{version}.js
...
```

The `{version}` part of the file name is pretty handy because it matches any version of the file specified and it uses the configuration of the application to select either the regular or minified version of the file (which I'll explain shortly). The version I installed of the jQuery library is 1.10.2, which means that the bundle will include the `/Scripts/jquery-1.10.2.js` file

The benefit of using `{version}` is that you can update the libraries you use to new versions without having to redefine your bundles. The drawback is that the `{version}` token isn't able to differentiate between two versions of the same library in the same directory. So, for example, if I were to add the `jquery-2.0.2.js` file to the `Scripts` folder, I would end up with both the 1.10.2 and 2.0.2 files being shipped to the client. Since this would undermine the goal of the optimization, I must ensure that only one version of the library is in the `/Scripts` folder.

■ **Note** The jQuery team has done something unusual with their version numbering and is maintaining two different development branches. As of jQuery 1.9, the jQuery 1.x and 2.x branches have the same API, but the jQuery 2.x release doesn't support older Microsoft browsers. You should use the 1.x release in your projects unless you are sure that none of your users are stuck with Internet Explorer versions 6, 7 or 8. For more details see my *Pro jQuery 2.0* book, published by Apress.

Since I used the Empty template to create the example project, I need to add a statement to the Global.asax file to call the RegisterBundles method in the BundleConfig class, as shown in Listing 26-6.

Listing 26-6. Setting Up the Bundles in the Global.asax File

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

namespace ClientFeatures {

    public class MvcApplication : System.Web.HttpApplication {
        protected void Application_Start() {
            AreaRegistration.RegisterAllAreas();
            RouteConfig.RegisterRoutes(RouteTable.Routes);
            BundleConfig.RegisterBundles(BundleTable.Bundles);,bv
        }
    }
}
```

Applying Bundles

The first thing I need to do, before I can apply the bundles, is make sure that the namespace that contains the bundle-related classes is available for use within my view. To do this, I added an entry to the pages/namespaces element in the Views/web.config file, as shown in Listing 26-7.

Listing 26-7. Adding the Bundles Namespace to the Web.config File

```
...
<pages pageBaseType="System.Web.Mvc.WebViewPage">
  <namespaces>
    <add namespace="System.Web.Mvc" />
    <add namespace="System.Web.Mvc.Ajax" />
    <add namespace="System.Web.Mvc.Html" />
    <add namespace="System.Web.Routing" />
    <add namespace="System.Web.Optimization"/>
    <add namespace="ClientFeatures" />
  </namespaces>
</pages>
...
```

You won't need to do this if you are using one of the more complex Visual Studio project templates, but Visual Studio doesn't set this up automatically when the Empty template is used.

The next step is to apply the bundles to the layout. You can see the changes I made to the _Layout.cshtml file in Listing 26-8.

Listing 26-8 Applying Bundles to the _Layout.cshtml File

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
    <style>
        .field-validation-error { color: #f00; }
        .validation-summary-errors { color: #f00; font-weight: bold; }
        .input-validation-error { border: 2px solid #f00; background-color: #fee; }
        input[type="checkbox"].input-validation-error { outline: 2px solid #f00; }
        div.hidden { display: none; }
        div.visible { display: block; }
    </style>
    @Styles.Render("~/Content/css")
    @Scripts.Render("~/bundles/clientfeaturesscripts")
    @RenderSection("Scripts", false)
</head>
<body>
    @RenderSection("Body")
</body>
</html>
```

Bundles are added using the @Scripts.Render and @Styles.Render helper methods and you can see how I have used these helpers to replace the link and script elements with the combined bundles of files.

■ **Tip** Notice that I have left the Scripts section in the layout so that the view can define its inline code. You can mix and match bundles and regular script and link elements freely, although you should consider moving inline code and styles to external files to maximize the optimization that the MVC Framework can perform, which I describe shortly.

You can see the HTML that these helper methods generate by starting the application, navigating to the /Home/MakeBooking URL, and viewing the page source. Here is the output produced by the Styles.Render method for the ~/Content/css bundle:

```
...
<link href="/Content/bootstrap-theme.css" rel="stylesheet"/>
<link href="/Content/bootstrap.css" rel="stylesheet"/>
...
```

And here is the output produced by the `Scripts.Render` method:

```
...
<script src="/Scripts/jquery-1.10.2.js"></script>
<script src="/Scripts/jquery.unobtrusive-ajax.js"></script>
<script src="/Scripts/jquery.validate.js"></script>
<script src="/Scripts/jquery.validate.unobtrusive.js"></script>
...
```

Optimizing the JavaScript and CSS Files

Organizing JavaScript and CSS files into related groups is a useful way to make sure that you don't forget to include a file and that your layouts include whatever version of a file that is included in the project. But the real magic of bundles is that they can be used to optimize the delivery of JavaScript and CSS content to the browser.

The key to this is contained in the `Web.config` file (the one in the root folder this time) and the `debug` attribute of the `compilation` element. Open the `Web.config` file and set the attribute value to `false`, as shown in Listing 26-9.

Listing 26-9. Disabling Debug Mode in the Web.config File

```
...
<system.web>
    <compilation debug="false" targetFramework="4.5.1" />
    <httpRuntime targetFramework="4.5.1" />
</system.web>
...
```

When the `debug` attribute is set to `true`, the HTML sent to the browser contains `link` and `script` elements for individual files. When the attribute is `false`, the minified versions of the files are selected and concatenated together so that they can be delivered to the client as a blob.

■ **Note** Minification processes a JavaScript or CSS file to remove whitespace and, in the case of JavaScript files, shortens the variable and function names so that the files require less bandwidth to transfer. Most libraries provide both debug (i.e., human-readable) and minified versions of their files, which is why the `Scripts` folder contains, for example, the `jquery-validate.js` and `jquery-validate.min.js` file. The extra `.min` in the file name denotes the minified file. The selection between the files is automatic and, for the most part, minification is a simple and successful process. Some advanced libraries (such as one of my favorites, AngularJS), however, require a special minification process. Caution is recommended.

Select `Start Without Debugging` from the Visual Studio `Debug` menu. (You can't run the debugger when the debug attribute is set to `false`.) Navigate to the `/Home/MakeBooking` URL and press the F12 key to bring up the developer tools. Switch to the network tab, clear the browser cache and then click the green arrow button to start recording the requests made by the browser. Reload the page to see the effect of setting the `debug` attribute to `false`, which Figure 26-4 illustrates.

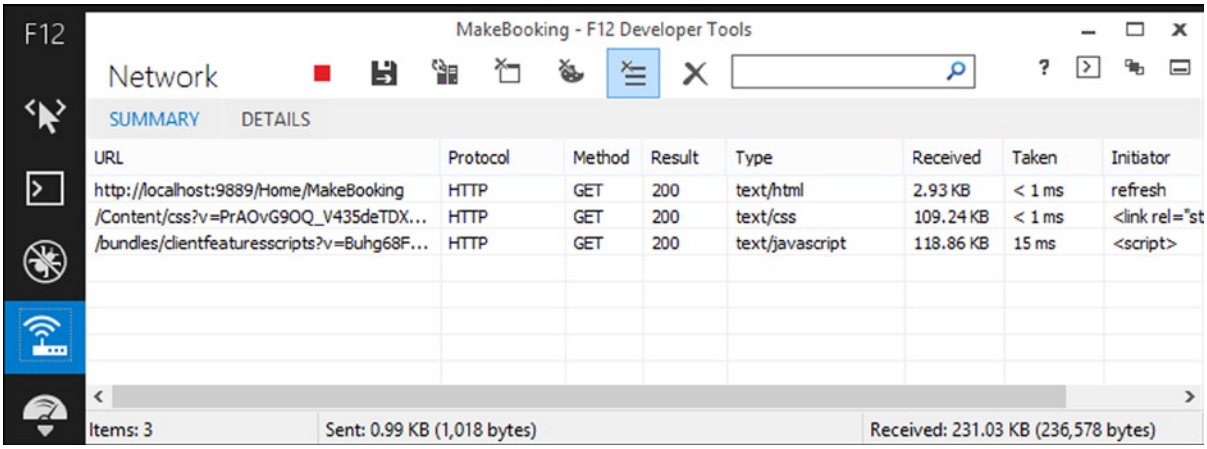

Figure 26-4. *Profiling with bundles*

Here is the summary of the profile information:

- The browser made three requests for the `/Home/MakeBooking` URL.
- There was one request for a CSS file.
- There was one request for a JavaScript file.
- A total of 1,018 bytes were sent from the browser to the server.
- A total of 236,578 bytes were sent from the server to the browser.

That's not bad. I have shaved about 50 percent off the amount of data that is sent to the browser just by letting ASP.NET and the MVC Framework optimize my JavaScript and CSS files. You can see how this works if you look at the HTML that the application renders. Here is the HTML that the `Styles.Render` method has produced:

```
...
<link href="/Content/css?v=PrAOvG9OQ_V435deTDX5p8RzKE4Gs8_LEeYxl29skhc1"
    rel="stylesheet"/>
...
```

And here is the HTML produced by the `Scripts.Render` method:

```
...
<script
    src="/bundles/clientfeaturesscripts?v=Buhg68FkCPk3xjXtPsE87M94MTb7DCZx3zKAYDOxRIA1">
</script>
...
```

These long URLs are used to request the contents of a bundle in a single blob of data. The MVC Framework minifies CSS data differently from JavaScript files, which is why I have to keep style sheets and scripts in different bundles.

The impact of the optimizations is significant. I have far fewer requests from the browser (which reduces the amount of data sent to the client) and I have less data sent in return—all of which helps keep down the cost of running the web application.

This is the point where I usually stop optimizing the requests. I could go further: moving the inline scripts into separate files so that they can be minified, for example. But I don't like to optimize too heavily unless I have a tangible problem to solve. Each optimization makes the application harder to debug and harder to maintain.

Summary

In this chapter, I showed you the bundles feature, which can help manage the JavaScript and CSS files in an application and optimize their delivery to the client. In the next chapter, I will show you the Web API (which makes it easy to create Web services that clients can consume), which is the foundation for *single page applications*.