Microsoft

Updated for ASP.NET MVC 3

Programming Microsoft® ASP.NET MVC 2

SECOND EDITION

Dino Esposito

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Control Number: 2011940367
ISBN: 978-0-7356-6284-1

Printed and bound in the United States of America.

First Printing

Microsoft Press books are available through booksellers and distributors worldwide. If you need support related to this book, email Microsoft Press Book Support at mspinput@microsoft.com. Please tell us what you think of this book at http://www.microsoft.com/learning/booksurvey.

Acquisitions Editor: Devon Musgrave
Developmental Editor: Devon Musgrave
Project Editor: Devon Musgrave
Copy Editor: Roger LeBlanc
Indexer: Christina Yeager
Editorial Production: Waypoint Press
Cover: Twist Creative • Seattle

To Silvia and my back for sustaining me.

Contents at a Glance

Table of Contents

Introduction

Get your facts first, and then you can distort them as much as you please.

—Mark Twain

Until late 2008, I was happy enough with Web Forms. I did recognize its weak points and could nicely work around them with discipline and systematic application of design principles. But a new thing called ASP.NET MVC was receiving enthusiastic reviews by a growing subset of the ASP.NET community. So I started to consider ASP.NET MVC and explore its architecture and potential while constantly trying to envision concrete business scenarios in which to employ it. I did this for about a year. Then I switched to ASP.NET MVC.

ASP.NET was devised in the late 1990s at a time when many companies in various industry sectors were rapidly discovering the Internet. For businesses, the Internet was a real breakthrough, making possible innovations in software infrastructure, marketing, distribution, and communication that were impractical or impossible before. Built on top of classic Active Server Pages (ASP), ASP.NET was the right technology at the right time, and it marked a turning point for the Web industry as a whole. For years, being a Web developer meant gaining a skill set centered on HTML and JavaScript and that was, therefore, radically different from the skills required for mainstream programming, which at the time was mostly based on C/C++, Java, and Delphi languages. ASP.NET combined the productivity of a visual and RAD environment with a component-based programming model. The primary goal of ASP.NET was to enable developers to build applications quickly and effectively without having to deal with low-level details such as HTTP, HTML, and JavaScript intricacies. That was exactly what the community loudly demanded in the late 1990s. And ASP.NET is what Microsoft delivered to address this request, exceeding expectations by a large extent.

Ten years later, today, ASP.NET is showing signs of age. The Web Forms paradigm still allows you to write functional applications, but it makes it harder and harder to stay in sync with new emerging standards, including both W3C recommendations and *de facto* industry standards. Today's sites raise the bar of features high and demand

things like full accessibility, themeability, Ajax, and browser independence, not to mention support for new tags and features as those coming up with HTML 5 and the fast-growing mobile space.

Today, you can still use Web Forms in one way or another to create accessible sites that can be skinned with CSS, offer Ajax capabilities, and work nearly the same across a variety of browsers. Each of these features, however, is not natively supported and incorporated in ASP.NET Web Forms, and this contributes to making the resulting application more fragile and brittle. For this reason, a new foundation for Web development is needed. ASP.NET MVC is the natural follow-up for ASP.NET developers—even though Web Forms will still be there and improved version after version to the extent that it is possible.

This leads me to another thought. From what I can see, most people using Web Forms are maintaining applications written for ASP.NET 2.0 and topped with some Ajax extensions. Web Forms will continue to exist for legacy projects; I'm not really sure that for new projects that the small changes we had in ASP.NET 4 and those slated for ASP.NET 5.0 will really make a difference. The real big change is switching to ASP.NET MVC. Again, that's just the natural follow up for ASP.NET developers.

Who Should Read This Book

This book is not for absolute beginners, but I do feel it is a book for everyone else, including current absolute beginners when they're no longer beginners. The higher your level of competency and expertise is, the less you can expect to find here that adds value in your particular case. However, this book comes after a few years of real-world practice, so I'm sure it has a lot of solutions that may appeal also the experts. What I can say today is that there are aspects of the solutions presented in this book that go beyond ASP.NET MVC 4, at least judging from the publicly available roadmap.

If you do ASP.NET MVC, I'm confident that you will find something in this book that makes it worth the cost.

Assumptions

The ideal reader of this book fits the following profile to some degree. The reader has played a bit with ASP.NET MVC (the version doesn't really matter) and is familiar with ASP.NET programming because of Web Forms development. The statement "Having

played a bit with ASP.NET MVC" raises the bar a bit higher than ground level and specifically means the following:

- The reader understands the overall structure of an ASP.NET MVC project (for example, what controllers and views are for).
- The reader compiled a HelloWorld site and modified it a bit.
- The reader can securely tweak a web.config or global.asax file.

Anything beyond this level of familiarity is not a contra-indication for using this book. I built the book (and the courseware based on it) so that everyone beyond a basic level of knowledge can find some value in it. Rest assured that the value a seasoned architect can get out of it is different from the value the book has for an experienced developer.

In addition, the book also works for everybody who is familiar with the MVC pattern but not specifically with the ASP.NET platform. Clearly, readers with this background won't find in this book a step-by-step guide to the ASP.NET infrastructure, but once they attain such knowledge from other resources (such as another recent book of mine published by Microsoft Press, *Programming Microsoft ASP.NET 4*), they can get the same value from reading this book as other readers.

Who Should Not Read This Book

The ideal reader of this book should not be looking for a step-by-step guide to ASP.NET MVC. The book's aim is to explain the mechanics of the framework and effective ways to use it. It skims through basic steps. If you think you need a beginner's guide, well, you probably will find this book a bit disappointing. You might not be able to see the logical flow of chapters and references and you could get lost quite soon. If you're a beginner, I recommend you flip through the pages and purchase a copy only if you see something that will help you in a specific or immediate way (for example, material that helps you solve a problem you are currently experiencing). In this case, the book has helped you accomplish something significant.

System Requirements

You will need the following hardware and software to compile and run the code associated with this book:

- One of Windows XP with Service Pack 3 (except Starter Edition), Windows Vista with Service Pack 2 (except Starter Edition), Windows 7, Windows Server 2003 with Service Pack 2, Windows Server 2003 R2, Windows Server 2008 with Service Pack 2, or Windows Server 2008 R2.
- Visual Studio 2010, any edition (multiple downloads may be required if using Express Edition products).
- SQL Server 2008 Express Edition or higher (2008 or R2 release), with SQL Server Management Studio 2008 Express or higher (included with Visual Studio, Express Editions require separate download). For a couple of examples, you might need to install the Northwind database within SQL Server. The database is included in the package. After installing the Northwind database in SQL Server, you might also want to edit the connection string as required.
- Computer that has a 1.6 GHz or faster processor (2 GHz recommended).
- 1 GB (32 Bit) or 2 GB (64 Bit) RAM (Add 512 MB if running in a virtual machine or SQL Server Express Editions, more for advanced SQL Server editions).
- 3.5 GB of available hard disk space.

Code Samples

This book features a companion website that makes available to you all the code used in the book. This code is organized by chapter, and you can download it from the companion site at this address:

http://go.microsoft.com/FWLink/?Linkid=230567

Follow the instructions to download the Mvc3-SourceCode.zip file.

Errata & Book Support

We've made every effort to ensure the accuracy of this book and its companion content. Any errors that have been reported since this book was published are listed on our Microsoft Press site at oreilly.com:

http://go.microsoft.com/FWLink/?Linkid=230565

If you find an error that is not already listed, you can report it to us through the same page.

If you need additional support, email Microsoft Press Book Support at *mspinput@microsoft.com*.

Please note that product support for Microsoft software is not offered through the addresses above.

We Want to Hear from You

At Microsoft Press, your satisfaction is our top priority, and your feedback our most valuable asset. Please tell us what you think of this book at:

http://www.microsoft.com/learning/booksurvey

The survey is short, and we read every one of your comments and ideas. Thanks in advance for your input!

Stay in Touch

Let's keep the conversation going. We're on Twitter: *http://twitter.com/MicrosoftPress*

Acknowledgments

The man who doesn't read good books has no advantage over the man who can't read them.

—Mark Twain

This is a book that I had no plans to write. It was Devon Musgrave who pushed me to update the previous edition, which was based on MVC 2. We looked at some Amazon reviews and we found out that there were some things in the previous edition that needed some fixing. Yes, feedback does help, and even though book reviews are not always crystal clear in their origin (there could be anybody behind a nickname), ideas expressed are always an asset.

So I looked over some of those reviews and critically reviewed the old book, chapter by chapter. And I found a few things to fix; not coincidentally, the same things I changed along the way in my ASP.NET MVC courseware. The fundamental change that hopefully makes this book far more valuable than the previous edition is that I managed to move the focus from the infrastructure to actual coding.

I wrote quite a few books that people found useful and helpful in their ability to understand the underlying machinery of a technology. This is not a winning point for a substantial part of the ASP.NET MVC audience. Most ASP.NET MVC developers have significant experience and excellent skills; they may not know ASP.NET MVC in detail, but they know a lot about Web programming and they're quick learners. They need to ramp up on ASP.NET MVC and understand its intricacies and they don't see the point of studying the underpinnings of the framework. So Devon guided me to refresh the book to give it a different slant. This book ended up as a complete rewrite; not simply a refresh. But now I'm really proud of this new baby. And I hope it addresses some of the nicknames (hopefully, real people) who reviewed and commented the MVC 2 book on Amazon a few months ago.

Marc Young took the responsibility of ensuring the technical quality of the book. And he pushed me hard on making the companion code a super-quality product, which is much better organized than in the past. (I admit I tend to be as lazy on companion code as I tend to be deep—and sometimes repetitive—on concepts.)

I have a joke about my English in every book. I write over and over again how bad my English is and how great Roger LeBlanc is in making it good. After a decade spent writing books in English I really think that it's now good enough to keep Roger's work to a minimum. And, in fact, in this book Roger played the wider role of managing editor.

Steve Sagman has been like a background task pushing notifications timely. I made most of the promised deadlines, but Steve has been flexible enough to adjust deadlines so that it seemed that I made all of them. Working with Steve is kind of relaxing; he never transmits pressure but he kicks in at the right time; which is probably the secret trick to not adding pressure.

Like millions of other Italian students, I spent many teenage hours trying to catch the spirit of the Divine Comedy. As you may know, the whole poem develops around a journey that Dante undertakes through the three realms of the dead guided by the Roman poet Virgilio. I too spent many hours of my past months trying to catch and express the gist of ASP.NET MVC. I began a journey through controllers, views, models and filters guided by a top-notch developer, trainer and friend—Hadi Hariri.

Loyal readers of my books may know about my (insane) passion for tennis. My wife Silvia told me once "OK, you like tennis so much, but is there any chance that you can make some money from it?" I never dared ask whether she meant "making money playing and winning tournaments" or "making money through software." To be on the safe side, I decided to train and play a lot more while spending many hours helping out Giorgio Garcia and the entire team at Crionet and e-tennis.net to serve better Web and mobile services to tennis tournaments and their fans. I joined Crionet as the Chief Technical Officer and I'm really enjoying going out for tournaments and focusing on domain logic of a tennis game. It was really nice last June to make it to the Wimbledon's Centre Court and claim it was for work and not for fun!

My son Francesco (13) is now officially a junior Windows Phone 7 developer with five applications already published to the marketplace. By the way, check out the nicest of his apps—ShillyShally, a truly professional tool for decision makers. He doesn't do much Web programming now, but he's pushing me hard for a mobile book—which is exactly one of my ongoing projects as I write these notes. If you do, or plan to do, mobile stay tuned or, better yet, get in touch.

Michela (10) is simply the perfect end user in this crazy technological world and a wonderful lover of German shepherds and baby tigers.

PART I

ASP.NET MVC Fundamentals

CHAPTER 1

ASP.NET MVC Controllers

They always say time changes things, but you actually have to change them yourself.

—Andy Warhol

ASP.NET Web Forms started getting old the day that Ajax conquered the masses. As some have said, Ajax has been the poisonous arrow shot in the heel of ASP.NET—another Achilles. Ajax made getting more and more control over HTML and client-side code a true necessity. Over time, this led to different architectures and made ASP.NET Web Forms a little less up to the task with each passing day.

Based on the same run-time environment as Web Forms, ASP.NET MVC makes developing web applications a significantly different experience. At its core, ASP.NET MVC just separates behavior from the generation of the response—a simple change, but one that has a huge impact on applications and developers. ASP.NET MVC is action-centric, disregards the page-based architecture of Web Forms, and pushes a web-adapted implementation of the classic Model-View-Controller pattern.

In ASP.NET MVC, each request results in the execution of an action—ultimately, a method on a specific class. Results of executing the action are passed down to the view subsystem along with a view template. The results and template are then used to build the final response for the browser. Users don't point the browser to a page; users just place a request. Doesn't that sound like a big change?

So everything looks different for developers in the beginning, but everything looks sort of familiar after a bit of practice. Your actions can serve HTML as well as any other type of response, including JSON, script, graphic, and binary files. You don't have to forgo using roles on methods, forms authentication, session state, and cache—the run-time environment is the same, and MVC and Web Forms applications can happily coexist on the same site.

Unlike Web Forms, ASP.NET MVC is made of various layers of code connected together but not intertwined and not forming a single monolithic block. For this reason, it's easy to replace any of these layers with custom components that enhance the maintainability as well as the testability of the solution. With ASP.NET MVC, you gain total control over the markup and can apply styles and inject script code at will using the JavaScript frameworks you like most.

The bottom line is that although you might decide to keep using Web Forms, for today's web development ASP.NET MVC is a much better choice. Worried about productivity? My best advice is

that you start making the transition as soon as possible. You don't need to invest a huge amount of time, but you need to understand exactly what's going on and the philosophy behind MVC. If you do that, any investment you make will pay you back sooner than you expect.

ASP.NET MVC doesn't change the way a web application works on the ASP.NET and Internet Information Services (IIS) platforms. ASP.NET MVC, however, changes the way developers write web applications. In this chapter, you'll discover the role and structure of the controller—the foundation of ASP.NET MVC applications—and how requests are routed to controllers.

Note This book is based on ASP.NET MVC 3. This version of ASP.NET MVC is backward compatible with the previous version, MVC 2. This means you can install both versions side by side on the same machine and play with the new version without affecting any existing MVC code you might have already. Of course, the same point holds for web server machines. You can install both ASP.NET MVC 2 and ASP.NET MVC 3 on the same server box without unpleasant side effects. The same level of backware compatibility is expected with the upcoming version, MVC 4.

Routing Incoming Requests

Originally, the whole ASP.NET platform was developed around the idea of serving requests for physical pages. It turns out that most URLs used within an ASP.NET application are made of two parts: the path to the physical Web page that contains the logic, and some data stuffed in the query string to provide parameters. This approach has worked for a few years, and it still works today. The ASP.NET run-time environment, however, doesn't limit you to just calling into resources identified by a specific location and file. By writing an ad hoc HTTP handler and binding it to a URL, you can use ASP.NET to execute code in response to a request regardless of the dependencies on physical files. This is just one of the aspects that most distinguishes ASP.NET MVC from ASP.NET Web Forms. Let's briefly see how to simulate the ASP.NET MVC behavior with an HTTP handler.

Note In software, the term *URI* (short for Uniform Resource Identifier) is used to refer to a resource by location or a name. When the URI identifies the resource by location, it's called a *URL*, or Uniform Resource Locator. When the URI identifies a resource by name, it becomes a *URN*, or Uniform Resource Name. In this regard, ASP.NET MVC is designed to deal with more generic URIs, whereas ASP.NET Web Forms was designed to deal with location-aware physical resources.

Simulating the ASP.NET MVC Runtime

Let's build a simple ASP.NET Web Forms application and use HTTP handlers to figure out the internal mechanics of ASP.NET MVC applications. You can start from the basic ASP.NET Web Forms application you get from your Microsoft Visual Studio project manager.

Defining the Syntax of Recognized URLs

In a world in which requested URLs don't necessarily match up with physical files on the web server, the first step to take is listing which URLs are meaningful for the application. To avoid being too specific, let's assume you support only a few fixed URLs, each mapped to an HTTP handler component. The following code snippet shows the changes required to be made to the default *web.config* file:

```
<httpHandlers>
   <add verb="*"
        path="home/test/*"
        type="MvcEmule.Components.MvcEmuleHandler" />
</httpHandlers>
```

Whenever the application receives a request that matches the specified URL, it will pass it on to the specified handler.

Defining the Behavior of the HTTP Handler

In ASP.NET, an HTTP handler is a component that implements the *IHttpHandler* interface. The interface is simple and consists of two members, as shown here:

```
public class MvcEmuleHandler : IHttpHandler
{
   public void ProcessRequest(HttpContext context)
   {
      // Logic goes here
      ...
   }

   public Boolean IsReusable
   {
      get { return false; }
   }
}
```

Most of the time, an HTTP handler has a hardcoded behavior influenced only by some input data passed over the query string. Nothing prevents us, however, from using the handler as an abstract factory for adding one more level of indirection. The handler, in fact, can use information from the request to determine an external component to call to actually serve the request. In this way, a single HTTP handler can serve a variety of requests and just dispatch the call among a few more specialized components.

The HTTP handler could parse out the URL in tokens and use that information to identify the class and the method to invoke. Here's an example of how it could work:

```
public void ProcessRequest(HttpContext context)
{
    // Parse out the URL and extract controller, action, and parameter
    var segments = context.Request.Url.Segments;
    var controller = segments[1].TrimEnd('/');
    var action = segments[2].TrimEnd('/');
```

```
    var param1 = segments[3].TrimEnd('/');

    // Complete controller class name with suffix and (default) namespace
    var fullName = String.Format("{0}.{1}Controller",
                       this.GetType().Namespace, controller);
    var controllerType = Type.GetType(fullName, true, true);

    // Get an instance of the controller
    var instance = Activator.CreateInstance(controllerType);

    // Invoke the action method on the controller instance
    var methodInfo = controllerType.GetMethod(action,
          BindingFlags.Instance |
          BindingFlags.IgnoreCase |
          BindingFlags.Public);
    var result = String.Empty;
    if (methodInfo.GetParameters().Length == 0)
    {
        result = methodInfo.Invoke(instance, null) as String;
    }
    else
    {
        result = methodInfo.Invoke(instance, new Object[] { param1 }) as String;
    }

    // Write out results
    context.Response.Write(result);
}
```

The preceding code just assumes the first token in the URL past the server name contains the key information to identify the specialized component that will serve the request. The second token refers to the name of the method to call on this component. Finally, the third token indicates a parameter to pass.

Invoking the HTTP Handler

Given a URL such as *home/test/**, it turns out that *home* identifies the class, *test* identifies the method, and whatever trails is the parameter. The name of the class is further worked out and extended to include a namespace and a suffix. According to the example, the final class name is *MvcEmule.Components.HomeController*. This class is expected to be available to the application. The class is also expected to expose a method named *Test*, as shown here:

```
namespace MvcEmule.Components
{
    public class HomeController
    {
        public String Test(Object param1)
        {
            var message = "<html><h1>Got it! You passed '{0}'</h1></html>";
            return String.Format(message, param1);
        }
    }
}
```

Figure 1-1 shows the effect of invoking a page-agnostic URL in an ASP.NET Web Forms application.

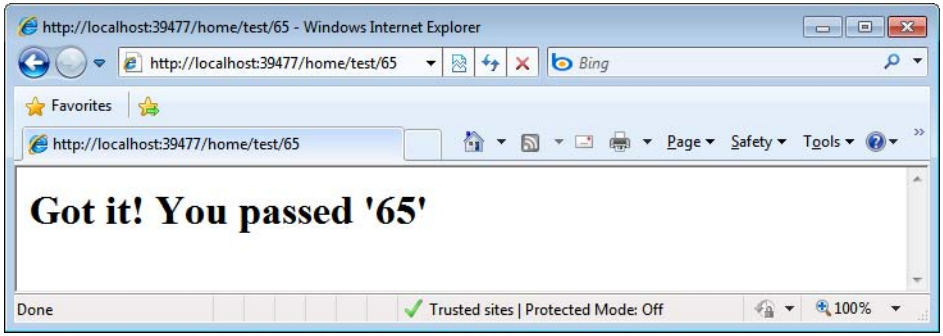

FIGURE 1-1 Processing page-agnostic URLs in ASP.NET Web Forms.

This simple example demonstrates the basic mechanics used by ASP.NET MVC.. The specialized component that serves a request is the controller. The controller is a class with just methods and no state. A unique system-level HTTP handler takes care of dispatching incoming requests to a specific controller class so that the instance of the class executes a given action method and produces a response.

What about the scheme of URLs? In this example, you just use a hardcoded URL. In ASP.NET MVC, you have a very flexible syntax you can use to express the URLs the application recognizes. In addition, a new system component in the run-time pipeline intercepts requests, processes the URL, and triggers the ASP.NET MVC HTTP handler. This component is the URL Routing HTTP module.

The URL Routing HTTP Module

The URL routing HTTP module processes incoming requests by looking at the URLs and dispatching them to the most appropriate executor. The URL routing HTTP module supersedes the *URL rewriting* feature of older versions of ASP.NET. At its core, URL rewriting consists of hooking up a request, parsing the original URL, and instructing the HTTP run-time environment to serve a "possibly related but different" URL.

Superseding URL Rewriting

URL rewriting comes into play if you need to make tradeoffs between needing human-readable and SEO-friendly URLs and needing to programmatically deal with tons of URLs. For example, consider the following URL:

```
http://northwind.com/news.aspx?id=1234
```

The *news.aspx* page incorporates any logic required to retrieve, format, and display any given news. The ID for the specific news to retrieve is provided via a parameter on the query string. As a developer, implementing the page couldn't be easier: you get the query string parameter, run the query, and create the HTML. As a user or as a search engine, by simply looking at the URL you can't really understand the intent of the page and you aren't likely to remember the address easily enough to pass it around.

URL rewriting helps you in two ways. It makes it possible for developers to use a generic front-end page, such as *news.aspx*, to display related content. In addition, it also enables users to request friendly URLs that will be programmatically mapped to less intuitive, but easier to manage, URLs. In a nutshell, URL rewriting exists to decouple the requested URL from the physical webpage that serves the requests.

In the latest version of ASP.NET 4 Web Forms, you can use URL routing to match incoming URLs to other URLs without incurring the costs of HTTP 302 redirects. In ASP.NET MVC, on the other hand, URL routing serves the purpose of mapping incoming URLs to a controller class and an action method.

Note Originally developed as an ASP.NET MVC component, the URL routing module is now a native part of the ASP.NET platform and, as mentioned, offers its services to both ASP.NET MVC and ASP.NET Web Forms applications, though through a slightly different API.

Routing the Requests

What happens exactly when a request knocks at the IIS gate? Figure 1-2 gives you an overall picture of the various steps involved and how things work differently in ASP.NET MVC and ASP.NET Web Forms applications.

The URL routing module intercepts any requests for the application that could not be served otherwise by IIS. If the URL refers to a physical file (for example, an ASPX file), the routing module ignores the request, unless it's otherwise configured. The request then falls down to the classic ASP.NET machinery to be processed as usual in terms of a page handler.

Otherwise, the URL routing module attempts to match the URL of the request to any of the application-defined routes. If a match is found, the request goes into the ASP.NET MVC space to be processed in terms of a call to a controller class. If no match is found, the request will be served by the standard ASP.NET runtime in the best possible way and likely results in an HTTP 404 error.

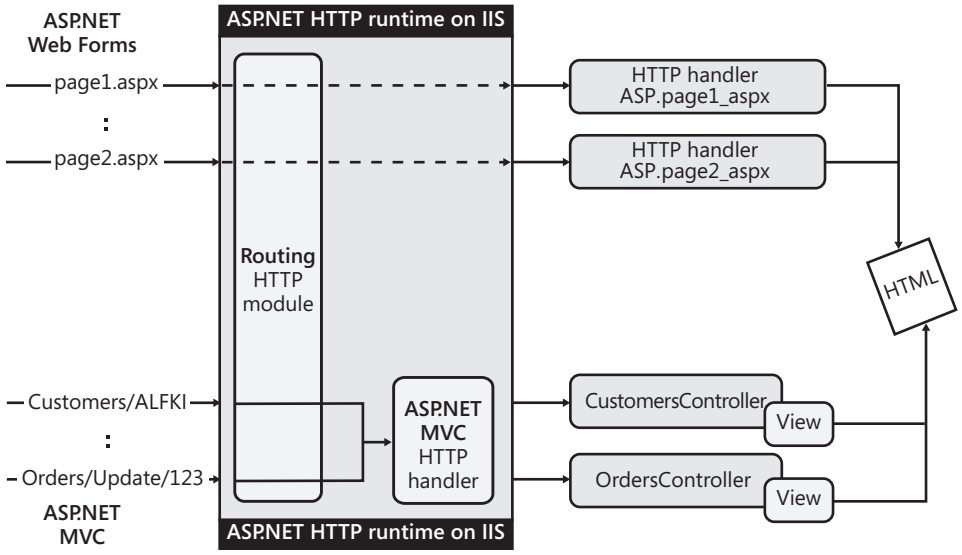

FIGURE 1-2 The role of the routing module in ASP.NET MVC.

In the end, only requests that match predefined URL patterns (also known as *routes*) are allowed to enjoy the ASP.NET MVC runtime. All such requests are routed to a common HTTP handler that instantiates a controller class and invokes a defined method on it. Next, the controller method, in turn, selects a view component to generate the actual response.

Internal Structure of the URL Routing Module

In terms of implementation, I should note that the URL routing engine is an HTTP module that wires up the *PostResolveRequestCache* event. The event fires right after checking that no response for the request is available in the ASP.NET cache.

The HTTP module matches the requested URL to one of the user-defined URL routes and sets the HTTP context to using the ASP.NET MVC standard HTTP handler to serve the request. As a developer, you're not likely to deal with the URL routing module directly. The module is system provided and doesn't need you to perform any specific form of configuration. You are responsible, instead, for providing the routes that your application supports and that the routing module will actually consume.

Application Routes

By design, an ASP.NET MVC application is not forced to depend on physical pages. In ASP.NET MVC, users place requests for acting on resources. The framework, however, doesn't mandate the syntax for describing resources and actions. I'm aware that the expression "acting on resources" will likely make you think of Representational State Transfer (REST). And, of course, you will not be too far off the mark in thinking so.

Although you can definitely use a pure REST approach within an ASP.NET MVC application, I would rather say that ASP.NET MVC is loosely REST-oriented in that it does acknowledge concepts like resource and action, but it leaves you free to use your own syntax to express and implement resources and actions. As an example, in a pure REST solution you would use HTTP verbs to express actions—GET, POST, PUT, and DELETE—and the URL to identify the resource. Implementing a pure REST solution in ASP.NET MVC is possible but requires some extra work on your part.

The default behavior in ASP.NET MVC is using custom URLs where you make yourself responsible for the syntax through which actions and resources are specified. This syntax is expressed through a collection of URL patterns, also known as *routes*.

URL Patterns and Routes

A *route* is a pattern-matching string that represents the absolute path of a URL—namely, the URL string without protocol, server, and port information. A route might be a constant string, but it will more likely contain a few placeholders. Here's a sample route:

```
/home/test
```

The route is a constant string and is matched only by URLs whose absolute path is */home/test*. Most of the time, however, you deal with parametric routes that incorporate one or more placeholders. Here are a couple of examples:

```
/{resource}/{action}
/Customer/{action}
```

Both routes are matched by any URLs that contain exactly two segments. The latter, though, requires that the first segment equals the string "Customer". The former, instead, doesn't pose specific constraints on the content of the segments.

Often referred to as a *URL parameter*, a placeholder is a name enclosed in curly brackets { }. You can have multiple placeholders in a route as long as they are separated by a constant or delimiter. The forward slash (/) character acts as a delimiter between the various parts of the route. The name of the placeholder (for example, *action*) is the key that your code will use to programmatically retrieve the content of the corresponding segment from the actual URL.

Here's the default route for an ASP.NET MVC application:

```
{controller}/{action}/{id}
```

In this case, the sample route contains three placeholders separated by the delimiter. A URL that matches the preceding route is the following:

```
/Customers/Edit/ALFKI
```

You can add as many routes as you want with as many placeholders as appropriate. You can even remove the default route.

Defining Application Routes

Routes for an application are usually registered in the *global.asax* file, and they are processed at the application startup. Let's have a look at the section of the *global.asax* file that deals with routes:

```
public class MvcApplication : HttpApplication
{
   protected void Application_Start()
   {
      RegisterRoutes(RouteTable.Routes);

      // Other code
      ...
   }

   public static void RegisterRoutes(RouteCollection routes)
   {
      // Other code
      ...

      // Listing routes
      routes.MapRoute(
         "Default",
         "{controller}/{action}/{id}",
         new {
               controller = "Home",
               action = "Index",
               id = UrlParameter.Optional
         });
   }
}
```

As you can see, the *Application_Start* event handler calls into a public static method named *RegisterRoutes* that lists all routes. Note that the name of the *RegisterRoutes* method, as well as the prototype, is arbitrary and can be changed if there's a valid reason.

Supported routes must be added to a static collection of *Route* objects managed by ASP.NET MVC. This collection is *RouteTable.Routes*. You typically use the handy *MapRoute* method to populate the collection. The *MapRoute* method offers a variety of overloads and works well most of the time. However, it doesn't let you configure every possible aspect of a route object. If there's something you need to set on a route that *MapRoute* doesn't support, you might want to resort to the following code:

```
// Create a new route and add it to the system collection
var route = new Route(...);
RouteTable.Routes.Add("NameOfTheRoute", route);
```

A route is characterized by a few attributes, such as name, URL pattern, default values, constraints, data tokens, and a route handler. The attributes you set most often are name, URL pattern, and default values. Let's expand on the code you get for the default route:

```
routes.MapRoute(
        "Default",
        "{controller}/{action}/{id}",
        new {
              controller = "Home",
              action = "Index",
              id = UrlParameter.Optional
        });
```

The first parameter is the name of the route; each route should have a unique name. The second parameter is the URL pattern. The third parameter is an object that specifies default values for the URL parameters.

Note that a URL can match the pattern even in an incomplete form. Let's consider the root URL—*http://yourserver.com*. At first sight, such a URL wouldn't match the route. However, if a default value is specified for a URL parameter, the segment is considered optional. As a result, for the preceding example, when you request the root URL, the request is resolved by invoking the method *Index* on the *Home* controller.

Processing Routes

The ASP.NET URL routing module employs a number of rules when trying to match an incoming requested URL to a defined route. The most important rule is that routes must be checked in the order they were registered in *global.asax*.

To ensure that routes are processed in the right order, you must list them from the most specific to the least specific. In any case, keep in mind that the search for a matching route always ends at the first match. This means that just adding a new route at the bottom of the list might not work and might also cause you a bit of trouble. In addition, be aware that placing a catch-all pattern at the top of the list will make any other patterns—no matter how specific—pass unnoticed.

Beyond order of appearance, other factors affect the process of matching URLs to routes. As mentioned, one is the set of default values that you might have provided for a route. Default values are simply values that are automatically assigned to defined placeholders in case the URL doesn't provide specific values. Consider the following two routes:

```
{Orders}/{Year}/{Month}
{Orders}/{Year}
```

If in the first route you assign default values for both *{Year}* and *{Month}*, the second route will never be evaluated because, thanks to the default values, the first route is always a match regardless of whether the URL specifies a year and a month.

A trailing forward slash (/) is also a pitfall. The routes *{Orders}/{Year}* and *{Orders}/{Year}/* are two very different things. One won't match to the other, even though logically, at least from a user's perspective, you'd expect them to.

Another factor that influences the URL-to-route match is the list of constraints that you optionally define for a route. A route constraint is an additional condition that a given URL parameter must fulfill to make the URL match the route. The URL not only should be compatible with the URL pattern, it also needs to contain compatible data. A constraint can be defined in various ways, including through a regular expression. Here's a sample route with constraints:

```
routes.MapRoute(
    "ProductInfo",
    "{controller}/{productId}/{locale}",
     new { controller = "Product", action = "Index", locale="en-us" },
     new { productId = @"\d{8}",
           locale = ""[a-z]{2}-[a-z]{2}" });
```

In particular, the route requires that the *productId* placeholder must be a numeric sequence of exactly eight digits, whereas the *locale* placeholder must be a pair of two-letter strings separated by a dash. Constraints don't ensure that all invalid product IDs and locale codes are stopped at the gate, but at least they cut off a good deal of work.

Route Handler

The route defines a bare-minimum set of rules according to which the routing module decides whether or not the incoming request URL is acceptable to the application. The component that ultimately decides how to remap the requested URL is another one entirely. Precisely, it is the *route handler*. The route handler is the object that processes any requests that match a given route. Its sole purpose in life is returning the HTTP handler that will actually serve any matching request.

Technically speaking, a route handler is a class that implements the *IRouteHandler* interface. The interface is defined as shown here:

```
public interface IRouteHandler
{
   IHttpHandler GetHttpHandler(RequestContext requestContext);
}
```

Defined in the *System.Web.Routing* namespace, the *RequestContext* class encapsulates the HTTP context of the request plus any route-specific information available, such as the *Route* object itself, URL parameters, and constraints. This data is grouped into a *RouteData* object. Here's the signature of the *RequestContext* class:

```
public class RequestContext
{
    public RequestContext(HttpContextBase httpContext, RouteData routeData);

    // Properties
    public HttpContextBase HttpContext { get; set; }
    public RouteData RouteData { get; set; }
}
```

The ASP.NET MVC framework doesn't offer many built-in route handlers, and this is probably a sign that the need to use a custom route handler is not that common. Yet, the extensibility point exists and, in case of need, you can take advantage of it. I'll return to custom route handlers and provide an example later in the chapter.

Handling Requests for Physical Files

Another configurable aspect of the routing system that contributes to a successful URL-to-route matching is whether or not the routing system has to handle requests that match a physical file.

By default, the ASP.NET routing system ignores requests whose URL can be mapped to a file that physically exists on the server. Note that if the server file exists, the routing system ignores the request even if the request matches a route.

If you need to, you can force the routing system to handle *all* requests by setting the *RouteExistingFiles* property of the *RouteCollection* object to *true*, as shown here:

```
// In global.asax.cs
public static void RegisterRoutes(RouteCollection routes)
{
     routes.RouteExistingFiles = true;
     ...
}
```

Note that having all requests handled via routing can create some issues in an ASP.NET MVC application. For example, if you add the preceding code to the *global.asax.cs* file of a sample ASP.NET MVC application and run it, you'll immediately face an HTTP 404 error when accessing *default.aspx*.

Preventing Routing for Defined URLs

The ASP.NET URL routing module doesn't limit you to maintaining a list of acceptable URL patterns. It also allows you to keep certain URLs off the routing mechanism. You can prevent the routing system from handling certain URLs in two steps. First, you define a pattern for those URLs and save it to a route. Second, you link that route to a special route handler—the *StopRoutingHandler* class. All it does is throw a *NotSupported* exception when its *GetHttpHandler* method is invoked.

For example, the following code instructs the routing system to ignore any *.axd* requests:

```
// In global.asax.cs
public static void RegisterRoutes(RouteCollection routes)
{
   routes.IgnoreRoute("{resource}.axd/{*pathInfo}");
   ...
}
```

All that *IgnoreRoute* does is associate a *StopRoutingHandler* route handler to the route built around the specified URL pattern.

Finally, a little explanation is required for the *{*pathInfo}* placeholder in the URL. The token *pathInfo* simply represents a placeholder for any content following the *.axd* URL. The asterisk (*),

though, indicates that the last parameter should match the rest of the URL. In other words, anything that follows the *.axd* extension goes into the *pathInfo* parameter. Such parameters are referred to as *catch-all parameters*.

The Controller Class

In spite of the explicit reference to the Model-View-Controller pattern in the name, the ASP.NET MVC architecture is essentially centered on one pillar—the controller. The controller governs the processing of a request and orchestrates the back end of the system (for example, business layer, services, data access layer) to grab raw data for the response. Next, the controller wraps up raw data computed for the request into a valid response for the caller. When the response is a markup view, the controller relies on the view engine module to combine data and view templates and produce HTML.

Aspects of a Controller

Any request that passes the URL routing filter is mapped to a controller class and served by executing a given method on the class. Therefore, the controller class is the place where developers write the actual code required to serve a request. Let's briefly explore some characterizing aspects of controllers.

Granularity of Controllers

An ASP.NET MVC application is usually made of a variety of controller classes. How many controllers should you have? The actual number is up to you and depends only on how you want to organize your application's actions. In fact, you could arrange an application around a single controller class that contains methods for any possible requests.

A common practice consists of having a controller class for each significant object your application manipulates. For example, you can have a *CustomerController* class that takes care of requests related to querying, deleting, updating, and inserting customers. Likewise, you create a *ProductController* class for dealing with products, and so forth. Most of the time, these objects are directly related to items in the application's main menu.

In general, you can say that the granularity of the controller is a function of the granularity of the user interface. Plan to have a controller for each significant source of requests you have in the user interface.

Stateless Components

A new instance of the selected controller class is instantiated for each request. Any state you might add to the class is bound to the same lifetime of the request. The controller class then must be able to retrieve any data it needs to work from the HTTP request stream and the HTTP context.

Further Layering Is Up to You

Often ASP.NET MVC and controller classes are presented as a magic wand you wave to write layered code that is cleaner and easier to read and maintain. The stateless nature of the controller class helps a lot in this regard, but it is not enough.

In ASP.NET MVC, the controller is isolated from both the user interface that triggered the request and the engine that produces the view for the browser. The controller sits in between the view and the back end of the system. Although this sort of isolation from the view is welcome and fixes a weak point of ASP.NET Web Forms, it alone doesn't ensure that your code will be respectful of the venerable principle of separation of concerns (SoC).

The system gets you a minimal level of separation from the view—everything else is up to you. Keep in mind that nothing, not even in ASP.NET MVC, prevents you from using direct ADO.NET calls and plain T-SQL statements directly in the controller class. The controller class is not the back end of the system, and it is not the business layer. It should be considered, instead, as the MVC counterpart of the code-behind class of Web Forms. As such, it definitely belongs to the presentation layer, not the business layer.

Highly Testable

The inherent statelessness of the controller, and its neat separation from the view, make the controller class potentially easy to test. However, the real testability of the controller class should be measured against its effective layering. Let's have a look at Figure 1-3.

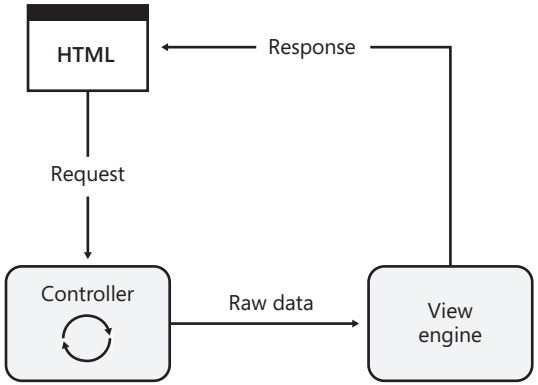

FIGURE 1-3 Controllers and views in ASP.NET MVC.

Although the controller class can be easily fed any fixed input you like and its output can be asserted without major issues, nothing can be said about the internal structure of action methods. The more the implementation of these methods is tightly bound to external resources (for example, databases, services, components), the less likely it is that testing a controller will be quick and easy.

Writing Controller Classes

The writing of a controller class can be summarized in two simple steps: creating a class that inherits (either directly or indirectly) from *Controller* and adding a bunch of public methods. However, a couple of important details must be clarified: how the system gets to know the controller class to instantiate and how it figures out the method to invoke.

From Routing to Controllers

Regardless of how you define your URL patterns, any request must always be resolved in terms of a controller name and an action name. This is one of the pillars of ASP.NET MVC. The controller name is automatically read from the URL if the URL includes a *{controller}* placeholder. The same happens for action names if the URL contains an *{action}* placeholder.

Having completely custom URLs devoid of such placeholders is still acceptable, though. In this case, however, it is your responsibility to indicate the controller and action through default values as shown here:

```
routes.MapRoute(
     "SampleRoute",
     "about",
     new { controller = "Home", action = "About"}
);
```

If controller and action names can't be resolved in a static way, you might want to write a custom route handler, explore the details of the request, and figure out controller and action names. Then you just store them in the *RouteData* collection, as shown here:

```
public class AboutRouteHandler : IRouteHandler
{
    public IHttpHandler GetHttpHandler(RequestContext requestContext)
    {
        if (requestContext.HttpContext.Request.Url.AbsolutePath == "/about")
        {
            requestContext.RouteData.Values["controller"] = "home";
            requestContext.RouteData.Values["action"] = "about";
        }
        return new MvcHandler(requestContext);
    }
}
```

For a route that requires a custom handler, the registration process is a bit different from what you saw earlier. Here's the code you need to have in *RegisterRoutes*:

```
public static void RegisterRoutes(RouteCollection routes)
{
    var aboutRoute = new Route("about", new AboutRouteHandler());
    routes.Add("SampleAboutRoute", aboutRoute);
    ...
}
```

Be sure to note that the controller name you obtain from the routing module doesn't match exactly the actual name of class that will be invoked. By default, the controller class is named after the controller name with a *Controller* suffix added. In the previous example, if *home* is the controller name, the class name is assumed to be *HomeController*. Note that conventions apply not just to the class name but also to the namespace. In particular, the class is expected to be scoped in the *Controllers* namespace under the default project namespace.

Note When you add a route based on a custom route handler that sets controller and action names programmatically, you might run into trouble with the links generated by the *Html.ActionLink* helper. You commonly use this helper to create route-based links for menus and other visual elements of the user interface. If you add a route with a custom handler, you might be surprised to see that the links you get from the helper are unexpectedly based on this route. To solve the issue, either you change *ActionLink* with *RouteLink* and expressly indicate which route you want the URL to be created after, or you specify in the custom route that *controller* and *action* are optional parameters.

From Routing to Actions

When the ASP.NET MVC run-time environment has a valid instance of the selected controller class, it yields to the action invoker component for the actual execution of the request. The action invoker gets the action name and attempts to match it to a public method on the controller class.

The *action* parameter indicates the name of the action to perform. Most of the time, the controller class just has a method with the same name. If this is the case, the invoker will execute it. Note, though, that you can associate an action name attribute to any public method, thus decoupling the method name from the action name. Here's an example:

```
public class HomeController : Controller
{
    // Implicit action name: Index
    public ActionResult Index()
    {
       ...
    }

    [NonAction]
    public ActionResult About()
    {
       ...
    }

    [ActionName("About")]
    public ActionResult LikeGermanSheperds()
    {
       ...
    }
}
```

The method *Index* is not decorated with attributes, so it is implicitly bound to an action with the same name. The third public method has a very fancy name, but it is explicitly bound to the action *about* via the *ActionName* attribute. Finally, note that to prevent a public controller method from being implicitly bound to an action name, you use the *NonAction* attribute. Given the previous code snippet, therefore, when the user requests the *about* action, the method *LikeGermanSheperds* runs regardless of the HTTP verb used to place the request.

Actions and HTTP Verbs

ASP.NET MVC is flexible enough to let you bind a method to an action for a specific HTTP verb. To associate a controller method with an HTTP verb, you either use the parametric *AcceptVerbs* attribute or direct attributes such as *HttpGet*, *HttpPost*, and *HttpPut*. The *AcceptVerbs* attribute allows you to specify which HTTP verb is required to execute a given method. Let's consider the following example:

```
[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Edit(Customer customer)
{
   ...
}
```

Given that code, it turns out that the *Edit* method can't be invoked using a GET. Note also that you are not allowed to have multiple *AcceptVerbs* attributes on a single method. Your code won't compile if you add multiple *AcceptVerbs* attributes (or analogous direct HTTP verb attributes) to an action method.

The *AcceptVerbs* attribute takes any value from the *HttpVerbs enum* type:

```
public enum HttpVerbs
{
    Get = 1,
    Post = 2,
    Put = 4,
    Delete = 8,
    Head = 0x10
}
```

The *HttpVerbs* enum is decorated with the *Flags* attribute so that you can combine multiple values from the enumeration using the bitwise OR (|) operator and still obtain another *HttpVerbs* value.

```
[AcceptVerbs(HttpVerbs.Post|HttpVerbs.Put)]
public ActionResult Edit(Customer customer)
{
   ...
}
```

You perform an HTTP GET command when you follow a link or type the URL into the address bar. You perform an HTTP POST when you submit the content of an HTML form. Any other HTTP command can be performed only via AJAX, or perhaps from a Windows client that sends requests to the ASP.NET MVC application.

The ability to assign a specific verb to a given action method naturally leads to duplicate method names. Two methods with the same name are acceptable in a controller class as long as they accept distinct HTTP verbs. Otherwise, an exception will be thrown because ASP.NET MVC doesn't know how to resolve the ambiguity.

Action Methods

Let's have a look at a sample controller class with a couple of simple but functional action methods:

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
       // Process input data
       ...

       // Perform expected task
       ...

       // Generate the result of the action
       return View();
    }

    public ActionResult About()
    {
       // Process input data
       ...

       // Perform expected task
       ...

       // Generate the result of the action
       return View();
    }
}
```

An action method grabs available input data using any standard HTTP channels. Next, it arranges for some action and possibly involves the middle tier of the application. The template of an action method can be summarized as follows:

- **Process input data** An action method gets input arguments from a couple of sources: route values and collections exposed by the *Request* object. ASP.NET MVC doesn't mandate a particular signature for action methods. For testability reasons, however, it's highly recommended that any input parameter is received through the signature. Avoid, if you can, methods that retrieve input data programmatically from *Request* or other sources. As you'll see later in this chapter, and even more thoroughly in Chapter 3, "The Model-Binding Architecture," an entire subsystem exists—the model binding layer—to map HTTP parameters to action method arguments.

- **Perform the task** The action method does its job based on input arguments and attempts to obtain expected results. In doing so, the method likely needs to interact with the middle tier. As we'll discuss further in Chapter 7, "Design Considerations for ASP.NET MVC Controllers," it is recommended that any interaction takes places through ad hoc dedicated services. At the end of the task, any (computed or referenced) values that should be incorporated in the response are packaged as appropriate. If the method returns JSON, data is composed into a JSON-serializable object. If the method returns HTML, data is packaged into a container object and sent to the view engine. The container object is often referred to as the *view-model* and can be a plain dictionary of name/value pairs or a view-specific, strongly typed class.
- **Generate the results** In ASP.NET MVC, a controller's method is not responsible for producing the response itself. It is, however, responsible for triggering the process that will use a distinct object (often, a view object) to render content to the output stream. The method identifies the type of response (file, plain data, HTML, JavaScript, or JSON) and sets up an *ActionResult* object as appropriate.

A controller's method is expected to return an *ActionResult* object or any object that inherits from the *ActionResult* class. Often, though, a controller's method doesn't directly instantiate an *ActionResult* object. Instead, it uses an action helper—that is, an object that internally instantiates and returns an *ActionResult* object. The method *View* in the preceding example provides an excellent example of an action helper. Another great example of such a helper method is *Json*, which is used when the method needs to return a JSON string. I'll return to this point in just a moment.

Processing Input Data

Controller action methods can access any input data posted with the HTTP request. Input data can be retrieved from a variety of sources, including form data, query string, cookies, route values, and posted files.

The signature of a controller action method is free. If you define parameterless methods, you make yourself responsible for programmatically retrieving any input data your code requires. If you add parameters to the method's signature, ASP.NET MVC will offer automatic parameter resolution. In particular, ASP.NET MVC will attempt to match the names of formal parameters to named members in a request-scoped dictionary that joins together values from the query string, route, posting form, and more.

In this chapter, I'll discuss how to manually retrieve input data from within a controller action method. I'll return to automatic parameter resolution—the most common choice in ASP.NET MVC applications—in Chapter 3.

Getting Input Data from the *Request* Object

When writing the body of an action method, you can certainly access any input data that comes through the familiar *Request* object and its child collections, such as *Form*, *Cookies*, *ServerVariables*, and *QueryString*. As you'll see later in the book, when it comes to input parameters of a controller method, ASP.NET MVC offers quite compelling facilities (for example, model binders) that you might want to use to keep your code cleaner, more compact, and easier to test. Having said that, though, nothing at all prevents you from writing old-style *Request*-based code as shown here:

```
public ActionResult Echo()
{
   // Capture data in a manual way
   var data = Request.Params["today"] ?? String.Empty;
   ...
}
```

In ASP.NET, the *Request.Params* dictionary results from the combination of four distinct dictionaries: *QueryString*, *Form*, *Cookies*, and *ServerVariables*. You can also use the *Item* indexer property of the *Request* object, which provides the same capabilities and searches dictionaries for a matching entry in the following order: *QueryString*, *Form*, *Cookies*, and *ServerVariables*. The following code is fully equivalent to the code just shown:

```
public ActionResult Echo()
{
   // Capture data in a manual way
   var data = Request["today"] ?? String.Empty;
   ...
}
```

Note that the search for a matching entry is case insensitive.

Getting Input Data from the Route

In ASP.NET MVC, you often provide input parameters through the URL. These values are captured by the routing module and made available to the application. Route values are not exposed to applications through the *Request* object. You have to use a slightly different approach to retrieve them programmatically:

```
public ActionResult Echo()
{
   // Capture data in a manual way
   var data = RouteData.Values["data"] ?? String.Empty;
   ...
}
```

Route data is exposed through the *RouteData* property of the *Controller* class. Also in this case, the search for a matching entry is conducted in a case-insensitive way.

The *RouteData.Values* dictionary is a String/Object dictionary. The dictionary contains only strings most of the time. However, if you populate it programmatically (for example, via a custom route handler), it can contain other types of values. In this case, you're responsible for any necessary type cast.

Getting Input Data from Multiple Sources

Of course, you can mix *RouteData* and *Request* calls in the same controller method. As an example, let's consider the following route:

```
routes.MapRoute(
       "EchoRoute",
       "echo/{data}",
       new { controller = "Home", action = "Echo", data = UrlParameter.Optional }
);
```

The following is a valid URL: *http://yourserver/echo/Sunday*. The code shown next will easily grab the value of the *data* parameter (Sunday). Here's a possible implementation of the *Echo* method in the *HomeController* class:

```
public ActionResult Echo()
{
   // Capture data in a manual way
   var data = RouteData.Values["data"];
   ...
}
```

What if you call the following URL instead?

```
http://yourserver/echo?today=3/27/2011
```

The URL still matches the route pattern, but it doesn't provide a value for the *data* parameter. Still, the URL adds some input value in the query string for the controller action to consider. Here's the amended version of the *Echo* method that supports both scenarios:

```
public ActionResult Echo()
{
   // Capture data in a manual way
   var data = RouteData.Values["data"] ??
              (Request.Params["today"] ?? String.Empty);
   ...
}
```

The question is, "Should I plan to have a distinct branch of code for each possible input channel, such as form data, query string, routes, cookies, and so forth?" Enter the *ValueProvider* dictionary.

The *ValueProvider* Dictionary

In the *Controller* class, the *ValueProvider* property just provides a single container for input data collected from a variety of sources. By default, the *ValueProvider* dictionary is fed by input values from the following sources (in the specified order):

- **Child action values** Input values are provided by child action method calls. A *child action* is a call to a controller method that originates from the view. A child action call takes place when the view calls back the controller to get additional data or to demand the execution of some special task that might affect the output being rendered. I'll discuss child actions in the next chapter.
- **Form data** Input values are provided by the content of the input fields in a posting HTML form. The content is the same as you would get through *Request.Form*.
- **Route data** Input values are provided by the content associated with parameters defined in the currently selected route.
- **Query String** Input values are provided by the content of parameters specified in the query string of the current URL.
- **Posted Files** Input values are represented by the file or files posted via HTTP in the context of the current request.

The *ValueProvider* dictionary offers a custom programming interface centered on the *GetValue* method. Here's an example:

```
var result = ValueProvider.GetValue("data");
```

Note that *GetValue* doesn't return a *String* or an *Object* type. Instead, it returns an instance of the *ValueProviderResult* type. The type has two properties to actually read the real parameter value: *RawValue* and *AttemptedValue*. The former is of type *Object* and contains the raw value as provided by the source. The *AttemptedValue* property, on the other hand, is a string and represents the result of an attempted type cast to *String*. Here's how to implement the *Echo* method using *ValueProvider*:

```
public ActionResult Echo()
{
   var data = ValueProvider.GetValue("data").AttemptedValue ??
               (ValueProvider.GetValue("today").AttemptedValue ?? String.Empty);
   ...
}
```

ValueProvider is a bit more demanding than *Request* and *RouteData* when it comes to parameter names. If you mistype the case of a parameter, you get a null object back from *GetValue*. This gets you an exception if you then just read the value without checking the result object for nullness.

Finally, note that by default you won't get access to cookies through the *ValueProvider* dictionary. However, the list of value providers can be extended programmatically by defining a class that implements the *IValueProvider* interface.

Note The value-provider mechanism can be useful for retrieving some request data packed into a comfortable collection of values. Default value providers save you from the burden of looking into the *QueryString* or *Form* collection.

What if you need to read data from a cookie or a request header? You can go the usual way and read the *Headers* or *Cookies* collection of the *Request* object and write the code that extracts individual values. However, if your application is extensively based on request headers or cookies, you might want to consider writing a custom value provider.

It is not hard to find working examples of both from the community. A good example of a value provider that exposes request headers can be found here: *http://blog.donnfelker.com/2011/02/16/asp-net-mvc-building-web-apis-with-headervalueprovider*.

Producing Action Results

An action method can produce a variety of results. For example, an action method can just act as a web service and return a plain string or a JSON string in response to a request. Likewise, an action method can determine that there's no content to return or that a redirect to another URL is required. In these two cases, the browser will just get an HTTP response with no significant body of content. This is to say that one thing is producing the raw result of the action (for example, collecting values from the middle tier); it is quite another case to process that raw result to generate the actual HTTP response for the browser. The *ActionResult* class just represents the ASP.NET MVC infrastructure for implementing this programming aspect.

Inside the *ActionResult* Class

An action method typically returns an object of type *ActionResult*. The type *ActionResult* is not a data container, though. More precisely, it is an abstract class that offers a common programming interface to execute some further operations on behalf of the action method. Here's the definition of the *ActionResult* class:

```
public abstract class ActionResult
{
    protected ActionResult()
    {
    }

    public abstract void ExecuteResult(ControllerContext context);
}
```

By overriding the *ExecuteResult* method, a derived class gains access to any data produced by the execution of the action method and triggers some subsequent action. Generally, this subsequent action is related to the generation of some response for the browser.

Predefined Action Result Types

Because *ActionResult* is an abstract type, every action method is actually required to return an instance of a more specific type. Table 1-1 lists all predefined action result types.

TABLE 1-1 Predefined ActionResult types in ASP.NET MVC

Type	Description
ContentResult	Sends raw content (not necessarily HTML) to the browser. The *ExecuteResult* method of this class serializes any content it receives.
EmptyResult	Sends no content to the browser. The *ExecuteResult* method of this class does nothing.
FileContentResult	Sends the content of a file to the browser. The content of the file is expressed as a byte array. The *ExecuteResult* method simply writes the array of bytes to the output stream.
FilePathResult	Sends the content of a file to the browser. The file is identified via its path and content type. The *ExecuteResult* method calls the *TransmitFile* method on *HttpResponse*.
FileStreamResult	Sends the content of a file to the browser. The content of the file is represented through a *Stream* object. The *ExecuteResult* method copies from the provided file stream to the output stream.
HttpNotFoundResult	Sends an HTTP 404 response code to the browser. The HTTP status code identifies a request that failed because the requested resource was not found.
HttpUnauthorizedResult	Sends an HTTP 401 response code to the browser. The HTTP status code identifies an unauthorized request.
JavaScriptResult	Sends JavaScript text to the browser. The *ExecuteResult* method of this class writes out the script and sets the content type accordingly.
JsonResult	Sends a JSON string to the browser. The *ExecuteResult* method of this class sets the content type to the application or JSON and invokes the *JavaScriptSerializer* class to serialize any provided managed object to JSON.
PartialViewResult	Sends HTML content to the browser that represents a fragment of the whole page view. A partial view in ASP.NET MVC is a concept very close to a user control in Web Forms.
RedirectResult	Sends an HTTP 302 response code to the browser to redirect the browser to the specified URL. The *ExecuteResult* method of this class just invokes *Response.Redirect*.
RedirectToRouteResult	Like *RedirectResult*, it sends an HTTP 302 code to the browser and the new URL to navigate to. The difference is in the logic and input data employed to determine the target URL. In this case, the URL is built based on action/controller pairs or route names.
ViewResult	Sends HTML content to the browser that represents a full page view.

Note that *FileContentResult*, *FilePathResult*, and *FileStreamResult* derive from the same base class, *FileResult*. You use any of these action result objects if you want to reply to a request with the download of some file content or even some plain binary content expressed as a byte array. *PartialViewResult* and *ViewResult* inherit from *ViewResultBase* and return HTML content. Finally, *HttpUnauthorizedResult* and *HttpNotFoundResult* represent two common responses for unauthorized access and missing resources. Both derive from a further extensible class, *HttpStatusCodeResult*.

Mechanics of Executing Action Results

To better comprehend the mechanics of action result classes, let's dissect one of the predefined classes. I've chosen the *JavaScriptResult* class, which provides some meaningful behavior without being too complex. (I'll return to this in Chapter 8, "Customizing ASP.NET MVC Controllers.")

The *JavaScriptResult* class represents the action of returning some script to the browser. Here's a possible action method that serves up JavaScript code:

```
public JavaScriptResult GetScript()
{
     var script = "alert('Hello')";
     return JavaScript(script);
}
```

In the example, *JavaScript* is a helper method in the *Controller* class that acts as a factory for the *JavaScriptResult* object. The implementation looks like this:

```
protected JavaScriptResult JavaScript(string script)
{
    return new JavaScriptResult() { Script = script };
}
```

The *JavaScriptResult* class supplies a public property—the *Script* property—that contains the script code to write to the output stream. Here's its implementation:

```
public class JavaScriptResult : ActionResult
{
    public String Script { get; set; }

    public override void ExecuteResult(ControllerContext context)
    {
        if (context == null)
            throw new ArgumentNullException("context");

        // Prepare the response
        HttpResponseBase response = context.HttpContext.Response;
        response.ContentType = "application/x-javascript";
        if (Script != null)
            response.Write(Script);
    }
}
```

As you can see, the ultimate purpose of the *ActionResult* class is preparing the *HttpResponse* object to return to the browser. This entails setting content type, expiration policies, and headers as well as content.

Returning HTML Markup

Most of the time, requests are served by sending back HTML markup. Composing the HTML for the browser is the core of a web framework. In ASP.NET Web Forms, the task of composing HTML is done through the page. Developers create ASPX pages as a mix of a view template and a code-

behind class. Both the action to grab results and the production of the actual response are blurred in a single run-time environment. In ASP.NET MVC, the production of the results is the responsibility of the action method; managing for the response to be composed and served is the responsibility of the framework. Finally, composing the HTML markup is the responsibility of yet another system component—the view engine.

I'll discuss view engines in the next chapter, but for now it suffices to say that a view engine knows how to retrieve a view template for a given action and how to process that to a plain HTML stream that mixes template information and raw data. The view engine dictates the syntax of the view template (ASPX, Razor, and Spark to name a few); the developer dictates the format of the raw data to be merged into the view. Let's consider a sample action method returning HTML:

```
public ActionResult Index()
{
   return View(); // same as View("index");
}
```

The *View* method is a helper method responsible for creating a *ViewResult* object. The *ViewResult* object needs to know about the view template, an optional master view, and the raw data to be incorporated into the final HTML. The fact that in the code snippet *View* is parameterless doesn't mean no data is actually passed on. Here's one of the signatures of the method:

```
protected ViewResult View(String viewName, String masterName, Object model)
```

By convention, the view template is a file named after the action name (*Index* in this case) and located in a specific folder. The exact location depends on the implementation of the currently selected view engine. By default, view templates are expected to be located under the Views folder in a directory that matches the name of the controller—for example, *Views/Home*. Note that you must maintain this directory structure when you deploy the site.

The extension of the view template file also depends on the implementation of the view engine. For the two predefined view engines you get with ASP.NET MVC, the extensions are *.aspx* if you opt for the ASPX view engine and *.cshtml* (or *.vbhtml*) if you opt for the Razor view engine. (I'll provide more details about this in Chapter 2, "ASP.NET MVC Views.")

Returning JSON Content

ASP.NET MVC lends itself very well to implementing simple web services to be called back from jQuery snippets in an Ajax context. All you need to do is set one or more action methods to return JSON strings instead of HTML. Here's an example:

```
public JsonResult GetCustomers()
{
    // Grab some data to return
    var customers = _customerRepository.GetAll();

    // Serialize to JSON and return
    return Json(customers);
}
```

The *Json* helper method gets a plain .NET object and serializes it to a string using the built-in *JavaScriptSerializer* class.

Note What if your controller action method doesn't return *ActionResult*? First and foremost, no exceptions are raised. Quite simply, ASP.NET MVC encapsulates any return value from the action method (numbers, strings, or custom objects) into a *ContentResult* object. The execution of a *ContentResult* object causes the plain serialization of the value to the browser. For example, an action that returns an integer or a string will get you a browser page that displays data as-is. On the other hand, returning a custom object displays any string resulting from the implementation of the object's *ToString* method. If the method returns an HTML string, any markup will not be automatically encoded and the browser will likely not properly parse it. Finally, a *void* return value is actually mapped to an *EmptyResult* object whose execution causes a no-op.

Special Capabilities of Controllers

As you have seen, the primary purpose of a controller is to serve the needs of the user interface. Any server-side functions you need to implement should be mapped to a controller method and triggered from the user interface. After performing its own task, a controller's method selects the next view, packs some data, and tells it to render.

This is the essence of the controller's behavior. However, other characteristics are often required in a controller, especially when controllers are employed in large and complex applications with particular needs, such as numerous commands to deal with, or long-running requests.

Grouping Controllers

In modern software development, we are progressively abandoning the idea of having just one big model to cover and explain everything. Complexity leads us to try to gain as much parallelism as possible in projects and to partition the logic into bound contexts to design and develop individually. How does this apply to controllers? Read on.

In large applications, it is not unusual that you have to deal with a given root entity—say, *Customer*—in various use-cases. Each scenario is going to add actions and requirements. Are you sure you can come up with an effective, lean-and-mean design? Are you sure you'll never end up with a rather bloated and unmanageable class? The concept of a *bound context* in some design methodologies addresses this issue by having you use the same entities in multiple contexts —just configured and behaving differently.

Applied to ASP.NET MVC, the idea of a bound context involves grouping controllers in *areas*.

The Rationale Behind *Areas*

Areas provide a means of partitioning large applications into multiple blocks (named *areas*), each of which can be developed independently. From the perspective of developers, an *area* provides a way to group controllers (and related views) in smaller and more manageable collections. All controllers always belong to an area, and any project consists of at least one default and unnamed area.

The ability to group controllers in areas is beneficial also because it leads you to partition your application into discrete functionalities. If you feel the need to go beyond the default single group of controllers, you are forced to think in terms of logical functionalities that emerge out of your requirements. When areas are used, an application grows up as a collection of distinct applets managed under the umbrella of a single solution.

Having said that, I feel a need to emphasize a key point about areas: areas are not for all applications. Areas come to the rescue when you are having a hard time taming dozens of controllers and views. If your application deals with logical sections such as blogs, forums, and news, you might want to dedicate an area to each in such a way that each area can be architected and developed in relative isolation, with no naming conflicts between controller classes and view templates.

Defining *Areas* in Your Project

In Visual Studio, you start with a classic ASP.NET MVC project and then add as many areas as you need. By default, a new ASP.NET MVC project comes only with the default area. By right-clicking on the project node, you can start adding new areas. At this stage, an area is identified by its name. Figure 1-4 shows a sample Visual Studio project with two additional areas defined: Account and Store.

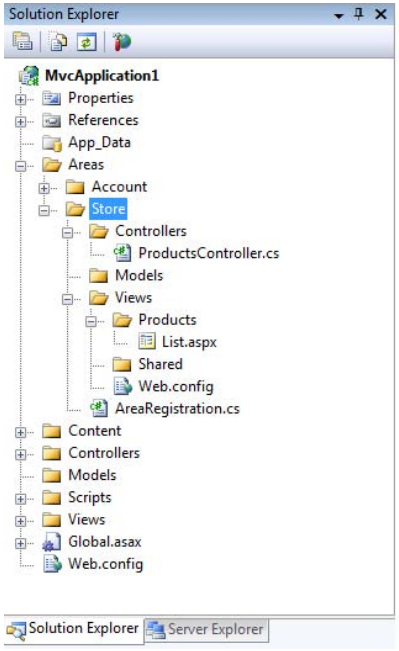

FIGURE 1-4 Areas in an ASP.NET MVC project.

Each area looks like a small subproject and owns its collection of controllers, views, and model classes. As you can see in the figure, each Views folder contains its own copy of the *web.config* file. In addition, a new *AreaRegistration.cs* class file is added for each area.

The next step for you as a developer is to add controller classes and views to the area. Doing this within an area is in no way different from doing the same in the context of the main application. Two other programming aspects make areas a little bit special—adding the area token to routes and linking views across different areas.

Adding Area Information to Routes

The use of areas is not transparent to the ASP.NET MVC machinery. Because areas are a way to group controllers, the routing subsystem must receive an additional piece of information that identifies the area the controller belongs to.

Imagine a URL that points to a generic *Home* controller you created to support your application. In a scenario where you have no explicit areas, that controller can be resolved only within a single environment. So if two controller classes with the same name and different namespaces are found, you just get an exception. When areas are used, though, you can have the same *Home* controller class defined in different namespaces and in different areas. As a result, the routing system definitely needs the area name along with the controller name and the action name.

This means that any helpers that produce URLs for the view must be extended to include area names—for example, the *Html.ActionLink* helper that you'll meet later in the book. It also means that you must define routes that send requests to the appropriate area based on the requested URL.

Each area comes with a system-provided registration file that defines the routes supported by the area. Here's an example:

```
public class StoreAreaRegistration : AreaRegistration
{
    public override string AreaName
    {
        get
        {
            return "Store";
        }
    }

    public override void RegisterArea(AreaRegistrationContext context)
    {
        context.MapRoute(
            "Store_default",
            "Store/{controller}/{action}/{id}",
            new { action = "Index", id = "" }
        );
    }
}
```

As you can see, the default route registered in *RegisterArea* includes an extra data token that matches the name of the area. The route, however, is fully customizable. In *global.asax*, you use a new helper method to register routes for all areas in the project. Here's the revised startup method in *global.asax*:

```
protected void Application_Start()
{
    AreaRegistration.RegisterAllAreas();
    RegisterRoutes(RouteTable.Routes);
}
```

The *RegisterAllAreas* method loops through all available areas and invokes *RegisterArea* for each of them.

Linking to Areas

As long as you navigate within the same area, no special measures are required to ensure that the link is followed correctly. However, to support cross-area links, you need to add area information to links. This is not an issue if you specify the target of anchor tags as a plain string—you update the string, and that's it.

In ASP.NET MVC, however, you often use HTML helpers to generate links and other visual elements. In particular, you use the *Html.ActionLink* method to generate the HTML for an anchor tag. When you use areas, you must resort to a different overload of this method. Here's how to use the helper with areas. (The snippet assumes you're using the ASPX view engine.)

```
<ul id="menu">
    <li><%= Html.ActionLink("Home", "Index", "Home",
                            new { area = "" }, null)%></li>
    <li><%= Html.ActionLink("Store", "List", "Product",
                            new { area = "Store" }, null)%></li>
</ul>
```

The first link displays "Home" as its text and points to the *Index* action on the *Home* controller within the default area. The second link displays "Store" as its text and links the *List* method on the *Product* controller within the *Store* area. The *area* token is optional as long as you don't cross the boundaries of the current area. Note that in a call to *Html.ActionLink* you also need to add a subsequent *null* argument to drive the compiler to using the right overload of the method.

Note When a controller with the same name is used in the default area as well as in an additional area, you might run into problems with the default route. It is required, in fact, that you add some namespace information so that the controller for the default area can be resolved, as shown here:

```
// Default route, default area
routes.MapRoute("Default",
                "{controller}/{action}/{id}",
                new
                {
                    controller = "Home",
                    action = "Index",
                    id = UrlParameter.Optional
                },
                new[] { "YourApp.Controllers" }
            );
```

The additional and optional parameter lets you list the namespaces that the system can use in order to resolve the controller name to a valid class.

Asynchronous Controllers

Especially for server-based applications, asynchronous operations are a fundamental asset to have on the way to providing scalability. In ASP.NET, asynchronous requests take advantage of asynchronous HTTP handlers, which have been a feature of the ASP.NET platform since the first version. Both ASP.NET Web Forms and ASP.NET MVC provide their own facilities to make it simpler for developers to implement asynchronous actions. In particular, ASP.NET MVC provides asynchronous controllers. So if you need to add at least one async method to a controller class, all you do is change the base controller class to *AsyncController*.

Important In ASP.NET, asynchronous pages are commonly associated with the idea of improving the performance of a page that is about to perform a potentially lengthy operation. While this assumption can't be denied, a couple of additional points should be cleared up. First, from the user's perspective, synchronous and asynchronous requests look nearly the same. If the requested operation is expected to take, say, 30 seconds to complete, the user will wait at least 30 seconds to get the new page back. This happens regardless of whether the implementation of the page is synchronous or asynchronous. Furthermore, don't be too surprised if an asynchronous page ends up taking a bit more time to complete a single request. So what's the benefit of asynchronous pages?

The benefit that asynchronous pages bring to the table is that they require much less work for the threads in the ASP.NET pool. This doesn't make lengthy requests run faster, but it does help the system to serve non-lengthy requests as usual—that is, with no special delays resulting from ongoing slow requests. Scalability is not quite the same as performance. Or, at least, scalability is about performance but as it applies to a different scale—that is, it applies to the whole application instead of the single request.

Mechanics of Asynchronous Actions

The overall programming model doesn't change when you define an asynchronous action: you still create a public method, optionally using a set of attributes. These methods don't need to be bound to special routes and return standard action result objects. Compared to a classic synchronous method, an asynchronous (or *async*, for short) action is made of only a pair of methods: *xxxAsync* and *xxxCompleted*, where *xxx* indicates the action name. I'll get into details in a moment. Let's focus on the mechanics of an async action first.

In general, an async ASP.NET request is served in two distinct steps, each requiring a thread from the ASP.NET pool. In the first step, half of the request proceeds from the beginning to the async point. The second half resumes from the async point and completes the processing. The two steps do not form a continuous sequence, and there's no guarantee that the same thread will be serving both steps. The first half (which I'll refer to as the *trigger*) prepares the execution of the request and stops when the lengthy operation begins. The second half begins after the lengthy operation has terminated and finalizes the request. (I'll refer to the final step as the *finalizer*.)

What's the async point, exactly?

The Async Point

The async point is the point in the execution flow when you release the thread in charge of the trigger to the ASP.NET pool. This means that the initial ASP.NET thread is now free to serve other incoming requests, and it is no longer bound to wait for the lengthy operation to complete. This is where the benefit of async operations lies.

What happens between the async point and the moment in which the request resumes and completes? Which thread is taking care of the lengthy operation? (You do need a thread—any thread, but a thread—to take care of any operations in Windows.)

The final step of the trigger method is to return an *IAsyncResult* object. An object that supports the *IAsyncResult* interface stores state information for an asynchronous operation, and it provides a synchronization object to allow threads to be signaled when the operation completes. In the Microsoft .NET Framework, there are a few common ways to get an *IAsyncResult* object. A typical example is invoking a *BeginXXX* method such as *BeginRead* on the *FileStream* class. Another great example is invoking the *BeginXXX* method on a service proxy. Another common scenario for asynchronous operations is when you explicitly start a custom thread or post your work item to a

pooled thread through the *ThreadPool* class. You can even provide your own implementation, but do so carefully and test it well.

In any case, the ultimate purpose of a trigger method is finding another thread (from outside ASP.NET) to take care of the lengthy operation and post the work item to it. When the post occurs, that is the async point.

After the potentially lengthy task has been started, what happens with the ASP.NET thread that took the request up to the async point? That thread has only to wait, in an idle state, until the operation completes elsewhere. Asynchronous HTTP handlers in ASP.NET manage to use an operating system thread, instead of an ASP.NET thread, to wait until the operation completes. This system thread is obtained through a Windows-specific mechanism known as *I/O completion ports*.

When the async point is reached, ASP.NET binds the pending request to an I/O completion port and registers a callback to get a notification when the request has terminated. The operating system will use one of its own dedicated threads to monitor the termination of the operation, thus freeing the ASP.NET thread from the need to wait in full idle. When the operation terminates, the operating system places a message in the completion queue. A message in the completion queue will trigger the ASP.NET callback, which will then pick up one of its own threads to resume and finalize the original request.

This is the general explanation of asynchronous request processing in ASP.NET. In ASP.NET MVC, the various steps are a bit abstracted to hide details such as the async point, HTTP handlers, and I/O completion ports. Let's review the mechanics of asynchronous requests in the context of ASP.NET MVC.

Async Actions in ASP.NET MVC

Let's figure out how ASP.NET MVC carries out async requests. When the system is about to execute the async action, the thread engaged is still the original ASP.NET thread that picked up the request from the web server queue. The code running at this point is the trigger method, which usually takes the form of an *xxxAsync* method, as the following code shows:

```
public void PerformLengthyTaskAsync(SomeData data)
{
   // Process input
   ...

   // Post a work item to a component that can result
   // in a lengthy operation (for example, invoke a web service)
   ...

   // That's all for now—the action is being executed elsewhere.
   // All that remains to be done is wait for it to terminate;
   // for this task, we don't want to squander an ASP.NET thread.
   return;
}
```

When the trigger method returns, the lengthy action is running in the care of some other thread, possibly on some other process. The asynchronous action invoker of ASP.NET MVC manages to sync

up with the underlying ASP.NET runtime so that a completion port is used to monitor the completion of the operation.

When this happens, the ASP.NET runtime puts the request back in circulation with a special flag that indicates it only needs to complete its second half. The first available ASP.NET thread picks up the request and begins processing it. In ASP.NET MVC, this means that the action is executed and the finalizer method is invoked. Here's the typical structure of a finalizer:

```
public ActionResult PerformLengthyTaskCompleted(SomeResponse data)
{
    // Manage the model state (if any)
    ...

    // Prepare and render the view
    ...
}
```

The finalizer receives a custom object (or a multitude of parameters) that contains the data it is expected to process and pass on to the view object. However, the signature of the finalizer must be known in some way to the trigger. Let's find out the details.

Designing Asynchronous Action Methods

Is there any difference between synchronous and asynchronous routes? In ASP.NET MVC, no distinction exists at the route level. You still use the *MapRoute* method to define both. A controller that exposes asynchronous methods is expected to derive from the new *AsyncController* class:

```
public class ServerFacadeController : AsyncController
{
    ...
}
```

Note that an *AsyncController* class can serve both synchronous and asynchronous requests. The name of the method conventionally indicates how the method has to be processed. You must be careful to avoid any ambiguity when you name your methods in an *AsyncController* class. Let's consider the following example that has a synchronous method and an asynchronous method:

```
public class ServerFacadeController : AsyncController
{
  public ActionResult PerformTask(SomeData data)
  {
     ...
  }
  public void PerformTaskAsync(SomeData data)
  {
     ...
  }
  public ActionResult PerformTaskCompleted(SomeResponse data)
  {
     ...
  }

}
```

The preceding code will throw an exception, as shown in Figure 1-5.

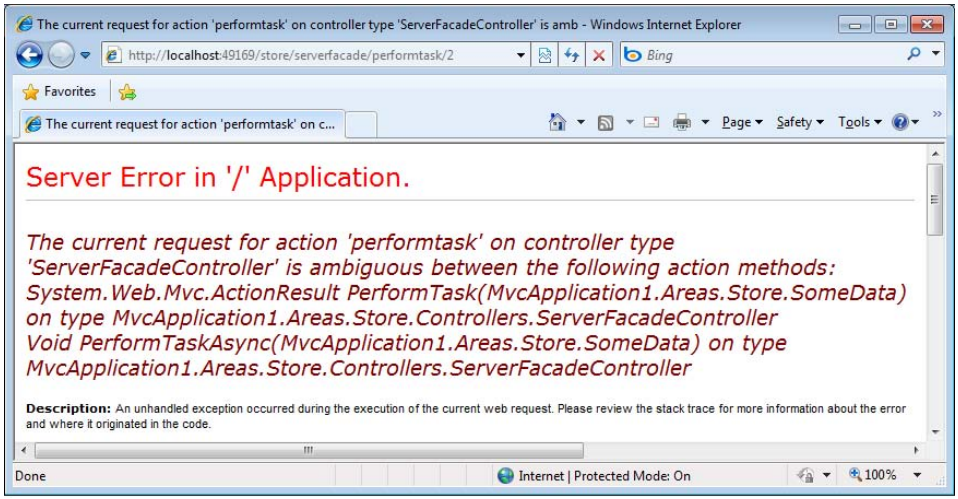

FIGURE 1-5 Ambiguous references in the name of the action.

An async action is identified by name, and the expected pattern is *xxxAsync*, where *xxx* indicates the default name of the action to execute. Clearly, if another method named *xxx* exists and is not disambiguated using HTTP verb or action name attributes, an exception is thrown.

The word *Async* is considered to be a suffix, and the URL required to invoke the *PerformTaskAsync* method contains only the prefix *PerformTask*. For example, the following URL invokes the method *PerformTaskAsync*, passing a value of *2* as a route parameter:

```
http://myserver/serverfacade/performtask/2
```

Whether it will be resolved as a synchronous or asynchronous action depends on the methods you actually have in the *AsyncController* class: for a given action name, you can only have either a synchronous or an asynchronous method match.

As mentioned, the *xxxAsync* method identifies the trigger of the operation. The finalizer of the request is another method in the controller class named *xxxCompleted*. You'll get another exception if a proper *xxxCompleted* method cannot be found.

Note the different signature of the two methods defining the asynchronous action. The trigger is expected to be a *void* method. If you define it to return any value, the return value will be simply ignored. The input parameters of the *xxxAsync* method are subject to model binding as usual. The finalizer method returns an *ActionResult* object as usual, and it receives a custom object that contains the data it is expected to process and pass on to the view object. A special protocol is necessary for matching the values calculated by the trigger to the parameters declared by the finalizer.

Coding Asynchronous Action Methods

In an asynchronous controller class, each asynchronous method is actually a pair of methods and an ad hoc invoker will call each at the right time. In particular, the invoker needs a counter to track the number of individual operations that compose the action so that it can synchronize results before declaring the overall action terminated. In light of this, here's the typical structure of the internal code of a pair of trigger/finalizer methods:

```
public void PerformTaskAsync(SomeData data)
{
    AsyncManager.OutstandingOperations.Increment();

    var response = new SomeResponse();
    ...

    // Do some remote work (for example, invoke a service)
    ...

    // Terminate operations, and prepare data for the finalizer
    AsyncManager.Parameters["data"] = response;
    AsyncManager.OutstandingOperations.Decrement();
}

public ActionResult PerformTaskCompleted(SomeResponse data)
{
   // Prepare the view (for example, massage received data into a view model class)
   var model = new PerformTaskViewModel(data);
   ...

   return View(model);
}
```

The *OutstandingOperations* member on the *AsyncManager* class provides a container that maintains a count of pending asynchronous operations. The *OutstandingOperations* member is an instance of the *OperationCounter* helper class and supplies an ad hoc API to increment and decrement. The *Increment* method is not limited to unary increments, as the following code demonstrates:

```
AsyncManager.OutstandingOperations.Increment(2);
service1.GetData(...);
AsyncManager.OutstandingOperations.Decrement();
service2.GetData(...);
AsyncManager.OutstandingOperations.Decrement();
```

The *Parameters* dictionary on the *AsyncManager* class is used to group values to be passed as arguments to the finalizer method of the asynchronous call. The *Parameters* dictionary will contain an entry for each parameter to be passed to the finalizer. If a match can't be found between entries in the dictionary and parameter names, a default value is assumed for the parameter. The default value results from the evaluation of the *default(T)* expression on the parameter's type. No exception is raised unless an attempt is made to access a null object.

Attributes of Asynchronous Action Methods

Any applicable filter attributes for an asynchronous method must be placed on the trigger method *xxxAsync*. Any attributes applied to the finalizer will be ignored. If an *ActionName* attribute is placed on *xxxAsync* to alias it, the finalizer must be named after the trigger method, not the action name. Consider the following code:

```
[ActionName("Test")]
public void PerformTaskAsync(SomeData data)
{
   ...
}
public ActionResult PerformTaskCompleted(SomeResponse data)
{
   ...
}
```

In addition, you can set a timeout on a per-controller or per-action basis by using the *AsyncTimeout* attribute:

```
[AsyncTimeout(3000)]
```

The attribute is invoked by ASP.NET MVC before the asynchronous action method executes. The duration is expressed in milliseconds and defaults to 45 seconds. By default, all methods are subject to this timeout. If you don't want any timeout, you set that preference explicitly by using the *NoAsyncTimeout* attribute. No timeout is equivalent to setting the timeout to the value of *System.Threading.Timeout.Infinite*.

By setting the *Timeout* property of the *AsyncManager* object, on the other hand, you can set a new global timeout value that applies to any call unless it's overridden by attributes at the controller or action level.

Candidates for Asynchronous Actions

Not all actions should be considered for an asynchronous behavior. Only I/O-bound operations are, in fact, good candidates to become asynchronous action methods on an asynchronous controller class.

An I/O-bound operation is an operation that doesn't depend on the local CPU for completion. When an I/O-bound operation is occurring, the CPU just waits for data to be processed (for example, downloaded) from external storage, such as a database or a remote web service. Operations in which the completion of the task depends on the activity of the CPU are, instead, referred to as *CPU-bound*.

The typical example of an I/O-bound operation is the invocation of a remote web service. In this case, the real work is being done remotely by another machine and another CPU. The ASP.NET thread would be stuck waiting and would be idle all the time. Releasing that idle thread from the duty of waiting, and making it available to serve other incoming requests, is the performance gain you can achieve by using asynchronous actions or pages.

It turns out that not all lengthy operations give you a concrete benefit if they're implemented asynchronously. A lengthy in-memory calculation, for example, doesn't provide you with any

significant benefit if it's implemented asynchronously, because the same CPU both serves the ASP.NET request and performs the calculation.

On the other hand, if remote resources are involved (or even multiple resources), using asynchronous methods can really boost the performance of the application, if not the performance of the individual request.

Summary

Controllers are the heart of an ASP.NET MVC application. Controllers mediate between the user requests and the capabilities of the server system. Controllers are linked to user-interface actions and are in touch with the middle tier. Controllers order the rendering of the page but don't run any rendering tasks themselves. This is a key difference from ASP.NET Web Forms. In a controller, the processing of the request is neatly separated from the display. In Web Forms, on the other hand, the page-processing phase incorporates both the execution of some tasks and the rendering of the response.

Although based on a different syntax, controller methods are not much different from the postback event handlers you have in ASP.NET Web Forms. In this regard, a controller class plays the same role of a code-behind class in Web Forms. The controller, as well as a Web Forms code-behind class, belongs to the presentation layer. For this reason, you should pay a lot of attention to how you code the behavior of the various action methods. Keep in mind that in ASP.NET MVC, any layering in the building of the solution is also up to you.

In this chapter, I skipped over all the details about how you add behavior to a controller method. I focused on an overview of what comes before and what comes after. In the next chapter, I'll delve deeper into what comes after; therefore, the focus will be on views, view engines, and the generation of the markup. Then, in Chapter 3, I'll discuss model binding and what happens before the behavior of an action method comes into play. And in Chapter 7, I'll come back to this topic with some design considerations on how to structure methods in a controller class.

This has been just the first pass on controllers. A lot more has to be said and learned.

CHAPTER 2

ASP.NET MVC Views

Design is not just what it looks like and feels like. Design is how it works.

—Steve Jobs

In ASP.NET MVC, any request is resolved in terms of an action being executed on some controller. Even for a newcomer, this point is relatively easy to understand and figure out. But there's another aspect of the request that the newcomer often has difficulty making sense of—the generation of the HTML for the browser.

In ASP.NET Web Forms, you don't even think of an action—you think of a page, and a page incorporates both logic and view. In ASP.NET Web Forms, you see the implementation of a functional need as the page that performs the task and generates the response you expect for it. Imagine you're working on the use-case for a user who registers with a given site. If the process completes successfully, you then want to display a thank-you screen. How do you design this behavior?

In classic ASP.NET, you start with a *register.aspx* page that the user reaches following a link from, say, the home page. The page unfolds its user interface, which ends with a submit button. The button originates a POST to the same page that takes care of posted data, modifies the state of the application as appropriate, and prepares the expected thank-you screen. As you can see, the entire process is rooted in the page resource.

In ASP.NET MVC, instead, you set up a *Register* action on some controller class. When the action is invoked over a GET command, it results in the display of the user interface for the data entry. When invoked over a POST, on the other hand, the action performs the desired server-side tasks and then manages to serve back the thank-you screen. The entire work flow is similar to what you have in a non-web scenario. In ASP.NET MVC, you just deal with two main flavors of components. One is the controller, which is in charge of executing the request and producing raw results in return for raw input. The other is the *view engine*, which is in charge of generating any expected HTML response based on the results calculated by the controller.

In this chapter, I'll first briefly discuss the internal architecture of the view engine and then move to more practical considerations about the engines available, the need for custom engines, and how you feed an engine with view templates and data.

Note As you saw in Chapter 1, "ASP.NET MVC Controllers," a controller action doesn't necessarily produce some HTML. An ASP.NET MVC application can be seen as a collection of components with the ability to serve various responses, including HTML, script, JSON, and plain text. In this chapter, I'll restrict the discussion to considering the subsystem responsible for the production of HTML. In Chapter 8, "Customizing ASP.NET MVC Controllers," I'll discuss in more detail other types of responses.

Structure and Behavior of a View Engine

The view engine is the component that physically builds the HTML output for the browser. The view engine kicks in for each request that returns HTML, and it prepares its output by mixing together a template for the view and any data the controller passes in. The template is expressed in an engine-specific markup language; the data is passed packaged in dictionaries or in strongly typed objects. Figure 2-1 shows the overall picture of how a view engine and controller work together.

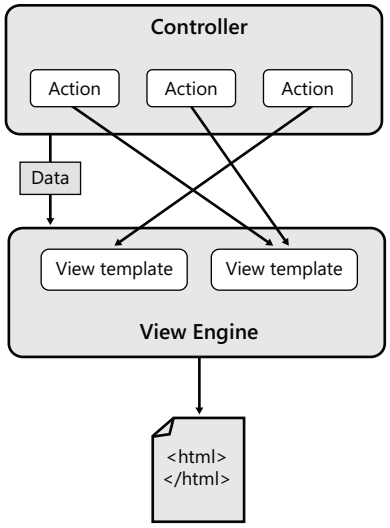

FIGURE 2-1 Controllers and view engines.

Mechanics of a View Engine

In ASP.NET MVC, a view engine is merely a class that implements a fixed interface—the *IViewEngine* interface. Each application can have one or more view engines. In ASP.NET MVC 3, each application is armed by default with two view engines. Let's find out more.

Detecting Registered View Engines

When you first create an ASP.NET MVC application, the Microsoft Visual Studio project wizard asks you to pick your favorite view engine. Figure 2-2 shows the specific dialog box as it shows up when ASP.NET MVC 3 is installed in Visual Studio 2010.

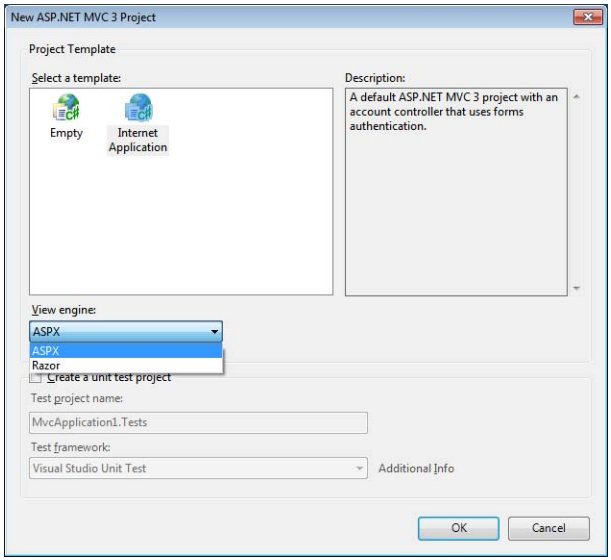

FIGURE 2-2 Choosing your favorite view engine.

In spite of appearances, the choice you make here has a limited impact on the application. Your choice, in fact, influences only the content of the project files the wizard will create for you. By default, any ASP.NET MVC 3 application will always load two view engines: ASPX and Razor.

The *ViewEngines* class is the system repository that tracks the currently installed engines. The class is simple and exposes only a static collection member named *Engines*. The static member is initialized with the two default engines. Here's an excerpt from the class:

```
public static class ViewEngines
{
   private static readonly ViewEngineCollection _engines =
           new ViewEngineCollection {
                  new WebFormViewEngine(),
                  new RazorViewEngine()
           };

   public static ViewEngineCollection Engines
   {
       get { return _engines; }
   }
   ...
}
```

In case you're interested in using *ViewEngines.Engines* to detect the installed engines programmatically, here's how to do it:

```
private static IList<String> GetRegisteredViewEngines()
{
   return ViewEngines
            .Engines
            .Select(engine => engine.ToString())
            .ToList();
}
```

The most likely scenario in which you might encounter *ViewEngines.Engines* is when you need to add a new view engine or unload an existing one. You do this in the application startup and precisely in the *Application_Start* event in *global.asax*.

Anatomy of a View Engine

A view engine is a class that implements the *IViewEngine* interface. The contract of the interface says it's all about the services the engine is expected to provide: the engine is responsible for retrieving a (partial) view object on behalf of the ASP.NET MVC infrastructure. A *view object* represents the container for any information that is needed to build a real HTML response in ASP.NET MVC. Here are the interface members:

```
public interface IViewEngine
{
    ViewEngineResult FindPartialView(
          ControllerContext controllerContext,
          String partialViewName,
          Boolean useCache);
    ViewEngineResult FindView(
          ControllerContext controllerContext,
          String viewName,
          String masterName,
          Boolean useCache);
    void ReleaseView(
          ControllerContext controllerContext,
          IView view);
}
```

Table 2-1 describes the behavior of the methods in the *IViewEngine* interface.

TABLE 2-1 Methods of the *IViewEngine* interface

Method	Description
FindPartialView	Creates and returns a view object that represents a fragment of HTML
FindView	Creates and returns a view object that represents an HTML page
ReleaseView	Releases the specified view object

Both *FindPartialView* and *FindView* return a *ViewEngineResult* object, which represents the results of locating a template for the view around the server directory tree and instantiating it. The class signature follows.

```
public class ViewEngineResult
{
    ...

    // Members
    public IEnumerable<String> SearchedLocations { get; private set; }
    public IView View { get; private set; }
    public IViewEngine ViewEngine { get; private set; }
}
```

The *ViewEngineResult* type just aggregates three elements: the view object, the view engine object used to create it, and the list of locations searched to find the template of the view. The content of the *SearchedLocations* property depends on the structure and behavior of the selected view engine. The *ReleaseView* method is intended to dispose of any references that the view object has in use.

Who Calls the View Engine?

Although Figure 2-1 seems to show direct contact between controllers and view engines, the two components never get in touch directly. The activity of both controllers and view engines, instead, is coordinated by an external manager object—the *action invoker*. The action invoker is triggered directly by the HTTP handler in charge of the request. The action invoker does two key things. First, it executes the controller's method and saves the action result. Next, it processes the action result. You see the sequence diagram in Figure 2-3.

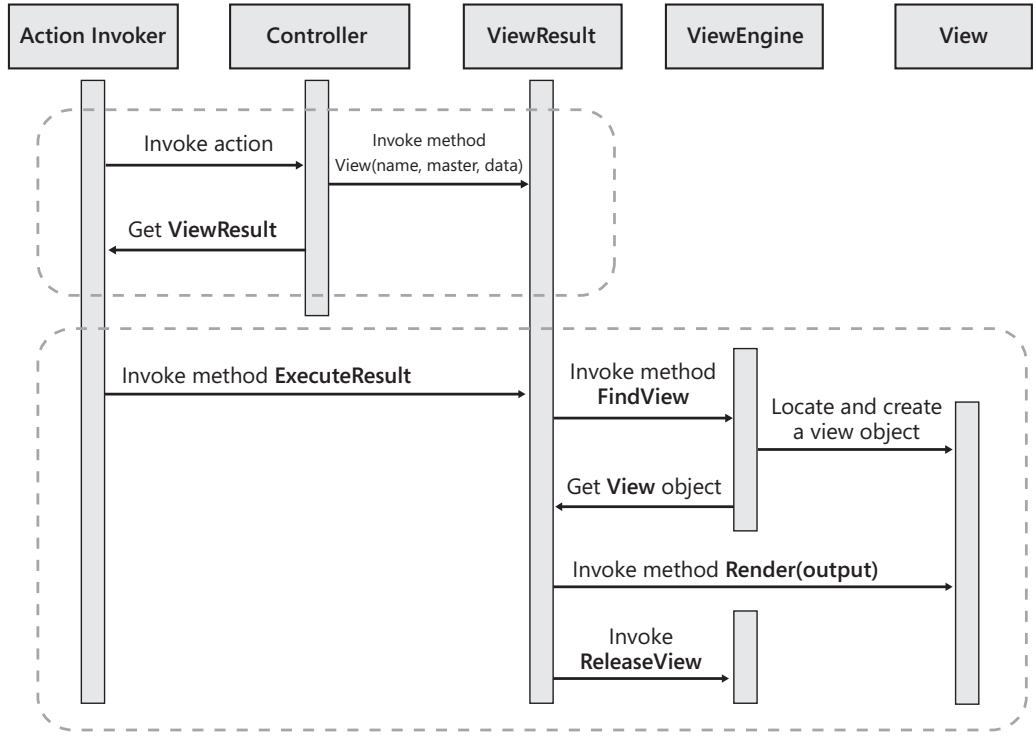

FIGURE 2-3 A sequence diagram that illustrates the request-servicing process.

Let's consider the typical code of a controller method as you saw it in Chapter 1.

```
public ActionResult Index()
{
    // Some significant code here
    ...

    // Order rendering of the next view
    return View();
}
```

The *View* method on the controller class packs a few pieces of data together in a single container—the *ViewResult* class. Information includes the name of the view template that the controller has selected as the next view to show to the user. An optional piece of data that goes into *ViewResult* is the name of the master view. Finally, the *ViewResult* container also incorporates the calculated data that will be displayed in the view. When the *View* method gets no parameters, as in the code snippet shown earlier, default values are provided. An instance of *ViewResult* object is delivered back to the action invoker.

Next, the action invoker invokes the *ExecuteResult* method on the *ViewResult* object. The method goes through the list of registered view engines to find one that can match the specified view and master view names. If no such view engine is found, an exception is thrown. Otherwise, the selected view engine is asked to create a view object based on the information provided.

Subsequently, the *ViewResult* object orders the view to render its content out to the provided stream—the actual response stream. Finally, the *ViewResult* object instructs the view engine to release the view.

The View Object

The view object is an instance of a class that implements the *IView* interface. The only purpose of a view object is for writing some HTML response to a text writer. Each view is identified by name. The name of the view is also associated with some physical file that defines the HTML layout to render. Resolving the association between the view name and actual HTML layout is the responsibility of the view engine.

The name of the view is one of the parameters the *View* method on the controller action is supposed to provide. If no such parameter is explicitly defined by the programmer, the system assumes by convention that the name of the view is the same as the action name. (As you saw in Chapter 1, the action name doesn't necessarily match the method name.)

The *IView* interface is shown here:

```
public interface IView
{
    void Render(ViewContext viewContext, TextWriter writer);
}
```

Under the hood, a view object is a wrapper around an object that describes a visual layout devoid of data. Rendering the view means populating the layout with data and rendering it out as HTML to

some stream. Of the two default view engines in ASP.NET MVC, one—the ASPX view engine—just uses a derivative of the ASP.NET *Page* class to represent the visual layout. The other view engine—the Razor engine—is based on a different class designed around the same core idea. The Razor engine uses the WebMatrix counterpart of an ASP.NET *Page* class.

Definition of the View Template

In ASP.NET MVC, everything being displayed to the user results from a view and is described in terms of a template file. The graphical layout is then transformed into HTML and styled via one or more cascading style sheet (CSS) files. The way in which the template file is written, however, depends on the view engine. Each view engine has its own markup language to define the template and its own set of rules to resolve a view name into a template file.

Resolving the Template

At the end of its job, the controller figures out the name of the next view to render to the user. The name of the view, however, has to be translated into some good HTML markup. This takes a couple more steps. First, the system needs to identify which view engine (if any) can successfully process the request for the view. Second, the view name has to be matched to an HTML layout and a view object must be successfully created from there.

Starting from the point that the name of the view is known, the *ViewResult* object (shown earlier in Figure 2-3) queries through all the installed view engines in the order in which they appear in the *ViewEngines.Engines* collection. Each view engine is asked whether it is capable of rendering a view with the given name.

By convention, each ASP.NET MVC view engine uses its own algorithm to translate the view name into a resource name that references the final HTML markup. For the two predefined view engines, the search algorithm attempts to match the view name to a physical file in one of a few fixed disk locations.

A custom view engine, however, can release both constraints and employ a different set of conventions. For example, it can load the view layout from, say, a database, or it can use a customized set of folders.

Default Conventions and Folders

Both the ASPX and Razor view engines use the same core conventions to resolve view names. Both match view names to file names, and both expect to find those files in the same set of predefined folders. The only difference between ASPX and Razor is the extension of the files that contain the view layout.

Unless you install a custom view engine, an ASP.NET MVC application finds its view templates under the *Views* folder. As shown in Figure 2-4, the *Views* folder has a number of subfolders—each named after an existing controller name. Finally, the controller folder contains physical files whose name is expected to match the view name and whose extension has to be *.aspx* for the ASPX view

engine and *.cshtml* for the Razor view engine. (If you're writing your ASP.NET MVC application in Microsoft Visual Basic, the extension will be *.vbhtml*.)

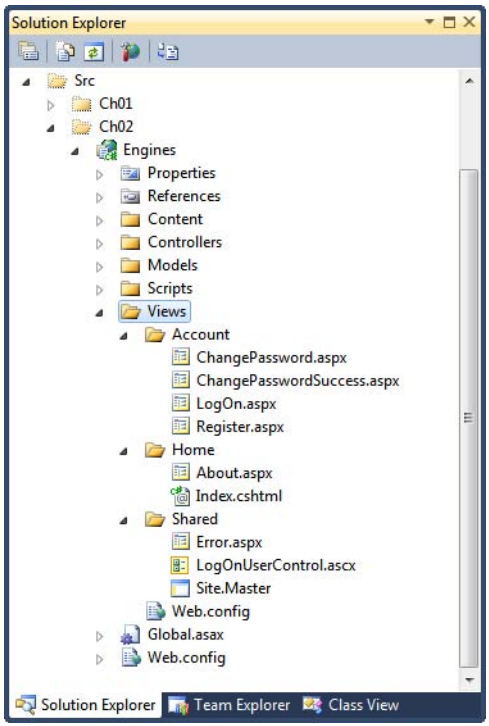

FIGURE 2-4 Locating view templates in an ASP.NET MVC application.

In Figure 2-4 under the *Views/Home* folder you find both *about.aspx* and *index.cshtml*—clearly two view files requiring distinct view engines. As long as all required view engines are registered with the application, you can happily mix and match template files written according to different syntaxes. The ASPX and Razor engines are natively incorporated in the ASP.NET MVC platform, but other engines exist (and can be successfully integrated in Visual Studio), such as Spark and NHAML.

Important Having view templates written using different markup languages certainly doesn't increase the level of consistency of your source code, but it can be a viable solution in cases where you have different skills within a team or when you need to incorporate some legacy code. Also, consider that when you have view templates for different engines, resolving the names can produce some surprises. View engines are called to claim the view in the order in which they are registered and, by default, the ASPX engine takes precedence over the Razor engine. To modify the order, you should clear the *Engines* collection during application startup and re-add engines in the order you prefer.

In general, ASP.NET MVC requires that you place each view template under the folder of the controller that uses it. If multiple controllers are expected to invoke the same view (or partial view), you should move the template file under the *Shared* folder.

Finally, note that the same hierarchy of directories that exists at the project level under the Views folder must be replicated on the production server. In an ASP.NET MVC project, Views and Content are content folders that contain resources the code points directly to. Folders such as Controllers and Models, on the other hand, are plain namespace containers used to better organize the source files, and they can be ignored in production.

The Template for the View

As mentioned, a view is nothing more than a template for the resulting HTML content. As you can see in Figure 2-4, the file *about.aspx* just describes the structure of the content being rendered. Here's a sample of its content:

```
<%@ Page Language="C#"
         MasterPageFile="~/Views/Shared/Site.Master"
         Inherits="System.Web.Mvc.ViewPage" %>

<asp:Content ID="aboutTitle" ContentPlaceHolderID="TitleContent" runat="server">
    About the book
</asp:Content>

<asp:Content ID="aboutContent" ContentPlaceHolderID="MainContent" runat="server">
    <h2><%: ViewBag.Message %></h2>
    <p>
        Put content here.
    </p>
</asp:Content>
```

Without beating around the bush, this looks nearly the same as an old, faithful ASP.NET Web Forms page. So what's the point of ASP.NET MVC?

Admittedly, the syntax you see here is the same as in an ASP.NET Web Forms page; the role of this file, however, is radically different. In ASP.NET MVC, *about.aspx* is not a public resource you can request by following a link or typing it in the address bar of the browser. It is, instead, an internal resource file used to provide the template for a view. In particular, the *about.aspx* template is picked up only when the user makes a request that the system maps to a controller method which, in turn, selects the *about* view for display, as shown here:

```
public ActionResult About()
{
    ViewBag.Message = "Thank you for choosing this book!";

    // By default, the view name is the same as the action name.
    return View();
}
```

Another huge difference is the data you display through the template. In Web Forms, each page is made of server controls and, as a developer, you set properties on server controls to display any data.

In ASP.NET MVC, you group the data you want to pass to the view in a container object and pass the container object as an argument in the controller's call that selects the view.

ASP.NET MVC allows you to pass a container object directly in the call to the *View* method. In any case, two predefined dictionaries are available for the controller method to stuff data. They are the *ViewData* and *ViewBag* dictionaries.

A key point to remember is that ideally the view object doesn't have to retrieve data on its own; the only data it has to deal with is the data it receives from the controller.

Note The purpose of the view template is to produce HTML, but the source of the template doesn't have to be HTML. The language used to write the template depends on the view engine. It can be nearly the same ASPX markup you might know from ASP.NET Web Forms if you opt for the ASPX view engine; it will be something significantly different if you choose the Razor engine or another engine.

The Master View

Just like in ASP.NET Web Forms, you need to decide whether you want to write the view template entirely from scratch or inherit some common markup from a master view. If you choose the latter option, you define the master template with respect to the syntax and constraints set by the view engine.

Specifying the master view is easy. You can leverage the conventions supported by the view engine, or you can just pass the name of the master view as an argument to the *View* method when you select the next view from the controller. Note that the master template might follow different conventions than plain views. For example, the ASPX view engine requires you to name it with a *.master* extension and place it in the *Shared* folder. The Razor view engine, instead, adds a *.cshtml* extension and requires you to specify the path in a special *.cshtml* file in the root of the *Views* folder.

I'll have more to say about the two default view engines in just a moment.

HTML Helpers

Each view engine has aspects that makes it unique compared to others. The location of templates is certainly one of these aspects. Another is the syntax view engines use to let developers describe the view they want.

Admittedly, the ASPX view engine is quite similar to ASP.NET Web Forms but doesn't support server controls in the same manner. This is not because of a partial implementation but has a deeper explanation. Server controls are, in effect, too tightly coupled to the life cycle of a Web Forms page to be easily reworked in a request-processing model that breaks up action into neatly distinct phases, such as getting input data, processing the request, and selecting the next view. As you'll see later, you can still use a few server controls with the ASPX engine, but not the most powerful of them.

On the other hand, server controls served a very important purpose in ASP.NET—favoring HTML-level code reuse. Even though ASP.NET MVC makes a point of letting developers gain control over every single HTML tag, a good deal of HTML can't be hard-coded in the view. It needs be built programmatically based on dynamically discovered data. What's a technology equivalent to server controls in ASP.NET MVC? Enter *HTML helpers*.

HTML helpers certainly are not the same as server controls, but they are the closest you can get to HTML-level code reuse with a view engine. An HTML helper method has no view state, no postbacks, and no page life cycle and events. HTML helpers are simple HTML factories. Technically speaking, an HTML helper is an extension method defined on a system class—the *HtmlHelper* class—that outputs an HTML string based on the provided input data. Internally, in fact, an HTML helper simply accumulates text into a *StringBuilder* object.

Basic Helpers

The ASP.NET MVC framework supplies a few HTML helpers out of the box, including *CheckBox*, *ActionLink*, and *RenderPartial*. The stock set of HTML helpers is presented in Table 2-2.

TABLE 2-2 Stock set of HTML helper methods

Method	Type	Description
BeginForm, BeginRouteForm	Form	Returns an internal object that represents an HTML form that the system uses to render the *<form>* tag
EndForm	Form	A void method, closes the pending *</form>* tag
CheckBox, CheckBoxFor	Input	Returns the HTML string for a check box input element
Hidden, HiddenFor	Input	Returns the HTML string for a hidden input element
Password, PasswordFor	Input	Returns the HTML string for a password input element
RadioButton, RadioButtonFor	Input	Returns the HTML string for a radio button input element
TextBox, TextBoxFor	Input	Returns the HTML string for a text input element
Label, LabelFor	Label	Returns the HTML string for an HTML label element
ActionLink, RouteLink	Link	Returns the HTML string for an HTML link
DropDownList, DropDownListFor	List	Returns the HTML string for a drop-down list
ListBox, ListBoxFor	List	Returns the HTML string for a list box
TextArea, TextAreaFor	TextArea	Returns the HTML string for a text area
Partial	Partial	Returns the HTML string incorporated in the specified user control
RenderPartial	Partial	Writes the HTML string incorporated in the specified user control to the output stream
ValidationMessage, ValidationMessageFor	Validation	Returns the HTML string for a validation message
ValidationSummary	Validation	Returns the HTML string for a validation summary message

As an example, let's see how to use an HTML helper to create a text box with programmatically determined text. You place the call in a code block if you're using the ASPX view engine:

```
<%: Html.TextBox("TextBox1", ViewBag.DefaultText) %>
```

Or you prefix the call with the @ symbol if you're using Razor. (I'll say more about Razor in a moment.)

```
@Html.TextBox("TextBox1", ViewBag.DefaultText)
```

Note The *Html* in the code snippets refers to a built-in property of the base classes used by both view engines to refer to a rendered view. The class is *ViewPage* for the ASPX view engine and *WebPage* for the Razor view engine. In both cases, the property *Html* is an instance of *HtmlHelper*.

Each HTML helper has a bunch of overloads to let you specify attribute values and other relevant information. For example, here's how to style the text box using the *class* attribute:

```
<%: Html.TextBox("TextBox1",
                 ViewBag.DefaultText,
                 new Dictionary<String, Object>{{"class", "coolTextBox"}}) %>
```

In Table 2-2, you see a lot of *xxxFor* helpers. In what way are they different from other helpers? An *xxxFor* helper differs from the base version because it accepts only a lambda expression, such as the one shown here:

```
<%: Html.TextBoxFor(model => model.FirstName,
                    new Dictionary<String, Object>{{"class", "coolTextBox"}}) %>
```

For a text box, the lambda expression indicates the text to display in the input field. The *xxxFor* variation is especially useful when the data to populate the view is grouped in a model object. In this case, your view results are clearer to read and strongly typed.

Let's see a few other examples of basic HTML helpers.

Rendering HTML Forms

The unpleasant work of rendering a form in ASP.NET MVC occurs when you have to specify the target URL. The *BeginForm* and *BeginRouteForm* helpers can do the ugliest work for you. The following code snippet shows how to write a simple input form with a couple of fields, *user* and *password*:

```
<% using (Html.BeginForm()) { %>
   <div>
     <fieldset>
         <legend>Account Information</legend>
         <p>
             <label for="userName">User name:</label>
             <%= Html.TextBox("userName") %>
             <%= Html.ValidationMessage("userName") %>
```

```
        </p>
        <p>
             <label for="password">Password:</label>
             <%= Html.Password("password") %>
             <%= Html.ValidationMessage("password") %>
        </p>
        ...
        <p>
            <input type="submit" value="Change Password" />
        </p>
    </fieldset>
  </div>
<% } %>
```

The *BeginForm* helper takes care of the opening *<form>* tag. The *BeginForm* method, however, doesn't directly emit any markup. It's limited to creating an instance of the *MvcForm* class, which is then added to the control tree for the page and rendered later.

To close the tag, you can use the *EndForm* helper or rely on the *using* statement as in the preceding example. The *using* pattern ends up invoking the *Dispose* method on the *MvcForm* object, which in turn emits the closing *</form>* tag.

By default, *BeginForm* renders a form that posts back to the same URL and, subsequently, to the same controller action. Other overloads on the *BeginForm* method allow you to specify the target controller's name and action, any route values for the action, HTML attributes, and even whether you want the form to perform a GET or a POST. The following example shows a form that posts to a controller named *Memo* to execute an action named *Update* and passes a collection of route values:

```
<% Html.BeginForm("Update", "Memo", new RouteValueDictionary{{"MemoID", 100}}); %>
   ...
<% Html.EndForm(); %>
```

After you have done this, generating the resulting URL and arranging the final markup is no longer a concern of yours.

BeginRouteForm behaves like *BeginForm* except that it can generate a URL starting from an arbitrary set of route parameters. In other words, *BeginRouteForm* is not limited to the default route based on the controller name and action.

Note In HTML, the *<form>* tag doesn't allow you to use anything other than the GET and POST verbs to submit some content. In ASP.NET MVC, a native method on the *HtmlHelper* class—*HttpMethodOverride*—comes to the rescue. The method emits a hidden field whose name is hard-coded to *X-HTTP-Method-Override* and whose value is PUT, DELETE, or HEAD. The content of the hidden field overrides the method set for the form, thus allowing you to invoke a REST API also from within the browser. The override value can also be specified in an HTTP header with the same *X-HTTP-Method-Override* name or in a query string value as a name/value pair. The override is valid only for POST requests.

Rendering Input Elements

All HTML elements that can be used within a form have an HTML helper to speed up development. Again, there's really no difference from a functional perspective between using helpers and using plain HTML. Here's an example of a check box element, initially set to *true*, but disabled:

```
<%= Html.CheckBox("ProductDiscontinued",
                  true,
                  new Dictionary<String, Object>() {{"disabled", "disabled"}})) %>
```

You also have facilities to associate a validation message with an input field. You use the *Html.ValidationMessage* helper to display a validation message if the specified field contains an error. The message can be indicated explicitly through an additional parameter in the helper. All validation messages are then aggregated and displayed via the *Html.ValidationSummary* helper.

I'll return to input forms and validation in Chapter 4, "Input Forms."

Action Links

As mentioned, creating URLs programmatically is a boring and error-prone task in ASP.NET MVC. For this reason, helpers are more than welcome, especially in this context. In fact, the *ActionLink* helper is one of the most frequently used in ASP.NET MVC views. Here's an example:

```
<%= Html.ActionLink("Home", "Index", "Home") %>
```

Typically, an action link requires the link text, the action name, and optionally the controller name. The HTML that results from the example is the following:

```
<a href="/Home/Index">Home</a>
```

In addition, you can specify route values, HTML attributes for the anchor tag, and even a protocol (for example, HTTPS), host, and fragment.

The *RouteLink* helper works in much the same way, except it doesn't require you to specify an action. With *RouteLink*, you can use any registered route name to determine the pattern for the resulting URL.

The text emitted by *ActionLink* is automatically encoded. This means you can't use any HTML tag in the link text that the browser will be led to consider as HTML. In particular, you can't use *ActionLink* for image buttons and image links. However, to generate a link based on controller and action data, you can use the *UrlHelper* class.

An instance of the *UrlHelper* class is associated with the *Url* property on the *ViewPage* type. The code here shows the *Url* object in action:

```
<a href="<%= Url.Action("Edit") %>">
    <img src="editMemo.jpg" alt="Edit memo" />
</a>
```

The *UrlHelper* class has a couple of methods that behave nearly similar to *ActionLink* and *RouteLink*. Their names are *Action* and *RouteUrl*.

Note ASP.NET MVC routing is not aware of subdomains; it always assumes you're in the application path. This means that if you want to use subdomains within a single application instead of virtual paths (for example, *dino.blogs.com* instead of *www.blogs.com/dino*), the extra work of figuring out which subdomain you're in belongs entirely to you.

You can address this task in a number of ways. A simple approach consists of creating a custom route handler that looks at the host passed in the URL and decides which controller to set up. This solution, however, is limited to incoming requests. It might not be enough for all of the helpers you have around to generate links to resources and actions.

A more complete solution, then, is creating your *Route* class so that it's aware of subdomains. A good example can be found here: *http://goo.gl/e6AOD*.

Partial Views

You use either the *Partial* or *RenderPartial* helper method to insert a partial view. Both methods take the name of the partial view as an argument. The only difference between the two is that *Partial* returns a string, whereas *RenderPartial* writes to the output stream and returns void. Because of this, the usage is slightly different:

```
<%: Html.Partial("login") %>
<% Html.RenderPartial("login"); %>
```

In ASP.NET MVC, a partial view is analogous to a user control in Web Forms. A partial view in ASP.NET MVC is rendered through the *ViewUserControl* class, which derives from ASP.NET's *UserControl* class. The typical location for a partial view is the *Shared* folder under Views. However, you can also store a partial view under the controller-specific folder.

A partial view is contained in a view, but it will be treated as an entirely independent entity. In fact, it is legitimate to have a view written for one view engine and a child partial view that requires another view engine.

The *HtmlHelper* Class

The *HtmlHelper* class owes most of its popularity to its numerous extension methods, but it also has a number of useful native methods. Some of them are listed in Table 2-3.

TABLE 2-3 Most popular native methods on *HtmlHelper*

Method	Description
AntiForgeryToken	Returns the HTML string for a hidden input field stored with the antiforgery token. (See Chapter 4 for more details.)
AttributeEncode	Encodes the value of the specified attribute using the rules of HTML encoding.
EnableUnobtrusiveJavaScript	Sets the internal flag that enables helpers to generate JavaScript code in an unobtrusive way.

Method	Description
EnableClientValidation	Sets the internal flag that enables helpers to generate code for client-side validation.
Encode	Encodes the specified value using the rules of HTML encoding.
HttpMethodOverride	Returns the HTML string for a hidden input field used to override the effective HTTP verb to indicate that a PUT or DELETE operation was requested.
Raw	Returns the raw HTML string without encoding.

In addition, the *HtmlHelper* class provides a number of public methods that are of little use from within a view but offer great support to developers writing custom HTML helper methods. A good example is *GenerateRouteLink*, which returns an anchor tag containing the virtual path for the specified route values.

Templated Helpers

Templated HTML helpers aim to make the display and editing of data quick to write and independent from too many HTML and CSS details. As you'll see in greater detail in Chapter 4, a best practice in ASP.NET MVC development entails building a view-specific model—the view model—and pass it down to the view object filled with data. How would you display or edit this data?

Writing HTML forms over and over again leads to repetitive, boring, and therefore error-prone code. Templated helpers take an object, read the value of properties, and decide how to best render those values. By decorating the view-model objects with special attributes, you provide the helper further guidance regarding user-interface hints and validation.

With templated helpers, you are not losing control over the user interface; more simply, attributes in the model establish a number of conventions and save you from a number of repetitive tasks.

Flavors of a Templated Helper

In ASP.NET MVC, you have two essential templated helpers: *Editor* and *Display*. They work together to make the code for labeling, displaying, and editing data objects easy to write and maintain. The optimal scenario for using these helpers is that you are writing your lists or input forms around annotated objects. However, templated helpers can work with both scalar values and composite objects.

Templated helpers actually come with three overloads. Using the *Display* helper as an example, you have the following more specific helpers: *Display*, *DisplayFor*, and *DisplayForModel*. There's no functional difference between *Display*, *DisplayFor*, and *DisplayForModel*. They differ only in terms of the input parameters they can manage.

The *Display* Helpers

The *Display* helper accepts a string indicating the name of the property in the *ViewData* dictionary, or on the view-model object, to be processed:

```
<%= Html.Display("FirstName") %>
```

The *DisplayFor* helper accepts a lambda expression and requires that a view-model object be passed to the view:

```
<%= Html.DisplayFor(model => model.FirstName) %>
```

Finally, *DisplayForModel* is a shortcut for *DisplayFor* getting the expression *model => model*:

```
<%= Html.DisplayForModel() %>
```

All flavors of templated helpers have the special ability to process metadata (if any) and adjust their rendering accordingly—for example, showing labels and adding validation. The display and editing capabilities can be customized using templates, as discussed in a moment. The ability of using custom templates applies to all flavors of a templated helper.

Important *ViewBag* is a property defined on the *ControllerBase* class defined to be of type *dynamic*. In .NET 4, the type *dynamic* indicates the site for dynamically interpreted code. In other words, whenever the compiler meets a reference to a *dynamic* object, it emits a chunk of code that checks at run time whether the code can be resolved and executed. Functionally speaking, this is similar to what happens with JavaScript objects.

Lambda expressions don't support dynamic members and therefore can't be used with data passed into the *ViewBag* dictionary. Also note that to successfully use *ViewBag* content in HTML helpers, you must cast the expression to a valid type.

The *Editor* Helpers

The purpose of the *Editor* helper is to let you edit the specified value or object. The editor recognizes the type of the value it gets and picks up a made-to-measure template for editing. Predefined templates exist for *object*, *string*, Boolean, and multiline text, while numbers, dates, and GUIDs fall back to the string editor. The helper editor works great with complex types. It generically iterates over each public property and builds up a label and an editor for the child value. Here's how to display in an editor the value of the *FirstName* property on some object being passed to the view:

```
<%= Html.EditorFor(person => person.FirstName) %>
```

You can customize the editor (and the visualizer) by creating a few partial views in the *EditorTemplates* folder of the view. It can be under a controller-specific subfolder or under the *Views\Shared* folder as well. The partial view is expressed as an *.ascx* template for the ASPX view engine and as a *.cshtml* template if you're using the Razor view engine. You can provide a custom template for each type you expect to support. For example, in Figure 2-5 you see a *datetime.ascx* template that will be used to modify the way dates are rendered in both editing and display. Likewise, you can provide a partial view for each type of property in the view-model object.

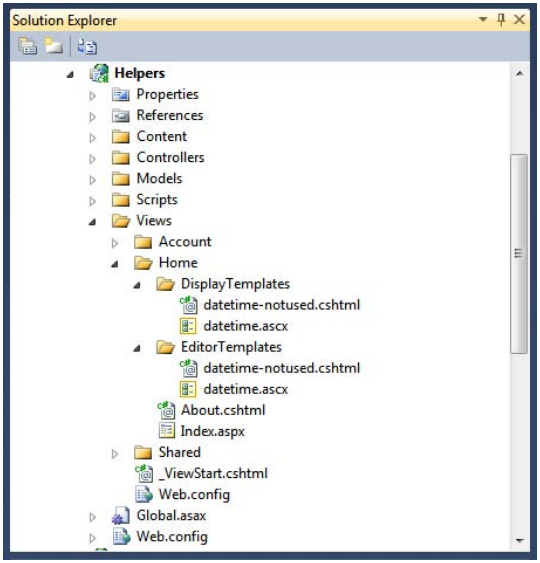

FIGURE 2-5 Custom templates for editors and visualizers.

If the name of the partial view matches the type name, the custom template is automatically picked up by the system.

You can also point the editor to your template by name and give the template the name you like. Here's an example that uses the *date.ascx* view to edit a *DateTime* property:

```
<%= Html.EditorFor(person => person.Birthdate, "date") %>
```

In the same ASP.NET MVC application, you can have views requiring different view engines. Note that ASP.NET MVC resolves each view and partial view independently. This means that if you're processing, say, the view *about*, you end up with the Razor engine (as shown in Figure 2-5). However, if the *about* view requires an editor for dates and you have a matching *.aspx* template, it'll be picked up anyway with no need for you to provide an analogous *.cshtml* template.

Finally, the *Editor* helper can recognize data annotation attributes on view-model objects and use that information to add special validation features, such as ensuring that a given value falls in the specified range or is not left empty.

Note When you use *DisplayForModel* and *EditorForModel*, the system uses reflection to find all the properties on the specified object and then generates a label and visualizer or editor for each of those properties. The overall template of the resulting view is a vertical sequence of labels and visualizers/editors. Each emitted piece of HTML is bound to a CSS class and can be easily styled as you prefer. Furthermore, if you'd like to change the template of the view, you need to provide an *object.aspx* (or *object.cshtml*) template and use reflection as well. I'll return to this topic with an example in Chapter 4.

Custom Helpers

The native set of HTML helper methods is definitely a great help, but it's probably insufficient for many real-world applications. Native helpers, in fact, cover only the markup of basic HTML elements. In this regard, HTML helpers are significantly different from server controls because they completely lack abstraction over HTML. Extending the set of HTML helpers is easy, however. All that is required is an extension method for the *HtmlHelper* class or for the *AjaxHelper* class if you're interested in creating an HTML factory that does some Ajax work.

Structure of an HTML Helper

An HTML helper is a plain method that is not tied to any forced prototype. You typically design the signature of an HTML helper method to make it receive just the data it needs. Having several overloads or optional parameters is common too.

Internally, the helper processes input data and starts building the output HTML by accumulating text in a buffer. This is the most flexible approach, but it is also quite hard to manage when the logic to apply gets complex. An alternate approach consists of generating the HTML using the *TagBuilder* class, which offers an HTML-oriented, string-builder facility. The *TagBuilder* class generates for you the text for HTML tags, thus allowing you to build chunks of HTML by concatenating tags instead of plain strings.

An HTML helper is expected to return an encoded HTML string.

Note In ASP.NET MVC 3, the *TagBuilder* class has moved out of the ASP.NET MVC main assembly. To use the class, you now also need to reference the *System.Web.WebPages* assembly.

MvcHtmlString Is Better Than Just a String

ASP.NET 4 introduced a new, more compact syntax to automatically HTML-encode any text being emitted to the output stream. Consider the following code:

```
<%: ViewData["UserName"] %>
```

It's equivalent to the following:

```
<% Html.Encode(ViewData["UserName"]) %>
```

What if you use the compact syntax on a piece of markup that is already encoded? Without countermeasures, the text inevitably will be double-encoded.

For this reason, it is becoming a best practice to write HTML helpers that return an *MvcHtmlString* wrapper object instead of a plain string. All native HTML helpers, in fact, have been refactored

to return *MvcHtmlString*. The change is no big deal for developers. You can easily obtain an *MvcHtmlString* object from a string through the following code:

```
var html = GenerateHtmlAsString();
return MvcHtmlString.Create(html);
```

The *MvcHtmlString* type is a smart wrapper for a string that has HTML content, and it exposes the *IHtmlString* interface. What's the purpose of *IHtmlString*?

Well, the auto-encoding feature in ASP.NET 4 doesn't apply to any values that implement *IHtmlString*. In light of this, by returning *MvcHtmlString* your HTML helper will always be shielded from double-encoding.

A Sample HTML Helper

Suppose your view receives some text that can be optionally empty. You don't want to render the empty string; you'd rather display some default text such as *N/A*. How do you do that? You can brilliantly resolve everything with an *if* statement. However, nesting an *if* statement in ASPX markup doesn't particularly help to make your code clean. And doing the same in Razor is only a little better.

An ad hoc helper can smooth things because it encapsulates the *if* statement and preserves the required logic, while delivering more compact and readable code. The following code demonstrates an HTML helper that replaces a given string with some default text if it's null or empty:

```
public static class OptionalTextHelpers
{
   public static MvcHtmlString OptionalText(this HtmlHelper helper,
        String text,
        String format="{0}",
        String alternateText="",
        String alternateFormat="{0}")
   {
        var actualText = text;
        var actualFormat = format;

        if (String.IsNullOrEmpty(actualText))
        {
            actualText = alternateText;
            actualFormat = alternateFormat;
        }

        return MvcHtmlString.Create(String.Format(actualFormat, actualText));
    }
}
```

The helper has up to four parameters, three of which are optional parameters. It takes the original text and its null replacement, plus a format string to embellish the text in both cases.

A Sample Ajax Helper

An Ajax helper differs from an HTML helper only because it is invoked in the context of an Ajax operation. Let's suppose, for example, that you want to use an image as a button. A click on the image should automatically trigger an Ajax call to some application URL.

How is this different from just attaching a bit of JavaScript to the *click* event of the image and then using jQuery to carry the call? If you know the URL to pass to jQuery, you don't need this helper. If, however, you find it better to express the URL as a controller/action pair, you need this helper to generate a link that takes the user to wherever the pair controller/action points:

```
public static class AjaxHelpers
{
    public static String ImgActionLink(this AjaxHelper ajaxHelper,
        String imageUrl,
        String imgAltText,
        String imgStyle,
        String actionName,
        String controllerName,
        Object routeValues,
        AjaxOptions ajaxOptions,
        Object htmlAttributes)
    {
        const String tag = "[xxx]";    // arbitrary string
        var markup = ajaxHelper.ActionLink(
             tag, actionName, controllerName, routeValues, ajaxOptions,
             htmlAttributes).ToString();

        // Replace text with IMG markup
        var urlHelper = new UrlHelper(ajaxHelper.ViewContext.RequestContext);
        var img = String.Format(
            "<img src='{0}' alt='{1}' title='{1}' style='{2}' />",
            urlHelper.Content(imageUrl),
            imgAltText,
            imgStyle);
        var modifiedMarkup = markup.Replace(tag, img);
        return modifiedMarkup;
    }
}
```

The helper first invokes the default *ActionLink* helper to get the URL as if it were to be a text-based hyperlink. In the first step, the hyperlink text is set to a known string acting as a placeholder. Next, when everything is ready to go, the helper strips off the placeholder string and replaces that with the URL of the image.

Why can't you just provide the <*img*> tag as the text of the original action link? Being a good citizen, *ActionLink* just HTML-encodes everything, so you won't see any images—just the text of the URL.

The Web Forms View Engine

ASP.NET MVC comes with a default view engine that is extensively based on a subset of the Web Forms machinery. In ASP.NET Web Forms, the generation of the response is based on the processing of a template file expressed using the ASPX markup. The Web Forms machinery is responsible for locating the ASPX source file and compiling it dynamically into a class. Next, the dynamically created class is processed, goes through the ASP.NET page life cycle, and writes any response out at the end of it all.

Inside the View Engine

The Web Forms view engine is a class named *WebFormViewEngine*. The class lives on top of a hierarchy of other classes that encapsulates most of the logic to locate an ASPX view template and render it as an ASP.NET page.

As mentioned, the ASPX view engine looks for view templates under the *Views* folder, and such a folder must be deployed to the production server as well. Let's review in detail the rules employed to resolve view names.

Search Locations

The *WebFormViewEngine* class inherits a bunch of interesting properties from the parent that are related to locations where view templates can be found. Table 2-4 describes these properties.

TABLE 2-4 Properties to express desired location formats for view templates

Property	Description
AreaMasterLocationFormats	Locations where master views are searched in the case of area-based applications
AreaPartialViewLocationFormats	Locations where partial views are searched in the case of area-based applications
AreaViewLocationFormats	Locations where views are searched in the case of area-based applications
MasterLocationFormats	Locations where master views are searched
PartialViewLocationFormats	Locations where partial views are searched
ViewLocationFormats	Locations where views are searched
FileExtensions	List of extensions supported for views, partial views, and master views

The *WebFormViewEngine* constructor sets these properties to default values. Each property is implemented as an array of strings.

You can change these values, and subsequently alter the logic for locating views, in the application startup or, better yet, in a custom view engine. (I'll say more about this later.)

Search Location Formats

Table 2-5 shows the default search locations and extensions as they are defined by the *WebFormViewEngine* class.

TABLE 2-5 Default location formats

Property	Default location format
AreaMasterLocationFormats	~/Areas/{2}/Views/{1}/{0}.master ~/Areas/{2}/Views/Shared/{0}.master
AreaPartialViewLocationFormats	~/Areas/{2}/Views/{1}/{0}.aspx ~/Areas/{2}/Views/{1}/{0}.ascx ~/Areas/{2}/Views/Shared/{0}.aspx ~/Areas/{2}/Views/Shared/{0}.ascx
AreaViewLocationFormats	~/Areas/{2}/Views/{1}/{0}.aspx ~/Areas/{2}/Views/{1}/{0}.ascx ~/Areas/{2}/Views/Shared/{0}.aspx ~/Areas/{2}/Views/Shared/{0}.ascx
MasterLocationFormats	~/Views/{1}/{0}.master ~/Views/Shared/{0}.master
PartialViewLocationFormats	~/Views/{1}/{0}.aspx ~/Views/{1}/{0}.ascx ~/Views/Shared/{0}.aspx ~/Views/Shared/{0}.ascx
ViewLocationFormats	~/Views/{1}/{0}.aspx ~/Views/{1}/{0}.ascx ~/Views/Shared/{0}.aspx ~/Views/Shared/{0}.ascx
FileExtensions	.aspx, .ascx, .master

As you can see, location formats are not fully qualified paths but contain up to three placeholders that an internal method takes care of expanding. The {0} placeholder indicates the view name, the {1} placeholder is for the controller name, and the {2} placeholder is for the area name, if any. The ASPX view engine uses the same location for views and partial views with or without areas.

Note The view engine also uses a cache to speed up the search. Any view name that is successfully resolved is stored in a view location cache. The cache is then checked first on any subsequent access.

The view location cache is abstracted by the *IViewLocationCache* interface and is exposed as a public read/write property named *ViewLocationCache*. The class that provides view location cache services by default is *DefaultViewLocationCache*. It stores any resolved view names in the ASP.NET *Cache* object.

For scalability reasons, you might want to introduce a distributed cache and stop using the ASP.NET cache entirely. You just create your *IViewLocationCache* implementation and use any caching mechanism you like. To register a custom view location cache, you set the *ViewLocationCache* property on the view engine instance. The view location cache saves the path to the file that contains the view template and does that on a controller/area basis.

Virtual Path Providers

The work done by view engines is articulated in two main steps—locating the view and building the response. Both default engines result from the class hierarchy shown in Figure 2-6.

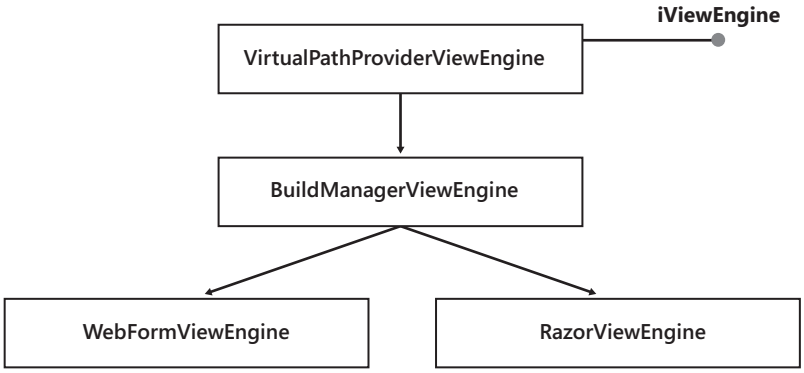

FIGURE 2-6 The class hierarchy of default view engines.

Both the ASPX and Razor view engines rely on the services of a special component—the *VirtualPathProvider* class, defined in the *System.Web.Hosting* namespace within the *system.web* assembly.

Basically, the default view engines leverage the virtual path provider to locate the view templates. These basic capabilities are incorporated in *VirtualPathProviderViewEngine*—the root class of Figure 2-6. The two engines, however, differ in terms of the approach chosen to parse templates and data to actual HTML. The *BuildManagerViewEngine* class abstracts the behavior that actual view engine classes implement.

Note Introduced with ASP.NET 2.0 to serve the needs of the Microsoft Office SharePoint Server development team, the virtual path provider mechanism in ASP.NET is a way to virtualize a bunch of files and even a structure of directories. Up to the latest version (4.0), ASP.NET doesn't read the content of any requested resources directly from disk; instead, ASP.NET gets it through the services of the built-in *VirtualPathProvider* class.

This class assumes a one-to-one correspondence between *.aspx* resources and disk files and serves ASP.NET with just the expected content. By deriving your own class from the system-provided *VirtualPathProvider* class, you can implement a virtual file system for your Web application. In such a virtual file system, you essentially abstract Web content away from the physical structure of the file system. As an example, you might serve incoming page requests based on the source code you have stored in a Microsoft SQL Server database. A virtual path provider takes a file name, directory name, or both, and it returns the content for it (or them). Where the content really comes from is a detail hidden in the implementation of the provider.

Most of the files involved with the processing of an ASP.NET request can be stored in a virtual file system. The list includes ASP.NET pages, themes, master pages, user controls, custom resources mapped to a build provider, and static Web resources such as HTML pages and images. A virtual path provider, however, can't serve global resources (such as *global.asax* and *web.config*) and the contents of reserved folders (such as *Bin*, *App_Data*, *App_GlobalResources*, *App_Browsers*, *App_Code*) and any *App_LocalResources*.

Designing a Sample View

The Web Forms view engine is entirely based on the ASPX markup you might know from ASP.NET Web Forms. The syntax for having a master view, for example, is exactly the same as in Web Forms. To build the body of the view, you use code blocks rather than server controls and resort to HTML helpers when you need to execute some code to produce HTML on the fly (for example, for data-binding purposes.)

Defining a Master View

Each master view template begins with the standard *@Master* directive, as do ASP.NET master pages. A master view template also contains *ContentPlaceHolder* server controls to define areas to be replaced by the actual view. Here's a basic but functional master view:

```
<%@ Master Language="C#" Inherits="System.Web.Mvc.ViewMasterPage" %>
<!DOCTYPE html>
<html>
<head runat="server">
    <title><asp:ContentPlaceHolder ID="TitleContent" runat="server" /></title>
    <link href="<%: Url.Content("~/Content/Site.css") %>"
          rel="stylesheet"
          type="text/css" />
    <script src="<%: Url.Content("~/Scripts/jquery.js") %>"
            type="text/javascript"></script>
</head>
<body>
   <asp:ContentPlaceHolder ID="MainContent" runat="server" />
</body>
</html>
```

Code blocks are used to resolve paths to content files such as style sheet, image, and script files. The *Url.Content* method parses the relative path and resolves it in terms of the actual root of the application.

Important The use of *Url.Content*, as well as the tilde (~) in the URL, is highly recommended to shield your application from annoying issues with paths when it has to support dual deployment as a virtual directory or root application. The tilde is an ASP.NET-specific idiom to indicate the actual root of the application. For this reason, browsers might not know about it and you need a code component to expand it properly. That's exactly what *Url.Content* does for you. Using *Url.Content*, however, is not enough in the case of subdomains within the application, In these cases, you need to rewrite *Url.Content* and a bunch of other helpers to make them return absolute paths that incorporate host information.

Code Blocks

Code blocks are fragments of executable code delimited by <% … %> tags. Within those tags, you can put virtually everything that the underlying build manager can understand and parse, including variable assignments, loop statements, function declarations and, of course, function calls. To be compatible with old Active Server Pages, the internal architecture of classic ASP.NET pages always supported this programming model, which appears unstructured, loose, not very rigorous, and inelegant to software purists and to, well, not just them. This overlooked approach to page construction, however, has been revamped to have new significance in ASP.NET MVC because of its inherent flexibility and because it allows full control over HTML.

Code blocks come in two flavors: inline code and inline expressions. Inline expressions are merely shortcuts for *Response.Write* and preface the expression with an = (equal) symbol. (As mentioned the : symbol can be used if you want auto-encoded output.)

```
<!-- Sample inline expression -->
<%= ViewData["Message"] %>
```

Inline code is plain code in code block brackets, and it requires a trailing semi-colon. An inline expression outputs the value of the expression in the output stream; an inline code block simply executes the specified code to create or modify some local state.

```
<!-- Sample inline code -->
<% RunThisCode(); %>
```

A golden rule of ASP.NET MVC development says that the view always remains disconnected from the machinery of the run-time environment. If the view needs to consume some data, that data must be passed explicitly to the view, using the view dictionary or the model.

Consider now a classic scenario where you need to display a grid of data. The data being passed is the bindable collection of data; however, there's no HTML element that can take it and display in a table-based format. You need to build and populate the <*table*> element yourself. To start with, here's the code through which the controller passes a collection of data down to the view:

```
public ActionResult Index()
{
    ViewBag.Header = "Cities in the world";

    // Prepare data to send
    var cities = new List<City>();
    cities.Add(new City() { Name = "New York", Country = "USA", Visited = true });
    cities.Add(new City() { Name = "Sydney", Country = "Australia", Visited = true });
    cities.Add(new City() { Name = "Madrid", Country = "Spain", Visited = false });
    cities.Add(new City() { Name = "Beijing", Country = "China", Visited = false });
    cities.Add(new City() { Name = "London", Country = "UK", Visited = true });
    return View(cities);
}
```

The view receives the data by declaring the type in the *Inherits* attribute of the content page, as the next block of code shows. Note that in the case of generic types (such as *ViewPage*<*T*>) the *Language* attribute can't be omitted because it helps the parser to figure out the meaning of the text assigned to the *Inherits* attribute:

```
<%@ Page Language="C#"
         MasterPageFile="~/Views/Shared/Site.Master"
         Inherits="System.Web.Mvc.ViewPage<IList<GridDemo.Models.City>>" %>

<asp:Content ID="Content1" ContentPlaceHolderID="TitleContent" runat="server">
    <%: ViewBag.Header %>
</asp:Content>

<asp:Content ID="Content2" ContentPlaceHolderID="MainContent" runat="server">
    <h2><%: ViewBag.Header %></h2>
    <hr />

    <table>
        <thead>
           <th>City</th>
           <th>Country</th>
           <th>Been there?</th>
        </thead>

    <% foreach (var city in Model)
       {%>
       <tr>
          <td><%: city.Name %></td>
          <td><%: city.Country %></td>
          <td><%: city.Visited ?"Yes" :"No"%></td>
       </tr>
    <%
       }%>
    </table>
</asp:Content>
```

Figure 2-7 shows the resulting page.

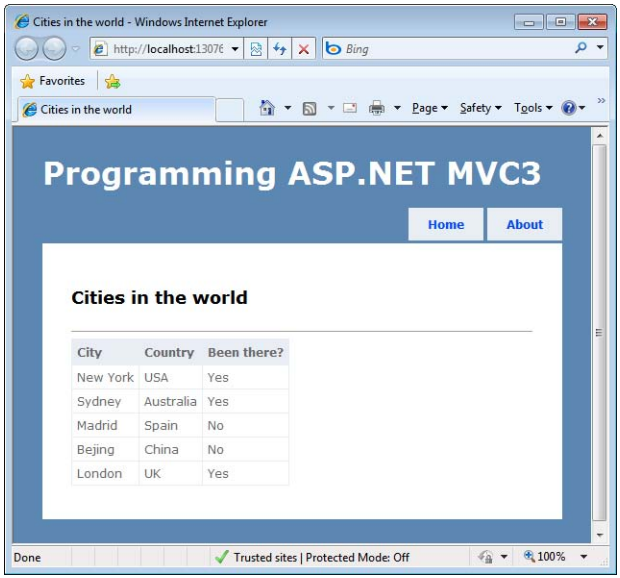

FIGURE 2-7 A data table built using code blocks.

Although the results are just fine, the code necessary to produce it is not. Code blocks interspersed with markup definitely make the source difficult to read and maintain. The way out, however, is not by using server controls, but by using HTML helpers.

HTML Helpers

As the name suggests, an HTML helper is just a productivity tool you use to make the generation of data-driven HTML a bit easier to manage. An HTML helper never reaches the same level of reusability and rapid development as server controls.

An HTML helper is the preferred way to add rendering logic to the view. An HTML helper receives only data the view already holds. An alternative to HTML helpers is to use render action methods, which I'll cover later in the chapter. Let's see how to rewrite the previous code with an ad-hoc helper:

```
public static class TableHelper
{
    public static MvcHtmlString Table(this HtmlHelper helper, ICollection<City> cities)
    {
        var builder = new StringBuilder();
        builder.AppendFormat("<table>");
        builder.AppendFormat("<thead>
                              <th>City</th>
                              <th>Country</th>
                              <th>Been there?</th>
                              </thead>");
```

```
        foreach(var city in cities)
        {
            builder.AppendFormat("<tr><td>{0}</td><td>{1}</td><td>{2}</td></tr>",
                city.Name, city.Country, city.Visited ?"Yes" :"No");
        }

        builder.AppendFormat("</table>");
        return MvcHtmlString.Create(builder.ToString());
    }
}
```

Admittedly, the name *Table* used here for the HTML helper is not a good choice. The HTML being generated is not a generic table; rather, it's just a table for a collection of *City* objects.

In ASP.NET Web Forms, you use a *DataGrid* control; no such helper exists natively for ASP.NET MVC. You find a lot of grid-like helpers from component vendors and from the developer community. The source code for this book also contains an example of a generic and pageable grid. Here's some sample code:

```
<%@ Page Language="C#"
         MasterPageFile="~/Views/Shared/Site.Master"
         Inherits="System.Web.Mvc.ViewPage<PagedViewModel<GridDemo.Models.City>>" %>
...
   <%: Html.Pager("pager1",
                  Model.ItemCount, Model.PageSize, "/home/index", Model.PageIndex) %>
   <%: Html.SimpleGrid(Model.DataSource) %>
</asp:Content>
```

Html.Pager and *Html.SimpleGrid* are sample helpers taken from the source code of this book. They assume the controller method (*Index*) is written to support pagination:

```
public ActionResult Index(Int32? pageIndex)
{
    // Prepare data to send
    ...

    // Set index page to display
    var index = pageIndex.HasValue ? pageIndex.Value : 1;
    index = index <1 ?1 :index;

    // Populate a specific view-model object that knows about pagination.
    var model = new PagedViewModel<City>
                    {
                        PageIndex = index,
                        PageSize = 2,
                        DataSource = cities
                    };
    return View(model);
}
```

The *PagedViewModel* type is also defined in the source code of the book and groups together parameters you need for paging data.

More often than not, HTML helpers are not written for extensive reuse. They are mostly used to keep the code in the view clean and readable. You still can achieve decent levels of reusability with HTML helpers, but this normally happens in a different way than with ASP.NET Web Forms—from the bottom up instead of from the top down.

You typically start with an HTML helper as you need it in a given scenario. Next, when you realize you're using a few similar helpers throughout the application, you can refactor the code to get just one. You never start (and I wouldn't recommend it either) with the idea of writing the super HTML helper that does every possible action. This approach was more common with server controls; it won't work that well here. In summary, don't be afraid of writing many HTML helpers, and don't be afraid of refactoring, either.

Server Controls

ASP.NET MVC doesn't prohibit you from using server controls. At the same time, though, most of the benefits of server controls vanish when they're used in ASP.NET MVC. You can happily use server controls to render a static page that doesn't interact with the user. For example, you can use a *GridView* control to flexibly create a table of data. However, you can't expect to enable advanced features of the *GridView* control such as sorting, paging, and inline editing.

Using server controls that operate postbacks (for example, drop-down lists whose selection becomes the input for successive requests or pageable datagrids) is highly problematic in ASP.NET MVC. You can still add a code-behind class to the ASPX view template and also add a few event handlers there. Your page class will receive events like *Init* and *Load* as usual, except that the *IsPostBack* property is constantly set to *false* and no postback event is ever triggered.

This fact violates a number of consolidated Web Forms practices and makes using server controls in MVC a hazardous approach. So avoid server controls in ASP.NET MVC. The only events that are safe to capture and handle are events raised by controls during the rendering phase. For example, the *RowDataBound* event fired by the *GridView* control during rendering works nicely in ASP.NET MVC. The same can't be said for any postback events, including selection changes in a list and page changes in a grid. Figure 2-8 shows a sample page that uses a *GridView* control to render a table of customer orders. The tooltip is added by intercepting the *RowDataBound* event. Paging doesn't work for free and, frankly, tricks to make it work are not nearly worth the pain. Have a look at the source code for this example and you'll agree. ASP.NET MVC is a different animal than ASP.NET Web Forms. Period.

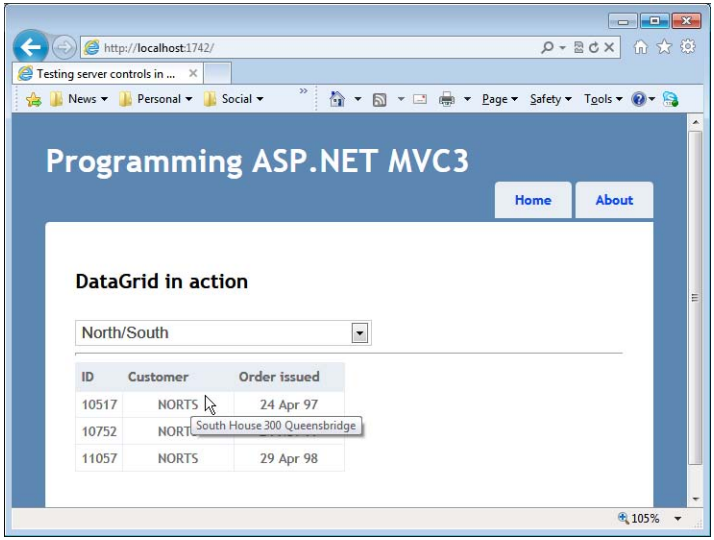

FIGURE 2-8 A tooltip added through a row data-binding event on a server control.

View and ASP.NET Intrinsics

From the view, you can certainly access some ASP.NET intrinsic objects, such as *Cache* and *Session*. The issue, though, is whether you should. And, no, you shouldn't. If the view needs to consume some data, that data must be passed explicitly to the view, using the view dictionary or the model.

Accessing any ASP.NET intrinsic object is the responsibility of the controller, as shown here:

```
public ActionResult AddToShoppingCart(ShoppingItem item)
{
    var cart = this.Session["CurrentShoppingCart"] as ShoppingCart;
    if (cart == null)
       throw new InvalidOperationException("Invalid shopping cart");

    // Do some work on the shopping cart
    cart.Items.Add(item);
    ...

    // Save cart back to the session state
    this.Session["CurrentShoppingCart"] = cart;

    // Show the current content of the cart.
    // The view receives any data it needs to display.
    // It doesn't have to retrieve any of it
    ViewData["CurrentCart"] = cart;
    return View("ShoppingCart");
}
```

As I've said a few times, the view should remain disconnected from the machinery of the run-time environment.

The Razor View Engine

The Web Forms view engine employs a syntax that on one hand is familiar to nearly all ASP.NET developers. On the other hand, the ASPX markup was devised as a way to support server controls, and it has severe limitations when used mostly with code blocks. Clearly, the major issue is lack of readability.

Introduced with ASP.NET MVC 3, the Razor view engine comes to the rescue, providing an alternate markup language to define the structure of view templates. According to Razor, a view template is an HTML page with a few placeholders and code snippets. Overall, the readability of the view template is greatly improved, and by combining Razor code snippets with HTML helpers you can arrange views to make them easier to read and maintain.

Inside the View Engine

Just like the Web Forms view engine, Razor is based on the currently registered ASP.NET virtual path provider. This means that Razor uses the same algorithm as the other view engine to retrieve view templates.

Search Locations

The Razor view engine supports the same set of properties you saw listed in Table 2-5. You can use these properties to set the default search locations and extensions for the engine. The settings employed by Razor are shown in Table 2-6.

TABLE 2-6 Default location formats

Property	Default location format
AreaMasterLocationFormats	~/Areas/{2}/Views/{1}/{0}.cshtml ~/Areas/{2}/Views/Shared/{0}.cshtml ~/Areas/{2}/Views/{1}/{0}.vbhtml ~/Areas/{2}/Views/Shared/{0}.vbhtml
AreaPartialViewLocationFormats	~/Areas/{2}/Views/{1}/{0}.cshtml ~/Areas/{2}/Views/{1}/{0}.vbhtml ~/Areas/{2}/Views/Shared/{0}.cshtml ~/Areas/{2}/Views/Shared/{0}.vbhtml
AreaViewLocationFormats	~/Areas/{2}/Views/{1}/{0}.cshtml ~/Areas/{2}/Views/{1}/{0}.vbhtml ~/Areas/{2}/Views/Shared/{0}.cshtml ~/Areas/{2}/Views/Shared/{0}.vbhtml
MasterLocationFormats	~/Views/{1}/{0}.cshtml ~/Views/Shared/{0}.cshtml ~/Views/{1}/{0}.vbhtml ~/Views/Shared/{0}.vbhtml
PartialViewLocationFormats	~/Views/{1}/{0}.cshtml ~/Views/{1}/{0}.vbhtml ~/Views/Shared/{0}.cshtml ~/Views/Shared/{0}.vbhtml

Property	Default location format
ViewLocationFormats	~/Views/{1}/{0}.cshtml ~/Views/{1}/{0}.vbhtml ~/Views/Shared/{0}.cshtml ~/Views/Shared/{0}.vbhtml
FileExtensions	.cshtml, .vbhtml

As you can see, everything is pretty much the same as with the Web Forms. As far as the view is concerned the only difference is the different file extension and, of course, the different syntax. Let's get familiar with it, then.

Code Nuggets

A Razor view template is essentially an HTML page with a few code snippets, also known as *code nuggets*. Code nuggets are similar to ASP.NET code blocks, but they feature a simpler and terser syntax. You denote the start of a Razor code block with the @ character. More importantly, you don't need to close those blocks explicitly. The Razor parser uses Microsoft Visual Basic or C# parsing logic to figure out where a line of code finishes. Here's an example:

```
<html>
<head>
    <title>@ViewBag.Title</title>
    <link href="@Url.Content("~/Content/Site.css")" rel="stylesheet" type="text/css" />
    <script src="@Url.Content("~/Scripts/jquery.js")" type="text/javascript"></script>
</head>
...
</html>
```

Any Razor code nugget can be mixed with plain markup even when the code nugget contains control flow statements such as an *if/else* or *for/foreach* statement. The following code rewrites the page rendered in Figure 2-7 using code nuggets:

```
<body>
    <h2>@ViewBag.Header</h2>
    <hr />

    <table>
        <thead>
           <th>City</th>
           <th>Country</th>
           <th>Been there?</th>
        </thead>
    @foreach (var city in Model) {
       <tr>
          <td><%: city.Name %></td>
          <td><%: city.Country %></td>
          <td><%: city.Visited ?"Yes" :"No"%></td>
       </tr>
    }
    </table>
</body>
```

Note that the closing brace (}),placed in the middle of the source, is correctly recognized and interpreted by the parser.

Generally, you should note that any C# (or Visual Basic) instructions can be used in a Razor template as long as they are prefixed by @. Here's an example of how you import a namespace and create a form block:

```
@using MvcGallery3.Extensions;
...
<body>
   @using (Html.BeginForm()) {
      <fieldset>
          <div class="editor-field">
                @Html.TextBox("TextBox1")
          </div>
      </fieldset>
   }
</body>
```

In the end, a Razor template is a plain HTML markup file intertwined with @ expressions that wrap executable statements and HTML helpers. However, a few shortcuts exist.

Special Expressions of Code Nuggets

You can insert a full segment of code made of multiple lines anywhere by wrapping it within an @{ code } block like the one shown here:

```
@{
   var user = "Dino";
}
...
<p>@user</p>
```

Any variable you create can be retrieved and used later as if the code belonged to a single block. The content of an @{...} block can mix code and markup. It is essential, however, that the parser can figure out exactly where code ends and markup begins and vice versa. Look at the following nugget:

```
@{
    var number = GetRandomNumber();
    if(number.IsEven())
      <p>Number is even</p>
    else
      <text>Number is odd</text>
}
```

If the markup content you're emitting is wrapped up by HTML tags, the parser will properly recognize it as markup. If it's plain text (for example, just a text string), you must wrap it in a Razor-specific *<text>* tag for the parser to handle it correctly.

A single statement that results from an expression can be combined in the same expression using round brackets:

```
<p> @("Welcome, " + user) </p>
```

Round brackets also work when you intend to place a function call:

```
<p> @(YourMethod(1,2,3)) </p>
```

Any content being processed by Razor is automatically encoded, so you don't need to take care of that. If your code returns HTML markup that you want to emit as is without being automatically encoded, you should resort to using the *Html.Raw* helper method:

```
@Html.Raw(Strings.HtmlMessage)
```

Finally, when inside multiline code nuggets *@{ ... }*, you use the C# or Visual Basic language syntax to place comments. You can comment out an entire block of Razor code using the @* ... *@ syntax:

```
@*
<div> Some Razor markup </div>
*@
```

The Visual Studio 2010 toolbar buttons for commenting blocks in and out support the Razor syntax nicely.

Note Most of the time, the Razor parser is smart enough to figure out from the context the reason why you're using the @ symbol, whether it's to denote a code nugget or perhaps a literal email address. If you hit a corner case that the parser can't successfully solve, using @@ makes it clear that you want the symbol @ to be taken literally and not as the start of a code nugget.

The Razor Syntax and Visual Basic

There are some minor differences between a C#-based Razor view and one based on the Visual Basic language. The most notable is that in a Visual Basic–based Razor view you end up using the @ symbol more often. In Visual Basic, in fact, you can directly write XML literals in the source code. The following C#-based expression is, then, ambiguous to solve for the Visual Basic parser:

```
@if (isFakeApp) {
    <h1>Hello world<h1>
}
```

In Visual Basic, the HTML embedded expression can be mistaken for inline XML. So you are required to prefix HTML literals with an additional @ symbol, as shown here:

```
@If (isFakeApp) Then
    @<h1>Hello world</h1>
End If
```

Another difference is the syntax for indicating a block of code. Instead of using *@{...}*, you resort to the following:

```
@Code
    ...
End Code
```

In addition, most Visual Basic constructs require a closing tag. This is the case for *For..Each*, *If*, *Using*, and more. In all of these cases, the closing tag doesn't need be prefixed with the @ symbol.

Note The Razor implementation is contained in the *System.Web.WebPages* assembly, which you should ensure is bound to the project. The assembly also features a bunch of interesting string extension methods to greatly simplify type conversion. You can use extensions such as *AsInt*, *AsBool*, and *AsDecimal* not just in Razor code nuggets but throughout an ASP.NET MVC project as long as the assembly is referenced and the namespace *System.Web.WebPages.StringExtensions* is imported.

The Razor View Object

When the Razor view engine is used, the resulting view object is an instance of the *WebViewPage* class defined in the *System.Web.Mvc* assembly. This class incorporates the logic to parse markup and render HTML. Public properties on this class are available to any code nuggets you might write in actual templates.

Table 2-7 provides a quick list of a few properties you might be interested in.

TABLE 2-7 Commonly used properties and methods of a Razor view object

Property	Description
Ajax	Gets an instance of the *AjaxHelper* class used to reference Ajax HTML helpers around the template.
Culture	Gets and sets the ID of the culture associated with the current request. This setting influences culture-dependent aspects of a page, such as date, number, and currency formatting. The culture is expressed in *xx-yy* format, where *xx* indicates the language and *yy* the culture.
Href	Converts paths that you create in server code (which can include the ~ operator) to paths that the browser understands.
Html	Gets an instance of the *HtmlHelper* class used to reference HTML helpers in the template.
Context	Gets the central repository to gain access to various ASP.NET intrinsic objects: *Request*, *Response*, *Server*, *User*, and the like.
IsAjax	Returns *true* if the current request was initiated by the browser's Ajax object.
IsPost	Returns *true* if the current request was placed through an HTTP POST verb.
Layout	Gets and sets the path to the file containing the master view template.

Property	Description
Model	Gets a reference to the view model object (if any) containing data for the view. This property is of type *dynamic*.
UICulture	Gets and sets the ID of the user-interface culture associated with the current request. This setting determines which resources are loaded in multilingual applications. The culture is expressed in *xx-yy* format, where *xx* indicates the language and *yy* the culture.
ViewBag	Gets a reference to the *ViewBag* dictionary that might contain data the controller needs to pass to the view object.
ViewData	Gets a reference to the *ViewData* dictionary that might contain data the controller needs to pass to the view object.

Note that not all of these properties are effectively defined directly on the *WebViewPage* class. Many of them are actually defined on parent classes.

Deriving from a Custom Base Class

It is not unusual that you might want to build your own hierarchy of view classes to reuse some application-specific rendering logic. In this case, the actual view class derives from your own class rather than from *WebViewPage*. To have this feature fully enabled in the Razor view engine, you need to be aware of a couple of things.

First, your base must be either marked as abstract or provide a void implementation for the *Execute* method, as shown here:

```
public class MyWebPage : WebViewPage
{
    public MyWebPage()
    {
        Author = "DinoE";
    }

    public override void Execute()
    {
        // No-op or make the class abstract and drop this method.
        return;
    }

    public String Author { get; set; }
}
```

The hierarchy of classes on top of *WebViewPage* leaves the *Execute* method abstract. Expected to contain the actual rendering logic, the *Execute* method is actually overridden only when the code for the view is generated. (Like in Web Forms, the actual view that renders HTML is generated on the fly.)

In the example, the custom view class has an extra property—*Author*. You can't simply reference this property in a Razor's nugget without receiving a compile error. This is because the run time assumes the parent view class is *WebViewPage*, which knows nothing about *Author*. You need to add the *@inherits* declaration and let the run-time system know about the effective base class to use:

```
@inherits RazorDemos.MyWebPage
...
@{
    var author = this.Author;
}
...
<h2>@ViewBag.Message (by @author)</h2>
```

You don't need to use *@inherits* if you're just taking the default *WebViewPage* as your parent view class.

Designing a Sample View

To create a Razor view of some reasonable complexity, you need to understand how to pass data to the view and how to define a master view.

Defining the Model for the View

As mentioned, the controller can pass data down to the view in various ways. It can use global dictionaries such as *ViewBag* or *ViewData*. Better yet, the controller can use a strongly typed object tailor-made for the specific view. I'll discuss the pros and cons of the various approaches in a moment.

To use *ViewBag* or *ViewData* from within a code nugget, you don't need to take any special measures. You just write @ expressions that read or write into the dictionaries. To use a strongly typed view model instead, you need to declare the actual type at the top of the template file, as shown here:

```
@model IList<GridDemo.Models.City>
```

The syntax you use to express the type is the same as the language you use throughout the template to write code nuggets. Next, you access properties in the view model object using the *Model* property as listed in Table 2-7. Here's an example:

```
@model IList<GridDemo.Models.City>

@{
   ViewBag.Title = ViewBag.Header;
}

<h2>@ViewBag.Header</h2>
<hr />
```

```
<table>
    <thead> ... </thead>
@foreach (var city in Model)
{
    <tr> ...  </tr>
}
</table>
```

Of course, if *Model* references an object with child properties, you use *Model.Xxx* to reference each of them.

Note In a Visual Basic–based Razor view, you define a view model object using a different syntax. Instead of using the keyword *@model*, you go with the *@ModelType* keyword.

Defining a Master View

In Razor, *layout pages* play the same role as master pages in the Web Forms view engine. A layout page is a standard Razor template that the view engine renders around any view you define, thus giving a uniform look and feel to sections of the site.

Each view can define its own layout page by simply setting the *Layout* property. The layout can be set to a hardcoded file or to any path that results from evaluating run-time conditions:

```
@{
    if (Request.Browser.IsMobileDevice) {
        Layout = "mobile.cshtml";
    }
}
```

You don't need to incorporate the code that determines the layout in each view file. Razor allows you to define a special file under the Views folder, which is processed before each view is built and rendered. This file, called *_ViewStart.cshtml*, is the ideal container of any view-related startup code, including the code that determines the layout to use. Here's a common implementation for the *_ViewStart.cshtml* file:

```
@{
    Layout = "~/Views/Shared/_Layout.cshtml";
}
```

According to the preceding code snippet, the file *_Layout.cshtml* defines the overall structure of each view in the site. (This is just the Razor counterpart to a master page.)

A layout page contains the usual mix of HTML and code nuggets and derives from *WebViewPage*. As such, it can access any properties on *WebViewPage*, including *ViewBag* and *ViewData*. The

layout template must contain at least one placeholder for injecting the code of a specific view. The placeholder is expressed through a call to the *RenderBody* method (defined on *WebViewPage*):

```
<html>
<head>
    <title>@ViewBag.Title</title>
    <link href="@Url.Content("~/Content/Site.css")" rel="stylesheet" type="text/css" />
    <script src="@Url.Content("~/Scripts/jquery-1.4.4.min.js")" type="text/javascript"></script>
</head>
<body>
    <div class="page">
    @RenderBody()
   </div>
</body>
</html>
```

Executing *RenderBody* causes any code in the actual view to flow into the layout template. Note that the code within the actual view template is processed before the merging of the view and layout. This means you can write view code that programmatically sets values in *ViewBag* that the layout can retrieve and consume. The typical example is the title of the page. Another good example is <*meta*> tags.

Note If you need to import one or more namespaces in the view you're creating, you can just add an *@using* directive at the top of the Razor file. Most notably, the directive must end with a semi-colon just as it should in a regular C# file. If you don't like having namespaces around each view, you can just list all the namespaces in the *web.config* file under the Views folder.

Defining Sections

The *RenderBody* method defines a single point of injection within the layout. Although this is a common scenario, you might need to inject content into more than one location. In the layout template, you define an injection point by placing a call to *RenderSection* at the locations you want those sections to appear:

```
<body>
    <div class="page">
    @RenderBody()
   </div>
   <div id="footer">
      @RenderSection("footer")
   </div>
</body>
```

Each section is identified by name and can be marked as optional. To declare a section optional, you do as follows:

```
<div id="footer">
   @RenderSection("footer", false)
</div>
```

The *RenderSection* method accepts an optional Boolean argument that denotes whether the section is required. The following code is functionally equivalent to the preceding code, but it's much better from a readability standpoint:

```
<div id="footer">
   @RenderSection("footer", required:false)
</div>
```

Note that *required* is not a keyword; more simply, it is the name of the formal parameter defined on the *RenderSection* method. (Its name shows up nicely thanks to IntelliSense.)

If the view template doesn't include a required section, you get a run-time exception. Here's how to define content for a section in a view template:

```
@section footer {
    <p>Written by Dino Esposito</p>
}
```

You can define the content for a section anywhere in a Razor view template.

You can also use the *RenderPage* method if you need to send an entire view straight to the output stream. The *RenderPage* method takes the URL of the view to render. The overall behavior is nearly identical to the *RenderPartial* extension method you might have used plenty of times in ASPX views.

Default Content for Sections

Master pages in the Web Forms view engine allow you to specify some default content for a placeholder to use in case the actual page doesn't fill it in. This same feature is not natively supported in Razor, but some quick workarounds can be arranged. In particular, the *WebViewPage* class provides a handy *IsSectionDefined* method you can leverage in a Razor template to know whether a given section has been specified or not. Here's some code you can use in a layout page to indicate default content for an optional section:

```
@* This code belongs to a layout page *@
<div id="footer">
    @if(IsSectionDefined("Copyright"))
    {
       @RenderSection("copyright")
    }
    else
    {
       <hr /><span>Rights reserved for a better use.</span>
    }
</div>
```

Note that section names are case insensitive.

Nested Layouts

Just as in Web Forms with master pages, you can nest Razor layouts to any level you want. Suppose you want to transform the *about* view created by the standard ASP.NET MVC application into something more sophisticated. The *about* view is based on the general site layout—the *_Layout.cshtml* file. Suppose that you want it to accept contact information from an external view template. Here's the structure you expect:

```
@{
    ViewBag.Title = "About Us";
}
<h2>About</h2>
<p>
     @RenderBody()
</p>
```

A layout template—that is, any Razor template that calls *RenderBody*—can't be requested directly. You get an exception if you try to do that. This means that you should rename the *about.cshtml* file, modified as shown in the preceding code, to something like *aboutLayout.cshtml*. In addition, you must explicitly mention the parent layout it is based on:

```
@{
    Layout = "~/Views/Shared/_Layout.cshtml";
    ViewBag.Title = "About Us";
}
<h2>About</h2>
<p>
     @RenderBody()
</p>
```

Finally, you create the actual *about.cshtml* view, which is the view directly requested by the application:

```
@{
    Layout = "~/Views/Home/AboutLayout.cshtml";
}
<fieldset>
    <legend>Contact</legend>
    <p>Follow me on Twitter: @@despos</p>
</fieldset>
```

Note that to render an @-prefixed Twitter name correctly, you must use a double @. This is not a scenario the Razor parser can solve effectively. Figure 2-9 shows a double-nested *about* view.

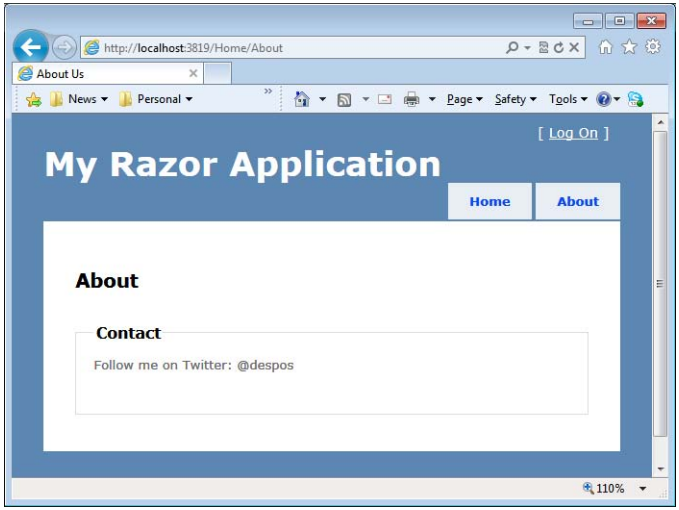

FIGURE 2-9 Nested Razor layouts.

Declarative HTML Helpers

As you saw earlier in the chapter, HTML helpers are a powerful tool to build reusable and parametric pieces of HTML. Written as extension methods for the *HtmlHelper* (or *AjaxHelper*) class, helpers can be easily referenced in a Razor template without requiring even a single code change. This rule applies, for example, to all built-in HTML helpers, such as those used to emit a form or an action link.

The downside of writing HTML helpers as extension methods is the limited flexibility they offer for expressing the graphical layout. To build, say, an HTML table, you need to compose markup into a string builder. It's doable, but not really effective. In Razor, you have an alternate approach—declarative helpers.

You start by creating a *.cshtml* file in the *App_Code* folder of the project. The project wizard doesn't create this folder automatically, so you have to do it yourself. (Curiously, the folder is not even listed as an ASP.NET folder.) The name you choose for the *.cshtml* file is important because you'll be using it when making calls to the helpers. Think of this file as one of your helper repositories. A classic name for this file is something like *MyHelpers.cshtml*. Here's some content:

```
@using RazorEngine.Models;

@helper CityGrid(IList<City> cities)
{
    <table>
        <thead> ... </thead>
        @foreach (var city in cities)
        {
            <tr>
                <td>@city.Name</td>
                <td>@city.Country</td>
```

```
                <td>@(city.Visited ? "Yes" : "No")</td>
            </tr>
        }
    </table>
}

@helper ShowHeader()
{
    <h2>I'm a Razor declarative helper!</h2>
}
```

The *@helper* keyword begins the HTML declaration. The keyword is followed by the signature and implementation of the method. The body of the helper is just an embedded fragment of a Razor template. You can have multiple helpers in a single file. (Note that the helper repository won't be detected if it's placed outside the *App_Code* folder.)

To invoke a declarative Razor helper, you do as follows:

```
@using RazorEngine.Models;
@model IList<City>

@MyHelpers.ShowHeader()
<p>
    @MyHelpers.CityGrid(Model)
</p>
```

Note that the name of the helper is composed of two parts: the repository name and the helper name.

Limitations of Declarative HTML Helpers

You should be aware that a notable limitation applies to Razor declarative HTML helpers. You won't have access to any standard ASP.NET MVC helpers, including all the HTML helpers listed in Table 2-2 and Table 2-3, Ajax helpers, and URL helpers. On more practical terms, you won't be able to use *Html.TextBox* to create a text box; instead, you should resort to using plain HTML elements. Likewise, you can't use *Url.Content* to properly expand a URL when you need to take into account the root ~ operator. Let's see why.

The reason is not easy to understand and explain, but it essentially boils down to Razor currently not being a fully integrated part of ASP.NET MVC. Razor was not originally created for ASP.NET MVC; it was created, instead, for WebMatrix, which is supposed to be yet another alternative for web development. Razor lives in *System.Web.WebPages.dll* and, reasonably, doesn't have a dependency on *System.Web.Mvc.dll*. When building ASP.NET MVC3, the development team decided to incorporate Razor and offer it as a new view engine. The team did a good job, and at first sight, it looks like Razor and ASP.NET MVC were designed together.

Declarative HTML helpers, though, introduce a lot of difficulty to this form of integration. When you create an *@helper* module, it is rendered as a *HelperPage* class, which lives outside the *System.Web.Mvc* assembly. This class offers a property named *Html* of type *HtmlHelper*, but no *Url* property.

The tricky point is that the type *HtmlHelper* on the *HelperPage* class is not the same *HtmlHelper* you have in *System.Web.Mvc*. It is, instead, a type with the same name created for the same purposes, but also to provide its services to WebMatrix. To make a long story short, the following code fails because of a null reference exception in *Html*:

```
@helper ShowMessage(String message)
{
    @* Html is defined on HelperPage *@
    <h2>@Html.Raw(message)</h2>
}
```

Internally, *Html* attempts to cast the current page to *WebPage* (the only view object defined in the Razor view engine in WebMatrix). Because it receives a *WebViewPage* (the view object defined for the Razor view engine in ASP.NET MVC) instead, the cast fails and you get an exception.

Which workarounds will work?

In a declarative Razor HTML helper, you use *Href* to replace *Url.Content*, which provides the same capabilities. As far as native HTML helpers are concerned, you can choose between two options.

First, you pass a valid *HtmlHelper* object to your helper and use the parameter to call built-in helpers. Alternatively, you can use the following type, which provides a comprehensive solution:

```
public static class MvcIntrinsics
{
    public static System.Web.Mvc.HtmlHelper Html
    {
        get {return ((System.Web.Mvc.WebViewPage)WebPageContext.Current.Page).Html;}
    }

    public static System.Web.Mvc.AjaxHelper Ajax
    {
        get { return ((System.Web.Mvc.WebViewPage)WebPageContext.Current.Page).Ajax; }
    }

    public static System.Web.Mvc.UrlHelper Url
    {
        get { return ((System.Web.Mvc.WebViewPage)WebPageContext.Current.Page).Url; }
    }
}
```

To use these properties, you end up with code that's nearly identical to what you would have written originally:

```
@helper ShowMessage(String message)
{
    <h2>@MvcIntrinsics.Html.Raw(message)</h2>
}
```

Note What's the lesson learned from this? Writing real-world code is hard, but adhering to certain principles helps you to keep your head above the big ball of mud. "Don't Repeat Yourself (DRY)" is more than a cool quotation.

Templated Delegates

A *templated delegate* is a Razor @ expression that includes code and markup. Implemented as a delegate, such an expression can be passed around as an argument. Let's see a few scenarios where these delegates can be helpful.

Templated Delegates and Formatted Text

Imagine you have an HTML helper that displays some data it receives from callers:

```
@helper ShowMessage(String message)
{
    <h2>@message</h2>
}
```

What if *message* contains HTML formatting? Try this code in a *.cshtml* view:

```
@{
    var message = "Hello, Razor";
    var formattedMessage = "<i>" + message + "</i>";
}
...
@MyHelpers.ShowMessage(message)
```

Any markup resulting from an @ expression is automatically HTML-encoded, so it should come as no surprise that you end up with the following markup being sent to the browser:

```
<h2>&lt;i&gt;Hello, Razor&lt;/i&gt;</h2>
```

Important If all you want to do is display formatted text, you can brilliantly and quickly do that by using *Html.Raw* as discussed earlier. Internally, *Html.Raw* just wraps any text you provide in a newly created instance of the *HtmlString* class. ASP.NET takes this object as it's already encoded and doesn't further encode it. Templated delegates go beyond formatting text and allow code and markup to be wrapped in a single object.

Let's see how templated delegates come to the rescue. First, a templated delegate is defined as shown here:

```
Func<T, TResult>
```

TResult must be *HelperResult*—the wrapper type Razor uses for rendering any @ expressions. The type *T*, instead, can be anything you like, including *Object* and *dynamic*. Here's how you can rewrite the HTML helper to make it accept templates:

```
@helper ShowTemplatedMessage(Func<String, HelperResult> funcTemplate, String message)
{
    <h2>@funcTemplate(message)</h2>
}
```

And here's how you can arrange the call to such a template from a Razor view:

```
@MyHelpers.ShowTemplatedMessage(@<i>@item</i>, @message)
```

To define a templated delegate, you use the @... syntax and place any markup you like past the initial @ symbol. Nicely enough, the markup can contain an embedded @ expression referring to variables and functions.

You should note that *@item*, which is used in the code snippet just shown, has a special meaning. It is used to refer to the only parameter that will be passed to the template. Put another way, *@item* refers to the value of type *T* you're passing to the template. Figure 2-10 shows the different effect of using templated delegates and plain HTML-formatted strings.

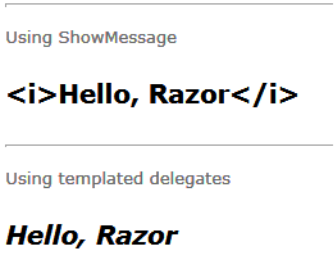

FIGURE 2-10 Passing HTML-formatted strings to helpers just doesn't work.

Templated Delegates and Optional Sections

Templated delegates can be used to provide a more elegant solution to the problem of giving default content to optional layout sections. You can write an extension method for *WebViewPage* that adds another overload to *RenderSection*. The code can go to an external library, to the App_Code folder, or to wherever you store helper classes in your project.

```
public static class WebPageExtensions
{
    public static HelperResult RenderSection(this WebViewPage thePage,
        String sectionName,
        Func<Object, HelperResult> funcDefaultContent)
    {
        // Use the section template if specified in the actual content page.
        if (thePage.IsSectionDefined(sectionName))
```

```
        {
            return thePage.RenderSection(sectionName);
        }

        // Go with default content as passed as an argument here.
        return funcDefaultContent(null);
    }
}
```

The method gets the section name and a templated delegate, and it renders the templated delegate if the section is not defined. Here's how you use the method:

```
<div id="footer">
    @RenderSection("footer", required:false)
    @if(IsSectionDefined("Copyright"))
    {
       @RenderSection("copyright")
    }
    else
    {
       <hr /><span>Rights reserved for a better use.</span>
    }
    @this.RenderSection("Privacy", @<u>Privacy policy</u>)
</div>
```

The preceding example shows different ways of dealing with optional sections. It is key to note that you need to use the *this* keyword in the call to the *RenderSection* extension method. That's because the method is an extension method—extension methods are a compiler trick you can use to add custom methods to specific instances of a class.

The *ViewStart* Page Class

If you don't like extension methods, an alternate approach for sharing default section content, as well as any other data around layouts and views, is to leverage the *PageData* dictionary of the *ViewStartPage* class.

The *ViewStartPage* class contains the basic behavior that results from code expressly written in the *_ViewStart.cshtml* file. This file represents the code being run at the beginning of the view rendering process. Anything you do here affects the rest, and anything you calculate or define here can be shared with layouts and views. Here's the typical content of the *_ViewStart.cshtml* file:

```
@{
    Layout = "~/Views/Shared/_Layout.cshtml";
}
```

This code sets the *Layout* file to be used by all views being called in the site. Likewise, you can add default content for optional sections. Unfortunately, *ViewStartPage* doesn't inherit from the same hierarchy as layouts, so you can't use the *@section* syntax to declare sections that other views

can override. What you can do, instead, is share templated delegates. Here's how you can share a delegate in *_ViewStart.cshtml*:

```
@{
    Layout = "~/Views/Shared/_Layout.cshtml";
    Func<object, HelperResult> privacyTemplate =
                @<u>
                    Our privacy policy from viewstart.
                 </u>;
    PageData["privacy"] = privacyTemplate;
}
```

PageData is a dictionary defined on *ViewStartPage* whose content is shared with layouts and views. The following code shows a revamped version of the *RenderSection* extension method that also checks *PageData* while looking for default section contents:

```
public static class WebPageExtensions
{
    public static HelperResult RenderSection(this WebViewPage thePage,
        String sectionName,
        Func<Object, HelperResult> funcDefaultContent)
    {
        // Render the section template as specified in the actual content page.
        if (thePage.IsSectionDefined(sectionName))
        {
            return thePage.RenderSection(sectionName);
        }

        //Render the default section template from the PageData dictionary
        if (thePage.PageData.ContainsKey(sectionName))
        {
            var templateFunc = thePage.PageData[sectionName] as Func<Object, HelperResult>;
            if (templateFunc != null)
                return templateFunc(null);
        }

        // Go with default content as passed as an argument here.
        return funcDefaultContent(null);
    }
}
```

As mentioned, *PageData* is a general-purpose dictionary of type *IDictionary<object, object>* and can be used to pass any sort of data down to actual views.

Coding the View

In this final section of the chapter, I'll delve into a couple of important points that affect views and controllers—how to effectively pass data to a view and how to add more flexibility to the view-rendering processing.

Modeling the View

Using Razor or the ASPX engine, you define the graphical layout of a view. Sometimes the view is just made of static content. More often, though, the view must incorporate real data resulting from some operation against the middle tier of the system or loaded from the application cache or session state. How would you make this data available to the view?

The golden rule of ASP.NET MVC design claims that the view receives, and doesn't compute, any data it has to display. Data can be passed in three nonexclusive ways: via the *ViewData* dictionary, the *ViewBag* dictionary, and a tailor-made container object.

Note The *ViewBag* dictionary is supported only in ASP.NET MVC 3, and it requires the application to be compiled for the Microsoft .NET Framework 4.

The *ViewData* Dictionary

Exposed directly by the *Controller* class, the *ViewData* property is a name-value dictionary object. Its programming model is analogous to using *Session* or other ASP.NET intrinsic objects:

```
public ActionResult Index()
{
    ViewData["PageTitle"] = "Programming ASP.NET MVC";
    return View();
}
```

Any data you store in a dictionary is treated as an object and requires casting, boxing, or both for it to be worked on in the view. You can create as many entries as you like in the dictionary. The lifetime of the dictionary is the same as the request.

The *ViewData* dictionary is packed into the view context—an internal structure through which the ASP.NET MVC infrastructure moves data from the controller level to the view level—and made available to the view engine. The view objects—*ViewPage* for the ASPX engine and *WebViewPage* for Razor—expose the *ViewData* dictionary to any code in the view templates. Here's how you retrieve *ViewData* content from a view template:

```
<head>
   <title> <%= ViewData["PageTitle"] %> </title>
</head>
```

In Razor, it works the same except for the different code nugget syntax:

```
<head>
   <title> @ViewData["PageTitle"] </title>
</head>
```

If the *ViewData* dictionary contains something other than strings, you might want to cast content to the proper type to invoke more specific methods.

Overall, the *ViewData* dictionary is straightforward to use and extremely flexible. In fact, it allows you to pass a new piece of data to the view by simply creating a new entry. At the same time, the name-based model forces you to use a lot of magic strings and, more importantly, to match them between the controller and view code. By using constants, you can reduce some of the inherent brittleness of magic strings, but you still have no defense against the possibility of picking up the wrong name. If you happen to reference the wrong dictionary entry, you'll find out only at run time. The *ViewData* dictionary is well suited for simple solutions and applications with a relatively short lifetime. As the number of dictionary entries and the number of views grow, maintenance becomes an issue and you should move away from *ViewData* when looking for other options.

The *ViewBag* Dictionary

Also defined on the *Controller* class, the *ViewBag* property offers an even more flexible facility to pass data to the view. The property is defined as a *dynamic* type—a new entry in the .NET 4 ecosystem:

```
public dynamic ViewBag { get; }
```

A .NET 4 compiler that encounters a *dynamic* type emits a special chunk of code instead of simply evaluating the expression. Such a special chunk of code passes the expression to the Dynamic Language Runtime (DLR) for a run-time evaluation. In other words, any expression based on the *dynamic* type is compiled to be interpreted at run time. Any member set or read out of *ViewBag* is always accepted by compilers but not actually evaluated until execution. Here's an example that compares the usage of *ViewData* and *ViewBag*:

```
public ActionResult Index()
{
    // Using ViewData
    ViewData["PageTitle"] = "Programming ASP.NET MVC";

    // Using ViewBag
    ViewBag.PageTitle = "Programming ASP.NET MVC";

    return View();
}
```

The compiler doesn't care whether a property named *PageTitle* really exists on *ViewBag*. All it does is pack a call to the DLR interpreter, where it asks the DLR to try to assign a given string to a certain *PageTitle* property. Similarly, when *PageTitle* is read out of *ViewBag*, the compiler instructs the DLR

to check whether such a property exists. If it doesn't exist, an exception is thrown. Here's how you consume content from *ViewBag* in a Razor view:

```
<head>
   <title> @ViewBag.PageTitle </title>
</head>
```

From a developer's perspective, which is better—*ViewBag* or *ViewData*? The *ViewBag* syntax is terser than the *ViewData* syntax, but as I see things, that's the entire difference. Just as with *ViewData*, you won't have compile-time checking on properties. The dependency of the *dynamic* type on the DLR doesn't save you run-time exceptions if you mistype a property name. In the end, it's purely a matter of preference. Note also that *ViewBag* requires ASP.NET MVC 3 and .NET 4, whereas *ViewData* works with any version of ASP.NET MVC and with .NET 2.0.

Note Because the *dynamic* type is resolved at run time, Visual Studio IntelliSense can't tell you anything about its properties. Visual Studio IntelliSense treats a *dynamic* type like a plain *Object* type. Some tools—most noticeably JetBrains ReSharper—are a bit smarter. ReSharper tracks all the properties met along the way in the scope where the *dynamic* variable is used. For any properties used, an entry is added to the IntelliSense menu of members.

Strongly Typed View Models

When you have dozens of distinct values to pass to a view, the same flexibility that allows you to quickly add a new entry, or rename an existing one, becomes your worst enemy. You are left on your own to track item names and values; you get no help from Microsoft IntelliSense and compilers.

The only proven way to deal with complexity in software is through appropriate design. So defining an object model for each view helps you track what that view really needs. I suggest you define a view-model class for each view you add to the application:

```
public ActionResult Index()
{
    ...
    // Pack data for the view using a view-specific container object.
    var model = new YourViewModel();

    // Populate the model.
    ...

    // Trigger the view.
    return View(model);
}
```

Having a view-model class for each view also creates the problem of choosing an appropriate class name. You could use a combination of controller and view names. For example, the view-model object for a view named *Index* invoked from the *Home* controller might be named

HomeIndexViewModel. (This particular approach is just a suggestion, though; you should just feel free to choose meaningful names.)

How would you shape up a view-model class?

First and foremost, a view-model object is a plain data-transfer object with only data and (nearly) no behavior. Ideally, properties on a view-model object expose data exactly in the format the view expects it to be. For example, if the view is expected to display only the date and status of pending orders, you might not want to pass a plain collection of full-blown *Order* objects as they result from the middle tier. The following view-model class is a better choice for modeling data for the view. It helps keeping the presentation layer and middle tier decoupled.

```
public class LatestOrderViewModel
{
    public DateTime OrderDate { get; set; }
    public String Status { get; set; }
}
```

The ASP.NET MVC infrastructure guarantees that *ViewData* and *ViewBag* collections are always made available to the view object without any developer intervention. The same is not true for custom view-model objects.

When a view-model object is used, the view-model type must be declared in the view template so that the actual view object can be created of type *ViewPage<T>* in the ASPX view engine and of type *WebViewPage<T>* if Razor is used. Here's what you need to have in an *.aspx* template if you use the ASPX Web Forms view engine:

```
<%@ Page Language="C#" MasterPageFile="~/Views/Shared/Site.Master"
         Inherits="System.Web.Mvc.ViewPage<LatestOrderViewModel>" %>
```

In the preceding code snippet, the type assigned to the *Inherits* attribute is not fully qualified. This means that you probably need to add an *@Import* directive to specify the namespace to import to locate the view-model type. If the view engine is Razor, here's what you need:

```
@model LatestOrderViewModel
```

To retrieve the view-model object in the view template, you use the *Model* property defined on both *WebViewPage* and *ViewPage*. Here's a Razor example:

```
<h2>
    Latest order placed
    <span class="highlight">@Model.OrderDate.ToString("dddd, dd MMM yyyy")</span>
    <br />
    Status is: <span class="highlight">@Model.Status</span>
</h2>
```

In this example, the format of the date is established in the view. It is also acceptable that the controller prepares the date as a string and passes it down to the view, ready for display. There's not a clear guideline on where this code belongs: it falls into a sort of gray area. My preference is to keep the view as simple as possible. If the format is fixed, and not dependent on run-time conditions,

it is acceptable for you to pass *DateTime* and let the view figure out the rest. When a bit of logic is required to format the date, in general I prefer to move it up to the controller.

Although each view should have its own model object, limiting the number of classes you deal with is always a good idea. To reuse model classes in multiple views, you typically build a hierarchy of classes. Here's an example of a view-model base class:

```
public class ViewModelBase
{
    public String Title { get; set; }
    public String Header { get; set; }
}
public class LatestOrderViewModel : ViewModelBase
{
    ...
}
```

Finally, it is always a good approach to devise the structure of the view-model class around the view rather than around the data. In other words, I always prefer to have a view-model class designed as a container.

Suppose you need to pass a list of orders to the view. The first option that springs to mind is using the following view-model class (in a Razor template):

```
@model IList<PendingOrder>
```

Functionally speaking, the approach is sound. But is it extensible? In my experience, you always end up stuffing a variety of heterogeneous data in a view. This might be required because of refactoring efforts, as well as new requirements that pop up. A more maintainable approach (that will let you refactor without changing controller/view interfaces) is the following:

```
@model YourViewModel
```

In this case, *YourViewModel* is defined as shown here:

```
public class YourViewModel
{
    public IList<PendingOrder> PendingOrders {get; set;}
}
```

A view-model class ultimately models the view, not the data.

Important I'm not sure if I stated it clearly enough, so let me rephrase it. Strongly typed view models are the only safe and sound solution for any ASP.NET MVC application of at least moderate complexity and duration. I do believe that using view models is a state of mind more than a way to fight complexity. However, if you can make it work with a couple of *ViewData* or *ViewBag* entries per view, and you'll be throwing the site away after a few months (for example, a site you set up for a specific event), by all means ignore view models.

Using Expando Objects to Control Class Proliferation

In the previous example, you have two distinct classes in action: *PendingOrder* and *YourViewModel*. The latter is the actual view-model class that the controller and view engine exchange. The former is a plain helper class. With a high number of views and rich data to pass, the number of such helper classes can grow quite large.

Such helper classes are not complex to write because they are bare data-transfer objects. Yet they are part of your application and contribute to project noise and pollution. At a minimum, having too many helper classes creates naming issues. There's a very real risk you'll end up with quite a few similarly named classes, which is never great for readability. (And, in turn, limited readability is never great for overall code maintainability.)

Here's an alternative (and quite pragmatic) approach you might want to consider to limit helper class proliferation. This shortcut is well suited to situations in which your view-model classes contain collections of other data-transfer objects:

```
public class PendingOrdersViewModel
{
    // Cleaner approach using a helper view-model object.
    // public ICollection<PendingOrder> Orders { get; set; }

    // More pragmatic approach
    public ICollection<dynamic> Orders { get; set; }
}
```

Instead of using child view-model objects for representing internal data, you resort to using the *dynamic* type. In the view template, you access child objects as shown here:

```
<ul>
@foreach (var o in Model.Orders)
{
    <li>@o.OrderDate.ToShortDateString(), @o.Status</li>
}
</ul>
```

Because elements in *Orders* are of type *dynamic*, there's no compile-time checking on properties being accessed. How would you stuff data in a *dynamic* variable?

```
public ActionResult Pending()
{
    var model = new PendingOrdersViewModel
                    {
                        Orders = new List<dynamic>()
                    };

    // Populate the view model with expando objects.
    dynamic o1 = new ExpandoObject();
    o1.OrderDate = DateTime.Today;
    o1.Status = "Pending";
    model.Orders.Add(o1);
```

```
    dynamic o2 = new ExpandoObject();
    o2.OrderDate = DateTime.Today.AddDays(-1);
    o2.Status = "Pending";
    model.Orders.Add(o2);

    return View(model);
}
```

A variable is a reference to a memory location. For a compiler of static-typed languages (for example, C#), each variable is bound to a fixed type that is known at compile time. A *dynamic* variable is a reference to a memory location whose actual type is determined at run time. You still need an object in which you pack data to be consumed later. This container is the *ExpandoObject* type. An *ExpandoObject* variable provides the same behavior as JavaScript objects—you can add properties and methods programmatically.

Packaging the View-Model Classes

Where should you define the view-model classes? This mostly depends on the size of the project. In a large project with a good deal of reusability and an expected long lifetime, you probably should create a separate class library with all view-model classes you use.

In smaller projects, you might want to isolate all the classes in a specific folder. This can be the Models folder that the default Visual Studio project templates create for you. Personally, I rename *Models* to *ViewModels* and group classes in controller-specific subfolders, as shown in Figure 2-11.

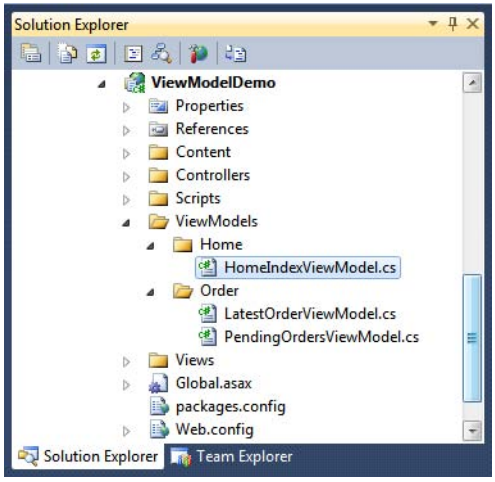

FIGURE 2-11 A suggested structure for the ViewModels folder.

Advanced Features

ASP.NET MVC is built around the *Convention-over-Configuration* pattern. As is typical of frameworks, the pattern saves developers a lot of programming details as long as they adhere to a number of fixed rules and conventions. This is particularly evident with views.

The view engine works by retrieving and processing view templates. How does it know about templates? In general, it can happen in either of two ways: you register known views with the engine (configuration), or you place views in specific locations so that the engine can retrieve them (convention). What if you want to organize your views according to a different set of conventions? Quite simply, you need your own view engine.

Note The need for a custom view engine is more frequent than one might think at first. You might create a custom view engine for two main reasons: you want to express views in a new markup language, or you want to apply a set of personalized conventions. I'm not saying that every application should use a custom markup language, but most applications might benefit from views organized in a custom way.

Custom View Engines

Most applications of mine employ their own view engine that just organize views in a slightly different way or need an extra layer of code to resolve view names to actual markup files. If you have reasons for using a different directory schema for some of your views, all you need to do is derive a simple class as shown here:

```
public class MyWebFormsViewEngine : WebFormViewEngine
{
    public MyWebFormsViewEngine()
    {
        // Ignoring areas in this example

        this.MasterLocationFormats = base.MasterLocationFormats;
        this.ViewLocationFormats = new string[]
                                        {
                                           "~/Views/{1}/{0}.aspx"
                                        };

        // Customize the location for partial views
        this.PartialViewLocationFormats = new string[]
                                        {
                                           "~/PartialViews/{1}/{0}.aspx",
                                           "~/PartialViews/{1}/{0}.ascx"
                                        };
    }
}
```

To use this class in lieu of the default view engine, you enter the following in *global.asax*:

```
protected void Application_Start()
{
    ...

    // Removes the default engines and adds the new one.
    ViewEngines.Engines.Clear();
    ViewEngines.Engines.Add(new MyWebFormsViewEngine());
}
```

After you do this, your application will fail if any of the partial views is located outside a PartialViews subfolder.

The example builds a slight variation of the default Web Forms view engine. You can do the same for Razor and for any other third-party view engine you happen to use.

Note If all you need to do is set location format properties on one of the default engines, you might not need to create a custom view engine. It might suffice that you retrieve the current instance in *Application_Start* and set location format properties directly.

Important If you have a custom view engine that supports custom folders (for example, a *PartialViews* folder for grouping partial views), you must add a *web.config* file to it. You can copy the same *web.config* file you find in the Views folder by default. That file contains critical information for the ASP.NET MVC run time to locate view classes correctly.

Render Actions

Complex views result from the composition of a variety of child views. When a controller method triggers the rendering of a view, it must provide all data the view needs for the main structure and all of the parts. Sometimes, this requires the controller to know a lot of details about parts of the application the class itself is not directly involved with. Want an example?

Suppose you have a menu to render in many of your views. Whatever action you take in relation to your application, the menu has to be rendered. Rendering the menu, therefore, is an action not directly related to the current ongoing request. How would you handle that? *Render actions* are a possible answer.

A render action is a controller method that is specifically designed to be called from within a view. A render action is therefore a regular method on the controller class that you invoke from the view by using one of the following HTML helpers: *Action* or *RenderAction*.

```
<%: Html.Action("action") %>
@Html.Action("action")
```

Action and *RenderAction* behave mostly in the same way; the only difference is that *Action* returns the markup as a string, whereas *RenderAction* writes directly to the output stream. Both methods support a variety of overloads through which you can specify multiple parameters, including route values, HTML attributes and, of course, the controller's name. You define a render action as a regular method on a controller class and define it to be the renderer of some view-related action:

```
public ActionResult Menu()
{
    var options = new MenuOptionsViewModel();
    options.Items.Add(new MenuOption {Url="...", Image="..."});
```

```
    options.Items.Add(new MenuOption {Url="...", Image="..."});
    return PartialView(options);
}
```

The content of the menu's partial view is not relevant here; all it does is get the model object and render an appropriate piece of markup. Let's see the view source code for one of the pages you might have in the application:

```
<asp:Content ID="Content1" ContentPlaceHolderID="MainContent" runat="server">
    ...
    <% Html.RenderAction("Menu"); %>
    <hr />
    <!-- Remainder of the view here -->
    ...
</asp:Content>
```

The *RenderAction* helper method calls the *Menu* method on the specified controller (or on the controller that ordered the current view to be rendered) and directs any response to the output stream. In this way, the view incorporates some logic and calls back the controller. At the same time, your controller doesn't need to worry about passing the view information that is not strictly relevant to the current request it is handling.

Child Actions

The execution of a render action is not simply a call made to a method via reflection. A lot more happens under the hood. In particular, a render action is a child request that originates within the boundaries of the main user request. The *RenderAction* method builds a new request context that contains the same HTTP context of the parent request and a different set of route values. This child request is forwarded to a specific HTTP handler—the *ChildActionMvcHandler* class—and is executed as if it came from the browser. The overall operation is similar to what happens when you call *Server.Execute* in general ASP.NET programming. There's no redirect and no roundtrip, but the child request goes through the usual pipeline of a regular ASP.NET MVC request.

By default, any action method can be invoked from a URL and via a render action. However, any action methods marked with the *ChildActionOnly* attribute won't be available to public callers, and their usage is limited to rendering actions and child requests.

Testing the View

In ASP.NET MVC, the HTML for the view is generated by the controller when it invokes the *View* method. If you consider the structure of the page trivial or just static, it might suffice that you ensure the correct data is passed to the view. This can be achieved easily through tests on the controller. (See Chapter 9, "Testing and Testability in ASP.NET MVC.")

Testing the front end of a Web application goes beyond classic unit testing and requires that you use ad hoc tools. In this regard, ASP.NET MVC is not much different from ASP.NET Web Forms, or even from Java or PHP web applications.

You need a tool that allows you to programmatically define a sequence of typical user actions and observe the resulting Document Object Model (DOM) tree. In other words, you want to test the layout and content of the response when the user performs a given series of actions.

Such tools have recording features, and they keep track of user actions as they are performed and store them as a reusable script to play back. Some tools also offer you the ability to edit test scripts or write them from scratch. Here's a sample test program written for one of the most popular of these front-end test tools—WatiN. The program tests the sample page we discussed earlier with a drop-down list and a grid:

```
public class SampleViewTests
{
   private Process webServer;

   [TestInitialize]
   public void Setup()
   {
      webServer = new Process();
      webServer.StartInfo.FileName = "WebDev.WebServer.exe";
      string path = ...;
      webServer.StartInfo.Arguments = String.Format(
             "/port:8080 /path: {0}", path);

      webServer.Start();
   }

   [TestMethod]
   public void CheckIfNicknameIsNotUsed()
   {
      using (IE ie = new IE("http://localhost:8080/Samples/Datagrid"))
      {
         // Select a particular customer ID
         ie.SelectList("ddCustomerList").Option("1").Select();

         // Check the resulting HTML on first row, second cell
         Assert.AreEqual(
                "A Bike Store",
                ie.Table(Find.ById("gridOrders").TableRow[0].TableCells[1].InnerHtml));
      }
   }

   [TestCleanup]
   public void TearDown()
   {
      webServer.Kill();
   }
}
```

The testing tool triggers the local Web server and points it to the page of choice. Next, it simulates some user actions and checks the resulting HTML.

Different tools might support a different syntax and might integrate with different environments and in different ways. However, the previous example gives you the gist of what it means to test the front end.

Web UI testing tools can be integrated as extensions into browsers (for example, Firefox), but they also offer an API for you to write test applications in C# or test harnesses using MSTest, NUnit, or other test frameworks. Table 2-8 lists a few popular tools.

TABLE 2-8 Tools for testing a web front end

Tools	More information
ArtOfTest	*http://www.telerik.com/automated-testing-tools.aspx*
Selenium	*http://seleniumhq.org*
WatiN	*http://watin.org*

Summary

ASP.NET MVC doesn't match URLs to disk files; instead, it parses the URL to figure out the next requested action to take. Each action terminates with an action result. Most common type of action result is the view result, which consists of a chunk of HTML markup.

Generated by the controller method, a view result is made of a template and model. The view engine takes care of parsing the view template and filling it in with model data. ASP.NET MVC comes with two default view engines supporting different markup languages for expressing the template and different disk locations to discover templates.

In this chapter, we first examined what it takes to process a view and then focused on development aspects, including using HTML helpers and templated helpers for the two default engines—ASPX and Razor. We also discussed best practices for modeling data and contrasted dictionaries with strongly typed view models.

CHAPTER 3

The Model-Binding Architecture

It does not matter how slowly you go, so long as you do not stop.

—Confucius

By default, the Microsoft Visual Studio standard project template for ASP.NET MVC applications includes a Models folder. If you look around for some guidance on how to use it and information about its intended role, you quickly reach the conclusion that the Models folder exists to store model classes. Fine, but which model is it for? Or, more precisely, what's the definition of a "model"?

In general, there are at least two distinct models—the domain model and the view model. The former describes the data you work with in the middle tier and is expected to provide a faithful representation of the domain entities. These entities are typically persisted by the data-access layer and consumed by services that implement business processes. This *domain model* (or *entity model* or even *data model*, if you like) pushes a vision of data that is, in general, distinct from the vision of data you find in the presentation layer. The view model just describes the data being worked on in the presentation.

Having said that, I agree with anyone who says that not every application needs a neat separation between the object models used in the presentation and business layers. Nonetheless, two distinct models logically exist, and coexist, in a typical layered web solution. You might decide that for your own purposes the two models nearly coincide, but you should always recognize the existence of two distinct models that operate in two distinct layers.

This chapter introduces a third type of model that, although hidden for years in the folds of the ASP.NET Web Forms runtime, stands on its own in ASP.NET MVC—the *input model*. The input model refers to the model through which posted data is exposed to controllers.

In Chapter 1, "ASP.NET MVC Controllers," we discussed request routing and structure of controller methods. In Chapter 2, "ASP.NET MVC Views," we discussed views as the primary result of action processing. We didn't discuss thoroughly yet how a controller method gets input data.

The Input Model

In ASP.NET Web Forms, we had server controls, view state, and the overall page life cycle working in the background to serve developers input data that was ready to use. With ASP.NET Web Forms, developers had no need to worry about an input model. Server controls in ASP.NET Web Forms provide a faithful server-side representation of the client user interface. Developers just need to write C# code to read from input controls.

This approach doesn't work well with the philosophy of ASP.NET MVC—which is more close to the metal with very thin abstraction over HTTP. Moreover, ASP.NET MVC makes a point of having highly testable controllers—which means that controllers should receive input data, not retrieve it. To pass input data to a controller, you need to package data in some way. This is precisely where the input model comes into play.

To better understand of the importance and power of the new ASP.NET MVC input model, let's start from where ASP.NET Web Forms left us.

Evolving from the Web Forms Input Processing

An ASP.NET Web Forms application is based on pages, and each server page is based on server controls. The page has its own life cycle that spans from processing the raw request data to arranging the final response for the browser. The page life cycle is fed by raw request data such as HTTP headers, cookies, the URL, and the body, and it produces a raw HTTP response containing headers, cookies, the content type, and the body.

Inside the page life cycle there are a few steps in which HTTP raw data is massaged into more easily programmable containers—server controls. In ASP.NET Web Forms, these "programmable containers" are never perceived as being part of an input object model. Furthermore, many developers don't realize the existence of such a model. In ASP.NET Web Forms, the input model is just based on server controls and the view state.

Role of Server Controls

Suppose you have a web page with a couple of *TextBox* controls to capture the user name and password. When the user posts the content of the form, there will likely be a piece of code to process the request as shown here:

```
public void Button1_Click(Object sender, EventArgs e)
{
   // You're about to perform requested action using input data.
   CheckUserCredentials(TextBox1.Text, TextBox2.Text);
   ...
}
```

The overall idea behind the architecture of ASP.NET Web Forms is to keep the developer away from raw data. Any incoming request data is mapped to properties on server controls. When this is not possible, data is left parked in general-purpose containers such as *QueryString* or *Form*.

What would you expect from a method like the *Button1_Click* just shown? That method is the Web Forms counterpart of a controller action. Here's how to refactor the previous code to use an explicit input model:

```
public void Button1_Click(Object sender, EventArgs e)
{
   // You're actually filling in the input model of the page.
   var model = new UserCredentialsInputModel();
   model.UserName = TextBox1.Text;
   model.Password = TextBox2.Text;

   // You're about to perform the requested action using input data.
   CheckUserCredentials(model);
   ...
}
```

The ASP.NET runtime environment breaks up raw HTTP request data into control properties, thus offering a control-centric approach to request processing.

Note that in the upcoming ASP.NET Web Forms 4.5, Microsoft is going to introduce some model binding capabilities just along the lines shown a moment ago. In particular, they suggest you call a method of yours from within Button1_Click (a go-without-saying practice) and give this method any signature you need. Parameters on this signature can be decorated with attributes to instruct the runtime to try to resolve those values from QueryString, Forms or other value providers.

Role of the View State

Speaking in terms of a programming paradigm, a key distinguishing character between ASP.NET Web Forms and ASP.NET MVC is the view state. In Web Forms, the view state plays a central role and helps server controls to always be up to date. Because of the view state, as a developer you don't need to care about segments of the user interface you don't touch in a postback. Suppose you display a list of choices for the user to drill down into. When the request for details is made, in Web Forms all you need to do is display the details. The raw HTTP request, however, posted the list of choices as well as key information to find. The view state makes it unnecessary for you to deal with the list of choices.

The view state and server control build a thick abstraction layer on top of classic HTTP mechanics, and they make you think in terms of page sequences rather than successive requests. This is neither wrong nor right; it is just the paradigm behind Web Forms. In Web Forms, there's no need for clearly defining an input model. If you do that, it's only because you want to keep your code cleaner and more readable.

Input Processing in ASP.NET MVC

In Chapter 1, you saw that a controller method can access input data through *Request* collections—such as *QueryString*, *Headers*, or *Form*—or value providers. Although it's functional, this approach is not ideal from a readability and maintenance perspective. You need an ad hoc model that exposes data to controllers.

Role of Model Binders

The input model has one main trait. It models any data that comes your way through an HTTP request into manageable and expressive classes. As a developer, you're largely responsible for designing these classes. What about mapping request data onto properties?

ASP.NET MVC provides an automatic binding layer that uses a built-in set of rules for mapping request data to properties from any value providers. The logic of the binding layer can be customized to a large extent, thus adding unprecedented heights of flexibility as far as the processing of input data is concerned.

Flavors of a Model

The ASP.NET MVC default project template offers just one Models folder, thus implicitly pushing the idea that "model" is just one thing—the model of the data the application is supposed to use. Generally speaking, this is a rather simplistic view, though it's effective in very simple sites.

If you look deeper into things, you can recognize three different types of "models" in ASP.NET MVC, as illustrated in Figure 3-1.

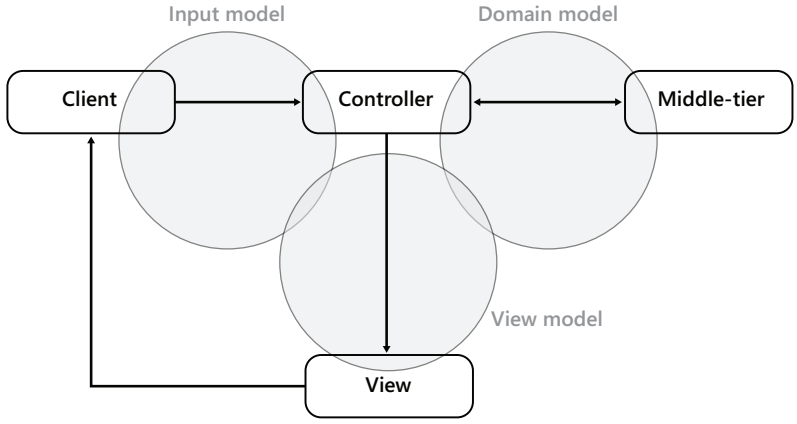

FIGURE 3-1 Types of models potentially involved in an ASP.NET MVC application.

The *input model* provides the representation of the data being posted to the controller. The *view model* provides the representation of the data being worked on in the view. Finally, the *domain model* is the representation of the domain-specific entities operating in the middle tier.

In this book, I'm not specifically covering the domain model because it results from the application of patterns and practices that would require a book of its own. I'll briefly touch on these topics in Chapter 7, "Design Considerations for ASP.NET MVC Controllers." You might want to check out the book I wrote with Andrea Saltarello called *Microsoft .NET: Architecting Applications for the Enterprise* (Microsoft Press, 2008) to read more about layered solutions. I covered the view model in Chapter 2 and will be discussing the input model in this chapter.

Note that the three models are not neatly separated, which Figure 3-1 shows to some extent. You might find overlap between the models. This means that classes in the domain model might be used

in the view, and classes posted from the client might be used in the view. The final structure and diagram of classes is up to you.

Model Binding

Model binding is the process of binding values posted over an HTTP request to the parameters used by the controller's methods. Let's find out more about the underlying infrastructure, mechanics, and components involved.

Model-Binding Infrastructure

The model-binding logic is encapsulated in a specific *model-binder* class. The binder works under the control of the action invoker and helps to figure out the parameters to pass to the selected controller method.

Analyzing the Method's Signature

As you saw in Chapter 1, each and every request passed to ASP.NET MVC is resolved in terms of a controller name and an action name. Armed with these two pieces of data, the action invoker—a native component of the ASP.NET MVC runtime shell—kicks in to actually serve the request. First, the invoker expands the controller name to a class name and resolves the action name to a method name on the controller class. If something goes wrong, an exception is thrown.

Next, the invoker attempts to collect all values required to make the method call. In doing so, it looks at the method's signature and attempts to find an input value for each parameter in the signature.

Getting the Binder for the Type

The action invoker knows the formal name and declared type of each parameter. (This information is obtained via reflection.) The action invoker also has access to the request context and to any data uploaded with the HTTP request—the query string, the form data, route parameters, cookies, headers, files, and so forth.

For each parameter, the invoker obtains a model-binder object. The model binder is a component that knows how to find values of a given type from the request context. The model binder applies its own algorithm—which includes the parameter name, parameter type, and request context available—and returns a value of the specified type. The details of the algorithm belong to the implementation of the model binder being used for the type.

ASP.NET MVC uses a built-in binder object that corresponds to the *DefaultModelBinder* class. The model binder is a class that implements the *IModelBinder* interface:

```
public interface IModelBinder
{
    Object BindModel(ControllerContext controllerContext, ModelBindingContext bindingContext);
}
```

Let's first explore the capabilities of the default binder and then see what it takes to write custom binders for specific types later.

The Default Model Binder

The default model binder uses convention-based logic to match the names of posted values to parameter names in the controller's method. The *DefaultModelBinder* class knows how to deal with primitive and complex types, as well as collections and dictionaries. In light of this, the default binder works just fine most of the time.

Note If the default binder supports primitive and complex types and the collections thereof, will you ever feel the need to use something other than the default binder? You will hardly ever feel the need to replace the default binder with another general-purpose binder. However, the default binder can't apply your custom logic to massage request data into the properties of a given type. As you'll see later, a custom binder is helpful when the values being posted with the request don't exactly match the properties of the type you want the controller to use. In this case, a custom binder makes sense and helps keep the controller's code lean and mean.

Binding Primitive Types

Admittedly, it sounds a bit magical at first, but there's no actual wizardry behind model binding. The key fact about model binding is that it lets you focus exclusively on the data you want the controller method to receive. You completely ignore the details of how you retrieve that data, whether it comes from the query string or the route.

Let's suppose you need a controller method to repeat a given string a given number of times. Here's what you do:

```
public class BindingController : Controller
{
   public ActionResult Repeat(String text, Int32 number)
   {
      var model = new RepeatViewModel {Number = number, Text = text};
      return View(model);
   }
}
```

Designed in this way, the controller is highly testable and completely decoupled from the ASP.NET runtime environment. There's no need for you to access the *Request* object or the *Cookies* collection directly.

Where do the values for *text* and *number* come from? And which component is actually reading them into text and number parameters?

The actual values are read from the request context, and the default model-binder object does the trick. In particular, the default binder attempts to match formal parameter names (*text* and *number* in the example) to named values posted with the request. In other words, if the request carries a form field, a query string field, or a route parameter named *text*, the carried value is automatically bound to the *text* parameter. The mapping occurs successfully as long as the parameter type and actual value are compatible. If a conversion cannot be performed, an argument exception is thrown. The next URL works just fine:

```
http://server/binding/repeat?text=Dino&number=2
```

Conversely, the following URL causes an exception:

```
http://server/binding/repeat?text=Dino&number=true
```

The query string field *text* contains *Dino*, and the mapping to the *String* parameter *text* on the method *Repeat* takes place successfully. The query string field *number*, on the other hand, contains *true*, which can't be successfully mapped to an Int32 parameter. The model binder returns a parameters dictionary where the entry for *number* contains *null*. Because the parameter type is Int32—that is, a non-nullable type—the invoker throws an argument exception.

Dealing with Optional Values

Note that an argument exception that occurs because invalid values are being passed is not detected at the controller level. The exception is fired before the execution flow reaches the controller. This means that you won't be able to catch it with *try/catch* blocks.

If the default model binder can't find a posted value that matches a required method parameter, it places a null value in the parameter dictionary returned to the action invoker. Again, if a value of null is not acceptable for the parameter type, an argument exception is thrown before the controller method is even called.

What if a method parameter has to be considered optional?

A possible approach entails changing the parameter type to a nullable type, as shown here:

```
public ActionResult Repeat(String text, Nullable<Int32> number)
{
    var model = new RepeatViewModel {Number = number.GetValueOrDefault(), Text = text};
    return View(model);
}
```

Another approach consists of using a default value for the parameter:

```
public ActionResult Repeat(String text, Int32 number=4)
{
    var model = new RepeatViewModel {Number = number, Text = text};
    return View(model);
}
```

Any decisions about the controller method's signature are up to you. In general, you might want to use types that are very close to the real data being uploaded with the request. Using parameters of type *Object*, for example, will save you from argument exceptions, but it will make it hard to write clean code to process the input data.

The default binder can map all primitive types, such as *String*, integers, *Double*, *Decimal*, *Boolean*, *DateTime*, and related collections. To express a Boolean type in a URL, you resort to the *true* or *false* strings. These strings are parsed using .NET native Boolean parsing functions, which recognize *true* and *false* strings in a case-insensitive manner. If you use strings such as *yes/no* to mean a *Boolean*, the default binder won't understand your intentions and places a null value in the parameter dictionary, which might cause an argument exception.

Value Providers and Precedence

The default model binder uses all the registered value providers to find a match between posted values and method parameters. By default, value providers cover the collections listed in Table 3-1.

TABLE 3-1 Request collections for which a default value provider exists

Collection	Description
Form	Contains values posted from an HTML form, if any.
RouteData	Contains values excerpted from the URL route.
QueryString	Contains values specified as the URL's query string.
Files	A value is the entire content of an uploaded file, if any.

Table 3-1 lists request collections being considered in the exact order in which they are processed by the default binder. Suppose you have the following route:

```
routes.MapRoute(
    "Test",
    "{controller}/{action}/test/{number}",
    new { controller = "Binding", action = "RepeatWithPrecedence", number = 5 }
);
```

As you can see, the route has a parameter named *number*. Now consider this URL:

```
/Binding/RepeatWithPrecedence/test/10?text=Dino&number=2
```

The request uploads two values that are good candidates to set the value of the *number* parameter in the *RepeatWithPrecedence* method. The first value is 10 and is the value of a route parameter named *number*. The second value is 2 and is the value of the QueryString element named *number*. The method itself provides a default value for the *number* parameter:

```
public ActionResult RepeatWithPrecedence(String text, Int32 number=20)
{
    ...
}
```

Which value is actually picked up? As Table 3-1 suggests, the value that actually gets passed to the method is 10—the value read from the route data collection.

Binding Complex Types

There's no limitation on the number of parameters you can list on a method's signature. However, a container class is often better than a long list of individual parameters. For the default model binder, the result is nearly the same whether you list a sequence of parameters or just one parameter of a complex type. Both scenarios are fully supported. Here's an example:

```
public class ComplexController : Controller
{
   public ActionResult Repeat(RepeatText inputModel)
   {
      var model = new RepeatViewModel
                      {
                        Title = "Repeating text",
                        Text = inputModel.Text,
                        Number = inputModel.Number
                      };
     return View(model);
   }
}
```

The controller method receives an object of type *RepeatText*. The class is a plain data-transfer object defined as follows:

```
public class RepeatText
{
    public String Text { get; set; }
    public Int32 Number { get; set; }
}
```

As you can see, the class just contains members for the same values you passed as individual parameters in the previous example. The model binder works with this complex type as well as it did with single values.

For each public property in the declared type—*RepeatText* in this case—the model binder looks for posted values whose key names match the property name. The match is case insensitive. Here's a sample URL that works with the *RepeatText* parameter type:

```
http://server/Complex/Repeat?text=Dino&number=5
```

Figure 3-2 shows the output the URL might generate.

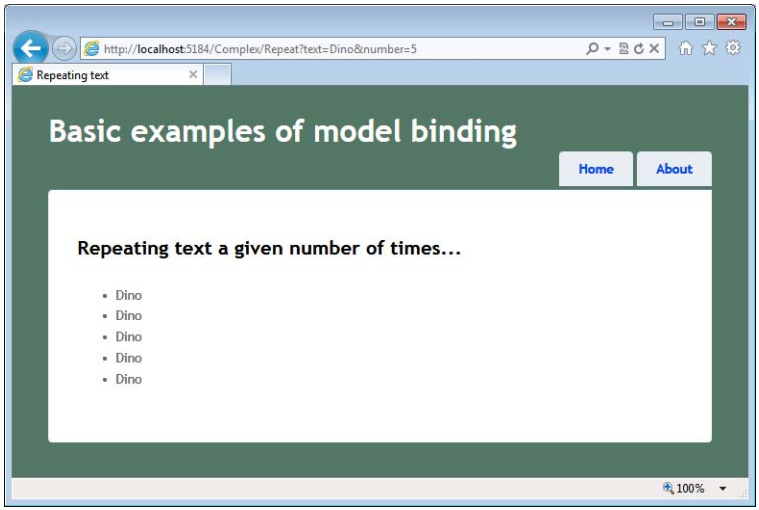

FIGURE 3-2 Repeating text with values extracted from a complex type.

Binding Collections

What if the argument that a controller method expects is a collection? For example, can you bind the content of a posted form to an *IList<T>* parameter? The *DefaultModelBinder* class makes it possible, but doing so requires a bit of contrivance of your own. Have a look at Figure 3-3.

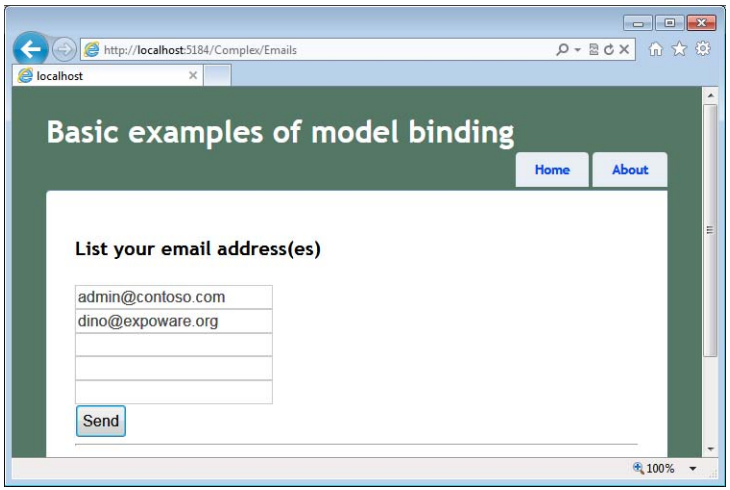

FIGURE 3-3 The page will post an array of strings.

When the user hits the Send button, the form submits its content. Specifically, it sends out the content of the various text boxes. If the text boxes have different IDs, the posted content takes the following form:

```
TextBox1=admin@contoso.com&TextBox2=&TextBox3=&TextBox4=&TextBox5=
```

In classic ASP.NET, this is the only possible way of working because you can't just assign the same ID to multiple controls. However, if you manage the HTML yourself, nothing prevents you from assigning the five text boxes in the figure the same ID. The HTML DOM, in fact, fully supports this scenario (though it is not recommended). Therefore, the following markup is entirely legal in ASP.NET MVC and produces HTML that works on all browsers:

```
@using (Html.BeginForm())
{
    <h2>List your email address(es)</h2>
    foreach(var email in Model.Emails)
    {
        <input type="text" name="emails" value="@email" />
        <br />
    }
    <input type="submit" value="Send" />
}
```

What's the expected signature of a controller method that has to process the email addresses typed in the form? Here it is:

```
public ActionResult Emails(IList<String> emails)
{
    ...
}
```

Figure 3-4 shows that an array of strings is correctly passed to the method thanks to the default binder class.

FIGURE 3-4 An array of strings has been posted.

As you'll see in greater detail in the next chapter, when you work with HTML forms you likely need to have a pair of methods—one to handle the display of the view (the verb GET), and one to handle the scenario in which data is posted to the view. The *HttpPost* and *HttpGet* attributes allow you to mark which scenario a given method is handling for the same action name. Here's the full implementation of the example, which uses two distinct methods to handle GET and POST scenarios:

```
[ActionName("Emails")]
[HttpGet]
public ActionResult EmailForGet(IList<String> emails)
{
    // Input parameters
    var defaultEmails = new[] { "admin@contoso.com", "", "", "", "" };
```

```
    if (emails == null)
        emails = defaultEmails;
    if (emails.Count == 0)
        emails = defaultEmails;
    var model = new EmailsViewModel {Emails = emails};
    return View(model);
}

[ActionName("Emails")]
[HttpPost]
public ActionResult EmailForPost(IList<String> emails)
{
    var defaultEmails = new[] { "admin@contoso.com", "", "", "", "" };
    var model = new EmailsViewModel { Emails = defaultEmails, RegisteredEmails = emails };
    return View(model);
}
```

Here's the full Razor markup for the view you see rendered in Figure 3-5:

```
@using BasicInput.ViewModels.Complex;
@model EmailsViewModel

@section title{
    @Model.Title
}

@using (Html.BeginForm())
{
    <h2>List your email address(es)</h2>
    foreach(var email in Model.Emails)
    {
        <input type="text" name="emails" value="@email" />
        <br />
    }
    <input type="submit" value="Send" />
}

    <hr />
    <h2>Emails submitted</h2>
    <ul>
    @foreach (var email in Model.RegisteredEmails)
    {
        if (String.IsNullOrWhiteSpace(email))
        {
            continue;
        }

        <li>@email</li>
    }
    </ul>
```

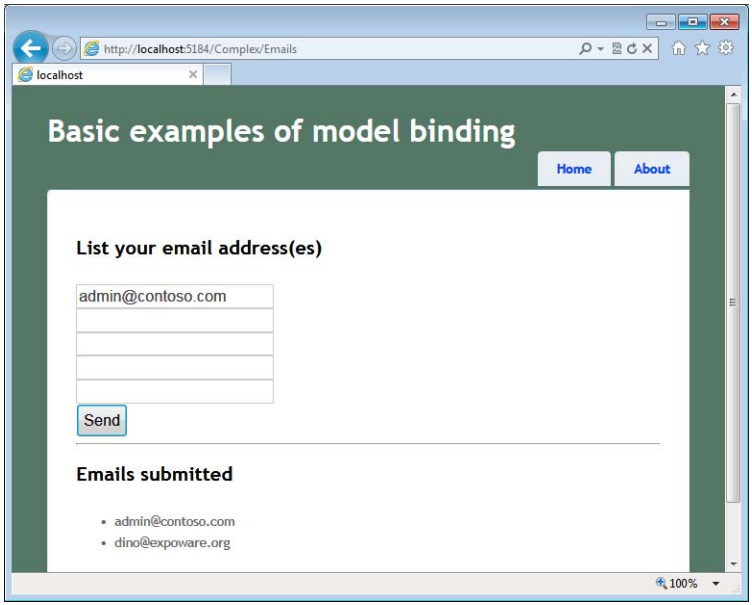

FIGURE 3-5 The page rendered after a POST.

In the end, to ensure that a collection of values is passed to a controller method, you need to ensure that elements with the same ID are emitted to the response stream. The ID, then, has to match to the controller method's signature according to the normal rules of the binder.

Note As you might have figured out already, the default model binder does a lot of work for you. However, it requires that you use fixed IDs in the HTML forms you create. That's the way in which the component works, and it's probably the only way to make it work in a rather generic way for a variety of classes.

Binding Collections of Complex Types

The default binder can also handle situations in which the collection contains complex types, even nested:

```
[ActionName("Countries")]
[HttpPost]
public ActionResult ListCountriesForPost(IList<Country> countries)
{
   ...
}
```

As an example, consider the following definition for type *Country*:

```
public class Country
{
    public Country()
```

```
    {
        Details = new CountryInfo();
    }
    public String Name { get; set; }
    public CountryInfo Details { get; set; }
}
public class CountryInfo
{
    public String Capital { get; set; }
    public String Continent { get; set; }
}
```

For model binding to occur successfully, all you really need to do is use a progressive index on the IDs in the markup. The resulting pattern is *prefix[index].Property*, where *prefix* matches the name of the formal parameter in the controller method's signature:

```
@using (Html.BeginForm())
{
    <h2>Select your favorite countries</h2>
    var index = 0;
    foreach (var country in Model.CountryList)
    {
        <fieldset>
        <div>
            <b>Name</b><br />
            <input type="text"
                   name="countries[@index].Name"
                   value="@country.Name" /><br />
            <b>Capital</b><br />
            <input type="text"
                   name="countries[@index].Details.Capital"
                   value="@country.Details.Capital" /><br />
            <b>Continent</b><br />
            @{
                var id = String.Format("countries[{0}].Details.Continent", index++);
             }
             @Html.TextBox(id, country.Details.Continent)
             <br />
        </div>
    </fieldset>
    }
    <input type="submit" value="Send" />
}
```

The index is numeric, 0-based, and progressive. In this example, I'm building user interface blocks for each specified default country. If you have a fixed number of user interface blocks to render, you can use static indexes:

```
<input type="text"
       name="countries[0].Name"
       value="@country.Name" />

<input type="text"
       name="countries[1].Name"
       value="@country.Name" />
```

Note that holes in the series (for example, 0 and then 2) stop the parsing, and all you get back is the sequence of data types from 0 to the hole.

The posting of data works fine as well. The POST method on the controller class will just receive the same hierarchy of data, as Figure 3-6 shows.

FIGURE 3-6 Complex and nested types posted to the method.

Rest assured that if you're having trouble mapping posted values to your expected hierarchy of types, it might be wise to consider a custom model binder.

Binding Content from Uploaded Files

Table 3-1 indicates that uploaded files can also be subject to model binding. The default binder does the binding by matching the name of the input file element used to upload with the name of a parameter. The parameter (or the property on a parameter type), however, must be declared of type *HttpPostedFileBase*:

```
public class UserData
{
    public String Name { get; set; }
    public String Email { get; set; }
    public HttpPostedFileBase Picture { get; set; }
}
```

The following code shows a possible implementation of a controller action that saves the uploaded file somewhere on the server machine:

```
public ActionResult Add(UserData inputModel)
{
    var destinationFolder = Server.MapPath("/Users");
    var postedFile = inputModel.Picture;
    if (postedFile.ContentLength > 0)
    {
        var fileName = Path.GetFileName(postedFile.FileName);
        var path = Path.Combine(destinationFolder, fileName);
        postedFile.SaveAs(path);
    }

    return View();
}
```

By default, any ASP.NET request can't be longer than 4 MB. This amount should include any uploads, headers, body, and whatever is being transmitted. The value is configurable at various levels. You do that through the *maxRequestLength* entry in the *httpRuntime* section of the *web.config* file:

```
<system.web>
   <httpRuntime maxRequestLength="6000" />
</system.web>
```

Obviously, the larger a request is, the more room you potentially leave for hackers to `prepare attacks on your site. Note also that in a hosting scenario your application-level settings might be ignored if the hoster has set a different limit at the domain level and locked down the *maxRequestLength* property at lower levels.

What about multiple file uploads? As long as the overall size of all uploads is compatible with the current maximum length of a request, you are allowed to upload multiple files within a single request. However, consider that web browsers just don't know how to upload multiple files. All a web browser can do is upload a single file, and only if you reference it through an input element of type file. To upload multiple files, you can resort to some client-side ad hoc component or place multiple INPUT elements in the form. If multiple INPUT elements are used, and properly named, a class like the one shown here will bind them all:

```
public class UserData
{
    public String Name { get; set; }
    public String Email { get; set; }
    public HttpPostedFileBase Picture { get; set; }
    public IList<HttpPostedFileBase> AlternatePictures { get; set; }
}
```

The class represents the data posted for a new user with a default picture and a list of alternate pictures. Here is the markup for the alternate pictures:

```
<input type="file" id="AlternatePictures[0]" name="AlternatePictures[0]" />
<input type="file" id="AlternatePictures[1]" name="AlternatePictures[1]" />
```

Note Creating files on the web server is not usually an operation that can be accomplished by relying on the default permission set. Any ASP.NET application runs under the account of the worker process serving the application pool the application belongs to. Under normal circumstances, this account is NETWORK SERVICE, and it isn't granted the permission to create new files. This means that to save files you must change the account behind the ASP.NET application or elevate the privileges of the default account.

For years, the identity of the application pool has been a fixed identity—the aforementioned NETWORKSERVICE account, which is a relatively low-privileged, built-in identity in Microsoft Windows. Originally welcomed as an excellent security measure, the practice of using a single account for a potentially high number of concurrently running services in the end created more problems than it helped to solve.

In a nutshell, services running under the same account could tamper with each other. For this reason, in Microsoft Internet Information Services 7.5, by default worker processes run under unique identities that are automatically and transparently created for each newly created application pool. The underlying technology is known as Virtual Accounts and is currently supported by Windows Server 2008 R2 and Windows 7. For more information, have a look at *http://technet.microsoft.com/en-us/library/dd548356(WS.10).aspx*.

Customizable Aspects of the Default Binder

Automatic binding stems from a convention-over-configuration approach. Conventions, though, sometimes harbor bad surprises. If, for some reason, you lose control over the posted data (for example, in the case of data that has been tampered with), it can result in undesired binding—any posted key/value pair will, in fact, be bound. For this reason, you might want to consider using the *Bind* attribute to customize some aspects of the binding process.

The *Bind* Attribute

The *Bind* attribute comes with three properties, which are described in Table 3-2.

TABLE 3-2 Properties for the *BindAttribute* class

Property	Description
Prefix	String property. It indicates the prefix that must be found in the name of the posted value for the binder to resolve it. The default value is the empty string.
Exclude	Gets or sets a comma-delimited list of property names for which binding is not allowed.
Include	Gets or sets a comma-delimited list of property names for which binding is permitted.

You apply the *Bind* attribute to parameters on a method signature.

Creating Whitelists of Properties

As mentioned, automatic model binding is potentially dangerous when you have complex types. In such cases, in fact, the default binder attempts to populate all public properties on the complex types for which it finds a match in the posted values. This might end up filling the server type with unexpected data, especially in the case of request tampering. To avoid that, you can use the *Include* property on the *Bind* attribute to create a whitelist of acceptable properties:

```
public ActionResult RepeatOnlyText([Bind(Include = "text")]RepeatText inputModel)
{
   ...
}
```

The binding on the *RepeatText* type will be limited to the listed properties (in the example, only *Text*). Any other property is not bound and takes whatever default value the implementation of *RepeatText* assigned to it. Multiple properties are separated by a comma.

Creating Blacklists of Properties

The *Exclude* attribute employs the opposite logic: it lists properties that must be excluded from binding. All properties except those explicitly listed will be bound:

```
public ActionResult RepeatOnlyText([Bind(Exclude = "number")]RepeatText inputModel)
{
   ...
}
```

You can use *Include* and *Exclude* in the same attribute if dong so allows you to better define the set of properties to bind. If, for instance, both attributes refer to the same property, *Exclude* will win.

Using a Prefix

The default model binder forces you to give your request parameters (for example, form and query string fields) given names that match formal parameters on target action methods. The *Prefix* attribute allows you to change this convention. By setting the *Prefix* attribute, you instruct the model binder to match request parameters against the prefix rather than against the formal parameter name. All in all, *alias* would have been a much better name for this attribute. Consider the following example:

```
[HttpPost]
[ActionName("Emails")]
public ActionResult EmailForPost([Bind(Prefix = "foo")]IList<String> emails)
{
   ...
}
```

For the emails parameter to be successfully filled, you need to have posted a field whose name is *foo*, not *emails*. The *Prefix* attribute makes particular sense on POST methods.

Finally, note that if a prefix is specified, it becomes mandatory and fields whose name is not prefixed are not bound.

Advanced Model Binding

So far, we've examined the behavior of the default model binder. The default binder does excellent work, but it is a general-purpose tool designed to work with most possible types in a way that is not specific to any of them. The *Bind* attribute gives you some more control over the binding process, but there are some reasonable limitations to its abilities. If you want to achieve total control over the binding process, all you do is create a custom binder for a specific type.

Custom Type Binders

There's just one primary reason you should be willing to create a custom binder: the default binder is limited to taking into account only a one-to-one correspondence between posted values and properties on the model.

Sometimes, though, the target model has a different granularity than the one expressed by form fields. The canonical example is when you employ multiple input fields to let users enter content for a single property—for example, distinct input fields for day, month, and year that then map to a single *DateTime* value.

Customizing the Default Binder

To create a custom binder from scratch, you implement the *IModelBinder* interface. Implementing the interface is recommended if you need total control over the binding process. If, say, all you need to do is keep the default behavior and simply force the binder to use a nondefault constructor for a given type, inheriting from *DefaultModelBinder* is the best approach. Here's the schema to follow:

```
public RepeatTextModelBinder : DefaultModelBinder
{
    protected override object CreateModel(
         ControllerContext controllerContext,
         ModelBindingContext bindingContext,
         Type modelType)
    {
         ...
         return new RepeatText( ... );
    }
}
```

Another common scenario for simply overriding the default binder is when all you want is the ability to validate against a specific type. In this case, you override *OnModelUpdated* and insert your own validation logic, as shown here:

```
protected override void OnModelUpdated(ControllerContext controllerContext,
           ModelBindingContext bindingContext)
{
   var obj = bindingContext.Model as RepeatText;
   if (obj == null)
      return;

   // Apply validation logic here for the whole model
   if (String.IsNullOrEmpty(obj.Text))
   {
      bindingContext.ModelState.AddModelError("Text", ...);
   }
   ...
}
```

You override *OnModelUpdated* if you prefer to keep in a single place all validations for any properties. You resort to *OnPropertyValidating* if you prefer to validate properties individually.

Important When binding occurs on a complex type, the default binder simply copies matching values into properties. You can't do much to refuse some values if they put the instance of the complex type in an invalid state.

A custom binder could integrate some logic to check the values being assigned to properties and signal an error to the controller method or degrade gracefully by replacing the invalid value with a default one. Although it's possible to use this approach, it's not commonly used because there are more powerful options in ASP.NET MVC that you can use to deal with data validation across an input form. And that is exactly the topic I'll address in the next chapter.

Implementing a Model Binder from Scratch

The *IModelBinder* interface is defined as follows:

```
public interface IModelBinder
{
    Object BindModel(ControllerContext controllerContext,
                     ModelBindingContext bindingContext);
}
```

Here's the skeleton of a custom binder that directly implements the *IModelBinder* interface. The model binder is written for a specific type—in this case, *MyComplexType*:

```
public class MyComplexTypeModelBinder : IModelBinder
{
  public Object BindModel(ControllerContext controllerContext,
                          ModelBindingContext bindingContext)
  {
     if (bindingContext == null)
         throw new ArgumentNullException("bindingContext");

     // Create the model instance (using the ctor you like best)
     var obj = new MyComplexType();

     // Set properties reading values from registered value providers
     obj.SomeProperty = FromPostedData<string>(bindingContext, "SomeProperty");
     ...
     return obj;
}

// Helper routine
private T FromPostedData<T>(ModelBindingContext context, String key)
{
   // Get the value from any of the input collections
   ValueProviderResult result;
   context.ValueProvider.TryGetValue(key, out result);

   // Set the state of the model property resulting from value
   context.ModelState.SetModelValue(key, result);
```

```
    // Return the value converted (if possible) to the target type
    return (T) result.ConvertTo(typeof(T));
}
```

The structure of *BindModel* is straightforward. You first create a new instance of the type of interest. In doing so, you can use the constructor (or factory) you like best and perform any sort of custom initialization that is required by the context. Next, you simply populate properties of the freshly created instance with values read or inferred from posted data. In the preceding code snippet, I assume you simply replicate the behavior of the default provider and read values from registered value providers based on a property name match. Obviously, this is just the place where you might want to add your own logic to interpret and massage what's being posted by the request.

Note that when writing a model binder, you are in no way restricted to getting information for the model only from the posted data—which represents only the most common scenario. You can grab information from anywhere—for example, from the ASP.NET cache and session state—parse it, and store it in the model.

Note ASP.NET MVC comes with two built-in binders beyond the default one. These additional binders are automatically selected for use when posted data is a Base64 stream (*ByteArrayModelBinder* type) and when the content of a file is being uploaded (*HttpPostedFileBaseModelBinder* type).

Registering a Custom Binder

You can associate a model binder with its target type globally or locally. In the former case, any occurrence of model binding for the type will be resolved through the registered custom binder. In the latter case, you apply the binding to just one occurrence of one parameter in a controller method.

Global association takes place in the *global.asax* file as follows:

```
void Application_Start()
{
    ...
    ModelBinders.Binders[typeof(MyComplexTypeModelBinder)] =
                          new MyCustomTypeModelBinder();
}
```

Local association requires the following syntax:

```
public ActionResult RepeatText(
           [ModelBinder(typeof(MyComplexTypeModelBinder))] MyComplexType info)
{
   ...
}
```

Local binders always take precedence over globally defined binders.

As you can tell clearly from the preceding code within *Application_Start*, you can have multiple binders registered. You can also override the default binder if required:

```
ModelBinders.Binders.DefaultBinder = new MyNewDefaultBinder();
```

Modifying the default binder, however, can have a large impact on the behavior of the application and should therefore be a very thoughtful choice.

A Sample *DateTime* Model Binder

In input forms, it is quite common to have users enter a date. You can sometimes use a jQuery UI to let users pick dates from a graphical calendar. The selection is translated to a string and saved to a text box. When the form posts back, the date string is uploaded and the default binder attempts to parse it to a *DateTime* object.

In other situations, you might decide to split the date into three distinct text boxes—for day, month, and year. These pieces are uploaded as distinct values in the request. The result is that the default binder can manage them only separately—the burden of creating a valid *DateTime* object out of day, month, and year values is up to the controller. With a custom default binder, you can take this code out of the controller and still enjoy the pleasure of having the following signature for a controller method:

```
public ActionResult MakeReservation(DateTime theDate)
```

Let's see how to arrange a more realistic example of a model binder.

The Displayed Data

The sample view we consider next shows three text boxes for the items that make up a date and a submit button. You enter a date, and the system calculates how many days have elapsed since or how many days you have to wait for the specified day to arrive. Here's the Razor markup:

```
@model DateEditorResponseViewModel
@section title{
    @Model.Title
}

@using (Html.BeginForm())
{
<fieldset>
    <legend>Date Editor</legend>
    <div style="margin:20">
        <table><tr>
        <td>@DateHelpers.InputDate("theDate", Model.DefaultDate)</td>
        <td><input type="submit" value="Find out more" /></td>
        </tr></table>
    </div>
</fieldset>
}
<hr />
@DateHelpers.Distance(Model.TimeToToday)
```

As you can see, I'm using a couple of custom helpers to better encapsulate the rendering of some view code. Here's how you render the date elements:

```
@helper InputDate(String name, DateTime? theDate)
{
    String day="", month="", year="";
    if(theDate.HasValue)
    {
        day = theDate.Value.Day.ToString();
        month = theDate.Value.Month.ToString();
        year = theDate.Value.Year.ToString();
    }
    <table cellpadding="0">
        <thead>
            <th>DD</th>
            <th>MM</th>
            <th>YYYY</th>
        </thead>
        <tr>
            <td><input type="text" name="@(name + ".day")"
                       value="@day" style="width:30px" /></td>
            <td><input type="text" name="@(name + ".month")"
                       value="@month" style="width:30px"></td>
            <td><input type="text" name="@(name + ".year")"
                       value="@year" style="width:40px" /></td>
        </tr>
    </table>
}
```

Figure 3-7 shows the output.

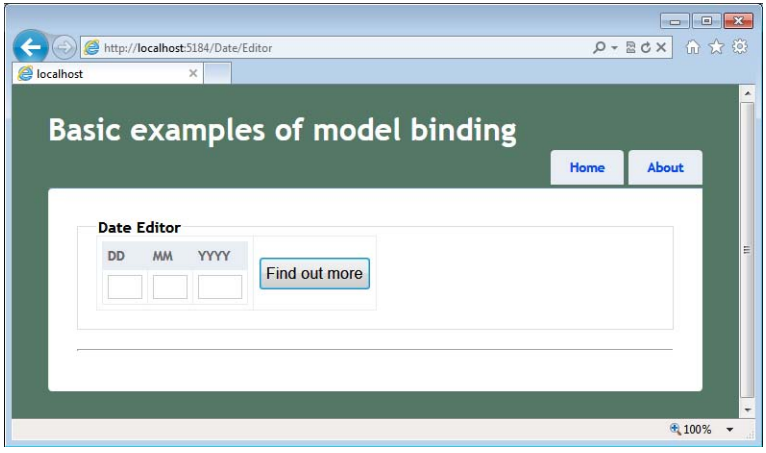

FIGURE 3-7 A sample view that splits date input text into day-month-year elements.

The Controller Method

The view in Figure 3-7 is served and processed by the following controller methods:

```
public class DateController : Controller
{
    [HttpGet]
    [ActionName("Editor")]
    public ActionResult EditorForGet()
    {
        var model = new DateEditorViewModel();
        return View(model);
    }

    [HttpPost]
    [ActionName("Editor")]
    public ActionResult EditorForPost(DateTime theDate)
    {
        var model = new DateEditorViewModel();
        if (theDate != default(DateTime))
        {
            model.DefaultDate = theDate;
            model.TimeToToday = DateTime.Today.Subtract(theDate);
        }
        return View(model);
    }
}
```

After the date is posted back, the controller action calculates the difference with the current day and stores the results back in the view model using a *TimeSpan* object. Here's the view model object:

```
public class DateEditorViewModel : ViewModelBase
{
    public DateEditorViewModel()
    {
        DefaultDate = null;
        TimeToToday = null;
    }
    public DateTime? DefaultDate { get; set; }
    public TimeSpan? TimeToToday { get; set; }
}
```

What remains to be examined is the code that performs the trick of transforming three distinct values uploaded independently into one *DateTime* object.

Creating the *DateTime* Binder

The structure of the *DateTimeModelBinder* object is not much different from the skeleton I described earlier. It is just tailor-made for the *DateTime* type:

```
public class DateModelBinder : IModelBinder
{
    public Object BindModel(ControllerContext controllerContext, ModelBindingContext
bindingContext)
    {
```

```
        if (bindingContext == null)
        {
            throw new ArgumentNullException("bindingContext");
        }

        // This will return a DateTime object
        var theDate = default(DateTime);

        // Try to read from posted data. xxx.Day|xxx.Month|xxx.Year is assumed.
        var day = FromPostedData<int>(bindingContext, "Day");
        var month = FromPostedData<int>(bindingContext, "Month");
        var year = FromPostedData<int>(bindingContext, "Year");

        return CreateDateOrDefault(year, month, day, theDate);
    }

    // Helper routines
    private static T FromPostedData<T>(ModelBindingContext context, String id)
    {
        if (String.IsNullOrEmpty(id))
            return default(T);

        // Get the value from any of the input collections
        var key = String.Format("{0}.{1}", context.ModelName, id);
        var result = context.ValueProvider.GetValue(key);
        if (result == null && context.FallbackToEmptyPrefix)
        {
            // Try without prefix
            result = context.ValueProvider.GetValue(id);
            if (result == null)
                return default(T);
        }

        // Set the state of the model property resulting from value
        context.ModelState.SetModelValue(id, result);

        // Return the value converted (if possible) to the target type
        T valueToReturn = default(T);
        try
        {
            valueToReturn = (T)result.ConvertTo(typeof(T));
        }
        catch
        {
        }

       return valueToReturn;
    }

    private DateTime CreateDateOrDefault(Int32 year, Int32 month, Int32 day, DateTime?
defaultDate)
    {
        var theDate = defaultDate ?? default(DateTime);
        try
        {
            theDate = new DateTime(year, month, day);
        }
```

```
        catch (ArgumentOutOfRangeException e)
        {
        }

        return theDate;
    }
}
```

The binder makes some assumptions about the naming convention of the three input elements. In particular, it requires that those elements be named *day*, *month*, and *year*—possibly prefixed by the model name. It is the support for the prefix that makes it possible to have multiple date input boxes in the same view without conflicts.

With this custom binder available, all you need to do is register it either globally or locally. Here's how to make it work with just a specific controller method:

```
[HttpPost]
[ActionName("Editor")]
public ActionResult EditorForPost([ModelBinder(typeof(DateModelBinder))] DateTime theDate)
{
   ...
}
```

Figure 3-8 shows the final page in action.

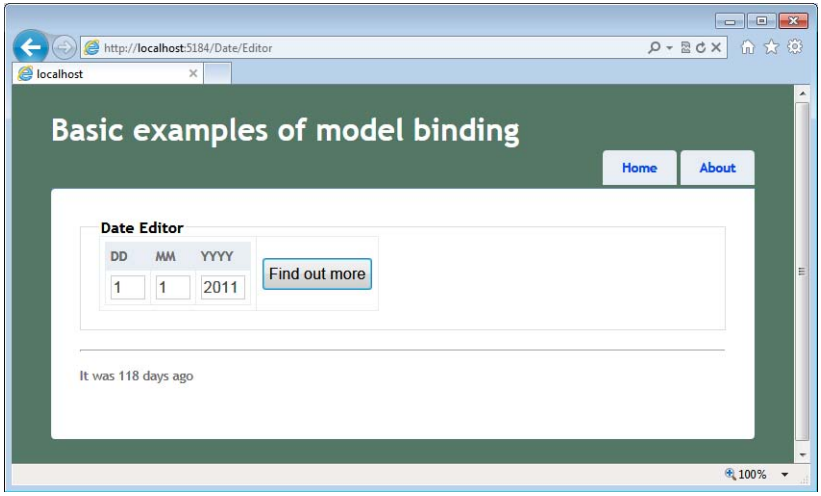

FIGURE 3-8 Working with dates using a custom type binder.

Summary

In ASP.NET MVC as well as in ASP.NET Web Forms, posted data arrives within an HTTP packet and is mapped to various collections on the *Request* object. To offer a nice service to developers, ASP.NET then attempts to expose that content in a more usable way.

In ASP.NET Web Forms, the content is parsed and passed on to server controls; in ASP.NET MVC, on the other hand, it is bound to parameters of the selected controller's method. The process of binding posted values to parameters is known as *model binding* and occurs through a registered model-binder class. Model binders provide you with complete control over the deserialization of form-posted values into simple and complex types.

In functional terms, the use of the default binder is transparent to developers—no action is required on your end—and it keeps the controller code clean. By using model binders, including custom binders, you also keep your controller's code free of dependencies on ASP.NET intrinsic objects, and thus make it cleaner and more testable.

The use of model binders is strictly related to posting and input forms. In the next chapter, we'll discuss aspects of input forms, input modeling, and data validation.

CHAPTER 4

Input Forms

Whatever you can do or dream, begin it.

—Wolfgang Goethe

Classic ASP.NET bases its programming model on the assumption that state is maintained across postbacks. This is not true at all at the HTTP protocol level, but it is brilliantly simulated using the page view-state feature and a bit of work in the Web Forms page life cycle. The view state, which is so often kicked around as a bad thing, is a great contribution to establishing a stateful programming model in ASP.NET, and that programming model was one of the keys to ASP.NET's success and rapid adoption. Data entry is a scenario in which server controls really shine and in which their postback and view-state overhead save you from doing a lot of work. Server controls also give you a powerful infrastructure for input validation.

If you've grown up with Web Forms and its server controls, you might be shocked when you're transported into the ASP.NET MVC model. In ASP.NET MVC, you have the same functional capabilities as in Web Forms, only they're delivered through a different set of tools. The ASP.NET MVC framework uses a different pattern, one that is not page based and relies on a much thinner abstraction layer than Web Forms. As a result, you don't have rich native components such as server controls to quickly arrange a nice user interface where elements can retain their content across postbacks. This fact seems to cause a loss of productivity, at least for certain types of applications, such as those heavily based on data entry.

Is this really true, though?

For sure, in ASP.NET MVC you write code that is conceptually and physically closer to the metal; therefore, it takes more lines, but it gives you a lot more control over the generated HTML and actual behavior of the run-time environment. You don't have to write everything from scratch, however. You have HTML helpers to automatically create (quite) simple but (still) functional viewers and editors for any primitive or complex type. You have data annotations to declaratively set your expectations about the content of a field and its display behavior. You have model binders to serialize posted values into more comfortable objects for server-side processing. You have tools for both server and client validation. Finally, you have some good tooling to scaffold controllers and views that support most common CRUD (Create, Read, Update, Delete) scenarios.

You have the tools, and although they're certainly different than in Web Forms, they're equally effective. So you should expect not only to use different tools for data entry but also to follow a

slightly different approach. This chapter aims to show you how to grab input data through forms in ASP.NET MVC 3 and then validate and process it against a persistence layer.

General Patterns of Data Entry

Input forms revolve around two main patterns: *Edit-and-Post* and *Select-Edit-Post*. The former displays an HTML form and expects users to fill the fields and post data when they're done filling the fields. The latter pattern just extends the former by adding an extra preliminary step. The users select an item of data, place it into edit mode, play with the content, and then save changes back to the storage layer. Let's illustrate the Select-Edit-Post pattern with an example.

> **Note** I'm not explicitly covering the Edit-and-Post pattern because it's merely a simpler version of the Select-Edit-Post pattern. In the upcoming "Editing Data" and "Saving Data" sections of this chapter, you'll find a description of the Select-Edit-Post pattern that works equally well as a description of the Edit-and-Post pattern.

A Classic Select-Edit-Post Scenario

I'll illustrate the Select-Edit-Post pattern through an example that starts by letting users pick a customer from a drop-down list. Next, the record that contains information about the selected customer is rendered into an edit form, where updates can be entered and eventually validated and saved.

In this example, the domain model consists of an Entity Framework model inferred from the canonical Northwind database. Figure 4-1 shows the initial user interface of the sample application.

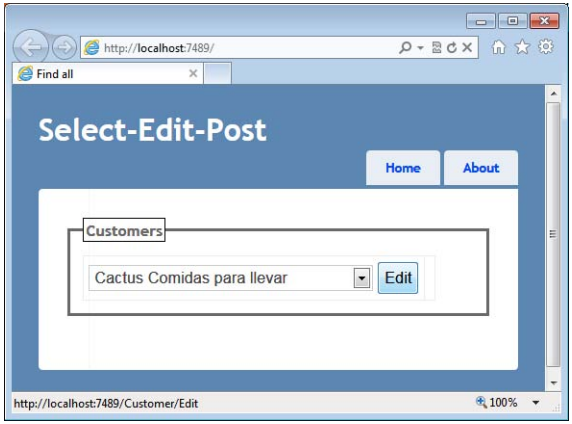

FIGURE 4-1 The initial screen of the sample application, where users begin by making a selection.

Presenting Data and Handling the Selection

The following listing shows the controller action that is used to populate the drop-down list to offer the initial screen to the user. Note that I'm designing controllers according to the Coordinator stereotype defined in the Responsibility Driven Design (RDD) methodology. (I'll talk more about RDD and MVC controllers as coordinators in Chapter 7, "Design Considerations for ASP.NET MVC Controllers." For now, suffice it to say that a *coordinator* is a component that reacts to events and delegates any further action to an external component.) A coordinator is limited to passing input and capturing output. Having controllers implemented as coordinators in ASP.NET MVC is beneficial because it decouples the layer that receives requests from the layer (or layers) that processes requests.

In the following code, the controller class receives a worker component via the constructor, and it uses the services of the component to process input data and receive a view model object to merge with the selected view template:

```
public class HomeController : Controller
{
    private readonly ICustomerWorkerServices _service;

    public HomeController() : this(new CustomerWorkerServices())
    {
    }
    public HomeController(ICustomerWorkerServices service)
    {
        _service = service;
    }

    public ActionResult Index()
    {
        var model = _service.FindAllCustomers();
        return View(model);
    }
}
```

The method *Index* obtains a view model object that contains the list of customers to display:

```
public class EditCustomerViewModelBase : ViewModelBase
{
    public IList<Customer> Customers { get; set; }
}
```

Here's the implementation of the *FindAllCustomers* method in the worker component:

```
public class CustomerWorkerServices : ICustomerWorkerServices
{
    private readonly ICustomerRepository _repository;
    public CustomerWorkerServices() : this(new CustomerRepository())
    {
    }
    public CustomerWorkerServices(ICustomerRepository repository)
    {
        _repository = repository;
    }
```

```
    public EditCustomerViewModelBase FindAllCustomers()
    {
        var model = new EditCustomerViewModelBase
                        {
                            Title="Find all",
                            Customers = _repository.FindAll()
                        };
        return model;
    }
}
```

The view that produces the interface in Figure 4-1 is shown here:

```
@{
    ViewBag.Title = Model.Title;
}
@model SEP.ViewModels.Customers.EditCustomerViewModelBase

<fieldset>
    <legend>Customers</legend>
    <table cellspacing="0" cellpadding="10">
        <tr>
            <td valign="top">
                @using(Html.BeginForm("Edit", "Customer"))
                {
                    @Html.DropDownList("customerList",
                            new SelectList(Model.Customers, "CustomerId", "CompanyName"))
                    <input type="submit" name="btnEdit" value="Edit" />
                }
            </td>
            <td valign="top">
            </td>
        </tr>
    </table>
</fieldset>
```

After the user has selected a customer from the list (by clicking a submit button), he submits a POST request for an *Edit* action on the *CustomerController* class.

Editing Data

The request for the *Edit* action moves the application into edit mode, and an editor for the selected customer is displayed. As you can see in Figure 4-2, you should also expect the successive view to retain the current drop-down list status. The following code shows a possible implementation for the *Edit* method on the *Customer* controller:

```
public ActionResult Edit([Bind(Prefix = "customerList")] String customerId)
{
    var model = _service.EditCustomer(customerId);
    return View(model);
}
```

As you saw in Chapter 3, "The Model-Binding Architecture," the *Bind* attribute instructs the default model binder to assign the value of the posted field named *customerList* to the specified parameter.

In this case, the *EditCustomer* method retrieves information about the specified customer and passes that to the view engine so that an input form can be arranged and displayed to the user:

```
public EditCustomerViewModel EditCustomer(String id)
{
    var model = new EditCustomerViewModel
                    {
                        Title = "Edit customer",
                        Customer = _repository.FindById(id),
                        Customers = _repository.FindAll()
                    };
    return model;
}
```

Why does the *EditCustomer* method also need to retrieve the list of customers?

This is a direct consequence of not having the view state around. Every aspect of the view must be recreated and repopulated each time. This is a key part of the HTTP contract and relates to the inherent HTTP stateless-ness. In ASP.NET Web Forms, most of the refilling work is done automatically by the abstraction layer of the framework through information stored in the view state. In ASP.NET MVC, it's just up to you.

Note that in the preceding code snippet I just place another call to the service layer; a more serious application would probably cache data and reload from there. The caching layer, however, could also be incorporated in the repository itself.

Here's the code for the view:

```
@{
    ViewBag.Title = Model.Title;
}
@model SEP.ViewModels.Customers.EditCustomerViewModel

<fieldset>
    <legend>Customers</legend>
    <table cellspacing="0" cellpadding="10">
        <tr>
            <td valign="top">
                @using(Html.BeginForm("Edit", "Customer"))
                {
                    @Html.DropDownList("customerList",
                        new SelectList(Model.Customers,
                                       "CustomerId",
                                       "CompanyName",
                                       Model.Customer.CustomerID))
                    <input type="submit" name="btnEdit" value="Edit" />
                }
            </td>
            <td valign="top">
                @Html.Partial("_customerEditor", Model.Customer)
            </td>
        </tr>
    </table>
</fieldset>
```

The structure of the view is nearly the same as in Figure 4-1, the only difference being the table-based editor on the right of the view. The editor (*_customerEditor.cshtml*) is created through the *Partial* HTML helper. If you look into the list of native HTML helpers, you find two apparently similar helpers: *Partial* and *RenderPartial*. What's the difference? As hinted at in Chapter 2, "ASP.NET MVC Views," *Partial* just returns a string, whereas *RenderPartial* performs the action of rendering a string. If the goal is only that of creating a view, they are nearly identical but still require a slightly different programming syntax. To call *RenderPartial*, you need the following in Razor:

```
@{ Html.RenderPartial(view) }
```

You need the following, however, if you use the ASPX view engine:

```
<% Html.RenderPartial(view); %>
```

Figure 4-2 shows the editor in action.

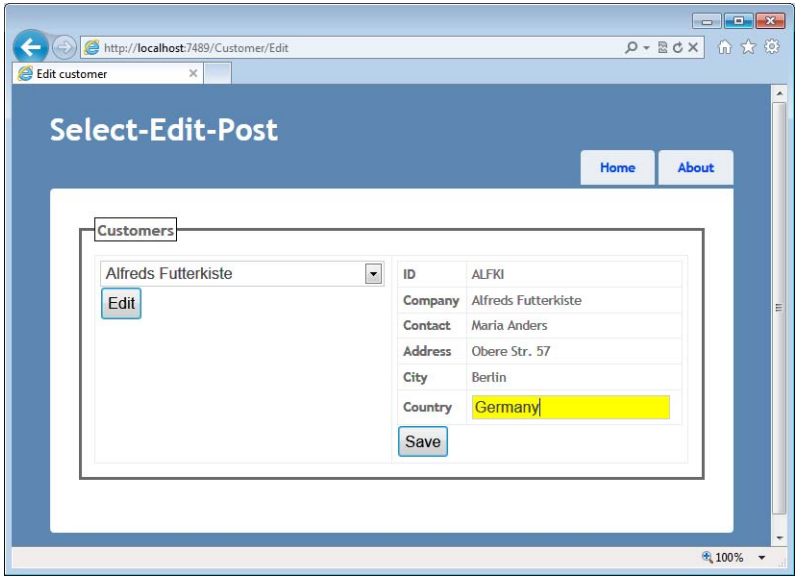

FIGURE 4-2 Users can make changes to the selected customer.

Saving Data

After the input form is displayed, the user enters any data she reckons to be valid and then presses the button that posts the content of the form to the server. Here's the markup for a typical form that posts changes. (The markup is the content of the *_customerEditor.cshtml* file.)

```
@model Northwind.Model.Customer

@using(Html.BeginForm("Update", "Customer", new { customerId = Model.CustomerID }))
{
    <table id="__tableCustomerEditAscx" rules="rows" frame="hsides">
        <tr>
```

```
            <td><b>ID</b></td>
            <td>@Model.CustomerID</td>
        </tr>
            ...
        <tr>
            <td><b>Country</b></td>
            <td>@Html.TextBox("Country", Model.Country,
                    new Dictionary<String, Object>() {{"class", "textBox"}} />
                <br />
                @Html.ValidationMessage("country")
            </td>
        </tr>
    </table>
    <input id="btnSave" type="submit" value="Save" />
}
```

Typically, you group any editable field (for example, a text box) with a *ValidationMessage* helper. The validation helper displays any message that originates from invalid values being entered in the field. Furthermore, you ensure that the resulting URL includes a key value that uniquely identifies the record to be updated. Here's an example:

```
Html.BeginForm("Update", "Customer", new { customerId = Model.CustomerID })
```

Internally, *BeginForm* matches the data it has received to the parameters of registered URL routes in an attempt to create the proper URL to post the form. The preceding code generates the following URL:

```
http://yourserver/customer/update?customerId=alfki
```

Thanks to the default model binder, the method *Update* receives the fields of the input form as members of the *Customer* class:

```
public ActionResult Update(Customer customer)
{
    var model = _service.TryUpdateCustomer(ControllerContext, customer);
    return View("Edit", model);
}
```

The method needs to do a couple of things: update the data layer, and display the edit view for the user to keep on making changes. Except perhaps for some unrealistically simple scenarios, the update operation requires validation. If validation of the data being stored is unsuccessful, detected errors must be reported to the end user through the user interface.

The ASP.NET MVC infrastructure offers built-in support for displaying error messages that result from validation. The *ModelState* dictionary—a part of the *Controller* class—is where methods add notifications of an error. Errors in the *ModelState* dictionary are then displayed through the *ValidationMessage* helper:

```
public EditCustomerViewModel TryUpdateCustomer(ControllerContext context, Customer customer)
{
    if (Validate(context, customer))
        Update(customer);
```

```
    return EditCustomer(customer.CustomerID);
}

private static Boolean Validate(ControllerContext context, Customer customer)
{
    var result = true;
    var modelState = context.Controller.ViewData.ModelState;

    // Any sort of specific validation you need ...
    if (!CountryIsValid(customer.Country)))
    {
        // For each detected error, add a message and set a new display value
        modelState.AddModelError("Country", "Invalid country.");
        result = false;
    }

    return result;
}

private static void Update(Customer customer)
{
    _repository.Update(customer);
}
```

As the name suggests, the *ModelState* dictionary is the key/value repository for any messages that relate to the state of the model behind the view. The value is the error message; the key is the unique name used to identify the entry (like the string *"Country"* in the preceding example). The key of a model state entry will match the string parameter of the *Html.ValidationMessage* helper. Figure 4-3 shows the system's reaction when the user attempts to enter an invalid value.

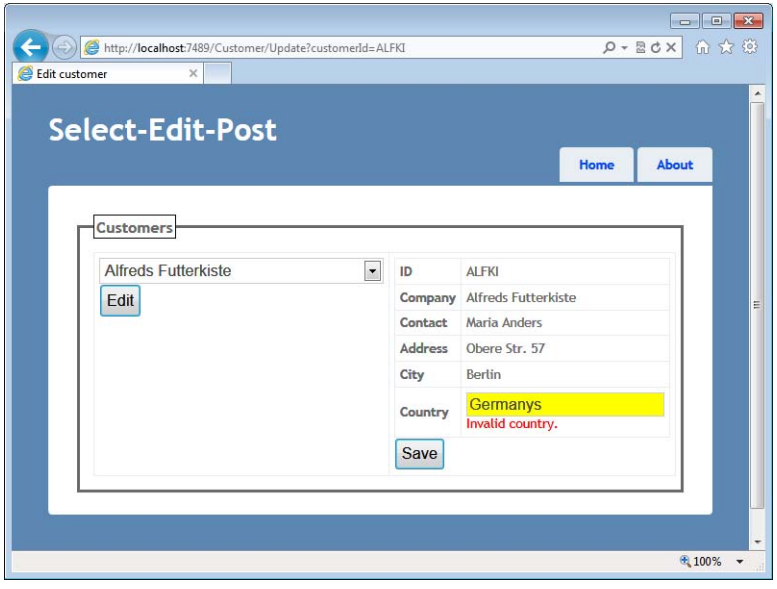

FIGURE 4-3 Dealing with invalid input.

During the validation process, every time you detect an error in the posted data, you add a new entry to the *ModelState* dictionary:

```
modelState.AddModelError("Country", "Invalid country.");
```

It is your responsibility to provide localized error messages.

Note that the model binder always sets parameters to *null* if a corresponding match cannot be found through the various value providers (for example, the form, route parameters, the query string). In particular, this means that string parameters or string members can be set to null. You should always check against nullness before you attempt to consume a value that came through model binding. The following code shows a possible way to deal with this condition when strings are involved:

```
if (String.IsNullOrEmpty(customer.Country) || !customer.Country.Equals("USA"))
{
   ...
}
```

With respect to the earlier code, you would place the test against nullness in the *CountryIsValid* method.

Applying the Post-Redirect-Get Pattern

The previous approach to input forms is functional, but it's not free of issues. First, the URL in the address bar might not reflect the data being displayed by the page. Second, repeating the last action (Refresh or an F5 keystroke) might not simply prompt the user with the annoying confirmation message shown in Figure 4-4. As the user goes ahead, she can easily get a run-time exception if the repeated operation is not implemented as *idempotent* (that is, if it doesn't always produce the same result regardless of how many consecutive times it is called).

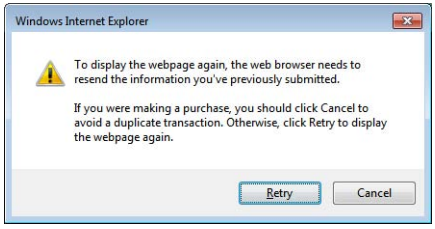

FIGURE 4-4 Confirmation message that browsers display when reposting a form.

These aspects of ASP.NET applications, however, are not specific to ASP.NET MVC. They exist in ASP.NET Web Forms too, but this is not a good reason to avoid using a better implementation.

The lack of synchronization between the URL in the browser's address bar and the content displayed might not be a problem in most cases. Your users might not even notice that. The confirmation dialog box that prompts the user when she refreshes the current page, in fact, is an old

(and not really pleasant) acquaintance of ASP.NET developers and the users of their applications. Let's see how the Post-Redirect-Get (PRG) pattern can help fix both aspects.

Keeping the URL and Content in Sync

In the previous example, the URL shown in the address bar after the user has selected, say, user ALFKI is the following:

```
// Action EDIT on CUSTOMER controller
http://yourserver/customer/edit
```

If the user repeats the last action (for example, by pressing F5), he gets the dialog box shown in Figure 4-4 and then the view updates as expected. Wouldn't it be preferable for the URL to reflect the selected customer and have the page be refreshed without side effects?

The side effect represented by the dialog box in Figure 4-4 has a well-known origin. When F5 is pressed, the browser just blindly reiterates the last HTTP request. And the selection of a customer from a drop-down list (shown in Figure 4-1) is an HTTP POST request for the action *Edit* on the *Customer* controller.

The PRG pattern recommends that each POST request, after having been processed, terminates with a redirect to a resource accessed via GET. If you do so, the URL stays nicely in sync with the customer displayed and your users forget the annoying dialog box that you saw in Figure 4-4.

Splitting POST and GET Actions

The first step on the way to applying PRG to ASP.NET MVC applications is neatly separating POST actions from GET actions. Here's how to rewrite the *Edit* action:

```
[HttpPost]
[ActionName("edit")]
public ActionResult EditViaPost([Bind(Prefix = "customerList")] String customerId)
{
    // POST, now REDIRECT via GET to Edit
    return RedirectToAction("edit", new {id = customerId});
}

[HttpGet]
[ActionName("edit")]
public ActionResult EditViaGet(String id)
{
    var model = _service.EditCustomer(id);
    return View("edit", model);
}
```

Every time the user posts the *Edit* method to select a given customer, all that happens is a redirect (HTTP 302) to the same action via GET. The GET method for the action *Edit* receives the ID of the customer to edit and does its job as usual.

The beautiful effect is that you can change the selection in either of two ways: by typing the URL in the address bar (as a command), or just by clicking on the drop-down list. Moreover, when the user

interface is updated, the last action tracked by the browser is a GET—and you can repeat the GET as many times as you like without incurring any boring warnings or annoying exceptions.

Updating Only via POST

Take a look back at Figure 4-3, which represents the next page after an update. In particular, the figure shows a failed update, but that is not relevant here. What's relevant instead is the URL:

```
http://yourserver/customer/update?customerId=ALFKI
```

This is the URL that will be repeated by pressing F5. I can hardly believe that any regular user will ever attempt to manually edit this URL and try to push updates to another customer. Anyway, as unlikely as that is, it's definitely a possibility.

The URL of an update operation should never be visible to users. The user performs the update, but the operation remains hidden between two displays of the same page. This is exactly what you get with the PRG pattern. Here's how to rewrite the *Update* action:

```
[HttpPost]
public ActionResult Update(Customer customer)
{
   _service.TryUpdateCustomer(ControllerContext, customer);
   return RedirectToAction("edit", new { id = customer.CustomerID });
}
```

As you can see, you need only the *HttpPost* leg. A user orders the update from a page created via a GET that displays the customer being edited. The update takes place, and the next view is obtained by redirecting again to the *Edit* action for the same customer. It's simple, clean, and effective.

Saving Temporary Data Across Redirects

There's one final issue to take into account. The PRG pattern makes the overall code look cleaner but requires two requests to update the view. This might or might not be a problem with regard to performance—I don't think it is. Anyway, this is primarily a functional problem: if the update fails, how do you pass feedback to the view? In fact, the view is being rendered in the GET action subsequent to the redirect—it's quite another blank request.

Overall, the best option you have is saving feedback messages to the *Session* object. ASP.NET MVC provides a slightly better option that still uses session state, but which does so through a smart wrapper—the *TempData* dictionary. Here's how to modify the code that validates before updating:

```
private static Boolean Validate(ControllerContext context, Customer customer)
{
    var result = true;

    if (String.IsNullOrEmpty(customer.Country) || !customer.Country.Equals("USA"))
    {
        var modelState = context.Controller.ViewData.ModelState;
        modelState.AddModelError("Country", "Invalid country.");
```

```
        // Save model-state to TempData
        context.Controller.TempData["ModelState"] = modelState;
        result = false;
    }

    return result;
}
```

You first add feedback messages to the *ModelState* dictionary as usual. Then you save a reference to the *ModelState* in the *TempData* dictionary. The *TempData* dictionary stores any data you provide in the session state for as long as two requests. After the second request past the storage has been processed, the container clears the entry.

You do the same for any other message or data you need to pass to the view object. For example, the following code adds a message if the update operation completes successfully (as you can see Figure 4-5):

```
private Boolean Update(ControllerContext context, Customer customer)
{
    // Perform physical update
    var result = _repository.Update(customer);

    // Add a message for the user
    var msg = result
                  ? "Successfully updated."
                  : "Update failed. Check your input data!";
    context.Controller.TempData["OutputMessage"] = msg;
    return result;
}
```

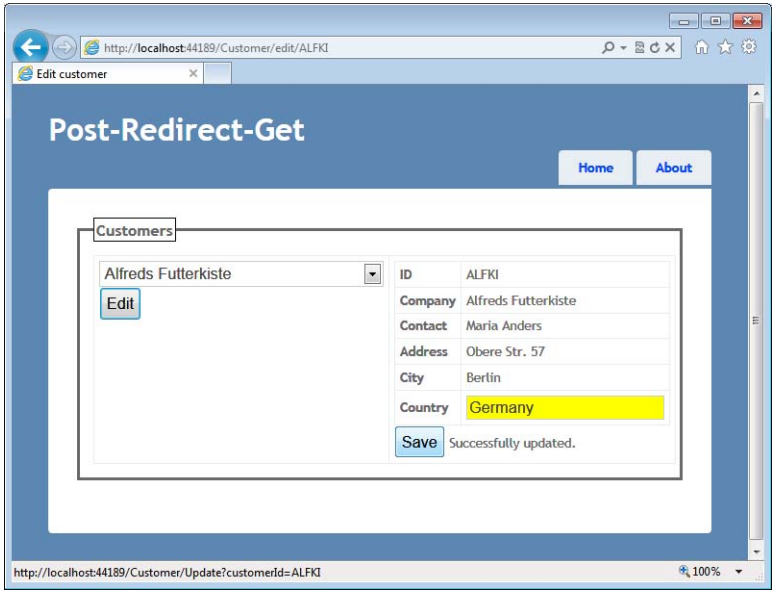

FIGURE 4-5 Post-update message.

You still might want to give some clear feedback to users about what has happened before redisplaying the same edit form for them to keep on working. If your update operation takes the user to a completely different page, you might not need the previous trick.

Saving data to the *TempData* dictionary is only half the effort. You also need to add code that retrieves the dictionary from the session state and merges that with the current model state. This logic belongs to the code that actually renders the view:

```
[HttpGet]
[ActionName("edit")]
public ActionResult EditViaGet(String id)
{
   // Merge current ModelState with any being recovered from TempData
   LoadStateFromTempData();

    ...
}
private void LoadStateFromTempData()
{
    var modelState = TempData["ModelState"] as ModelStateDictionary;
    if (modelState != null)
        ModelState.Merge(modelState);
}
```

With these few changes, you can arrange input forms that are clean to write and read and that work effectively. The only wrinkle in an input form built in accordance with the PRG pattern is that you need a second request—the redirect—for each operation. Furthermore, you still need to pack the view model with any data required to render the view, including data calculated by the ongoing operation and any other data around the page, such as menus, breadcrumbs, and lists.

To go beyond this level, however, you have to embrace the Ajax approach.

Ajax-Based Forms

Ajax is no longer a plus for the web; Ajax is a native part of the web. Now when you discuss use-cases and requirements with a customer, as long as a web front end is involved, Ajax capabilities are an obvious part of the deal. So Ajax is here, and you can leverage it to build even better input forms.

You can add Ajax to ASP.NET MVC views in two nonexclusive ways. You can simply incorporate script code in the view and use jQuery, or other libraries to grab data from a remote endpoint. Likewise, you can leverage a built-in infrastructure that basically brings the core idea of Web Forms partial rendering to ASP.NET MVC.

The core idea of partial rendering is the *HTML Message* pattern—you make an Ajax request, and the endpoint returns a response that contains plain HTML. The primary aspect of the HTML Message pattern is that it greatly simplifies the code that developers need to write. In particular, as a developer you don't need to write much JavaScript code to refresh the view with downloaded markup. The framework injects any required script into the view and just leaves you responsible for a bunch of semideclarative statements.

In particular, ASP.NET MVC implements the HTML Message pattern for anchors and forms. You can emit hyperlinks that, when followed, automatically insert the response into a given placeholder. Likewise, you can post a form and update a given placeholder with the HTML being returned.

Creating an Ajax Form

An Ajax form is nearly identical to a regular form; the only small difference is that you emit the markup of the form using *Ajax.BeginForm* instead of *Html.BeginForm*. *Ajax.BeginForm* gets one more argument than *Html.BeginForm*—the *AjaxOptions* object.

The *AjaxOptions* class allows you to specify a couple of key parameters—the target element of the current DOM that will be populated after the call, and the element that will be displayed during the call to indicate any progress. The *UpdateTargetId* property indicates the ID of the target element; the *LoadingElementId* indicates the progress element.

More important than the syntax differences is the structure of the views. Ajax is all about making partial changes to the existing page. For this reason, you don't likely need multiple views. All you do is integrate the initial view with the markup returned by Ajax calls.

The following markup shows a view equivalent to that of Figure 4-1:

```
<table cellspacing="0" cellpadding="10">
    <tr>
        <td valign="top">
            @using (Ajax.BeginForm("Edit", "Customer",
                new AjaxOptions
                {
                    UpdateTargetId = "customerEditor",
                    LoadingElementId = "loading"
                }))
            {
                @Html.DropDownList("customerList",
                   new SelectList(Model.Customers, "CustomerId", "CompanyName"))
                <input type="submit" name="btnEdit" value="Edit" />
                <br />
                <img id="loading"
                     src="@Url.Content("~/images/loading.gif")"
                     alt=""
                     class="loader" />
            }
        </td>
        <td valign="top">
            <div id="customerEditor"></div>
        </td>
    </tr>
</table>
```

The right-most cell now contains an empty *DIV*, which will be filled with the markup of the customer editor downloaded through an Ajax call. Figure 4-6 shows the schema of an Ajax-based form for the input form we considered so far.

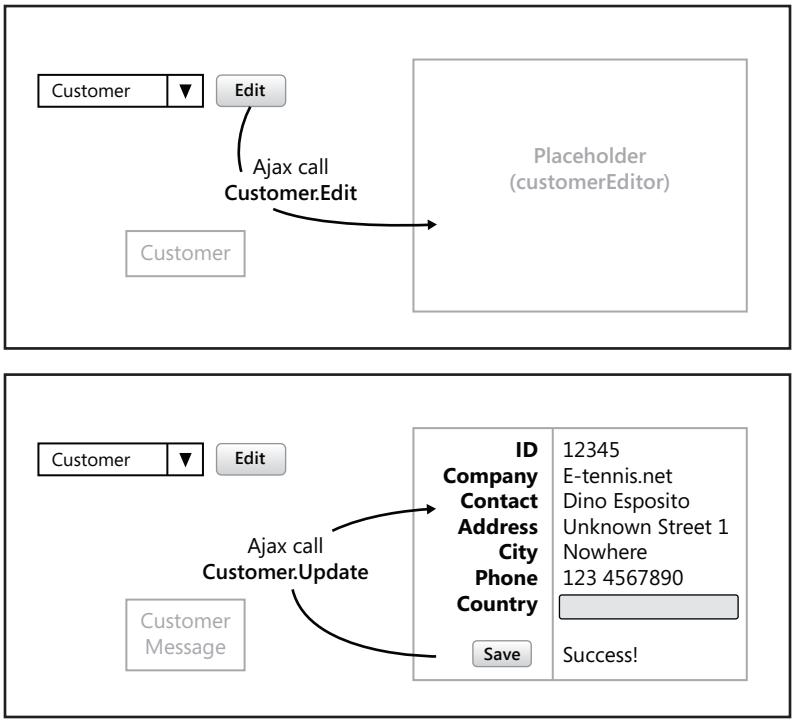

FIGURE 4-6 Schema of an Ajax form.

Infrastructure for Ajax-Based Forms

To use the *Ajax.BeginForm* helper successfully, you must link both *MicrosoftAjax.js* and *MicrosoftMvcAjax.js* from your application. Both files can be linked from your site or from the Microsoft content delivery network (CDN). These files are also automatically added to the project you create via Microsoft Visual Studio and are available from the Scripts folder. If you don't want to host them on your site, you can find the exact link to the Microsoft CDN here: *http://www.asp.net/ajaxlibrary/cdn.ashx*.

You use the *AjaxOptions* class to indicate additional parameters for an Ajax operation. Table 4-1 describes the role played by the properties of the *AjaxOptions* class.

TABLE 4-1 Members of the *AjaxOptions* class

Property	Description
Confirm	Indicates the JavaScript function to call to have a confirmation before the request executes.
HttpMethod	Indicates the HTTP method to use for the request.
InsertionMode	Indicates the insertion mode for any content downloaded that has to be injected in the current DOM.
LoadingElementId	Indicates the ID of the DOM element to be displayed while the request is ongoing.
LoadingElementDuration	Indicates the time (in milliseconds) it should take to show and hide the progress user interface.
OnBegin	Indicates the JavaScript function to call before the request executes. By returning *false*, you cancel the operation.
OnComplete	Indicates the JavaScript function to call when the request has completed. This call is made in addition to more specific callbacks that deal with the success or failure of the operation. (The sequence of these calls depends on whether you're using unobtrusive JavaScript or not, which is discussed later in this chapter.) By returning *false*, you cancel the page update.
OnFailure	Indicates the JavaScript function to call when the request completes with a failure.
OnSuccess	Indicates the JavaScript function to call when the request completes successfully.
UpdateTargetId	Indicates the ID of the DOM element to be updated with any HTML content downloaded.
Url	Indicates the target URL of the request if it is not already specified in the markup, such as when a link or a form is used.

Not all of the properties are to be set, however. At a minimum, you might want to specify the *UpdateTargetId* and *LoadingElementId* properties to update the user interface and display a progress message. During the execution of an Ajax request, three JavaScript callbacks might be involved. The first is *OnBegin*, which fires just before the request is placed. Next, you receive *OnComplete*, followed by either *OnSuccess* or *OnFailure*. All these callbacks receive an *AjaxContext* object. Members of the *AjaxContext* class are listed in Table 4-2.

Important The order in which Ajax callbacks run, and their prototype, is different if you opt for the unobtrusive JavaScript model. I'll cover the unobtrusive JavaScript model for Ajax in ASP.NET MVC 3 later in the chapter.

TABLE 4-2 Members of the *AjaxContext* class for JavaScript

Property	Description
data	Indicates the response being returned
insertionMode	Indicates the insertion mode for the response
loadingElement	Indicates the DOM element used to show feedback during the request
request	Indicates the library object that incorporates the web request
response	Indicates the internal object used to execute the request
updateTargetId	Indicates the DOM element used to update the user interface

The member named *data* contains the response. Note that *data* is implemented as a string. If it is a JSON string, you must use the *eval* function to transform it into a usable JavaScript object. You might use the member *request* in a preliminary event such as *OnBegin*, whereas you might need to access *response* in a completion event such as *OnComplete*. Note that *request* is an object of type *Sys.Net.WebRequest*, whereas *response* is an object of type *Sys.Net.WebRequestExecutor*. Both types are defined in the *MicrosoftAjax.js* library.

Important In ASP.NET MVC 3, the *AjaxOptions* class is not used when you opt for unobtrusive JavaScript. See the upcoming section "Unobtrusive JavaScript code." Until then, take any statement as specific to the ASP.NET MVC Ajax model as it worked in earlier versions of the framework.

Ajax-Aware Controller Methods

The form posts to method *Edit* on the *Customer* controller. The method addressed by an Ajax call can't return just a regular view; it has to return a fragment of HTML, not a full page. That's why you use *PartialView* to refer to a partial view such as a user control:

```
public ActionResult Edit([Bind(Prefix = "customerList")] String customerId)
{
    var model = _service.EditCustomer(customerId);
    return PartialView("_customerEditor", model);
}
```

The *_customerEditor* view is the same user control discussed earlier. Also, this user control requires some minor changes. For example, you need to transform the form into an Ajax form by using *Ajax.BeginForm* to post to the *Update* method:

```
Ajax.BeginForm("Update", "Customer",
    new { customerId = Model.Customer.CustomerID },
    new AjaxOptions
        {
            UpdateTargetId = "customerEditor"
        })
```

Note that the customer editor replaces itself after posting. As you can see, the value of the *UpdateTargetId* property is the same ID as the element that appears where the editor was originally inserted. Figure 4-7 shows the same input form you considered earlier in the chapter rewritten as a single view (*index.cshtml*) plus a partial view (*_customerEditor.cshtml*).

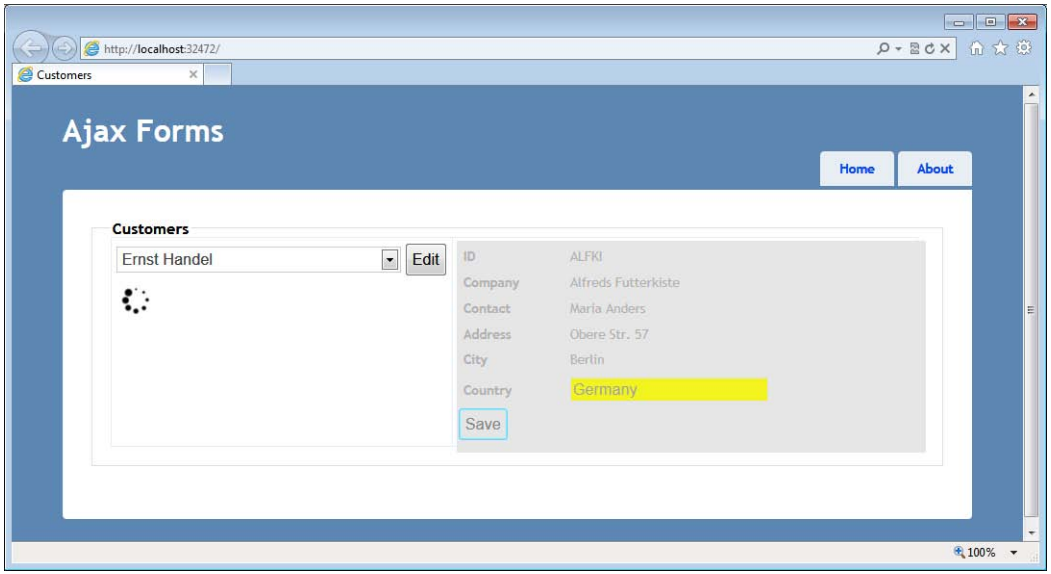

FIGURE 4-7 The same form rewritten as an Ajax form.

In general, Ajax forms can be functionally equivalent to HTML forms. Implementing Ajax forms, however, might require some changes to the view models and the overall data flow. Both Figure 4-6 and Figure 4-7 suggest the view model for the *Edit* action needs to pass only the *Customer* object; there's no need to also pass the list of customers. In Ajax, in fact, that part of the view is never refreshed. Likewise, the view model used to render the view past the *Update* action also needs an additional property for representing the overall message that indicates whether the update was successful or whether it failed. (Validation error messages are incorporated in the child form and don't need to make it to the view model.)

Updating the Page Title After an Ajax Request

A known issue with ASP.NET MVC Ajax operations is updating the title of the current page. In a non-Ajax implementation, you add the title string to the view model and have the view update the title. In Ajax, you return only an HTML fragment and need some ad hoc script to update the title. Moreover, if the new title depends on the context—a very likely scenario indeed—you must find a way to add it to the response.

If your Ajax operation returns JSON, you pack title information in the object being returned and process it via script. But if you're using the ASP.NET MVC Ajax infrastructure, you have little control over the data being returned over the wire in the context of the request; more importantly, you don't control the script that processes the response. Some tricks are necessary.

When you create Ajax forms through *Ajax.BeginForm* and want to update the page title after an operation, you must just add the title string to the response in some way and then retrieve it on the client. As mentioned, the *AjaxOptions* object has a property named *OnSuccess* that refers to a JavaScript function automatically invoked if the Ajax operation completes successfully with a status code in the range of 200:

```
Ajax.BeginForm("Edit", "Customer",
                    new AjaxOptions
                    {
                        UpdateTargetId = "customerEditor",
                        LoadingElementId = "loading",
                        OnSuccess = "onEditCompleted"
                    })
```

Here's a possible skeleton for the *onEditCompleted* JavaScript function:

```
function onEditCompleted(context) {
   document.title = ...;
}
```

How can you pass title information? A simple but effective solution is to use a custom HTTP response header. According to the World Wide Web Consortium (W3C), in fact, a response header should be used to pass additional information about the response, which cannot be placed in the status line. Here's how to set a custom header for Ajax requests:

```
public ActionResult Edit([Bind(Prefix = "customerList")] String customerId)
{
    var model = _service.EditCustomer(customerId);
    if (Request.IsAjaxRequest())
       HttpContext.Response.AddHeader("Content-Title", model.Customer.CompanyName);
    return PartialView("_customerEditor", model);
}
```

The name of the header (*Content-Title* in the example) is arbitrary. The following code shows how to retrieve the header from JavaScript and set the page title:

```
function onEditCompleted(context) {
    var response = context.get_response();
    var title = response.getResponseHeader('Content-Title');
    if (title.length >0)
       document.title = title;
}
```

Figure 4-8 shows the finalized Ajax form.

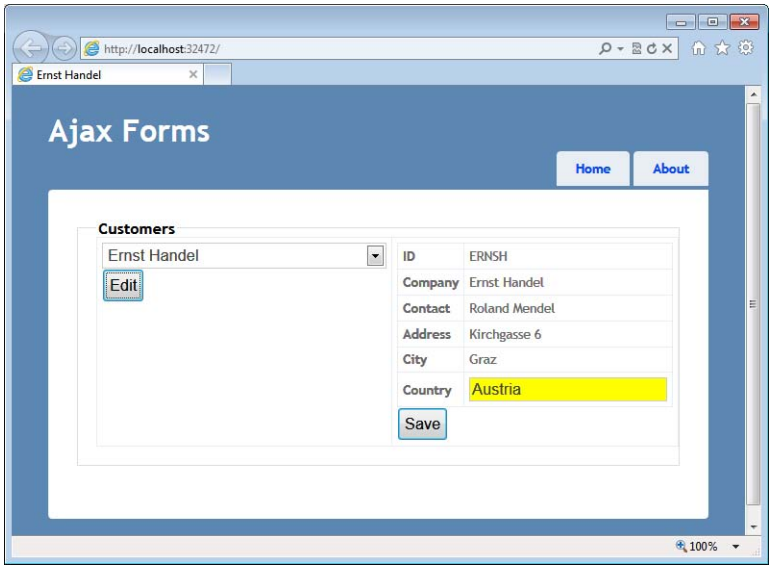

FIGURE 4-8 Ajax form with title updates.

Unobtrusive JavaScript Code

Just like any piece of Ajax functionality, Ajax input forms are extensively based on JavaScript calls. At its core, an Ajax form works by placing the HTTP request using *XMLHttpRequest* instead of leveraging the browser's network services. The *Ajax.BeginForm* helper takes care of emitting any required JavaScript code and binds it to the *onsubmit* event of the *FORM* tag. Here's a sample of this code:

```
<form action="/Customer/Edit"
      id="form0"
      method="post"
      onclick="Sys.Mvc.AsyncForm.handleClick(this, new Sys.UI.DomEvent(event));"
      onsubmit="Sys.Mvc.AsyncForm.handleSubmit(this, new Sys.UI.DomEvent(event),
                    { insertionMode: Sys.Mvc.InsertionMode.replace,
                      loadingElementId: &#39;loading&#39;,
                      updateTargetId: &#39;customerEditor&#39;,
                      onSuccess: Function.createDelegate(this, onEditCompleted) });">
      ...
</form>
```

From a functional perspective, there's nothing wrong with this code, except that it's a bit too intrusive in the HTML layout. It adds explicit event handlers to the markup and references JavaScript objects in external libraries. In this context, "intrusive" means having the HTML markup tightly coupled to external dependencies and not limited to the basic syntax of HTML. In ASP.NET MVC 3, you can instruct the Ajax infrastructure to rewrite this code to be unobtrusive. Unobtrusive JavaScript is a pattern for webpages that essentially consists of keeping JavaScript separated from HTML elements. Any required event handlers are not hardcoded as attributes but are added programmatically

through automatically running script. In ASP.NET MVC 3, you can declaratively turn the previous code into the following:

```
<form action="/Customer/Edit"
      data-ajax="true"
      data-ajax-loading="#loading"
      data-ajax-mode="replace"
      data-ajax-success="onEditCompleted"
      data-ajax-update="#customerEditor"
      id="form0"
      method="post">
...
</form>
```

You enable unobtrusive Ajax JavaScript by linking the file *jquery.unobtrusive-ajax.js*, or its minified version, to the view. The file contains the script that processes the *data-ajax-XXX* attributes and dynamically adds event handlers:

```
<script src="@Url.Content("~/Scripts/jquery.unobtrusive-ajax.min.js")" type="text/javascript" />
```

This is the only JavaScript file you need to link in an unobtrusive mode. You can get rid of the files I mentioned earlier: *MicrosoftAjax.js* and *MicrosoftMvcAjax.js*.

In addition, you add an entry to the *web.config* file, as shown here:

```
<appSettings>
   ...
  <add key="UnobtrusiveJavaScriptEnabled" value="true" />
</appSettings>
```

Note that unobtrusive JavaScript is the default configuration when you create a new ASP.NET MVC 3 project through the Visual Studio wizard.

Note If you miss the unobtrusive JavaScript file, the worst thing you'll experience is that the request is processed as a regular HTML request. You won't see any errors when attempting to submit the Ajax request; this is because no actual JavaScript file is linked to the page that the browser knows about.

Troubles with Unobtrusive Ajax

There are some important differences in Ajax forms if you decide to opt for the unobtrusive model. In a nutshell, porting to the unobtrusive model is seamless as long as you don't use callbacks. If you're using callbacks such as *OnSuccess* or *OnCompleted*, be prepared to rewrite your code a bit.

The file that unobtrusively attaches event handlers to your DOM elements uses jQuery and, more importantly, uses the jQuery model for Ajax operations. The jQuery Ajax model and the original ASP.NET MVC Ajax model are different.

A first difference you might notice is that the *OnComplete* callback is invoked after (and not before, as mentioned in Table 4-1) the callbacks that denote success or failure. By returning *false* from the *OnBegin* callback, you can still cancel the operation. Any value you return from *OnComplete*, instead, is just ignored. The signature of the *OnSuccess* callback is different. Here's how to rewrite the *OnSuccess* callback that updates the title of the page:

```
function onEditCompletedWithSuccessUnobtrusive(data, status, xhr) {
    var title = xhr.getResponseHeader('Content-Title');
    if (title.length >0)
       document.title = title;
}
```

The first argument indicates the response of the call; the second argument is a string that indicates the outcome of the operation. Finally, the third parameter is the *XMLHttpRequest* object through which the operation was carried out. You use the *getResponseHeader* method to read a particular response header. Table 4-3 shows the mapping between ASP.NET MVC callbacks and jQuery Ajax callbacks. Based on that, you can figure out signatures from the *http://api.jquery.com/jQuery.ajax* site.

TABLE 4-3 Members of the *AjaxOptions* class for JavaScript

ASP.NET MVC Callback	jQuery Callback
OnBegin	beforeSend(jqXHR, settings)
OnComplete	complete(jqXHR, textStatus)
OnSuccess	success(data, textStatus, jqXHR)
OnFailure	error(jqXHR, textStatus, errorThrown)

Ajax Hyperlinks

An action link is an HTML helper that emits a hyperlink bound to a piece of JavaScript code. As a result, when you click on the hyperlink, the URL is invoked asynchronously and a JavaScript callback runs when the response is ready. Here's an example:

```
@Ajax.ActionLink("Show catalog", "Index", "Customer",
            new AjaxOptions
            {
               OnSuccess="fillProductList"
            })
```

In this case, the *ActionLink* method generates a hyperlink that points to the *Index* action and displays the "Show catalog" text. What about the controller?

When the *ActionLink* code block is processed, the name of the controller is resolved to the controller that is processing the view, if no other controller is specified. If the controller is different, you simply pick up another overload of the method that allows you to indicate the controller name.

The *ActionLink* method emits the following JavaScript call for the previous code block:

```
<a href="/Products/Index"
   onclick="Sys.Mvc.AsyncHyperlink.handleClick(
          this,
          new Sys.UI.DomEvent(event),
          {
            insertionMode: Sys.Mvc.InsertionMode.replace,
            onSuccess: Function.createDelegate(this, fillProductList)
          }
   );">
   Show catalog
</a>
```

To use the *ActionLink* method successfully, you must link both *MicrosoftAjax.js* and *MicrosoftMvcAjax.js* from your application. If unobtrusive JavaScript is enabled, you must link the file *jquery.unobtrusive-ajax.js* instead. All considerations we just covered regarding callbacks apply in the same way.

Automating the Writing of Input Forms

Input forms play a central part in the organization of views. Often input forms contain images, require the specific placement of fields, and are enriched with client-side goodies. In all of these cases, you probably need to write the template of the input form from scratch.

Generally speaking, however, there are a number of other situations in which the building of the input form can be automated. This is often the case, for example, when you write the back office of a website. A back-office system typically consists of a bunch of forms for editing records; however, as a developer, you don't care much about the graphics. You focus on effectiveness rather than style. In a nutshell, a back-office system is the perfect case where you don't mind using an automatic generator of form input templates.

Predefined Display and Editor Templates

In Chapter 2, you encountered templated HTML helpers, such as *DisplayXxx* and *EditorXxx* helpers. These helpers can take an object (even the whole model passed to the view) and build a read-only, or editable, form. You use the following expression to specify the object to display or edit:

```
Html.DisplayFor(model => model.Customer)
Html.EditorFor(model => model.Customer)
```

The model argument is always the model being passed to the view. You can use your own lambda expression to select a subset of the entire model. To select the entire model (say, for editing), you can choose either of the following, functionally equivalent, expressions:

```
Html.EditorFor(model => model)
Html.EditorForModel()
```

Attributes placed on public members of the model class provide guidance on how to display individual values. Let's find out how it works in practice.

Annotating Data Members for Display

In ASP.NET MVC, templated helpers use metadata associated with class members to decide how to display or edit your data. Metadata is read through a metadata provider object; the default metadata provider grabs information from data annotations attributes. The most commonly used attributes are listed in Table 4-4.

TABLE 4-4 Some annotations that affect the rendering of data

Attribute	Description
DataType	Indicates the presumed data type you'll be editing through the member. It accepts values from the *DataType* enumeration. Supported data types include *Decimal*, *Date*, *DateTime*, *EmailAddress*, *Password*, *Url*, *PhoneNumber*, and *MultilineText*.
DisplayFormat	Allows you to indicate a format through which to display (and/or edit) the value. For example, you might use this annotation to indicate an alternate representation for null or empty values. In this case, you use the *NullDisplayText* property.
DisplayName	Indicates the text to use for the label that presents the value.
HiddenInput	Indicates whether a hidden input field should be displayed instead of a visible one.
UIHint	Indicates the name of the custom HTML template to use when displaying or editing the value.

Annotations are spread across a variety of namespaces, including *System.ComponentModel* and *System.ComponentModel.DataAnnotations*. If you explore these (and other) namespaces, you can find even more attributes, but some of them don't seem to work with ASP.NET MVC—at least not in the way that one would expect.

Data annotations are attributes, and attributes don't typically contain code. They just represent meta information for other modules to consume. By using data annotations, you decorate your model objects with metadata. This isn't really expected to produce any visible effect: it all depends on how other components consume metadata.

In ASP.NET MVC, default display and editor helpers simply consume only a few of the possible annotations. However, metadata information is there, and if you override default templates you have available a lot more meta information to consume at your leisure. The *ReadOnly* attribute is a good example of an annotation that default templates ignore but that ASP.NET MVC regularly understands and exposes to helpers.

Note Data annotations include descriptive attributes that instruct listeners how to display or edit data as well as validation attributes that instruct listeners how to validate the content of a model class. I'll discuss validation attributes later.

The following code shows a view-model class decorated with annotations:

```
public class CustomerViewModel : ViewModelBase
{
    [DisplayName("Company ID")]
    [ReadOnly(true)]  // This will be blissfully ignored by default templates!
    public Int32 Id { get; set; }

    [DisplayName("Is a Company (or individual)?")]
    public Boolean IsCompany { get; set; }

    [DisplayFormat(NullDisplayText = "(empty)")]
    public String Name { get; set; }

    [DataType(DataType.MultilineText)]
    public String Notes { get; set; }

    [DataType(DataType.Url)]
    public String Website { get; set; }

    [DisplayName("Does this customer pay regularly?")]
    public Boolean? IsReliable { get; set; }
}
```

Note that you don't need data annotations if you are not using *DisplayForModel* or *EditorForModel* to generate input forms automatically. Like any other metadata, annotations are transparent to any code that is not specifically designed to consume them.

Figure 4-9 shows the input form that *EditorForModel* creates for the previous model object.

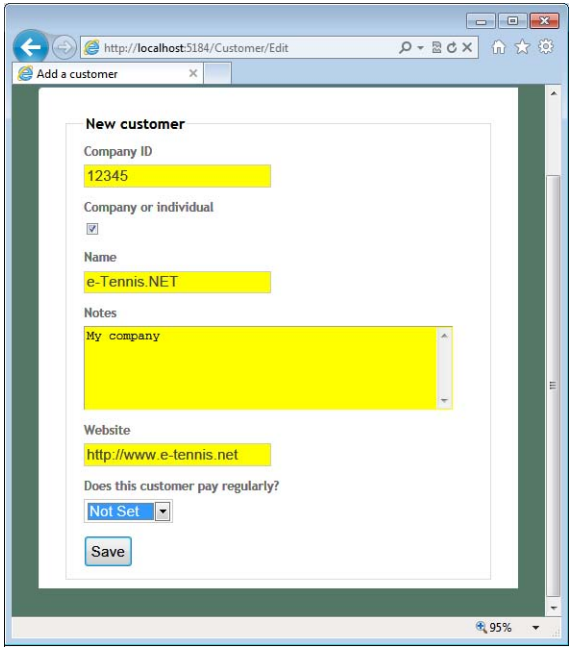

FIGURE 4-9 An autogenerated input form.

Meta information tells the Editor/Display helpers how to edit and display values. This results in ad hoc HTML templates being used, such as a *TextArea* element for multiline types and check boxes for Boolean values. For nullable Boolean values, helpers also automatically display a tri-state drop-down list. Not all data types show up in edit mode or display mode. For example, the *EmailAddress* and *Url* data types are reflected only in display mode.

To be precise, the form displayed in the figure also passed through a slight set of graphical changes due to the application of a style sheet. The Editor/Display helpers automatically read style information from a few cascading style sheet (CSS) classes with a conventional name, as shown here:

```
.display-label, .editor-label {
    margin: 1em 0 0 0;
    font-weight: bold;
}
.display-field, .editor-field {
    margin: 0.5em 0 0 0;
}
.text-box {
    width: 30em;
    background: yellow;
}
.text-box.multi-line {
    height: 6.5em;
}
.tri-state {
    width: 6em;
}
```

In particular, classes named *display-label* and *editor-label* refer to labels around input values. You can insert these styles either inline in views or inherit them from a globally shared CSS file.

Default Templates for Data Types

Display/Editor helpers work by mapping the data type of each member to render to a predefined display or edit template. Next, for each of the template names, the system asks the view engines to return a proper partial view. Predefined display templates exist for the following data types: *Boolean*, *Decimal*, *EmailAddress*, *HiddenInput*, *Html*, *Object*, *String*, *Text*, and *Url*. Data types are resolved through the *DataType* annotation or the actual type of the value. If no match can be found, the default template is used, which consists of plain text for display and a text box for editing. Let's see what a template looks like.

The following listing shows a sample implementation of the ASPX template for displaying the *Url* data type:

```
<%@ Control Language="C#" Inherits="System.Web.Mvc.ViewUserControl" %>
<a href="<%: Model %>">
   <%: Model %>
</a>
```

The *Url* data type is rendered through a hyperlink in which the *Model* object references the data being displayed—most likely, a string representing a website. The real code being used by ASP.NET MVC is a bit more sophisticated, like the code shown here:

```
<%@ Control Language="C#" Inherits="System.Web.Mvc.ViewUserControl" %>
<a href="<%: Model %>">
    <%: ViewData.TemplateInfo.FormattedModelValue) %>

</a>
```

As you can see, the real value is used to set the URL but not the text of the hyperlink. The *FormattedModelValue* property on the *TemplateInfo* object is either the original raw model value or a properly formatted string, if a format string is specified through annotations.

Note The *TemplateInfo* property is defined on the *ViewData* object, but it is always null, except when you're inside of a template.

The *Url* template is a fairly simple one. The template for Boolean values is a much more interesting example to consider:

```
<%@ Control Language="C#" Inherits="System.Web.Mvc.ViewUserControl" %>
<script runat="server">
private bool? ModelValue {
   get
   {
      bool? value = null;
      if (ViewData.Model != null) {
         value = Convert.ToBoolean(ViewData.Model, CultureInfo.InvariantCulture);
      }
      return value;
   }
}
</script>
<% if (ViewData.ModelMetadata.IsNullableValueType) { %>
   <select class="list-box tri-state" disabled="disabled">
      <option value=""
              <%= ModelValue.HasValue
                     ? "" : "selected='selected'" %>> Not Set </option>
      <option value="true"
             <%= ModelValue.HasValue && ModelValue.Value
                     ? "selected='selected'" : "" %>> True </option>
      <option value="false"
             <%= ModelValue.HasValue && !ModelValue.Value
                     ? "selected='selected'" : "" %>> False </option>
    </select>
<% } else { %>
    <input class="check-box" disabled="disabled" type="checkbox"
           <%= ModelValue.Value ? "checked='checked'" : "" %> />
```

The example contains a mix of logic and markup as the template applies to *Boolean* and *Nullable<Boolean>* values. The value is first converted to a *Boolean* type, if possible, and the result is stored in the *ModelValue* property. Next, the template queries the *ModelMetadata* property on the *ViewData* object to learn whether the type is nullable. If so, it emits a drop-down list; otherwise, it proceeds with creating a classic check box as shown in Figure 4-9. The *ModelMetadata* property stores everything the metadata provider could figure out through data annotations.

Note The default metadata provider reads data annotation attributes. A custom metadata provider, which is always an option, could read metadata from any other source, including databases or XML files.

Editor templates are not that different from display templates, at least in terms of structure. The code, however, might be a bit more complex. ASP.NET MVC provides a few predefined editors for *Boolean*, *Decimal*, *HiddenInput*, *Object*, *String*, *Password*, and *MultilineText*. Here's a sample editor for a password:

```
<%@ Control Language="C#" Inherits="System.Web.Mvc.ViewUserControl" %>
<%= Html.Password("",
        ViewData.TemplateInfo.FormattedModelValue,
        new { @class = "text-box single-line password" }) %>
```

As you can see, conventions regarding styles descend from the implementation of default templates. By changing the default template for a given data type, you can choose to style it according to different rules.

Object.aspx is the template used to recursively loop through the public members of a type and build an entire form using type-specific editors. The default implementation of *Object.aspx* vertically stacks labels and editors. Although it's valuable for quick prototyping, it's not usually something you want to seriously consider for a realistic application. Let's move ahead and see how to customize editor templates.

Note An insider's perspective of templated helpers, and a lot of valuable details, can be found on Brad Wilson's blog at *http://bradwilson.typepad.com/blog/2009/10/aspnet-mvc-2-templates-part-3-default-templates.html*. Although the post refers to ASP.NET MVC 2, it contains up-to-date information.

Custom Templates for Data Types

Display/Editor helpers are customizable to a great extent. Any custom template consists of a custom view located under the Views*[controller]*\DisplayTemplates folder for display helpers and under the Views*[controller]*\EditorTemplates folder for editor helpers. If you want templates shared by all controllers, you place them under Views\Shared. If the name of the view matches the data type, the view

becomes the new default template for that data type. If it doesn't, the view won't be used until it's explicitly called through the *UIHint* annotation, as shown here:

```
public CustomerViewModel
{
    ...
    [UIHint("CustomerViewModel_Url")]
    public String Url {get; set;}
}
```

The *Url* property is now displayed or edited using the *CustomerViewModel_Url* template. Properties whose data type is *Url* will keep on being served, instead, by the default template.

Let's see how to create a custom display and editor template for a type that doesn't even have predefined templates—the *DateTime* type. Here's the content for an ASPX display template:

```
<%@ Control Language="C#" Inherits="System.Web.Mvc.ViewUserControl" %>
<%: ((DateTime)Model).ToString("ddd, dd MMM yyyy") %>
```

In this example, you just display the date in a fixed format (but one controlled by you) that includes day of the week, name of the month, and year. Here's a sample editor template for a date:

```
<%@ Control Language="C#" Inherits="System.Web.Mvc.ViewUserControl<DateTime>" %>
<table><tr>
    <td><%= Html.Label("Day") %></td>
    <td><%= Html.TextBox("Day", Model.Day)%></td>
</tr><tr>
    <td><%= Html.Label("Month")%></td>
    <td><%= Html.TextBox("Month", Model.Month) %></td>
</tr><tr>
    <td><%= Html.Label("Year")%></td>
    <td><%= Html.TextBox("Year", Model.Year)%></td>
</tr></table>
```

You can style elements in the template using any class names you like; as long as those classes are defined in some style sheets in the application, they will be automatically used. Note that IDs you set in your templates are always automatically prefixed in the emitted markup by the name of the member. For example, if the preceding date template is applied to a property named *BirthDate*, the actual IDs emitted will be *BirthDate_Day*, *BirthDate_Month*, and *BirthDate_Year*.

Note It's probably not a very realistic scenario, but if you happen to have two editors for the same model in the same page, well, in this case you're going to have ID conflicts that the model binder might not be able to solve without a bit of effort on your own. If you are in this position, you might want to reconsider the tricks we discussed in Chapter 3 for binding a collection of custom types to a controller method.

How do you name these templates? If the *UIHint* attribute is specified, its value determines the template name. If it's not specified, the *DataType* attribute takes precedence over the actual type name of the member. Given the following class definition, the expected template name is *date*. If

none of the view engines can provide such a view, the default template is used instead. Figure 4-10 shows the table-based date editor defined earlier.

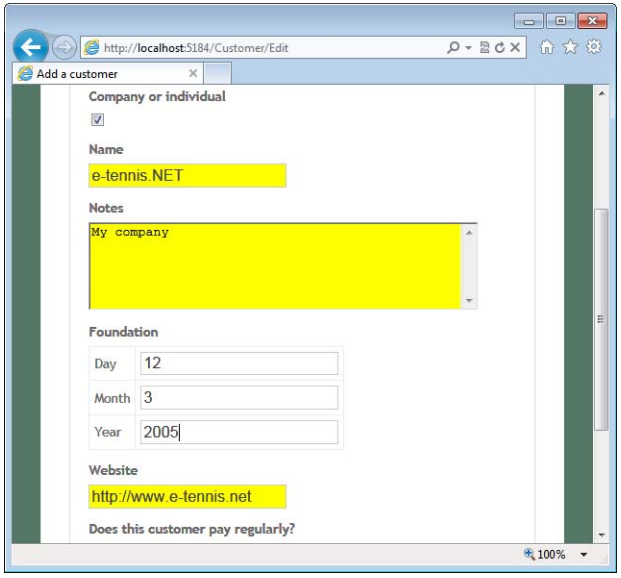

FIGURE 4-10 A custom template for editing dates.

Speaking of custom templates for dates, another typical example is creating a custom template where you integrate a jQuery UI to pick up the date from a popup calendar.

Data Type Validation

Unlike what the name suggests, the *DataType* attribute doesn't have any effect on validation. If you set a model member to be, say, a *Date* or a *Url*, no validation is performed on either the client or the server. However, this doesn't mean that type-based validation is not possible in an automatic way via annotations.

In particular, the model binder always attempts to cast the provided value to the specified property type and adds an error to the model state if the attempt fails. If you assign a string to a property of type *Int32*, you'll get an error message, meaning that these basic forms of validation are implemented, anyway. (If you use HTML 5–specific input elements, you are guaranteed by the browser that no invalid value is being passed.)

For some types, specifically numbers such as *Int32* and *Decimal*, basic data type validation occurs on the client also, and the form won't post if a string is inserted where a numeric property is expected. Checks on dates, however, are not enforced on the client by default. I'll introduce client-side validation later in the chapter.

Read-Only Members

If you decorate a member of the view model with the *ReadOnly* attribute, you probably expect it not to be editable in the editor. You probably expect that the display template is used within the editor for the model. You'll be surprised to see that this is not the case. The *ReadOnly* attribute is properly recognized by the metadata provider, and related information is stored in the metadata available for the model. For some reason, though, this is not transformed into a template hint.

As weird as it might sound, you have data annotations to indicate a given member is read-only, but this is not reflected by default templates. There are a few workarounds to this problem. First and foremost, you can use the *UIHint* annotation to specify a read-only template like the one shown here, named *readonly.ascx*:

```
<%@ Control Language="C#" Inherits="System.Web.Mvc.ViewUserControl" %>
<span><%: Model %></span>
```

You need to place the *readonly.ascx* template in the editor folder.

Although it's effective, this solution bypasses the use of the *ReadOnly* attribute. Not that this is a big problem, but you might wonder if there's a way to solve the issue by forcing the metadata provider to process the *ReadOnly* attribute differently. You need a different metadata provider, like the one shown here:

```
public class ExtendedAnnotationsMetadataProvider : DataAnnotationsModelMetadataProvider
{
    protected override ModelMetadata CreateMetadata(IEnumerable<Attribute> attributes,
        Type containerType,
        Func<Object> modelAccessor,
        Type modelType,
        String propertyName)
    {
        var metadata = base.CreateMetadata(
            attributes,
            containerType,
            modelAccessor,
            modelType,
            propertyName);

        if (metadata.IsReadOnly)
            metadata.TemplateHint = "readonly";   // Template name is arbitrary
        return metadata;
    }
}
```

You create a new class that inherits the *DataAnnotationsModelMetadataProvider* class and overrides the *CreateMetadata* method. The override is simple—you call the base method and then check what the *IsReadOnly* property returns. If the member is declared read-only, you can programmatically set the *TemplateHint* property to your custom read-only template. (It is then your responsibility to ensure that such a template is available under the Views folder.)

A custom metadata provider must be registered with the system. You can do that in either of two ways. You can just store an instance of the new provider in the *Current* property of the *ModelMetadataProviders* class, as shown next. (This needs be done in *Application_Start* in *global.asax*.)

```
ModelMetadataProviders.Current = new ExtendedAnnotationsMetadataProvider();
```

In ASP.NET MVC 3, you have another equally effective option: using dependency resolvers. I'll return to the topic of dependency resolvers in Chapter 7. For now, it suffices to say that all internal components of ASP.NET MVC that were pluggable in earlier versions, and a bunch of new ones, are in ASP.NET MVC 3 and are discovered by the system using a centralized service (the *dependency resolver*) that acts like a *service locator*. A service locator is a general-purpose component that gets a type (for example, an interface) and returns another (for example, a concrete type that implements that interface).

In ASP.NET MVC 3, the *ModelMetadataProvider* type is discoverable through the dependency resolver. So all you have to do is register your custom provider with a made-to-measure dependency resolver. A dependency resolver is a type that knows how to get an object of a requested type. A simple dependency resolver tailor-made for this scenario is shown here:

```
public class SampleDependencyResolver : IDependencyResolver
{
    public object GetService(Type serviceType)
    {
        try
        {
            return serviceType == typeof(ModelMetadataProvider)
                ? new ExtendedAnnotationsMetadataProvider()
                : Activator.CreateInstance(serviceType);
        }
        catch
        {
            return null;
        }
    }

    public IEnumerable<object> GetServices(Type serviceType)
    {
        return Enumerable.Empty<Object>();
    }
}
```

You register the resolver in *global.asax* like so:

```
DependencyResolver.SetResolver(new SampleDependencyResolver());
```

Note that ASP.NET MVC 3 attempts to find a valid implementation of *ModelMetadataProvider* first by looking into registered resolvers and then by looking into the *ModelMetadataProviders.Current* property. Double registration for the *ModelMetadataProvider* type is not allowed, and an exception will be thrown if that occurs.

Custom Templates in Razor

Writing display and editor templates in Razor is nearly the same as using the ASPX view engine. In ASPX, you always begin your views with the following code:

```
<%@ Control Language="C#" Inherits="System.Web.Mvc.ViewUserControl<TModel>" %>
```

In this code, *TModel* stands for the actual type of the model. This code translates literally to Razor:

```
@inherits System.Web.Mvc.WebViewPage<TModel>
...
```

This syntax allows you to consume data in the model as well as gaining you access to template properties via *ViewData*. In addition, Razor allows a more compact syntax that you might want to leverage when you are not going to consume directly any template metadata. Here it is:

```
@model TModel
...
```

In practice, you focus exclusively on the model type and ignore any other metadata available.

Custom Templates for Model Data Types

So far, we've dug out a lot of details about display and editor templates that work on primitive types of values. The internal architecture of helpers such as *EditorForModel* and *DisplayForModel*, however, provides for some inherent ability to reflect through the public interface of the model type and build a hierarchical view. This behavior is hardcoded, but it's too simple to be useful in real-world scenarios. In this section, I'll show you how to rewrite the display and editor template for a generic type to make it look tabular. In other words, instead of getting a vertically stacked panel of label and value *DIVs*, you get a two-column table with labels on the left and values on the right. The template to overwrite is named *object*.

Tabular Templates

Writing a table-based layout is a relatively simple exercise that consists of looping over the properties of the model object. Here is the Razor code. (I'm using the Razor code here because it is much easier to read and understand than the ASPX counterpart.)

```
@inherits System.Web.Mvc.WebViewPage
@if (Model == null)
    <span>@ViewData.ModelMetadata.NullDisplayText</span>
else
{
    <table cellpadding="0" cellspacing="0" class="display-table">
    @foreach (var prop in ViewData
                        .ModelMetadata
                        .Properties
                        .Where(pm => pm.ShowForDisplay && !ViewData.TemplateInfo.Visited(pm)))
    {
```

```
        <tr>
            <td>
                <div class="display-label">
                    @prop.GetDisplayName()
                </div>
            </td>
            <td>
                <div class="display-field">
                    <span>@Html.Display(prop.PropertyName)</span>
                </div>
            </td>
        </tr>
    }
    </table>
}
```

If the model is null, you just emit the default null text for the model class. If it's not, you proceed to looping all the properties and creating a table row for each property that has not been previously visited and that is set up for display. For each property, you then recursively call the *Display* HTML helper. You can style every little piece of markup at will. In this example, I'm introducing a new table-level style called *display-table*. Figure 4-11 shows the results.

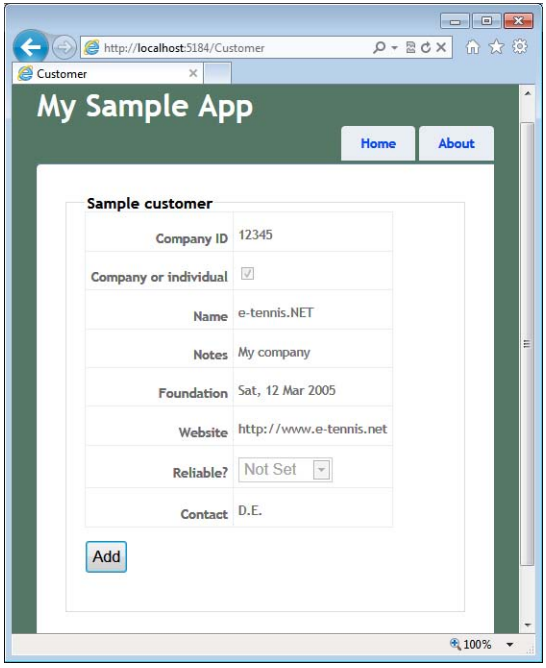

FIGURE 4-11 A tabular default display template.

The editor template is a bit more sophisticated because it takes care of validation errors and required fields. The template has an extra column to indicate which fields are required (based on

the *Required* annotation), and each editor is topped with a label for validation messages. Here is the source code:

```
@inherits System.Web.Mvc.WebViewPage
@if (Model == null)
    <span>@ViewData.ModelMetadata.NullDisplayText</span>
else {
    <table cellpadding="0" cellspacing="0" class="editor-table">
    @foreach (var prop in ViewData
                        .ModelMetadata
                        .Properties
                        .Where(pm => pm.ShowForDisplay && !ViewData.TemplateInfo.Visited(pm)))
    {
        <tr>
            <td>
                <div class="editor-label">
                    @prop.GetDisplayName()
                </div>
            </td>
            <td width="10px"> @(prop.IsRequired ? "*" : "") </td>
            <td>
                <div class="editor-field">
                        <span>@Html.Editor(prop.PropertyName)</span>
                        <span>@Html.ValidationMessage(prop.PropertyName, "*")</span>
                </div>
            </td>
        </tr>
    }
    </table>
}
```

Figure 4-12 shows the editor in action.

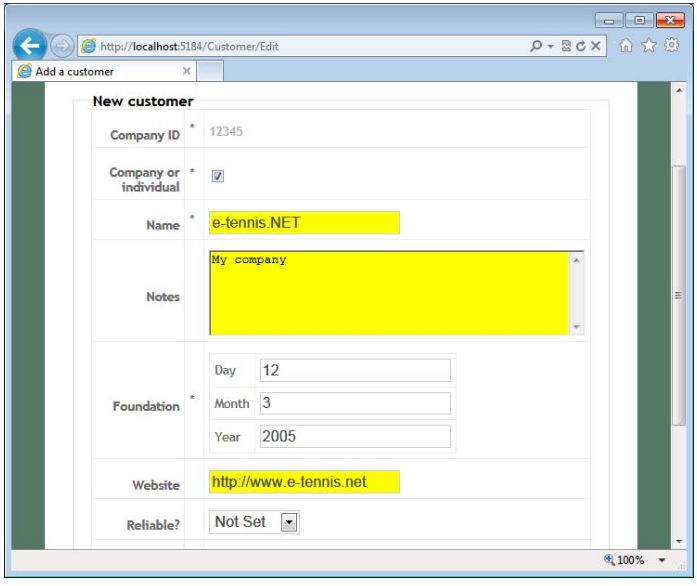

FIGURE 4-12 A tabular default editor template.

Dealing with Nested Models

One final point to cover is the expected behavior of the templates when the model type has nested models. The code shown so far recursively digs out properties of nested models and renders them as usual. Imagine you have the following model:

```
public class CustomerViewModel
{
   ...
   public ContactInfo Contact {get; set;}
}

public class ContactInfo
{
   public String FullName {get; set;}
   public String PhoneNumber {get; set;}
   public String Email {get; set;}
}
```

Figure 4-13 shows the results.

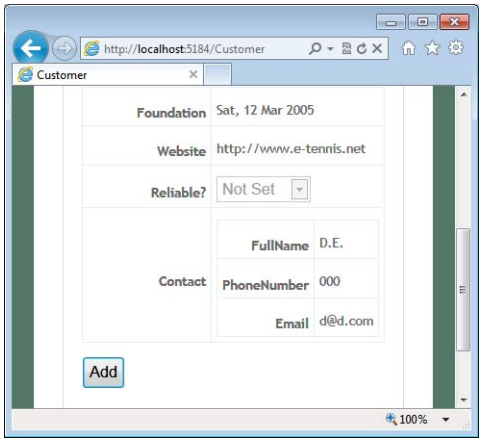

FIGURE 4-13 Nested models.

As a developer, you can control this behavior to some extent. By adding an extra *if* branch to the code of the template, you can limit the nesting level to what you like (1 in this example):

```
@if (Model == null)
    <span>@ViewData.ModelMetadata.NullDisplayText</span>
else
if (ViewData.TemplateInfo.TemplateDepth > 1)
{
    <span>@ViewData.ModelMetadata.SimpleDisplayText</span>
}
else
{ ... }
```

The property *TemplateDepth* measures the allowed nesting level. With this code enabled, if the nesting level is higher than 1, you resort to a simplified representation of the child model. This representation can be controlled by overriding *ToString* on the child model type:

```
public class ContactInfo
{
    ...
    public override string ToString()
    {
        return "[ContactInfo description here]";
    }
}
```

Alternatively, you can select one column and use that for rendering. You select the column using the *DisplayColumn* attribute on the child model type:

```
[DisplayColumn("FullName")]
public class ContactInfo
{
   ...
}
```

The attribute *DisplayColumn* takes precedence over the override of *ToString*.

Note As you've seen, the *HiddenInput* attribute gives you a chance to hide a given member from view while uploading its content through a hidden field. If you just want the helpers to ignore a given member, you can use the *ScaffoldColumn* attribute and pass it a value of *false*.

Input Validation

Programming books are full of statements like "Input is evil" and, of course, a chapter mostly dedicated to input forms can't leave out a section on input validation. If you want to discuss it in terms of what's really a necessity, you should focus on server-side validation and think about the best technology available that allows validation in the most effective way for your application *and* business domain.

Having said that, however, there's really no reason for skipping a client-side validation step that would simply prevent patently invalid data from making it to the server and consume a few valuable CPU cycles. So you're likely interested in both client-side and server-side validation. Even in moderately complex web applications, however, validation applies at least at two levels—to validate the input received from the browser, and to validate the data the back end of your system is going to store.

These two levels *might* sometimes be just one, but this is not what *typically* happens in the real world and outside books and tutorials about some cool pieces of technology. Unless your domain model is essentially aligned one-to-one with the storage, and the user interface is mostly CRUD, you have to consider (and make plans for) two levels of validation: presentation and business. The technology to use is an architectural choice and, guess what, it depends on the context.

In the rest of this chapter, I'll review a few options for validation in the presentation layer. Most of the space is reserved for a bunch of other data annotations attributes that are well integrated with the ASP.NET MVC plumbing. Toward the end, I'll also briefly summarize options you have to build your own validation layer for the business layer. (This topic, however, belongs in an entirely different, more architecture-oriented book, such as *Microsoft .NET: Architecting Applications for the Enterprise*, which I wrote with Andrea Saltarello (Microsoft Press, 2008).

Using Data Annotations

As mentioned, data annotations are a set of attributes you can use to annotate public properties of any .NET class in a way that any interested client code can read and consume. Attributes fall into two main categories: display and validation. We've just discussed the role that display attributes play with metadata providers in ASP.NET MVC. Before we dig out validation attributes, however, let's learn a bit more about data validation in ASP.NET MVC.

Validation Provider Infrastructure

ASP.NET MVC comes with a built-in layer that provides validation services to controllers. Such a validation layer follows the golden rule of ASP.NET MVC—prefer convention over configuration. In other words, you end up taking advantage of this layer without needing to write any validation-specific code; instead, you just declare what you need.

As you saw in Chapter 3, controllers receive their input data through the model-binding subsystem. The model binder maps request data to model classes and, in doing so, it validates input values against validation attributes set on the model class.

Validation occurs through a provider. The default validation provider is based on data annotations. The default validation provider is the *DataAnnotationsModelValidatorProvider* class. Let's see which attributes you can use that the default validation provider understands.

Table 4-5 lists the most commonly used data annotation attributes that express a condition to verify on a model class.

TABLE 4-5 Data annotation attributes for validation

Attribute	Description
Compare	Checks whether two specified properties in the model have the same value.
CustomValidation	Checks the value against the specified custom function.
EnumDataType	Checks whether the value can be matched to any of the values in the specified enumerated type.
Range	Checks whether the value falls in the specified range. It defaults to numbers, but it can be configured to consider a range of dates, too.
RegularExpression	Checks whether the value matches the specified expression.
Remote	Makes an Ajax call to the server, and checks whether the value is acceptable.
Required	Checks whether a non-null value is assigned to the property. It can be configured to fail if an empty string is assigned.
StringLength	Checks whether the string is longer than the specified value.

All of these attributes derive from the same base class—*ValidationAttribute*. As you'll see in a moment, you can also use this base class to create your own custom validation attributes.

You use these attributes to decorate members of classes being used in input forms. For the whole mechanism to work, you need to have controller methods that receive data in complex data types, as shown here for the *Memo* controller:

```
public ActionResult Edit()
{
    var memo = new Memo();
    return View(memo);
}

[HttpPost]
public ActionResult Edit(Memo memo)
{
    // ModelState dictionary contains error messages
    // for any invalid value detected according to the annotations
    // you might have in the Memo model class.

    return View(memo);
}
```

The model binder object edits the *ModelState* dictionary while binding posted values to the *Memo* model class. For any invalid posted value being mapped to an instance of the *Memo* class, the binder automatically creates an entry in the *ModelState* dictionary. Whether the posted value is valid or not depends on the outcome returned by the currently registered validation provider. The default validation provider bases its response on the annotations you might have set on the *Memo* model class. Finally, if the next view makes use of *ValidationMessage* helpers, error messages show up automatically. This is exactly the case if you use *EditorForModel* to create the input form.

Decorating a Model Class

The following listing shows a sample class—the aforementioned *Memo* class—that makes extensive use of data annotations for validation purposes:

```
public class Memo : ViewModelBase
{
    public Memo()
    {
        Created = DateTime.Now;
        Category = Categories.Work;
    }

    [Required]
    [StringLength(100)]
    public String Text { get; set; }

    [Required]
    [Range(1, 5]
    public Int32 Priority { get; set; }

    [Required]
    public DateTime Created { get; set; }

    [EnumDataType(typeof(Categories))]
    [Required]
    public Categories Category { get; set; }

    [StringLength(50, MinimumLength=4)]
    [RegularExpression(@"\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}\b")]
    public String RelatedEmail { get; set; }
}

public enum Categories
{
   Work,
   Personal,
   Social
}
```

Given the class, you should expect to receive error messages if the *Text* property is longer than 100 characters or if it is left empty. Likewise, the *Priority* members must be an integer between 1 and 5 (extremes included) and the *Created* date cannot be omitted. The *RelatedEmail* member can be empty; if given any text, however, the text has to be between 4 and 50 characters long and matching the regular expression. Finally, the *Category* member must contain a string that evaluates to one of the constants in the *Categories* enumerated type. Figure 4-14 shows the validation of a sample memo.

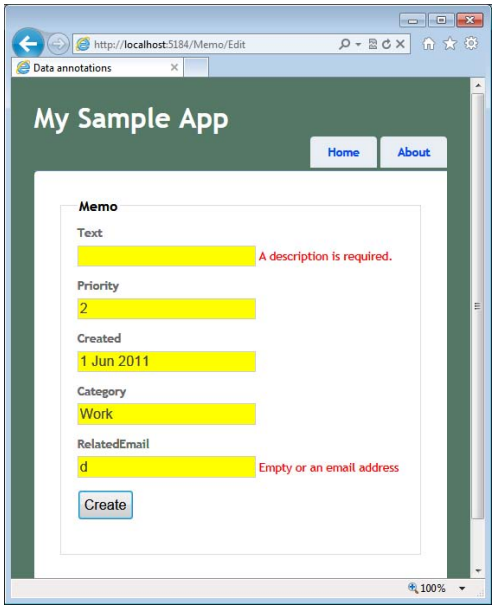

FIGURE 4-14 Validation messages.

Dealing with Enumerated Types

You might wonder why the Category field is being edited as a plain string. It would be smarter if a drop-down list could be provided. That won't just happen for free. The *EnumDataType* is recognized by the validation provider, which ensures the value belongs to the enumeration; it is ignored by editors. If you want a drop-down list with enumerated values, you need to write a *String.cshtml* custom template and place it in an EditorTemplates folder. Because enumerated types are paired to strings, you overwrite *String.cshtml* (or *.aspx*) to change the way in which an enumeration is edited. In the code, you determine the markup based on the actual type of the *Model* property. Here's the simple code you need:

```
@model Object
@if (Model is Enum)
{
   <div class="editor-field">
      @Html.DropDownList("", new SelectList(Enum.GetValues(Model.GetType())))
      @Html.ValidationMessage("")
   </div>
}
else
{
    @Html.TextBox("", ViewData.TemplateInfo.FormattedModelValue,
          new { @class = "text-box single-line" })
}
```

Figure 4-15 shows the same form when the just-created editor template is used.

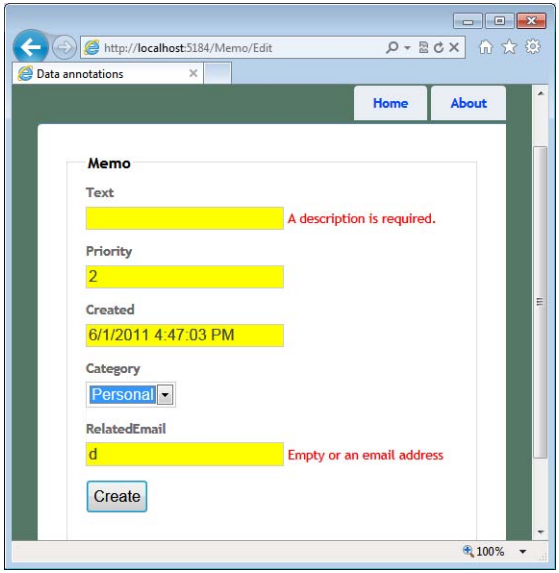

FIGURE 4-15 Mixing validation and display data annotations.

Controlling Error Messages

In Figure 4-15, you see some messages that attempt to explain what went wrong. You have total control over those messages. Each attribute has an *ErrorMessage* property for you to set the text. Note, though, that attribute properties can accept only constant strings.

```
[Required(ErrorMessage = "A description is required.")]
public String Text { get; set; }
```

You might not like the idea of having plain strings interspersed with class definition. A way to decouple the class definition from error messages is by using resources. Note that using resources allows you to make offline changes to the text without touching classes. It also allows localization, but it still doesn't give you programmatic control over the text being displayed. For programmatic control, the only option you have is editing the *ModelState* dictionary in the controller method, as shown here:

```
[HttpPost]
public ActionResult Edit(Memo memo)
{
    if (!ModelState.IsValid)
    {
        var newModelState = new ModelStateDictionary();

        // Create a new model-state dictionary with values you want to overwrite
        newModelState.AddModelError("Text", "...");
        newModelState.AddModelError("Priority", "...");
        ...
```

```
        // Merge your changes with the existing ModelState
        ModelState.Merge(newModelState);
    }
    return View(memo);
}
```

Basically, you create a new model-state dictionary and then merge it with one calculated during the model-binding stage. When you merge the dictionaries, the values in the new dictionary are copied into the *ModelState* dictionary, overwriting any existing value.

If having constant messages works for you and all you need to do is avoid magic strings in software, you can use a resource file which, incidentally, also sets the stage for easy localization. Each validation attribute accepts an error message expressed as a resource index:

```
[Required(ErrorMessageResourceName="MustSetPriority",
      ErrorMessageResourceType = typeof(Strings))]
[Range(1, 5, ErrorMessageResourceName="InvalidPriority",
      ErrorMessageResourceType=typeof(Strings))]
public Int32 Priority { get; set; }
```

You indicate the resource through a pair of arguments—the resource container data type, and the name of the resource. The former is expressed through the *ErrorMessageResourceType* property; the latter is expressed by *ErrorMessageResourceName*. When you add a resource file to an ASP.NET MVC project, the Visual Studio designer creates a container type that exposes strings as public members. This is the type to assign to the *ErrorMessageResourceType* property. By default, the name of this autogenerated type matches the name of the resource (*.resx*) file.

Advanced Data Annotations

The beauty of data annotations is that once you have defined attributes for a model type, you're pretty much done. Most of what follows happens automatically, thanks to the deep understanding of data annotations that the ASP.NET MVC infrastructure has. However, in this imperfect world none of the things you really need are completely free. So it is for data annotations, which cover a lot of relatively simple situations very well, but leave some extra work for you to do on your own in many realistic applications. Let's explore a few more advanced features you can build on top of data annotations.

Cross-Property Validation

The data annotations we've considered so far are attributes you use to validate the content of a single field. This is definitely useful, but it doesn't help that much in a real-world scenario where you likely need to validate the content of a property in light of the value stored in another. Cross-property validation requires a bit of context-specific code. The problem is, how and where do you write it?

The solution that springs to mind is probably not too far from the following outline:

```
public ActionResult Edit(Memo memo)
{
    if (ModelState.IsValid)
    {
        // If here, then properties have been individually validated.
        // Proceed with cross-property validation, and merge model-state dictionary
        // to reflect feedback to the UI.
        ...
    }
}
```

If the model state you receive from the binder is valid, all of the decorated properties passed the validation stage. So, taken individually, each property is OK. You proceed with context-specific, cross-property validation, add errors to a new state dictionary, and merge it to the existing *ModelState*.

You might be pleased to know that the *CustomValidation* attribute serves exactly this purpose and, overall, can be considered a shortcut for the approach I just suggested. Consider the following code:

```
[EnumDataType(typeof(Categories))]
[Required]
[CustomValidation(typeof(Memo), "ValidateCategoryAndPriority")]
public Categories Category { get; set; }
```

The *CustomValidation* attribute takes two parameters: a type and a method name. The type can also be the same model you're decorating. The method must be public and static with any of the following signatures:

```
public static ValidationResult MethodName(Categories category)
public static ValidationResult MethodName(Categories category, ValidationContext context)
```

Using the first overload is the same as defining custom validation logic for an individual value. It is sort of equivalent to creating a custom data annotations attribute—it's quicker to write but more rigid as far the signature is concerned. Much more interesting is the second overload. Through the *ValidationContext* parameter, in fact, you can get a reference to the model object and check as many properties as you like:

```
public static ValidationResult ValidateCategoryAndPriority(
                Categories category, ValidationContext context)
{
    // Grab the model instance
    var memo = context.ObjectInstance as Memo;
    if (memo == null)
        throw new NullReferenceException();

    // Cross-property validation
    if (memo.Category == Categories.Personal && memo.Priority > 3)
        return new ValidationResult("Category and priority are not consistent.");
    return ValidationResult.Success;
}
```

You can attach the *CustomValidation* attribute to an individual property as well as to the class. In the preceding example, the attribute is attached to *Category*, but it ensures that if the value is *Personal* the property *Priority* must be set to a value not higher than 3. If the validation fails, the entry in the model-state dictionary uses *Category* as the key. If you attach *CustomValidation* to the class, be aware that the validation is performed only if everything went fine for all individual properties. This is exactly the declarative counterpart of the pattern outlined earlier:

```
[CustomValidation(typeof(Memo), "ValidateMemo")]
public class Memo
{
   ...
}
```

Here's the signature of the method if the *CustomValidation* attribute is attached to the class:

```
public static ValidationResult ValidateMemo(Memo memo)
{
   ...
}
```

When you use *CustomValidation* at the class level, you have the problem of capturing error messages, because error messages are usually associated with properties. You can easily solve the issue by using the helper *Html.ValidationSummary*, which brings up all error messages regardless of the originating properties. I'll return to this topic—class level validation—later in the chapter.

Finally, the *Compare* attribute is available to serve far more quickly a specific cross-property scenario: when you need to ensure that the value of a property matches the value of another. The canonical example is retyping a new password:

```
[Required]
[DataType(DataType.Password)]
public String NewPassword {get; set;}

[Required]
[DataType(DataType.Password)]
[Compare("NewPassword"])
public String RetypePassword {get; set;}
```

The comparison is made using the *Equals* method as implemented on the specific type.

Creating Custom Validation Attributes

The *CustomValidation* attribute forces you to validate the value stored in the property without the possibility of adding any extra parameter. The problem here is not so much gaining access to other properties on the model object—which you can do through *ValidationContext*—but enriching the signature of the attribute that defines additional attribute-level parameters. For example, suppose you want to validate a number to ensure that it is an even number. Optionally, however, you want to

enable the attribute to check whether the number is also a multiple of 4. You want an attribute like the one shown here:

```
[EvenNumber(MultipleOf4=true)]
public Int32 MagicNumber {get; set;}
```

There's no way to pass an optional Boolean value in the signature recognized by *CustomValidation*. In the end, the difference between the *CustomValidation* attribute and a custom validation attribute is that the latter is designed to be (easily) reusable. Let's see how to write a custom data annotation attribute:

```
[AttributeUsage(AttributeTargets.Property)]
public class EvenNumberAttribute : ValidationAttribute
{
    // Whether the number has to be checked also for being a multiple of 4
    public Boolean MultipleOf4 { get; set; }

    public override Boolean IsValid(Object value)
    {
        if (value == null)
            return false;

        var x = -1;
        try
        {
            x = (Int32) value;
        }
        catch
        {
            return false;
        }

        if (x % 2 > 0)
            return false;
        if (!MultipleOf4)
            return true;

        // Is multiple of 4?
        return (x % 4 == 0);
    }
}
```

You create a class that inherits from *ValidationAttribute* and override the method *IsValid*. If you need extra parameters such as *MultipleOf4*, you just define public properties.

In ASP.NET MVC 3, you can create custom attributes that perform cross-property validation. All you do is override a slightly different overload of the *IsValid* method:

```
protected override ValidationResult IsValid(Object value, ValidationContext context)
```

Using the properties on the *ValidationContext* object, you can gain access to the entire model object and perform a full validation.

Enabling Client-Side Validation

All of the examples considered work across a postback or through an Ajax form. To use them with Ajax, you need to make just a small change to the view that displays the editor so that it uses *Ajax.BeginForm* instead of *Html.BeginForm*. In addition, the controller method should return a partial view instead of a full view, as shown here:

```
[HttpPost]
public ActionResult Edit(Memo memo)
{
    if (Request.IsAjaxRequest())
        return PartialView("edit_ajax", memo);
    return View(memo);
}
```

Here's how to convert the original view to an *edit_ajax* partial view:

```
@model DataAnnotations.ViewModels.Memo.Memo
@{
    Layout = "";  // Drop the master page
}

@using (Ajax.BeginForm("Edit", "Memo",
     new AjaxOptions() { UpdateTargetId="memoEditor" }))
{
    <div id="memoEditor">
    <fieldset>
        <legend>Memo</legend>
        @Html.EditorForModel()
        <p>
        <input type="submit" value="Create" />
        </p>
    </fieldset>
    </div>

}
```

In this way, your form is validated on the server but doesn't refresh the full page. At least for a few basic annotations, however, it can be helpful in enabling client-side validation so that if the data is patently invalid (for example, a required field is empty), no HTTP request is ever started.

To enable client-side validation, you need to perform a couple of easy steps. First, ensure that your *web.config* file contains the following:

```
<appSettings>
    <add key="ClientValidationEnabled" value="true" />
</appSettings>
```

Next, ensure that the following files are linked to the view: *MicrosoftAjax.js* and *MicrosoftMvcValidation.js*. That's it.

Turning on client-side validation injects a lot of script code and JSON data into the page. To avoid that, and replace the JSON blob with neat attributes, you can choose the unobtrusive way:

```
<appSettings>
   <add key="ClientValidationEnabled" value="true" />
   <add key="UnobtrusiveJavaScriptEnabled" value="true" />
</appSettings>
```

When you opt for unobtrusive JavaScript, you must replace the aforementioned JavaScript files with the following two: *jquery.validate.js* (the jQuery validation plugin) and *jquery.validate.unobtrusive.js*, or their minified versions. The unobtrusive file uses the jQuery Validation plugin for performing client-side validation.

For further clarity, Table 4-6 summarizes the script files required case by case when you're using rich client input forms.

TABLE 4-6 Script files required by input forms

Scenario	Files (minified form)
Ajax-based forms	*MicrosoftAjax.js, MicrosoftMvcAjax.js*
Unobtrusive Ajax-based forms	*jquery.js, jquery.unobtrusive-ajax.min.js*
Client-side validation	*MicrosoftAjax.js, MicrosoftMvcValidation.js*
Unobtrusive Client-side validation	*jquery.js, jquery.unobtrusive-ajax.min.js, jquery.validate.js, jquery.validate.unobtrusive.js*

The name *jquery.js* is meant to be a placeholder for the latest version of the jQuery library.

Important When you enable client-side validation, all built-in data annotation attributes gain a client-side behavior and, to the extent that it is possible, perform their validation in JavaScript within the browser. If the validation fails, no request is made to the web server. Custom validation attributes such as *EvenNumber*, however, don't automatically work in this way. To add client-side validation for custom attributes also, you need to implement an additional interface—*IClientValidatable*—which I'll cover in a moment.

Culture-Based, Client-Side Validation

When it comes to data-type validation, you likely have the additional problem of globalization. For example, consider the following:

```
public Decimal Price {get; set;}
```

The editor correctly handles the type and displays a default format of "0.00". If you enter "0,8" where the decimal separator is the comma, however, your input is rejected and the form won't post. As you can see, it is a problem of setting the correct culture on the client-side validation. The jQuery

Validation plugin defaults to the US culture on the client; on the server-side, instead, it depends on the value of the *Culture* property on the thread. (See Chapter 5, "Aspects of ASP.NET MVC Applications," for more details on localization and globalization.)

To support a specific culture on the client, you must first link the official jQuery globalization plugin as well as the script file for the specific culture you're interested in. Both files must be included after regular validation scripts. In addition, you must tell the globalization plugin which culture you intend to use. Finally, the validator plugin must be informed that it needs to use globalization information when parsing a number:

```
<script type="text/javascript">
    $.validator.methods.number = function (value, element) {
        if (Globalization.parseFloat(value)) {
            return true;
        }
        return false;
    }
    $(document).ready(function () {
        $.culture = jQuery.cultures['it-IT'];
        $.preferCulture($.culture.name);
        Globalization.preferCulture($.culture.name);
    });
</script>
```

With this code in place, you can enter decimals using the culture-specific settings. Taking this approach further, you can customize on a culture basis most of the client-side validation work.

Validating Properties Remotely

Validation might happen on the client, but you should see it as a way to save a few HTTP-heavy requests for pages. To be on the safe side, you should always validate any data on the server. However, to give users a nicer experience, you might want to perform a server-side validation without leaving the browser. The typical example is when users are registering to some service and enter a nickname. That name has to be unique, and uniqueness can be verified only on the server. Wouldn't it be cool if you could tell the user in real time whether the nickname is already taken? In this way, she could change it and avoid annoying surprises when finally posting the registration request.

Data annotations offer an attribute that helps code this feature—the *Remote* attribute. Attached to a property, the attribute invokes a method on some controller and expects a *Boolean* response. The controller method receives the value to validate plus an additional list of related fields. Here's an example:

```
[Remote("CheckCustomer", "Memo",
        AdditionalFields="Country",
        ErrorMessage="Not an existing customer")]
public String RelatedCustomer { get; set; }
```

When validating *RelatedCustomer* on the client, the code silently places a jQuery call to the method *CheckCustomer* on the *Memo* controller. If the response is negative, the specified error message is displayed:

```
public ActionResult CheckCustomer(String relatedCustomer)
{
    if (CustomerRepository.Exists(relatedCustomer))
        return Json(true, JsonRequestBehavior.AllowGet);
    return Json(false, JsonRequestBehavior.AllowGet);
}
```

The controller must return true/false wrapped up in a JSON payload. If additional fields have been specified, they are added to the query string of the URL and are subject to the classic model-binding rules of ASP.NET MVC. Multiple fields are separated by a comma. Here's a sample URL:

```
http://yourserver/memo/checkcustomer?relatedCustomer=dino&Country=...
```

The Ajax call is placed every time the input field loses the focus after having been modified. The property decorated with the *Remote* attribute honors the client-side validation contract and doesn't allow the form to post back until a valid value is entered.

Important Data annotations can be specified only statically at compile time. Currently, there's no way to read attributes from an external data source and bind them to model properties on the fly. To enable this feature, you probably need to consider replacing the default validation provider with a custom one that reads metadata for validation from a different source.

I've found this to be a real issue more in business-layer validation than in presentation. In business-layer validation, you likely need to inject business rules as requirements change, and more flexibility is therefore welcome. In business-layer validation, I often use the Enterprise Library Validation Application Block, which can read validation attributes from the configuration file. (I'll return to Enterprise Library in a moment.)

Self-Validation

Data annotations attempt to automate the validation process around data being posted from forms. Most of the time, you use only stock attributes and get error messages for free. In other cases, you create your own attributes at a bit higher development cost, but still what you do is create components that fit nicely in an existing infrastructure. In addition, data annotations are designed to work mostly at the property level. In ASP.NET MVC 3, you can always access the entire model via the validation context; however, ultimately, when the validation is complex, many developers prefer to opt for a handcrafted validation layer.

In other words, you stop fragmenting validation in a myriad of combinations of data annotations and move everything into a single place—a method that you call on the server from within the controller:

```
public ActionResult Edit(Memo memo)
{
   if (!Validate(memo))
      ModelState.AddModelError(...);
   ...
}
```

The *Memo* class might or might not have property annotations. As I see things, if you opt for self-validation, for reasons of clarity you should just stop using data annotations. In any case, self-validation doesn't prevent you from using data annotations as well.

The *IValidatableObject* Interface

When it comes to building a layer of self-validation, you can unleash your creativity and choose the application model that best suits you. ASP.NET MVC 3 attempts to suggest an approach. Basically, ASP.NET MVC guarantees that any model class that implements the *IValidatableObject* interface is automatically validated, with no need for the developer to call out validation explicitly. The interface is shown here:

```
public interface IValidatableObject
{
    IEnumerable<ValidationResult> Validate(ValidationContext validationContext);
}
```

If the interface is detected, the *Validate* method is invoked by the validation provider during the model-binding step. The parameter of type *ValidationContext* makes available the entire model for any sort of cross-property validation.

Important If the model is also decorated with data annotations, the *Validate* method is not invoked if some of the properties are not in a valid state. (To avoid pitfalls, I suggest that you drop annotations entirely if you opt for *IValidatableObject*.)

Using *IValidatableObject* is functionally equivalent to using the *CustomValidation* attribute at the class level. The only difference is that with *IValidatableObject* you can implement a validation layer with a single method, using your own architecture and remaining independent of data annotations.

Benefits of Centralized Validation

When you have really complex validation logic in which cross-property validation is predominant, mixing per-property validation and class-level validation might result in an unpleasant experience for the end user. As mentioned, class-level validation won't fire until properties are individually validated.

This means that users initially see a few error messages related to properties and are led to think that they are safe once these errors are fixed. Instead, they might get a whole new bunch of errors because of cross-property validation. Users won't have any clue about these other possible errors until they show up. To avoid this, if class-level validation is predominant, just focus on that and drop per-property validation.

To implement class-level validation, choosing between *IValidatableObject* or *CustomValidation* at the class level is entirely your call. For years, I used my own interface, which looked nearly identical to today's *IValidatableObject*.

The *IClientValidatable* Interface

Custom validation attributes won't produce any visible effect when client-side validation is enabled. In other words, a custom attribute is not able, per se, to run any client-side code to attempt to validate values in the browser. It doesn't mean, however, that this ability can't be added. It just takes more code. Here's how to extend a custom attribute to enable it on the client.

```
[AttributeUsage(AttributeTargets.Property)]
public class ClientEvenNumberAttribute : ValidationAttribute, IClientValidatable
{
    ...

    // IClientValidatable interface members
    public IEnumerable<ModelClientValidationRule> GetClientValidationRules(
            ModelMetadata metadata, ControllerContext context)
    {
       var errorMessage = ErrorMessage;
       if (String.IsNullOrEmpty(errorMessage))
            errorMessage = (MultipleOf4 ? MultipleOf4ErrorMessage : EvenErrorMessage);

       var rule = new ModelClientValidationRule
                      {
                         ValidationType = "iseven",
                         ErrorMessage = errorMessage
                      };
       rule.ValidationParameters.Add("multipleof4", MultipleOf4);
       yield return rule;
   }
}
```

You simply make it implement a new interface—the *IClientValidatable* interface. The method returns a collection of validation rules. Each rule is characterized by an error message, a JavaScript function name used for validation, and a collection of parameters for the JavaScript code. Names of parameters and functions should be lowercase.

Next, you need to write some JavaScript code that performs the validation on the client. Preferably, you should do this in unobtrusive JavaScript mode and use jQuery and the jQuery validation plugin. Here's the JavaScript you need to enable the previous *EvenNumber* attribute to work on the client too:

```
$.validator.addMethod('iseven', function (value, element, params) {
    var mustBeMultipleOf4 = params.multipleof4;
    if (mustBeMultipleOf4)
        return (value % 4 === 0);
    return (value % 2 === 0);
});

$.validator.unobtrusive.adapters.add('iseven', ['multipleof4', 'more_parameters'],
    function (options) {
        options.rules['iseven'] = options.params;
        options.messages['iseven'] = options.message;
    });
```

The *addMethod* function registers the validation callback for the *iseven* rule. The validation callback receives the value to validate and all parameters previously added to the rule in the implementation of *IClientValidatable*.

In addition, you need to specify adapters to generate JavaScript-free markup for the validation scenario. To add an adapter, you indicate the name of the function, its parameters, and error message.

You don't need anything else; if it doesn't work, first check that all required scripts are available and then check function and parameter names in JavaScript code and in the C# implementation of *IClientValidatable*. Be aware that you won't be receiving any JavaScript error if something goes wrong.

Dynamic Server-Side Validation

Validation is about conditional code so, in the end, it is a matter of combining a few *if* statements and returning Booleans. Writing a validation layer with plain code and without any ad hoc framework or technology might work in practice, but it might not be a great idea in design. Your resulting code would be hard to read and hard to evolve, even though some recently released fluent-code libraries are making it easier.

Subject to real business rules, validation is a highly volatile matter, and your implementation must account for that. In the end, it's not simply about writing code that validates, but code that is open to validating the same data against different rules. Data Annotations are a possible choice; another valid choice is the Validation Application Block (VAB) in the Microsoft Enterprise Library.

Data Annotations and VAB have a lot in common. Both frameworks are attribute-based, and both frameworks can be extended with custom classes representing custom rules. In both cases, you can define cross-property validation. Finally, both frameworks have a validator API that evaluates an instance and returns the list of errors. So what's the difference?

Data Annotations are part of the .NET Framework and don't need any separate download. Enterprise Library is a separate download, which is largely simplified by an appropriate NuGet package. It might not be a big deal in a large project, but it's still an issue because it might require additional approval in some corporate scenarios.

In my opinion, VAB is superior to Data Annotations in one regard—it can be fully configured via XML rulesets. An XML ruleset is an entry in the configuration file where you describe the validation you want. Needless to say, you can change things declaratively without even touching your code. Here's a sample ruleset:

```
<validation>
  <type assemblyName="..." name="Samples.DomainModel.Customer">
    <ruleset name="IsValidForRegistration">
      <properties>
        <property name="CompanyName">
          <validator type="NotNullValidator" />
          <validator lowerBound="6" lowerBoundType="Ignore"
                     upperBound="40" upperBoundType="Inclusive"
                     messageTemplate="Company name cannot be longer ..."
                     type="StringLengthValidator" />
        </property>
        <property name="Id">
          <validator type="NotNullValidator" />
        </property>
        <property name="PhoneNumber">
          <validator negated="false"
                     type="NotNullValidator" />
          <validator lowerBound="0" lowerBoundType="Ignore"
                     upperBound="24" upperBoundType="Inclusive"
                     negated="false"
                     type="StringLengthValidator" />
        </property>
        ...
      </properties>
    </ruleset>
  </type>
</validation>
```

A ruleset lists the attributes you want to apply to a given property on a given type. In code, you validate a ruleset as follows:

```
public virtual ValidationResults ValidateForRegistration()
{
    var validator = ValidationFactory
            .CreateValidator<Customer>("IsValidForRegistration");
    var results = validator.Validate(this);
    return results;
}
```

The preceding method applies the validators listed in the *IsValidForRegistration* ruleset to the specified instance of the class. (The *ValidateForRegistration* method is expected to be a method on an entity class of yours.)

Note Validation is twofold: sometimes you want to yell out when invalid data is passed; sometimes you want to collect errors and report that to other layers of code. In .NET 4, Code Contracts are another technology you want to take into account for validation. Unlike annotations and VAB, however, Code Contracts just check conditions and then throw at first failure. You need to use a centralized error handler to recover from exceptions and degrade gracefully.

In general, I recommend using Code Contracts in a domain entity only to catch potentially severe errors that can lead to inconsistent states. For example, it makes sense to use Code Contracts in a factory method. In this case, if the method is passed patently invalid data, you want it to throw an exception. Whether you also use Code Contracts in the setter methods of properties is your call. I prefer to take a softer route and validate via attributes.

Summary

Input forms are common in any web application, and ASP.NET MVC applications are no exception. For a while, there was a sentiment in the industry that ASP.NET MVC was not well suited to support data-driven applications because they required a lot of data entry and validation. Ultimately, ASP.NET MVC measures up nicely to the task. It does use a different set of tools than Web Forms, but it is still effective and to the point.

ASP.NET MVC offers templated helpers for autogenerated, repetitive forms and server-side and client-side forms of input validation. Input validation can be largely streamlined if you build your user interface around view-model objects annotated with display and validation attributes. These attributes—known as data annotations—are recognized by the ASP.NET MVC infrastructure (model metadata and validation providers), and they're processed to produce templated helpers and feedback messages for the user.

With this chapter, we almost covered all the fundamentals of ASP.NET MVC programming. In the next chapter, we'll cover what remains to be dealt with, including localization, security, and integration with intrinsic objects, primarily *Session* and *Cache*.

PART II

ASP.NET MVC Software Design

CHAPTER 5

Aspects of ASP.NET MVC Applications

A multitude of rulers is not a good thing. Let there be one ruler, one king.

—Homer

There's a lot more to a web application than just a request/response sequence. If it were as simple as that, with a well-done set of ASP.NET MVC controllers you would be all set. Unfortunately, things are a bit more complex. An effective ASP.NET MVC application results from the effective consideration and implementation of various aspects, including Search Engine Optimization (SEO), state management, error handling, and localization, just to name a few (and fairly important) ones.

This chapter is a collection of distinct and, to some extent, self-contained topics, each touching on an aspect that many ASP.NET MVC applications out there already have or are considering to have.

Important Not every web application needs to manage session or global state. Likewise, not every application is written for an international audience or makes a point of being highly ranked by search engines. Having said that, though, there's no application I can think of that needs none of these aspects.

ASP.NET Intrinsic Objects

ASP.NET MVC works and thrives on top of the classic ASP.NET infrastructure. To a good extent, ASP.NET MVC can be considered as a specialization of the classic ASP.NET runtime environment that just supports a different application and programming model. ASP.NET MVC applications have full access to any built-in components that populate the ecosystem of ASP.NET, including *Cache*, *Session*, *Response*, and the authentication and error-handling layer.

Nothing is different in ASP.NET MVC in the way in which these components can be accessed. But what about the way these components should be used from within ASP.NET MVC? Describing that is precisely the purpose of this chapter.

Note In ASP.NET—and in this book—the expression *intrinsic objects*, or just *intrinsics*, is frequently used to indicate the whole set of fundamental objects wrapped up by *HttpContext*. The list includes *HttpRequest*, *HttpResponse*, *HttpSessionState*, *Cache*, and the objects that identify the logged-on user.

SEO and HTTP Response

Let's start our analysis of ASP.NET intrinsic objects with the *HttpResponse* object. The object is used to describe the HTTP response being sent back to the browser at the end of request processing. The public interface of the *HttpResponse* object allows you to set cookies and content type, append headers, and pass instructions to the browser regarding the caching of the response data. In addition, the *HttpResponse* object allows redirecting to other URLs.

To customize any aspects of the response stream, you write a custom action result object and make a controller method return an instance of your type instead of *ViewResult* or any other predefined typed deriving from *ActionResult*. I'll discuss this in detail in Chapter 8, "Customizing ASP.NET MVC Controllers."

SEO is an extremely valid reason to pay more attention to the features of the ASP.NET *HttpResponse* object. Permanent redirection is the first practical programming aspect to take into account.

Permanent Redirection

In ASP.NET, when you invoke *Response.Redirect* you return to the browser an HTTP 302 code indicating that the requested content is now available from another specified location. Based on that, the browser makes a second request to the specified address and gets any content. A search engine that visits your page, however, takes the HTTP 302 code literally. The actual meaning of the HTTP 302 status code is that the requested page has been *temporarily* moved to a new address. As a result, search engines don't update their internal tables, and when someone later clicks to see your page, the engine returns the original address. As a result, the browser receives an HTTP 302 code and needs to make a second request to finally get to display the desired page.

If the redirection is used to convey requests to a given URL, permanent redirection is a better option because it represents a juicier piece of information for a search engine. To set up a permanent redirection, you return the HTTP 301 response code. This code tells search agents that the location has been permanently moved. Search engines know how to process an HTTP 301 code and use that information to update the page URL reference. The next time they display search results that involve the page, the linked URL is the new one. In this way, users can get to the page quickly and a second

roundtrip is saved. The following code shows what's required to arrange a permanent redirection programmatically:

```
void PermanentRedirect(String url, Boolean endRequest)
{
    Response.Clear();
    Response.StatusCode = 301;
    Response.AddHeader("Location", url);
    ...

    // Optionally end the request
    if (endRequest)
       Response.End();
}
```

In ASP.NET 4, the *HttpResponse* class features a new method for such a thing. It is named *RedirectPermanent*. You use the method in the same way you used the classic *Response.Redirect*, except that this time the caller receives an HTTP 301 status code. For the browser, it makes no big difference, but it is a key difference for search engines.

Before the release of ASP.NET MVC 3, many developers created their own action result type to perform a permanent redirection. In ASP.NET MVC 3, things are much easier because a new *Boolean* member named *Permanent* has been added to the *RedirectResult* type. The whole thing is wrapped by the *RedirectPermanent* method on the *Controller* class that you'll likely use:

```
public ActionResult Index()
{
   ...
   return RedirectPermanent(url);
}
```

As mentioned, I'll return to the topic of advanced customization of action results in Chapter 8.

Devising Routes and URLs

A huge difference between ASP.NET Web Forms and ASP.NET MVC is that in the latter URLs look more like commands you send to the Web application than server paths to pages and resources. Because the routing mechanism is under your total control as a developer, you are responsible for devising URLs for your application properly.

If you have SEO in mind, devising URLs properly mostly means guaranteeing URL uniqueness. One of the primary purposes of a search engine is determining how relevant the content pointed to by a given URL is. Of course, a given piece of information is much more relevant if it can be found only in one place and through a unique URL. Sometimes, however, even if the content is unique, it can be reached through multiple, subtly different, URLs. In this case, you run the risk of getting a lower ranking from search engines and, worse yet, having references to that portion of your site lost in the last result pages and hardly noticed by potential visitors. The problem here does not have much to do

with storage and page content, but with the shape and format of URLs. Even though the World Wide Web Consortium (W3C) suggests you consider using case-sensitive URLs, from an SEO perspective using single-case (and lowercase) URLs are a better choice. If you can keep all of your URLs lowercase, that adds consistency to the site while reducing duplicate URLs.

What about inbound links?

Well, there's not much you can do to avoid having external sites link to pages in your site using the case they prefer. Most likely, they will just copy your URLs, thus repeating the same case you have chosen. If this is not the case, you can always force a permanent redirect via an HTTP module that intercepts the *BeginRequest* event. Forcing all inbound links to use the same case saves you from splitting traffic across multiple URLs instead of concentrating all of it on a single URL with a higher rank. (We can call this strategy "Unite and Conquer" as opposed to the "Divide and Conquer" strategy that is so popular in other software scenarios.)

To address this problem, the *canonical URL* format also has been defined. The canonical URL describes your idea of a URL in the form of a preferred URL scheme. All you do is add a *<link>* tag to the *<head>* section, as shown here:

```
<link rel="canonical" href="http://myserver.com/" />
```

If your site has a significant amount of content that can be accessed through multiple URLs, the canonical URL gives more information to search engines so that they can treat similar URLs as a single one and come to a more appropriate ranking of the content of the resource. A possible effect of the canonical URL feature (which has zero cost on your side) is that it can clear up the controversy between having or not having the trailing slash. With a canonical URL that defaults to either choice, it makes no difference to a search engine which one is actually linked.

The Trailing Slash

There are some SEO concerns related to the trailing slash. In particular, a search engine incorporates a filter that detects and penalizes duplicate content in search results. Duplicate content is any page (that is, any distinct URL) in the search results that is regarded as serving the same content as others. A URL with a trailing slash and the same URL without the trailing slash are, to the search engine's eyes, just two URLs serving the same content.

To serve the most relevant content possible to the user, a search engine tries to rank lower the pages that seem nearly the same as others. But this process can accidentally reduce the rank of good pages.

What about ASP.NET MVC and the routing system? Should you force a trailing slash?

Ultimately, an ASP.NET MVC application is entirely responsible for its URLs and, subsequently, for what a search engine will ask for. In a new application, it's ultimately up to you because your routes determine how the request is processed. Helpers that are used to generate URL in the markup tend to avoid trailing slashes, so let's say that not having trailing slashes is a more common solution in ASP.NET MVC. But keep in mind that the other approach is equally valid. In ASP.NET MVC, it's up to you to resolve (or not resolve) URLs with and without the trailing slash in the same way. You ultimately

decide about your page rank. Whether you use or don't use the trailing slash is not as important as being consistent with whatever choice you make.

If you're porting an existing site to ASP.NET MVC, you might have a bunch of legacy URLs to maintain. You can install a custom route handler and permanently redirect (HTTP 301) from legacy URLs to new URLs. This approach works, but in practice it might take weeks for the search engine to physically update the internal tables of links to reflect all of your permanent redirects. Meanwhile, you might lose quite a bit of income because of that.

The search engine always likes to deal with the existing URLs. In this case, you might want to install a rewrite module in Microsoft Internet Information Services (IIS) to map an ASP.NET MVC URL to a legacy one. The following post provides some details: *http://www.hanselman.com/blog/ASPNETMVCAndTheNewIIS7RewriteModule.aspx*.

Managing the Session State

All real-world applications of any shape and form need to maintain their own state to serve users' requests. Web applications are no exception. However, unlike other types of applications, Web applications need special system-level tools to achieve the result. The reason for this peculiarity lies in the stateless nature of the underlying protocol that web applications still rely upon. As long as HTTP remains the transportation protocol for the web, all applications will run into the same trouble—figuring out the most effective way to persist state information.

In ASP.NET, the *HttpSessionState* class provides a dictionary-based model of storing and retrieving session-state values. The class doesn't expose its contents to all users operating on the application at a given time. Only the requests that originate in the context of the same session—that is, generated across multiple page requests made by the same browser instance—can access the session state. The session state can be stored and published in a variety of ways, including in a web farm or web garden scenario. By default, though, the session state is held within the ASP.NET worker process.

Using the *Session* Object

As an ASP.NET MVC developer, you have no technical limitations on your way to the intrinsic *Session* object. You have just the same issues and benefits as a developer of an ASP.NET Web Forms application. The ASP.NET MVC infrastructure uses the session state internally, and you can do so also in your code. In particular, the ASP.NET MVC infrastructure uses the session state to persist the content of the *TempData* dictionary, as you saw in Chapter 4, "Input Forms."

So if you feel the need to store data across sessions, you store it and then read it back through the familiar *Session* object:

```
public ActionResult Config()
{
   Session[StateEntries.PreferredTextColor] = "Green";
   ...
}
```

As you can see, nothing is different from classic ASP.NET programming. Just keep in mind that the session dictionary is a name/value collection, so it requires plain strings to identify entries. Using constants in code is a good technique to prevent nasty errors. When you read from session state, casting and null-checking are unavoidable:

```
var preferredTextColor = "";
var data = Session[StateEntries.PreferredTextColor];
if (data ! null)
      preferredTextColor = (String) data;
```

Any data stored in the session state is returned as an *Object*.

Never Outside the Controller

The most important aspect that relates to accessing the session state in ASP.NET MVC is that you should be using it only from within the controller. Generally speaking, data stored in the session state can be consumed in either of two ways. It can be used to drive some back-end calculation on input data, or it can be passed as is to the view.

In the latter case, make sure you copy individual values to proper members on the view model class or to whatever predefined view data structures you use (*ViewData* or *ViewBag*). Technically speaking, you could access the session state (and other intrinsic objects) also from within a Razor or ASPX view. Although any code like this works, you should avoid doing this so that you can preserve a strong separation of concerns between controllers and views. The golden rule is that the view receives from the outside world whatever data it needs to incorporate.

Note The golden rule just mentioned holds also for render actions, which are special controller methods you invoke directly from the view. Render actions add a bit of logic to your views but don't break the separation between controllers and views—the view keeps on having its only contact in the controller.

Caching Data

Caching indicates the application's ability to save frequently used data to an intermediate storage medium. In a typical web scenario, the canonical intermediate storage medium is the web server's memory. However, you can design caching around the requirements and characteristics of each application, thus using as many layers of caching as needed to reach your performance goals.

In ASP.NET, built-in caching capabilities come through the *Cache* object. The *Cache* object is created on a per-AppDomain basis, and it remains valid while that AppDomain is up and running. The object is unique in its capability to automatically scavenge the memory and get rid of unused items. Cached items can be prioritized and associated with various types of dependencies, such as disk files, other cached items, and database tables. When any of these items change, the cached item is automatically invalidated and removed. Aside from that, the *Cache* object provides the same

dictionary-based and familiar programming interface as *Session*. Unlike *Session*, however, the *Cache* object does not store data on a per-user basis.

The Bright Side and Dark Side of the Native *Cache* Object

Using the *Cache* object in ASP.NET MVC is just the same as in ASP.NET Web Forms. You should be accessing the *Cache* object preferably from a controller class or from infrastructure classes such as *global.asax*, as shown here:

```
protected void Application_Start()
{
   var data = LoadFromSomeRepository();
   Cache[CacheEntries.CustomerRecords] = data;
   ...
}
```

To read from cache, you use the same pattern you just saw for session state, check for nullness, and cast to a known valid type:

```
var customerRecords = new CustomerRecords();
var data = Cache[CacheEntries.CustomerRecords];
if (data ! null)
      customerRecords = (CustomerRecords) data;
```

If cached data needs to be displayed in the view, you just add that data to the view model class for the specific view. Alternatively, you define render actions on the controller class.

So far, so good. What's the dark side of caching in ASP.NET MVC?

As mentioned, the *Cache* object is limited to the current AppDomain and, subsequently, to the current process. This design was fairly good a decade ago, but it shows more and more limitations today. If you're looking for a global repository object that, like *Session*, works across a web farm or web garden architecture, the native *Cache* object is not for you. You have to resort to AppFabric Caching services or to some commercial frameworks (such as ScaleOut or NCache) or open-source frameworks (such as Memcached or Shared Cache).

The issue, however, is that the implementation of the *Cache* object is not based on the same popular provider model as session state. This means you can't replace the cache data holder if you need to scale it up, not even for testing your controllers.

Injecting a Caching Service

The currently recommended approach for caching in ASP.NET MVC consists of injecting caching capabilities into the application. You define a contract for an abstract caching service and make your controllers work against this contract. Injection can happen the way you like—either through your favorite Inversion-of-Control (IoC) framework or via an ad hoc controller constructor. The latter is jokingly called "the poor man's dependency injection approach."

The following code shows a minimal, but functional, example of how to abstract the caching layer:

```
public interface ICacheService
{
    Object Get(String key);
    void Set(String key, Object data);
    Object this[String key] { get; set; }
    ...
}
```

You are responsible for making this interface as rich and sophisticated as you need. For example, you might want to add members to support dependencies, expiration, and priorities. Just keep in mind that you are not writing the caching layer for the entire ASP.NET subsystem, but simply writing a segment of your application. In this respect, the YAGNI principle (You Aren't Gonna' Need It) holds true as never before.

Any controller class that needs caching will accept an *ICacheService* object through the constructor:

```
private ICacheService _cacheService;
public HomeController(ICacheService cacheService)
{
    _cacheService = cacheService;
}
```

The next step consists of defining a few concrete implementations of the *ICacheService* interface. The concrete type simply uses a particular cache technology to store and retrieve data. Here's the skeleton of a class that implements the interface using the native ASP.NET *Cache* object:

```
public class AspNetCacheService : ICacheService
{
   private readonly Cache _aspnetCache;
   public AspNetCacheService()
   {
       if (HttpContext.Current != null)
           _aspnetCache = HttpContext.Current.Cache;
   }

   public Object Get(String key)
   {
       return _aspnetCache[key];
   }

   public void Set(String key, Object data)
   {
       _aspnetCache[key] = data;
   }

   public object this[String name]
   {
       get { return _aspnetCache[name]; }
       set { _aspnetCache[name] = value; }
   }
   ...
}
```

Finally, let's complete the controller's code that uses this caching service so that the service can be properly injected:

```
public class HomeController
{
   private readonly ICacheService _cacheService;
   public HomeController() : this(new AspNetCacheService())
   {
   }
   public HomeController(ICacheService cacheService)
   {
      _cacheService = cacheService;
   }
   ...
}
```

In this way, your controller classes that need caching are not tightly bound to a specific implementation of a cache object and are, at a minimum, easier to test.

A Better Way of Injecting a Caching Service

Injecting a cache service into a controller instance requires that a new cache service be created for each request. Because the caching service is a plain wrapper around an existing and external cache data holder (for example, the ASP.NET Cache object or AppFabric Caching Services), this will not have a huge impact on the performance of the request. The cache data holder, in fact, is initialized only once, at application startup.

Can you manage things so that you save your application a few CPU cycles per request and expose a global cache object that is, in turn, based on a replaceable provider? You bet! Try adding the following code to *global.asax*:

```
public class MvcApplication : HttpApplication
{
    ...

    // Internal reference to the cache wrapper object
    private static ICacheService _internalCacheObject;

    // Public method used to inject a new caching service into the application.
    // This method is required to ensure full testability.
    public void RegisterCacheService(ICacheService cacheService)
    {
       _internalCacheObject = cacheService;
    }

    // Use this property to access the underlying cache object from within
    // controller methods. Use this instead of native Cache object.
    public static ICacheService CacheService
    {
        get { return _internalCacheObject;  }
    }
```

```
    protected void Application_Start()
    {
        ...

        // Inject a global caching service
        RegisterCacheService(new AspNetCacheService());

        // Store some sample app-wide data
        CacheService["StartTime"] = DateTime.Now;
    }
}
```

When using this approach, you have no need to inject the actual caching service in the selected controller for each request. The caching service is initialized and injected once at application startup. Controllers use a public static method on the application object (as defined in *global.asax*) to access the cache:

```
var data = MvcApplication.CacheService[...]
```

The public method *RegisterCacheService* preserves testability. In any unit test where you want to test a cache-aware controller, you place the following call in the preliminary phase of the unit test:

```
MvcApplication.RegisterCacheService(new FakeCacheService());
```

Next, you proceed with calling the controller method, which will transparently use the fake cache service.

Distributed Caching

Easier testing is not the only benefit you gain by using a public contract for your cache-related tasks. By keeping your controllers aware of a cache interface—not a cache implementation—you keep them able to work with any object that provides caching services through the specified interface. In other words, you can replace the aforementioned *AspNetCacheService* class with another similar-looking class that relies on a different caching infrastructure.

For example, you can transparently plug in a cache service based on a distributed framework, such as Microsoft AppFabric Caching Services or another open-source or commercial framework. Nearly all of these frameworks expose a public API that is analogous to the basic ASP.NET *Cache* object, so there's not really much work that you need to do to set it up beyond configuration.

Nicely enough, if you're just not interested in distributed caching, you can still replace the ASP.NET native *Cache* object with the newest *MemoryCache* object introduced in the .NET Framework 4 with the precise purpose of giving caching capabilities to any .NET applications. For this reason, the class is defined outside the ASP.NET realm in a brand new assembly named *System.Runtime.Caching*. The *MemoryCache* object works like the ASP.NET *Cache* except that it throws an exception if you try to store null values. The *MemoryCache* class inherits from a base class, *ObjectCache*. By deriving your own cache object, you can take control of the internal storage and management of cached data. This is not a recommended approach for everybody, but it's definitely possible. Note, however, that

ObjectCache and derived types are not designed to provide the behavior of a distributed cache. If you intend to create your own distributed cache, the hard work of maintaining multiple caches in sync is entirely up to you.

Caching the Method Response

The classic mechanism of ASP.NET output caching survived in ASP.NET MVC too. It takes the form of the *OutputCache* attribute you can attach to a controller method or to the controller class to affect all action methods:

```
[OutputCache(Duration=10, VaryByParam="None")]
public ActionResult Index()
{
   ...
}
```

The *Duration* parameter indicates, in seconds, how long the method's response should stay cached in memory. The *VaryByParam* attribute, on the other hand, indicates how many distinct versions of the response you should cache—one for each distinct value of the specified property. If you use *None*, you tell the system you don't want multiple versions of the same method's response.

Table 5-1 lists the properties supported by the attribute. They are a subset of the attributes of the *@OutputCache* directive of ASP.NET. Missing attributes are those limited to ASP.NET user controls.

TABLE 5-1 Properties of the *OutputCache* attribute

Attribute	Description
CacheProfile	Associates a response with a group of output-caching settings specified in the *web.config* file.
Duration	The time, in seconds, that the response is cached.
Location	Specifies the location (browser, proxy, or server) to store the response of the method call. The attribute takes its value from the *OutputCacheLocation* enumeration.
NoStore	Indicates whether to send a *Cache-Control:no-store* header to prevent browser-side storage of the response.
SqlDependency	Indicates a dependency on the specified table on a given Microsoft SQL Server database. Whenever the contents of the table changes, the response is removed from the cache.
VaryByContentEncoding	Content encoding by which you intend to differentiate cached responses.
VaryByCustom	A semicolon-separated list of strings that lets you maintain distinct cached copies of the response based on the browser type or user-defined strings.
VaryByHeader	A semicolon-separated list of HTTP headers.
VaryByParam	A semicolon-separated list of strings representing query string values sent with GET method attributes, or parameters sent using the POST method.

These properties communicate the same output-caching infrastructure of the ASP.NET runtime and work exactly the same in ASP.NET Web Forms.

Note It should be pretty obvious, but let's state it clearly. Methods for which you set the *OutputCache* attribute are not executed for requests that hit the server when a valid cached response is available.

Partial Output Caching

In ASP.NET MVC 3, partial caching is no longer limited to the entire response of the method. You can now attach the *OutputCache* attribute to child actions also. A child action is a method on the controller that the view can call back using the *Html.RenderAction* helper. The *RenderAction* helper can invoke any method on the controller; some methods, however, can be marked as exclusive child actions. You do this using the *ChildActionOnly* attribute:

```
[ChildActionOnly]
public ActionResult RenderSiteMap()
{
   ...
}
```

Such a method is clearly designed to render a small section of the view. It is logically equivalent to a user control in ASP.NET Web Forms. By decorating this method with the *OutputCache* attribute, you can cache the response for the specified duration. This feature was not supported in versions of ASP.NET MVC prior to version 3, even though some workarounds were found and have been blogged about.

Finally, note that this is the only way you have to implement partial output caching in ASP.NET MVC. You have no way to configure partial views to cache their output, as you can do with user controls in Web Forms.

Error Handling

Because ASP.NET MVC works on top of the classic ASP.NET runtime environment, you can't expect to find a radically different infrastructure to handle run-time errors. This means that you can still opt for the classic ASP.NET strategy of mapping any 400 or 500 HTTP status codes to a specific URL that provides error information. Switching programmatically to a different page in cases of error requires an HTTP redirect. You control the mapping through the <*customErrors*> section of the *web.config* file.

As I see things, while functional, this approach is less than ideal in ASP.NET MVC, where you can easily switch to an error interface by simply changing the name of the view template invoked by the controller.

Let's see what ASP.NET MVC has to offer when it comes to error handling. Overall, error handling in ASP.NET MVC spans two main areas: the handling of program exceptions and route exceptions. The former is concerned with catching errors in controllers and views; the latter is more about redirection and HTTP errors.

Handling Program Exceptions

Most of the code you write in ASP.NET MVC applications resides in controller classes. In a controller class, you can deal with possible exceptions in a number of equivalent ways: handling exceptions directly via *try/catch* blocks, overriding the *OnException* method, and using the *HandleError* attribute.

Handling Exceptions Directly

In the first place, you can use local *try/catch* blocks to protect yourself against a possible exception in a specific section of the code. This is the approach that gives you maximum flexibility, but it does so at the cost of adding some noise to the code. I'm not one to question the importance of exception handling—which, by the way, is the official .NET approach to error handling. However, the fact is, the presence of *try/catch* blocks makes reading code a bit harder. For this reason, I always welcome any alternative solution that aims to centralize exception-handling code to the extent it is legitimately possible.

To execute a piece of code with the certainty that any (or just some) exceptions it might raise will be caught, you use the following code:

```
try
{
    // Your regular code here
    ...
}
catch
{
    // Your recovery code for all exceptions
    ...
}
```

The preceding snippet will catch any exceptions originated by the code in the *try* block. Because of its extreme generality, it doesn't put you in the position of implementing an effective recovery strategy. The sample code snippet can have a number of variations and extensions. For example, you can list multiple *catch* blocks, one per each significant exception. You can also add a *finally* block, which will finalize the operation and run regardless of whether the execution flow went through the *try* block or the *catch* block:

```
try
{
    // Your regular code here
    ...
}
catch(NullReferenceException nullReferenceException)
{
    // Your recovery code for the null reference exception
    ...
}
catch(ArgumentException argumentException)
{
```

```
    // Your recovery code for the argument exception
    ...
}
finally
{
    // Finalize here, but DON'T throw exceptions from here
    ...
}
```

Exceptions will be listed from the most specific to the least specific. From a *catch* block, you are also allowed to swallow the exception so that other top-level modules will never know about it. Alternatively, you can handle the situation gracefully and recover. Finally, you can do some work and then re-throw the same exception or arrange a new one with some extra or modified information in it.

When it comes to writing direct code for handling exceptions, you might want to keep a few guidelines in mind. First, the *catch* block is fairly expensive if your code gets into it. Therefore, you should use the *catch* block judiciously—use it only when it's really needed and without overcatching.

Furthermore, you should never throw an exception as an instance of the root *System.Exception* class. It is strictly recommended that you try to use built-in exception types—such as *InvalidOperationException*, *NullReferenceException*, and *ArgumentNullException*—whenever these types apply. You should resist the temptation of having your own exceptions all the way through, although for program errors you should consider defining your own exceptions. In general, you should be very specific with exceptions. *ArgumentNullException* is more specific than *ArgumentException*. An exception comes with a message, and the message must be targeted to developers and, ideally, localized.

For a long time, Microsoft said you should derive your exception classes from *System.ApplicationException*. More recently, there's been a complete turnaround on this point: the new directive says the opposite. You should ignore *ApplicationException* and derive your exception classes from *Exception* or other, more specific, built-in classes. And don't forget to make your exception classes serializable.

Note Another key guideline from Microsoft about exception handling is to use built-in types for general errors (for example, null references, invalid arguments, and I/O or network exceptions) and create application-specific types for exceptions that are specific to the application you're creating.

Overriding the *OnException* Method

As discussed in Chapter 1, "ASP.NET MVC Controllers," the execution of each controller method is governed by a special system component known as the *action invoker*. An interesting aspect of the

default action invoker is that it always executes controller methods within a *try/catch* block. Here's some pseudocode that illustrates the behavior of the default action invoker:

```
try
{
    // Try to invoke the action method
    ...
}
catch(ThreadAbortException)
{
    throw;
}
catch(Exception exception)
{
   // Prepare the context for the current action
   var filterContext = PrepareActionExecutedContext( ..., exception);

   // Go through the list of registered action filters, and give them a chance to recover
   ...

   // Re-throw if not completely handled
   if (!filterContext.ExceptionHandled)
   {
       throw;
   }
}
```

If an exception is thrown at some point during the method's execution or during the rendering of the view, the control passes to the code in the *catch* block as long as the exception is not a *ThreadAbortException*. Handling the exception entails looping through the list of registered action filters and giving each its own chance to fix things. At the end of the loop, if the exception has not been marked as handled, the caught exception is re-thrown.

An *action filter* is a piece of code that can be registered to handle a few events fired during the execution of an action method. One of these system events is fired when the invoker intercepts an exception. (I'll cover action filters in detail in Chapter 8.) To have your own code added to the list of filters, the simplest thing you can do is override the *OnException* method on the controller class:

```
protected override void OnException(ExceptionContext filterContext)
{
   ...
}
```

Defined in any of your controller classes (or in a base class of yours), this method is always invoked when an unhandled exception occurs in the course of the action method.

Note No exception will be caught by *OnException* that originates outside the realm of the controller, such as null references resulting from a failure in the model-binding layer or a not-found exception resulting from an invalid route. I'll tackle this aspect in a more specific way later on in the chapter.

Overall, there's just one reason for you to override *OnException* in a controller class: you want to control the behavior of the system and degrade gracefully in the case of an exception. This means that the code in *OnException* is given the power of controlling the entire response for the request that just failed. The method receives a parameter of type *ExceptionContext*. This type comes with a *Result* property of type *ActionResult*. As you can guess, the property refers to the next view or action result. If the code in *OnException* omits setting any result, the user won't see any error screen (neither the system's nor the application's); the user will see just a blank screen. Here's the typical way to implement *OnException*:

```
protected override void OnException(ExceptionContext filterContext)
{
    // Let other exceptions just go unhandled
    if (filterContext.Exception is InvalidOperationException)
    {
        // Default view is "error"
        filterContext.SwitchToErrorView();
    }
}
```

The *SwitchToErrorView* method is an extension method for the *ExceptionContext* class coded as shown here:

```
public static void SwitchToErrorView(this ExceptionContext context,
                                     String view = "error", String master = "")
{
    var controllerName = context.RouteData.Values["controller"] as String;
    var actionName = context.RouteData.Values["action"] as String;
    var model = new HandleErrorInfo(context.Exception, controllerName, actionName);
    var result = new ViewResult
                        {
                            ViewName = view,
                            MasterName = master,
                            ViewData = new ViewDataDictionary<HandleErrorInfo>(model),
                            TempData = context.Controller.TempData
                        };
    context.Result = result;

    // Configure the response object
    context.ExceptionHandled = true;
    context.HttpContext.Response.Clear();
    context.HttpContext.Response.StatusCode = 500;
    context.HttpContext.Response.TrySkipIisCustomErrors = true;
}
```

Altogether, this code provides an effective framework-level *try/catch* block that is not limited to catching the exception but switches to an error view. In the code just shown, the default error view is *error*, but you can change it at will, as well as its layout.

Using the *HandleError* Attribute

As an alternative to overriding the *OnException* method, you can decorate the class (or just individual methods) with the *HandleError* attribute or any custom class that derives from it:

```
[HandleError]
public class HomeController
{
    ...
}
```

Note that *HandleError* is a bit more than a simple attribute; it is an action filter. As such, it contains executable code and is not limited to providing meta information to some other modules. In particular, *HandleError* implements the *IExceptionFilter* interface:

```
public interface IExceptionFilter
{
   void OnException(ExceptionContext filterContext);
}
```

The interface is the same one that all controllers implement. On the base *Controller* class, however, *OnException* has just an empty body.

Internally, *HandleError* implements the method *OnException* using a piece of code very similar to *SwitchToErrorView*. The only difference is in the conditions under which the change of view is operated. The *HandleError* attribute traps the specified exceptions only if they haven't been previously fully handled and are not resulting from child actions. To control the exceptions you want to handle, you do as follows:

```
[HandleError(ExceptionType=typeof(NullReferenceException), View="SyntaxError")]
```

Each method can have multiple occurrences of the attribute, one for each exception you're interested in. The *View* and *Master* properties indicate the view to display after the exception. By default, *HandleError* switches to a view named *error*. (Such a view is purposely created by the Microsoft Visual Studio ASP.NET MVC standard template.)

Important For *HandleError* to produce any visible results in debug mode, you need to enable custom errors at the application level, as shown here:

```
<customErrors mode="On">
</customErrors>
```

If you leave on the default settings for the <*customErrors*> section of the configuration file, only remote users will get the selected error page. Local users (for example, developers doing some debugging) will receive the classic error page with detailed information about the stack trace as produced by the normal ASP.NET exception handler.

Just like any other action filter in ASP.NET MVC, the *HandleError* attribute can be automatically applied to any method of any controller class by registering it as a global filter. Incidentally, this is exactly what happens within the code generated by the Visual Studio tooling for ASP.NET MVC 3. Here's an excerpt from *Application_Start* in *global.asax*:

```
public class MvcApplication : System.Web.HttpApplication
{
    protected void Application_Start()
    {
        RegisterGlobalFilters(GlobalFilters.Filters);
        ...
    }

    public static void RegisterGlobalFilters(GlobalFilterCollection filters)
    {
        filters.Add(new HandleErrorAttribute());
    }
}
```

Introduced in ASP.NET MVC 3, a global filter is an action filter that the default action invoker automatically adds to the list of filters before it invokes any action method. I'll have more to say on global action filters in Chapter 8.

Global Error Handling

For the most part, *OnException* and the *HandleError* attribute provide controller-level control over error handling. This means each controller that requires error handling must host some exception-handling code. More important than this, however, is that dealing with errors at the controller level doesn't ensure that you intercept all possible exceptions that might be raised around your application.

You can create a global error handler at the application level that catches all unhandled exceptions and routes them to the specified error view.

Global Error Handling from *global.asax*

Since the very first version of the ASP.NET runtime, the *HttpApplication* object—the object behind *global.asax*—has featured an *Error* event. The event is raised whenever an unhandled exception reaches the outermost shell of code in the application. Here's how to write such a handler:

```
void Application_Error(Object sender, EventArgs e)
{
   ...
}
```

You could do something useful in this event handler, such as sending an email to the site administrator or writing to the Microsoft Windows event log to say that the page failed to execute properly. Here's an example:

```
void Application_Error(Object sender, EventArgs e)
{
    var exception = Server.GetLastError();
    if (exception == null)
        return;

    var mail = new MailMessage { From = new MailAddress("automated@contoso.com") };
    mail.To.Add(new MailAddress("administrator@contoso.com"));
    mail.Subject = "Site Error at " + DateTime.Now;
    mail.Body = "Error Description: " + exception.Message;
    var server = new SmtpClient { Host = "your.smtp.server" };
    server.Send(mail);

    // Clear the error
    Server.ClearError();

    // Redirect to a landing page
    Response.Redirect("home/landing");
}
```

With the preceding code, the administrator will receive an email message (as shown in Figure 5-1), but the user will still receive a system error page. If you want to avoid that, after you manage the error you can redirect the user to a landing page. Finally, note that if the SMTP server requires authentication, you need to provide your credentials through the *Credentials* property of the *SmtpClient* class.

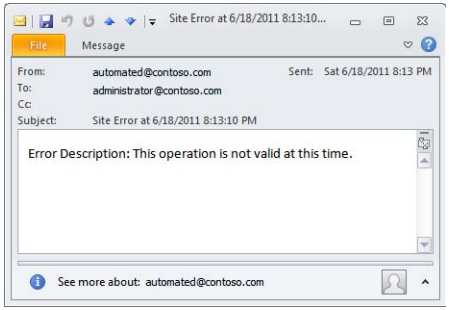

FIGURE 5-1 An email message to the site administrator.

Global Error Handling Using an HTTP Module

In ASP.NET, there's just one way to capture any fatal exceptions—writing a handler for the *HttpApplication* object *Error* event. This can be done in two ways, however. You can write code directly in the application's *global.asax* file, or you can plug a made-to-measure HTTP module to the *web.config* file. The HTTP module registers its own handler for the *Error* application event. The

two solutions are functionally equivalent, but the one based on the HTTP module can be enabled, disabled, and modified without recompiling the application. It is, in a way, less obtrusive.

When you consider a global error handler, you have a couple of ideas in mind—alerting the administrator about an exception, and logging the exception. Especially for the second task, an HTTP module seems like an easier-to-manage solution than having code in *global.asax*.

A tool that is very popular among ASP.NET developers is Error Logging Modules And Handlers (ELMAH). ELMAH is essentially made of an HTTP module that, once configured, intercepts the *Error* event at the application level and logs it according to the configuration of a number of back-end repositories. ELMAH comes out of an open-source project (*http://code.google.com/p/elmah*) and has a number of extensions, mostly in the area of repositories. ELMAH offers some nice facilities, such as a web page you can use to view all recorded exceptions and drill down into each of them. Any error-reporting system specifically designed for ASP.NET can't be, architecturally speaking, much different from ELMAH.

Intercepting Model-Binding Exceptions

A centralized error handler is also good at catching exceptions that originate outside the controller, such as exceptions caused by incorrect parameters. If you declare a controller method with one, say, integer argument and the current binder can't match any posted value to it, you get an exception. Technically, the exception is not raised by the model binder itself; it is raised by an internal component of the action invoker while preparing the method call. If this component doesn't find a value for a nonoptional method parameter, it just throws an exception.

This exception is not fired from within the controller code, but it still falls under the control of the overall *try/catch* block in the action invoker. Why doesn't a global (or local) *HandleError* do the trick of trapping the exception? It does, but only if you turn on the custom error flag in the *web.config* file. With the custom error flag disabled, your only chance to intercept a model-binding error is through the centralized error handler in *global.asax*. Figure 5-2 shows the page served gracefully to the user and the email message sent to the administrator.

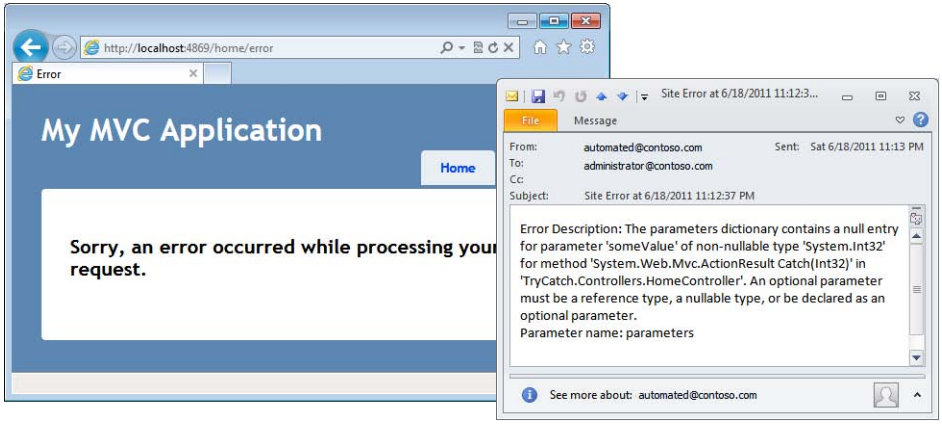

FIGURE 5-2 A model-binder exception trapped by the *Application_Error* handler.

Handling Route Exceptions

In addition to any detected program errors, your application might be throwing exceptions because the URL of the incoming request doesn't match any of the mapped routes—either because of an invalid URL pattern (an invalid action or controller name) or a violated constraint. In this case, your users get an HTTP 404 error. Letting users receive the default 404 ASP.NET page is something you might want to avoid for a number of reasons—primarily, to be friendlier to end users.

The typical solution enforced by the ASP.NET framework consists of defining custom pages (or routes in ASP.NET MVC) for common HTTP codes such as 404 and 403. Whenever the user types or follows an invalid URL, she is redirected to another page where some, hopefully, useful information is provided. Here's how to register ad hoc routes in ASP.NET MVC:

```
<customErrors mode="On">
    <error statusCode="404" redirect="/error/show" />
    ...
</customErrors>
```

This trick works just fine, and there's no reason to question it from a purely functional perspective. So where's the problem, then?

The first problem is with security. By mapping HTTP errors to individualized views, hackers can distinguish between the different types of errors that can occur within an application and use this information for planning further attacks. So you explicitly set the *defaultRedirect* attribute of the *<customErrors>* section to a given and fixed URL and ensure that no per-status codes are set.

A second issue with per-status code views has to do with SEO. Imagine a search engine requesting a URL that doesn't exist in an application that implements custom error routing. The application first issues an HTTP 302 code and tells the caller that the resource has been temporarily moved to another location. At this point, the caller makes another attempt and finally lands on the error page. This approach is great for humans, who ultimately get a pretty message; it is less than optimal from an SEO perspective because it leads search engines to conclude the content is not missing at all—just harder than usual to retrieve. And an error page is catalogued as regular content and related to similar content.

On the other hand, route exceptions are a special type of error and deserve a special strategy distinct from program errors. Ultimately, route exceptions refer to some missing content.

Dealing with Missing Content

The routing subsystem is the front end of your application and the door at which request URLs knock to get some content. In ASP.NET MVC, it is easy to treat requests for missing content in the same way as valid requests. No redirection and additional configuration are required if you create a dedicated controller that catches all requests that would go unhandled.

Catch-All Route

A common practice to handle this situation consists of completing the route collection in *global.asax* with a catch-all route that traps any URLs sent to your application that haven't been captured by any of the existing routes:

```
public static void RegisterRoutes(RouteCollection routes)
{
    // Main routes
    ...

    // Catch-all route
    routes.MapRoute(
      "Catchall",
      "{*anything}",
      new { controller = "Error", action = "Missing" }
    );
}
```

Obviously, the catch-all rule needs to go at the very bottom of the routes list. This is necessary because routes are evaluated from top to bottom and parsing stops at the first match found. The catch-all route maps the request to your application-specific *Error* controller. (You don't have an *Error* controller available out of the box in ASP.NET MVC, but it is highly recommended that you create one on your own.) The *Error* controller, in turn, looks at content and headers and then decides which HTTP code to return. Here's an example of such an *Error* controller:

```
public class ErrorController : Controller
{
    public ActionResult Missing()
    {
       HttpContext.Response.StatusCode = 404;
       HttpContext.Response.TrySkipIisCustomErrors = true;

       // Log the error (if necessary)
       ...

       // Pass some optional information to the view
       var model = ErrorViewModel();
       model.Message = ...;
       ...

       // Render the view
       return View(model);
    }
}
```

The *ErrorViewModel* class in the example is any view-model class you intend to use to pass data to the underlying view in a strongly typed manner. Using the *ViewData* dictionary is fine as well, and overall it's an acceptable compromise in this specific and relatively simple context.

By using an error controller, you can improve the friendliness of the application and optimize it for search engines. In fact, you actually serve a pretty user interface to users while returning a direct (that is, not redirected) error code to any callers.

Important A catch-all route is simply a route selected for a URL that didn't match any other routes. Many routes, however, are matched by the standard route, which overall is quite a generic *catch-almost-all* route. In other words, a URL like */foo* matches the default route and never reaches the catch-all route. Then, if it's missing a *Foo* controller, it results in a 404 error. To intercept 404 errors that occur because of an invalid controller name, you should override the controller factory. (I'll say more about this in Chapter 7.)

Skipping IIS Error-Handling Policies

In the preceding code snippet, the *Missing* method on the *ErrorController* class at some point sets to *true* the *TrySkipIisCustomErrors* property on the *Response* object. It is a property introduced with ASP.NET 3.5 that specifically addresses a feature of the IIS 7 integrated pipeline.

When an ASP.NET application (either Web Forms or ASP.NET MVC) runs under IIS 7 within an integrated pipeline, some of the ASP.NET configuration settings will be merged with the settings defined at the IIS level. (See Figure 5-3.)

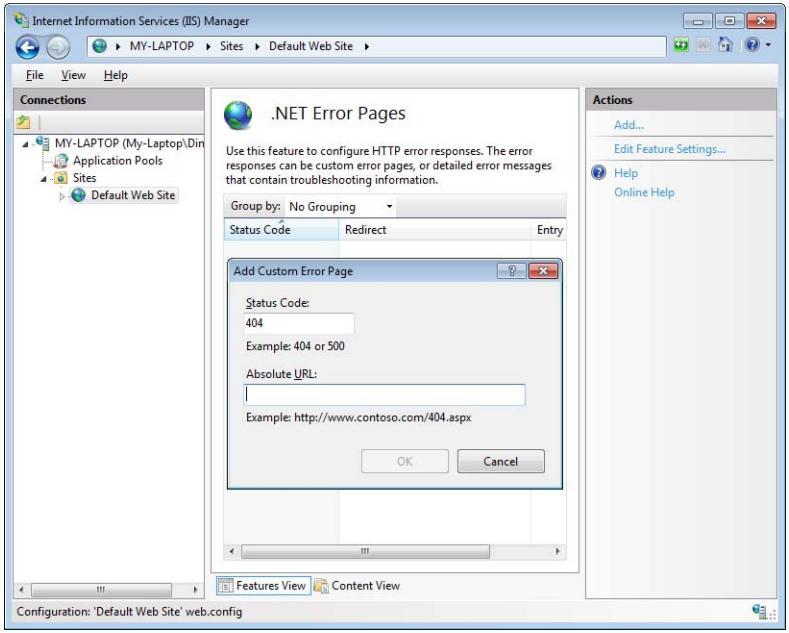

FIGURE 5-3 Defining custom error pages at the IIS level.

In particular, if error pages are defined in IIS for common HTTP status codes, in the default case these pages will take precedence over the ASP.NET-generated content. As a result, your application might trap an HTTP 404 error and serve a nice-looking ad hoc page to the user. Like it or not, your page never makes it to the end user because it is replaced by another page that might be set at the IIS level.

To make sure that the IIS error handling is always bypassed, you set the *TrySkipIisCustomErrors* property to *true*. The property is useful only for applications that run under IIS 7 in integrated pipeline mode. In integrated pipeline mode, the default value of the property is *false*. The implementation of the *HandleError* exception filter, for example, takes this aspect into careful consideration and sets the property to *true*.

Localization

The whole theme of localization is nothing new in the .NET Framework, and ASP.NET is no exception. You have had tools to write culture-specific pages since the very first version of ASP.NET. The beauty is that nothing has changed, so adding localization capabilities to ASP.NET MVC applications is neither more difficult nor different than in classic ASP.NET.

Considering localization from the perspective of an entire application with a not-so-short expectation of life, there are three aspects of it that need to be addressed: how to make (all) resources localizable, how to add support for a new culture, and how to use (or whether to use) databases as a storage place for localized information.

Using Localizable Resources

A localizable ASP.NET MVC view, as well as an ASP.NET Web Form, uses resources instead of hardcoded text and images to flesh out the user interface. After a resource assembly is linked to the application, the ASP.NET runtime selects the correct value at run time according to the user's language and culture. Even though the general idea of a resource hasn't changed since ASP.NET Web Forms, in ASP.NET MVC you probably don't want to use some of the bolted-on tooling built since older versions of Visual Studio. In particular, the distinction between folders for local and global resources, while still acceptable, makes little sense to me today.

No Global, No Local—Just Resources

In ASP.NET Web Forms, you create resource assemblies by simply creating ad hoc resource files in appropriate folders: *App_LocalResources* for resources local to a single group of pages, and *`App_GlobalResources* for resources visible from within all pages. The benefit of this separation is twofold: you can use specific methods on the *HttpContext* object and query for resource items (such as strings, menus, and images), and you can use special syntax to bind resource values to server control properties. For this reason, specific ASP.NET folders were introduced a few versions ago. In the context of an ASP.NET MVC application, I like to move toward a different approach that doesn't distinguish between global and local resources, but aims to organize resources in the most effective way for deployment, updates, and usage.

The definition of a resource, however, needs be reworked a bit.

Resources are just user interface items that you might want to adapt to a specific culture. Resources span the entire application and include *mostly* text to translate. Localizable resources, however, are certainly not limited to plain strings and are not even limited to server-side code. That's why I recommend forgetting about old-fashioned ASP.NET Web Forms practices and outlining your own resource management strategy specific to modern web applications. Such a strategy is based on two pillars: the organization of resource files and the granularity of the content.

What's the role of the RESX file?

An RESX file is ultimately an XML file that gets compiled on the fly by the Microsoft Visual Studio designer. As a developer, you have some control over the namespace and the access modifier of the class members. In other words, when you add a resource to the project you can choose whether to make all the properties public or internal (the default) and decide which namespace will group them. A *public* modifier is necessary if you're compiling resources in their own assembly. (See Figure 5-4.) Otherwise, you can opt for an *internal* modifier.

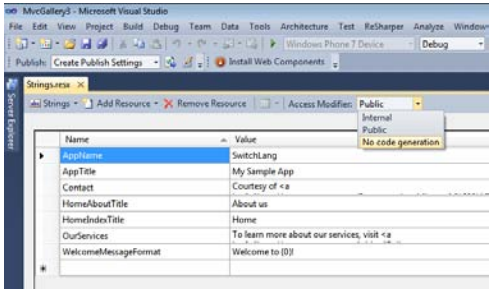

FIGURE 5-4 Selecting the access modifier for resource strings.

Also, make sure that in the project the resource file is associated with the Embedded Resource build action. (See Figure 5-5.)

All in all, the classic RESX resource file devised as a centralized repository of text, images, menus, and other auxiliary resources is probably a thing of the past. The RESX file is still used but mostly as a string repository for localizable text. A complex user interface might not simply be translated by replacing a word or two and a few bitmaps. You might want to use different cascading style sheets (CSS), if not different view templates. This makes the idea of an RESX single repository obsolete and, at the same time, puts pressure on us to create more powerful tools that can apply culture-based logic to the loading of views, CSS files, and images.

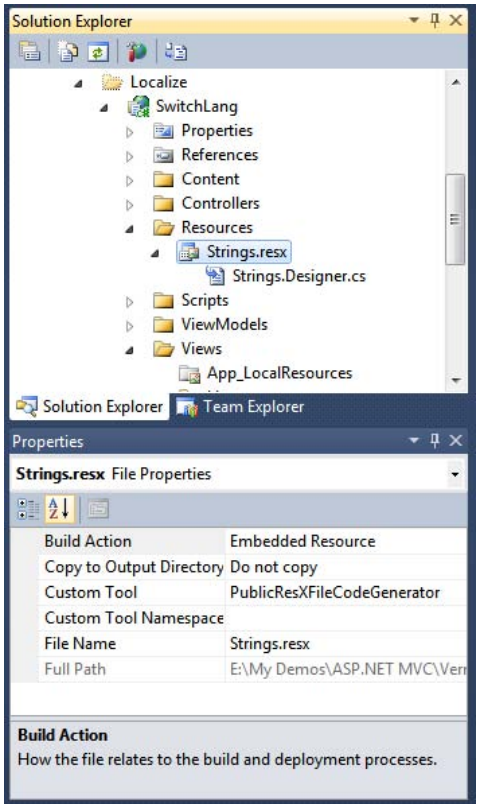

FIGURE 5-5 Adding a resource file as an embedded resource.

Important You can store an image or a text file (for example, a CSS file) into an RESX file. However, this approach ends up storing the files into the assembly. To edit style sheets or images, you must recompile the assembly. In my opinion, this makes managing auxiliary resources quite cumbersome. The alternative approach I'm presenting here is based on a very simple idea—extending the culture-detection mechanism that ASP.NET applies natively to RESX files to other individual resources, such as images, views, and CSS files.

Localizable Text

Localizable text is the only type of resource that can still comfortably sit in a classic RESX file. From what I have learned on the battlefield, having a single global file to hold all localizable resources turns into a not-so-pleasant experience even for a moderately complex application.

One issue is the size of the file, which grows significantly; another issue, which is even more painful, is the possible concurrent editing that multiple developers might be doing on the same file with the subsequent need for a continuous merge. However, I encourage you not to overlook the naming issue. When you have hundreds of strings that cover the entire application scope, how do you

name them? Many strings look the same or differ only in subtle points. Many strings are not entire strings with some sensible meaning; they often are bits and pieces of some text to be completed with dynamically generated content. Trust me: naming a few of them in the restricted context of only some pages is doable; handling hundreds of them for the entire application is really painful.

Overall, the best approach seems to be having multiple resource files. You might start with a resource file for each view, and then merge strings and other resources into a more global resource file as you find them referenced from multiple views.

In more concrete terms, this means creating a *Resources* folder in your project that contains multiple RESX files distributed in a bunch of subfolders organized according to some criteria. It could be per logical functional area or even on a per-view or per-controller basis. Figure 5-6 shows a Resources folder that groups multiple files.

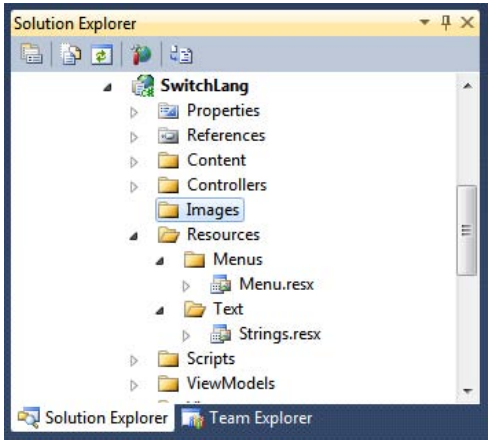

FIGURE 5-6 A custom Resources folder.

Any string you place in such RESX files is global and can be referenced from any view and controller class. Here's what you need in Razor:

```
@using SwitchLang.Resources.Text;
...
<p>@Strings.OurServices</p
```

The preceding expression guarantees that either the language-neutral value or the localized value is retrieved and displayed. The resource manager will pick up the right assembly resource for the current culture.

All RESX files that use the default language are compiled to the same assembly as the application. This is the case for files whose name doesn't include a culture reference, such as *errors.resx*, *strings.resx*, *menus.resx*, and so forth. Culture-specific resources are compiled in separate assemblies, one per culture. You localize text to a culture by adding a new RESX file named according to the pattern shown here:

```
filename.XX.resx
```

Here, the *XX* stands for the two-letter name of the culture—for example, *it*, *fr*, *en*, *es*, *de*, and the like. Those localized files are copies of the original (culture-neutral) file and just translate text into the language.

Important I also suggest you consider keeping even default resources in their own assembly. All you need to do is create a new class library project, drop the *Resources* folder in it (including localized versions), and reference the library from the main application.

In ASP.NET MVC, another effective way of organizing files within a project is grouping all resources under the default *Content* folder. In my current project, I have *Scripts*, *Css*, *Images*, and *Text* subfolders under *Content*. The *Text* subfolder contains RESX files and RESX files are limited to strings.

Localizable Files

I'm not sure about you, but I by far prefer to manage images, style sheets, and scripts as distinct files, not embedded in a compiled RESX file. Beyond the manageability point, referencing files from the resource section of an assembly requires some additional configuration work that must be kept in mind. I'll return to embedded files in a moment.

In ASP.NET MVC, it is common to use the *Url.Content* method to reference content files such as images or style sheets. The major benefit of this method is that it transparently converts a relative path to an absolute path. More specifically, the method understands the tilde (~) operator. When used in a URL, the tilde operator indicates the root path regardless of what it is and regardless of whether your application is deployed as a root application or a subapplication. For this reason, I always recommend you reference external files through *Url.Content*.

Wouldn't it be great if this method could handle localization too? You request the path for, say, *welcome.jpg* and you get back *welcome.it.jpg* if the current culture is IT. The method itself can't be altered; but writing an extension method that extends the *Url.Content* syntax is easy:

```
public static class UrlExtensions
{
    public static String Content(this UrlHelper helper,
                                 String contentPath, Boolean localizable=false)
    {
        var url = contentPath;
        if (localizable)
            url = GetLocalizedUrl(helper, url);
        return helper.Content(url);
    }

    public static String GetLocalizedUrl(UrlHelper helper, String resourceUrl)
    {
        var cultureExt = String.Format("{0}{1}",
            Thread.CurrentThread.CurrentUICulture.TwoLetterISOLanguageName,
            Path.GetExtension(resourceUrl));
        var url = Path.ChangeExtension(resourceUrl, cultureExt);
```

```
        // Check if localized URL exists, and return file.XX.jpg (or whatever)
        return VirtualFileExists(helper, url) ? url : resourceUrl;
    }

    public static Boolean VirtualFileExists(UrlHelper helper, String url)
    {
        var fullVirtualPath = helper.Content(url);
        var physicalPath = helper.RequestContext.HttpContext.Server.MapPath(fullVirtualPath);
        return File.Exists(physicalPath);
    }
}
```

Basically, the new *Content* method is a thin wrapper around *Url.Content*. The method checks whether a localized resource exists. If one does exist, the method returns the localized URL; otherwise, it returns the original URL. Here's an example of how to use the method:

```
@using MvcGallery.Extensions.Localization;
...
<link href="@Url.Content("~/Content/Site.css", localizable:true)"
      rel="stylesheet"
      type="text/css" />
```

Note that the type of the resource you reference (image, style sheet, or script) is not relevant; the *Content* method just processes URLs and changes the extension if conditions apply.

Referencing Embedded Files

When you embed a file (a script, CSS, or image) in the resource section of an assembly, you need to do some extra work to retrieve it. On the up side, however, you don't need to deploy the files separately; deploying the assembly is all you need to do.

There are two ways of embedding a resource in an assembly. You can add the resource file to an RESX file. You open up the RESX file, and use the interface of the designer to pick up an existing image or create a new one. The image is stored as a bitmap object in the resources of the assembly. You can retrieve this information using the *ResourceManager* class or the *GetGlobalResourceObject* method on the *HttpContext* class. (The latter only works if the RESX files is placed in the old-fashioned *App_GlobalResources* folder.) In both cases, you can't transform the content in the assembly into a URL you can attach to an HTML tag. One possibility is to create an ad hoc controller method like the one shown here:

```
public Object Image(String name)
{
    var bits = (Bitmap) HttpContext.GetGlobalResourceObject("AllResources", name);
    Response.ContentType = "image/jpeg";
    bits.Save(Response.OutputStream, ImageFormat.Jpeg);
    return bits;
}
```

The *<img>* tag looks like this:

```
<img alt="" src="/home/image?name=trees" />
```

A better approach consists of adding the resource to the assembly as an individual embedded resource. Figure 5-7 shows how to do it with a JPEG image. You add the image to your project and then change the build action to Embedded Resource.

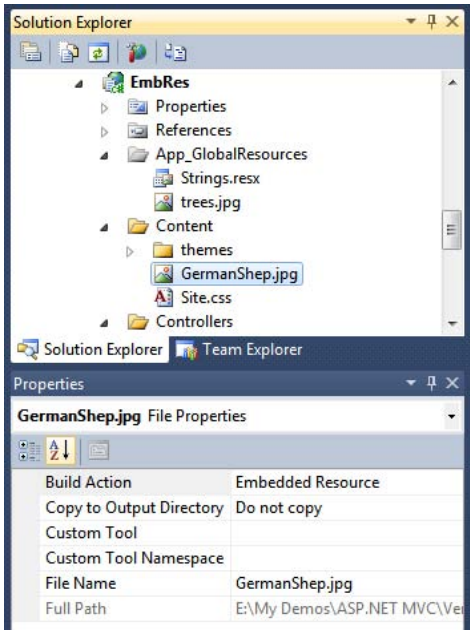

FIGURE 5-7 Adding an image as an embedded resource.

Next, and more importantly, you decorate your assembly as one that contains embedded resources. In the AssemblyInfo file under the Properties project folder, add the following:

```
[assembly: WebResource("EmbRes.Content.GermanShep.jpg", "image/jpg")]
```

The path is the just the fully qualified name of the resource. To retrieve the resource, you need some help from the ASP.NET *Page* class. The trick is easy to do if you use the ASPX view engine:

```
<img alt=""
     src="<%= Page.ClientScript.GetWebResourceUrl(typeof(MvcApplication),
                                          'EmbRes.Content.GermanShep.jpg'); %>" />
```

In Razor, this requires one extra step because there's no *Page* object that can be used. However, all you need to do is get yourself a dynamically created instance:

```
@{
    var p = new Page();
    var url = p.ClientScript.GetWebResourceUrl(typeof(MvcApplication),
                                               "EmbRes.Content.GermanShep.jpg");
}
...
<img alt="" src="@url" />
```

Another tricky point to consider is the first argument of *GetWebResourceUrl*. Setting it to the application's type works just fine. Figure 5-8 shows the image displayed.

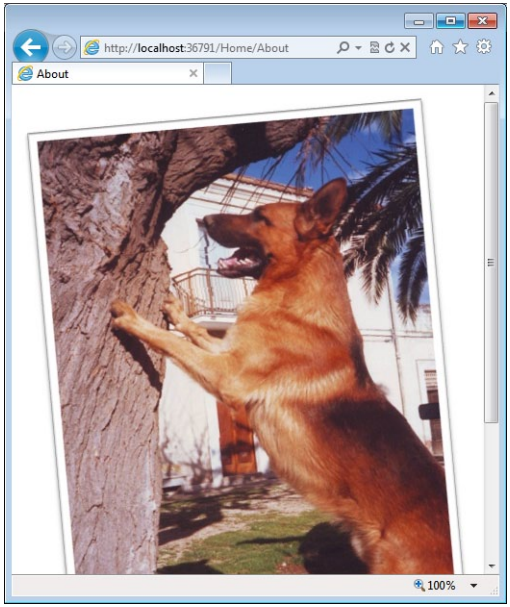

FIGURE 5-8 Displaying an image from embedded resources.

Localizable Views

Views are another part of the application that might need to be adapted to the current locale. In ASP.NET MVC, you call out the view from within an action method. In addition, each view invoked from the controller can include partial views that might need to be localized as well. This means that you need to add localization capabilities at two levels: using the action method and the extension methods commonly used to link partial views.

The best way to add localization logic to action methods is through an action filter. In the end, all you need to do is determine the name of the view algorithmically and call it. An action filter keeps the code clean and moves localization logic elsewhere, where it can be managed separately. I'll return to this particular action filter in Chapter 8. For now, let's focus on HTML extensions to invoke localized partial views.

In ASP.NET MVC, you use *Html.RenderPartial* when you just want to render a view; you use *Html.Partial* when you want to get back the HTML markup and write it yourself to the stream. Adding localization logic here is nearly the same process as we used earlier for resource files. Here's a new HTML extension method that extends *Partial*:

```
public static class PartialExtensions
{
    public static MvcHtmlString Partial(this HtmlHelper htmlHelper,
             String partialViewName, Object model, ViewDataDictionary viewData,
            Boolean localizable = false)
```

```
    {
        // Attempt to get a localized view name
        var viewName = partialViewName;
        if (localizable)
            viewName = GetLocalizedViewName(htmlHelper, viewName);

        // Call the system Partial method
        return System.Web.Mvc.Html.PartialExtensions.Partial(htmlHelper, viewName, model,
viewData);
    }

    public static MvcHtmlString Partial(this HtmlHelper htmlHelper, String partialViewName,
                                        Boolean localizable=false)
    {
        // Attempt to get a localized view name
        var viewName = partialViewName;
        if (localizable)
            viewName = GetLocalizedViewName(htmlHelper, viewName);

        // Call the system Partial method
        return htmlHelper.Partial(viewName, null, htmlHelper.ViewData);
    }

    public static String GetLocalizedViewName(HtmlHelper htmlHelper, String partialViewName)
    {
        var urlHelper = new UrlHelper(htmlHelper.ViewContext.RequestContext);
        return UrlExtensions.GetLocalizedUrl(urlHelper, partialViewName);
    }
}
```

The structure of the code is quite intuitive. The new method checks whether a localized view exists according to the established convention. If one does exist, the method proceeds and calls the original *Partial* method with the localized name, otherwise, everything proceeds as usual. The extra step can be controlled through the *localizable* Boolean parameter:

```
@using MvcGallery3.Extensions.Localization;
...
@Html.Partial("_aboutdetails", localizable:true)
```

The code for *RenderPartial* is nearly identical and can be found in the source code download.

Dealing with Localizable Applications

So far, you've seen what you can do to deal with individual resources that must be localized. Localizing an application, however, requires more than just localizing a collection of single resources. In particular, you should have a clear idea of what you intend to accomplish by localizing an application. Do you want your application to support multiple languages but to configure each language at setup time? Or do you want the user to be able to switch between a few predefined languages? Or, finally, is an auto-adapting application what you want to have? Let's examine the three scenarios.

Auto-Adapting Applications

An auto-adapting application is an application that in some way decides the culture to use based on user-provided information. Such an application supports a number of predefined cultures, and it falls back to the neutral culture when the detected locale doesn't match any of the supported cultures. The neutral culture is just the native default language of the application. Neutral resources are those that don't have any culture ID in the name.

The most characteristic aspect of an auto-adapting application is how the application determines the culture to use. This is commonly done in a couple of ways. The most common approach entails having the application read the list of accepted languages that the browser sends with each request. Another approach is based on geo-localization, where the server-side part of the application in some way gets the current location of the user (the IP address or just the client-side geo-localization) and chooses the culture accordingly. The first approach is gently offered by the ASP.NET runtime; the second approach requires some extra work on your part and likely uses some extra framework. Let's examine the first approach.

In ASP.NET, you use the *Culture* and *UICulture* properties to get and set the current culture. You do that on a per-request basis—for example, from within the constructor of each controller class or in the constructor of some base controller class. The property *Culture* governs all applicationwide settings, such as dates, currency, and numbers. The property *UICulture* governs the language being used to load resources. These string properties are publicly available from the view classes in both the ASPX and Razor view engines.

Note that the two properties default to the empty string, which means the default culture is the culture set on the web server. If culture properties are set to *auto*, the first preferred language sent by the browser through the *Accept-Languages* request header is picked up. To make your views automatically localized (if resources for that culture are available), you must add a line to the *web.config* file:

```
<system.web>
   ...
  <globalization culture="auto" uiCulture="auto" />
</system.web>
```

That's all you need to do (in addition to using localized resources) to get an auto-adapting localized application up and running.

Multilingual Applications

Another possible scenario is when you have a multilingual application that is deployed with multiple localized assemblies but is configured to use only one set of resources. Also, in this case you don't need to write ad hoc code anywhere. All you need to do is write the correct information in the *globalization* section of the *web.config* file:

```
<system.web>
   ...
  <globalization culture="it" uiCulture="it" />
</system.web>
```

In addition, of course, you need to have culture-specific resources available so that they can be invoked automatically by the ASP.NET framework.

Note When your application must be available in various languages but uses just one language at a time, you might also want to isolate all resources to RESX and other auxiliary files and then replace them with locale-specific files without recompiling the entire application. Frankly, having tried that, I don't much like this approach. The risk of accidentally overwriting files is high, confusion can grow unacceptably high, and the work to apply a fix is significant. I prefer to design a multi-assembly application and perhaps deploy only the resource assemblies I really need in a particular deployment scenario.

Changing Culture Programmatically

Most of the time, though, what you really want is the ability to set the culture programmatically and the ability to change it on the fly as the user switches to a different culture by clicking an icon or using a culture-specific URL.

To change the culture programmatically as you go, you need to satisfy two key requirements. First, define the policies you'll be using to retrieve the culture to set. The policy can be a value you read from some database table or perhaps from the ASP.NET cache. It can also be a value you retrieve from the URL. Finally, it can even be a parameter you get via geo-location—that is, by looking at the IP address the user is using for connecting. In any case, at some point you know the magic string that identifies the culture to set. How do you apply that?

The following code shows what you need to instruct the ASP.NET runtime about the culture to use:

```
var culture = "...";    // i.e., it-IT
var cultureInfo = CultureInfo.CreateSpecificCulture(culture);
Thread.CurrentThread.CurrentCulture = cultureInfo;
Thread.CurrentThread.CurrentUICulture = cultureInfo;
```

You pick up the ASP.NET current thread and set the *CurrentCulture* and *CurrentUICulture* properties. Note that the two culture properties might or might not have the same value. For example, you can switch the language of text and messages according to the browser's configuration while leaving globalization settings (such as dates and currency) constant.

The culture must be set for each request because each request runs on its own thread. In ASP.NET MVC, you can achieve this in a number of ways. For example, you can embed the preceding code in a base controller class. This forces you to derive any controllers from a given base class. If you find this unacceptable, or you just prefer to take another route, you can go for a custom action invoker or a global action filter. (I'll say more about invokers and action filters in Chapter 8.) In both cases,

you write the code once and attach it to all controllers in a single step. Let's see what it takes to use a global filter for localization:

```
[AttributeUsage(AttributeTargets.Class|AttributeTargets.Method, AllowMultiple=false,
Inherited=true)]
public class CultureAttribute : ActionFilterAttribute
{
    private const String CookieLangEntry = "lang";

    public String Name { get; set; }
    public static String CookieName
    {
        get { return "_LangPref"; }
    }

    public override void OnActionExecuting(ActionExecutingContext filterContext)
    {
        var culture = Name;
        if (String.IsNullOrEmpty(culture))
            culture = GetSavedCultureOrDefault(filterContext.RequestContext.HttpContext.
Request);

        // Set culture on current thread
        SetCultureOnThread(culture);

        // Proceed as usual
        base.OnActionExecuting(filterContext);
    }

    public static void SavePreferredCulture(HttpResponseBase response, String language,
                                            Int32 expireDays=1)
    {
        var cookie = new HttpCookie(CookieName) { Expires = DateTime.Now.AddDays(expireDays) };
        cookie.Values[CookieLangEntry] = language;
        response.Cookies.Add(cookie);
    }

    public static String GetSavedCultureOrDefault(HttpRequestBase httpRequestBase)
    {
        var culture = "";
        var cookie = httpRequestBase.Cookies[CookieName];
        if (cookie != null)
            culture = cookie.Values[CookieLangEntry];
        return culture;
    }

    private static void SetCultureOnThread(String language)
    {
        var cultureInfo = CultureInfo.CreateSpecificCulture(language);
        Thread.CurrentThread.CurrentCulture = cultureInfo;
        Thread.CurrentThread.CurrentUICulture = cultureInfo;
    }
}
```

The *CultureAttribute* class offers public static methods to read and write a specific culture string to a custom cookie. The filter overwrites the *OnActionExecuting* method, meaning that it might kick in before any controller method runs. For this to happen, though, the filter must be registered as a global filter. In the implementation of *OnActionExecuting*, the filter reads the user's preferred culture that was previously stored to a cookie and sets it to the current request thread.

The following code shows how to register the filter as a global filter that applies to all controller methods within the application:

```
public class MvcApplication : HttpApplication
{
    public static void RegisterGlobalFilters(GlobalFilterCollection filters)
    {
        filters.Add(new HandleErrorAttribute());
        filters.Add(new CultureAttribute());
    }
    ...
}
```

With this infrastructure in place, you can add links to your pages (typically, the master page) to switch languages on the fly:

```
@Html.ActionLink(Menu.Lang_IT, "Set", "Language", new { lang = "it" }, null)
@Html.ActionLink(Menu.Lang_EN, "Set", "Language", new { lang = "en" }, null)
```

To follow the actions just shown, you need an action method. I prefer to isolate this code to a specific controller, such as the *LanguageController* class shown here:

```
public class LanguageController : Controller
{
    public void Set(String lang)
    {
        // Set culture to use next
        CultureAttribute.SavePreferredCulture(HttpContext.Response, lang);

        // Return to the calling URL (or go to the site's home page)
        HttpContext.Response.Redirect(HttpContext.Request.UrlReferrer.AbsolutePath);
    }
}
```

The action method simply stores the newly selected language in the store you selected—a custom cookie in the example—and redirects. Figure 5-9 shows a sample page whose content can be displayed in multiple languages.

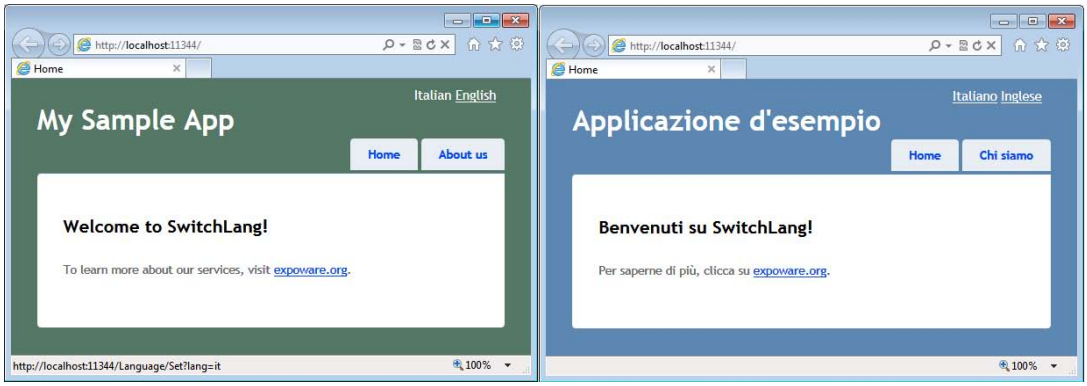

FIGURE 5-9 An application that allows users to switch languages on the fly.

The approach discussed can be applied regardless of the technique you use to determine the desired language, whether by choice, IP address, or geo-location.

Note More and more websites check the location from where a user is connected and suggest a language and a culture. This feature requires an API that looks up the IP address and maps that to a country and then a culture. Some browsers (for example, Firefox 3.5, Safari, iPhone, and Opera) have built-in geo-location capabilities that work according to the W3C API. (See *http://www.mozilla.com/firefox/geolocation*.)

To support other browsers (including Internet Explorer), you can resort to third-party services such as Google Gears. Google Gears is a plug-in that extends your browser in various ways, including adding a geo-location API that returns the country of the user from the current geographical location. Note that Google returns the ISO 3166 code of the country (for example, *GB* for the United Kingdom) and its full name. From here, you have to determine the language to use. The country code doesn't always match the language. For the United Kingdom, the language is *en*.

To install Google Gears, pay a visit to *http://gears.google.com*. Google Gears is currently marked as a deprecated API because Google is focusing on HTML5 support. An effective approach you can take today is illustrated at *http://diveintohtml5.org/geolocation.html*; it consists of using HTML5 API with a fallback that either displays an error message or goes with Google Gears. The check against the browser's support of the HTML5 geo-location API and subsequent fallback is conducted via the Modernizr JavaScript library. You find this library already embedded in any new ASP.NET MVC project created via Visual Studio.

Storing Localized Resources in a Database

While discussing localization, it seems inevitable that you have to talk about databases as a possible store for localized data. Is this an option? You bet. However, there are some pros and cons to consider.

In the first place, using a database adds latency even though you don't make a database call for each segment of a view to be localized. Most likely, instead, you'll read a bunch of records and probably cache them for a long time. The performance hit represented by using the database in this way is therefore less devastating than one might think at first.

Storing localization data inside a database requires a custom localization layer, whereas going through the classic XML-based approach of resource files doesn't lead you to writing much extra code and offers you excellent support from the Visual Studio designers.

When the number of views becomes significant (for example, in the hundreds), the number of resource items will be at least in the thousands. At this point, managing them can be problematic. You can have too many assemblies loaded in the AppDomain consuming run-time memory, and that will have an impact on the overall performance of the site. Hence, a database is probably the best way to go for a large share of localizable content.

Data stored within a relational database is easier to manage, query, and cache, and the size is not an issue. In addition, with a database and a custom localization layer you gain more flexibility in the overall retrieval process of local resources. In fact, you can ask the layer for a group of strings—or, better yet, for raw data—to then be formatted for the needs of the UI. In other words, a custom localization layer decouples you from maintaining a direct binding between a resource item and specific pieces of the user interface.

Summary

An application built with ASP.NET MVC is primarily a web application. Modern web applications have more numerous requirements than only a few years ago. For example, a web application today has to be SEO-friendly and very likely must support full localization to be able to drive the user's actions using the user's specific language and culture. Finally, serving a notorious yellow-screen-of-death (namely, one of those default error pages of ASP.NET) is hardly acceptable; it still happens, but it is really a bad statement about the site. (An unhandled error has always been a bad thing, but the level of forgiveness that users were willing to give only a few years ago is definitely a thing of the past.)

For all these reasons, the infrastructure of any web application (and, in this context, the infrastructure of ASP.NET MVC applications) needs to be stronger and richer. In particular, you need to pay more attention to the URLs you recognize and design both for SEO and error handling. You need to design views and controllers to check the current locale and adjust graphics and messages automatically. You also need to detect the culture and let users switch among the languages you support.

This chapter offered a detailed overview of how to proceed. The next chapter is about securing an ASP.NET MVC application.

CHAPTER 6

Securing Your Application

Security is a kind of death.

—Tennessee Williams

Security means a variety of things to users and developers. In a web context, security is related to preventing injection of malicious code in the running application. Likewise, security relates to actions aimed at preventing disclosure of private data. Finally, security relates to building applications (and sections of an application) that only authenticated and authorized users can access.

The aspect of security that application developers deal with more frequently is certainly the authentication and authorization of users. In fact, although they are based on a number of common patterns, the first two aspects of web security need to be tackled while keeping an eye on the overall architecture of the application. In general, strict control over input forms and accepted input data significantly cuts down the risk of security hazards.

Security in ASP.NET MVC

ASP.NET provides a range of authentication and authorization mechanisms implemented in conjunction with Internet Information Services (IIS), the Microsoft .NET Framework, and the underlying security services of the operating system. If IIS and the ASP.NET application are working in integrated mode—the most common scenario these days with IIS 7 and newer versions—the request goes through a single pipeline that includes an authentication step and an optional authorization step. If IIS and ASP.NET are running their own process, some requests might be authenticated or authorized at the gate by IIS while others (for example, requests for ASPX pages) are handed over to ASP.NET along with the IIS security token of the authenticated, or anonymous, user.

Originally, ASP.NET supported three types of authentication methods: Windows, Passport, and Forms. A fourth possibility is None, meaning that ASP.NET does not even attempt to perform its own authentication and completely relies on the authentication already carried out by IIS. In this case, anonymous users can connect and resources are accessed using the default ASP.NET account. In ASP.NET 4, Passport authentication is marked as obsolete because its purposes are now better served by emerging security standards such as OAuth and OpenID. (I'll say more on both later in the chapter.)

Authentication and Authorization

Windows authentication is seldom practical for real-world Internet applications. Windows authentication is based on Microsoft Windows accounts and NTFS ACL tokens and, as such, assumes that clients are connecting from Windows-equipped machines. Useful and effective in intranet scenarios and possibly in some extranet scenarios, Windows authentication is simply unrealistic in more common situations because web-application users are required to have Windows accounts in the application's domain.

Forms authentication is the most commonly used way to collect and validate user credentials—for example, against a database of user accounts.

Configuring Authentication in ASP.NET MVC

In ASP.NET MVC as well as in Web Forms, you choose the authentication mechanism using the *<authentication>* section in the root *web.config* file. Child subdirectories inherit the authentication mode chosen for the application. By default, ASP.NET MVC applications are configured to use Forms authentication. The following code snippet shows an excerpt from the autogenerated *web.config* file in ASP.NET MVC 3. (I just edited the login URL.)

```
<authentication mode="Forms">
   <forms loginUrl="~/Auth/LogOn" timeout="2880" />
</authentication>
```

Configured in this way, the application redirects the user to the specified login URL every time the user attempts to access a URL reserved to authenticated users. How would you mark an ASP.NET MVC URL (for example, a controller method) to require authentication?

Restricting Access to Action Methods

You use the *Authorize* attribute when you want to restrict access to an action method and make sure that only authenticated users can execute it. Here's an example:

```
[Authorize]
public ActionResult Index()
{
   ...
}
```

If you add the *Authorize* attribute to the controller class, any action methods on the controller will be subject to authentication:

```
[Authorize]
public class HomeController
{
   public ActionResult Index()
   {
      ...
   }

   ...
}
```

The *Authorize* attribute is inheritable. This means that you can add it to a base controller class of yours and ensure that any methods of any derived controllers are subject to authentication. You should never use, instead, the *Authorize* attribute as a global filter. The following code, in fact, would restrict access to any resource—including the login page!

```
public class MvcApplication : System.Web.HttpApplication
{
    public static void RegisterGlobalFilters(GlobalFilterCollection filters)
    {
        // Don't do this!
        filters.Add(new AuthorizeAttribute());
        ...
    }
}
```

Handling Authorization for Action Methods

The *Authorize* attribute is not limited to authentication; it also supports a basic form of authorization. Any methods marked with the attribute can be executed only by an authenticated user. In addition, you can restrict access to a specific set of authenticated users with a given role. You achieve this by adding a couple of named parameters to the attribute, as shown here:

```
[Authorize(Roles="admin", Users="DinoE, FrancescoE")]
public ActionResult Index()
{
   ...
}
```

If a user is not authenticated or doesn't have the required user name *and* role, the attribute prevents access to the method and the user is redirected to the login URL. In the example just shown, only users *DinoE* or *FrancescoE* (if they are in the role of *Admin*) can have access to the method. Note that *Roles* and *Users*, if specified, are combined in a logical AND operation.

Authorization and Output Caching

What if a method that requires authentication and/or authorization is also configured to support output caching? Output caching (specifically, the *OutputCache* attribute) instructs ASP.NET MVC to not really process the request every time, but return any cached response that was previously calculated and that is still valid. With output caching enabled, it might happen that a user requests a protected URL already in the cache. What should the behavior be in this case? ASP.NET MVC ensures that the *Authorize* attribute takes precedence over output caching. In particular, the output caching layer returns any cached response for a method subject to *Authorize* only if the user is authenticated and authorized.

Extending the *Authorize* Attribute

The *Authorize* attribute is the only security-related action filter supported by ASP.NET MVC. It handles authentication and authorization, but it sometimes misses some details.

Not Logged In or Not Authorized?

To enable the execution of the method, the *Authorize* attribute requires, first and foremost, that the user be authenticated. Next, if *Users*, *Roles*, or both are specified, it checks whether the user is also authorized (by name and role) to access the method.

The net effect is that the attribute doesn't ultimately distinguish between users who are not logged in and logged-in users who do not have the rights to invoke a given action method. In both cases, in fact, the attempt to call the action method redirects the user to the login page.

You might or might not like this behavior. If you do not, one thing you can do is create an enhanced attribute class, as shown here:

```
public class AuthorizedOnlyAttribute : AuthorizeAttribute
{
    public AuthorizedOnlyAttribute()
    {
        View = "error";
        Master = String.Empty;
    }

    public String View { get; set; }
    public String Master { get; set; }

    public override void OnAuthorization(AuthorizationContext filterContext)
    {
        base.OnAuthorization(filterContext);
        CheckIfUserIsAuthenticated(filterContext);
    }

    private void CheckIfUserIsAuthenticated(AuthorizationContext filterContext)
    {
        // If Result is null, we're OK: the user is authenticated and authorized.
        if (filterContext.Result == null)
            return;

        // If here, you're getting an HTTP 401 status code
        if (filterContext.HttpContext.User.Identity.IsAuthenticated)
        {
            if (String.IsNullOrEmpty(View))
               return;
            var result = new ViewResult {ViewName = View, MasterName = Master};
            filterContext.Result = result;
        }
    }
}
```

In the new class, you override the *OnAuthorization* method and run some extra code to check whether you're getting an HTTP 401 message. If this is the case, you then check whether the current user is authenticated and redirect to your own error page (if any). You have *View* and *Master* properties to configure the target error view with instructions for the user.

The net effect is that if you're getting an HTTP 401 error because the user is not logged in, you'll go to the log-in page. Otherwise, if the request failed because of authorization permissions, the user receives a friendly error page. Using the new attribute couldn't be easier:

```
[AuthorizedOnly(Roles="admin", Users="DinoE")]
public ActionResult Index()
{
   ...
}
```

Hiding Critical User Interface Elements

You might also want to prevent users from accessing restricted resources by simply disabling or hiding action links and buttons that might trigger restricted action methods. By checking the authentication state of the current user and any assigned role, you can just turn off the visibility flag of critical input elements if users don't have enough privileges.

I like this approach, but I would never implement it as the only solution to handle roles and permissions. In the end, hiding UI elements (which is more effective and simpler than disabling them) is fine as long as you still restrict access to action methods also by using programmatic checks.

A Quick Look at Windows Authentication

Although Forms authentication is by far the most common authentication mechanism for ASP.NET applications, there are some scenarios in which you want to opt for Windows authentication. Typically, you use the Windows authentication method in intranet scenarios when the users of your application also have Windows accounts that can be authenticated by the web server.

When using Windows authentication, ASP.NET works in conjunction with IIS. The real authentication is performed by IIS, which uses one of its two authentication methods: Basic or Integrated Windows. After IIS has authenticated the user, it passes the security token on to ASP.NET. When configured to operate in Windows authentication mode, ASP.NET does not perform any further authentication steps and just uses the IIS token to authorize access to the resources.

For example, let's assume that you configured the web server to work with the Integrated Windows authentication mode and that you disabled anonymous access. What happens when a user connects to the ASP.NET application? If the account of the local user doesn't match any accounts on the web server or in the trusted domain, IIS pops up a dialog box asking the user to enter valid credentials. Next, if credentials are determined to be valid, IIS generates a security token and hands it over to ASP.NET.

Implementing a Membership System

To authenticate a user, you need some sort of a membership system that supplies methods to manage the account of any users. Building a membership system means writing the software and the user interface to create a new user and update or delete existing users. It also means writing the software for editing any information associated with a user, such as the user's email address, password, and roles.

How do you create a user? Typically, you add a new record to some data store. Each data store can have its own set of properties, but core tasks are common and to a large extent abstracted by the ASP.NET native membership API. In ASP.NET MVC, you build a membership system integrating the ASP.NET membership API with one or two specific account controllers. A bunch of views and view model classes complete the infrastructure.

Note The Microsoft Visual Studio tooling for ASP.NET MVC 3 generates a sample application with full support for authentication and authorization. Although it's fully functional, the sample code you get is not exactly a paragon of software virtue. In any significant applications of mine, I just wipe out all the autogenerated files and start from scratch, taking the approach described next.

Defining a Membership Controller

At a minimum, you need to have a controller that knows how to log users on and off. The following sample *AuthController* class shows a possible way of getting one. This code reworks the sample *AccountController* class created by the Visual Studio tooling for ASP.NET MVC:

```
public class AuthController : Controller
{
    [HttpGet]
    public ActionResult LogOn()
    {
        // Just displays the login view
        return View();
    }

    [HttpPost]
    public ActionResult LogOn(LogOnViewModel model, String returnUrl)
    {
        // Gets posted credentials and proceeds with
        // actual validation
        ...
    }

    public ActionResult LogOff(String defaultAction="Index", String defaultController="Home")
    {
        // Logs out and redirects to the home page
        FormsAuthentication.SignOut();
        return RedirectToAction(defaultAction, defaultController);
    }
}
```

The *LogOn* method must be split in two: one overload to simply display the login view, and one to handle posted credentials and proceed with the actual validation. The *LogOff* method signs out of the application and redirects to the specified page—typically, the application's home page. To sign out, you use the native forms authentication services of ASP.NET. To top it off, you can also consider adding the following overload for *LogOff*:

```
public ActionResult LogOff (String defaultRoute)
{
    FormsAuthentication.SignOut();
    return RedirectToRoute(defaultRoute);
}
```

The method accepts a route name instead of a controller/action pair to identify the return URL.

Validating User Credentials

Figure 6-1 shows a canonical user interface for a login view. It contains two text boxes—one for the user name and password, and a check box in case the user wants to be remembered on the site.

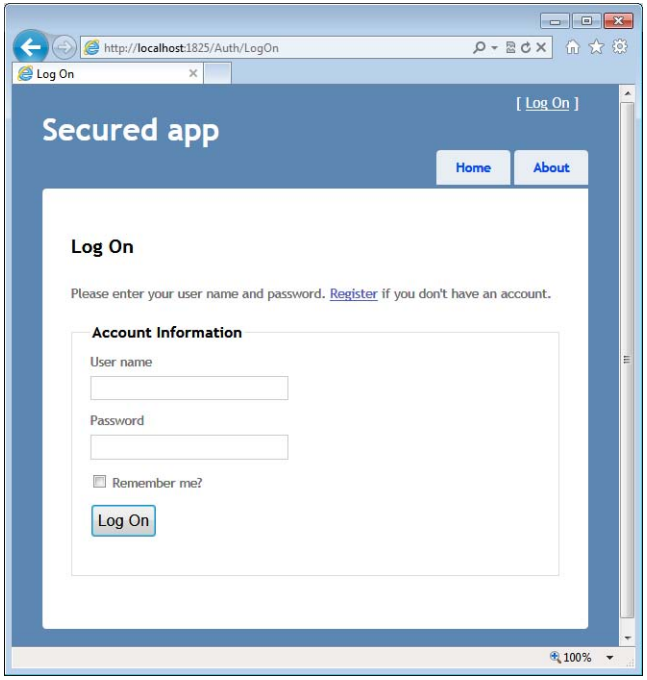

FIGURE 6-1 A canonical login view.

And here's a possible implementation for the controller method that handles data posted from this form:

```
[HttpPost]
public ActionResult LogOn(LogOnViewModel model, String returnUrl,
                          String defaultAction="Index", String defaultController="Home")
{
    var isValidReturnUrl = IsValidReturnUrl(returnUrl);
    if (!ModelState.IsValid)
    {
        ModelState.AddModelError("", "The user name or password provided is incorrect.");
        return View(model);
    }

    // Validate and proceed
    if (Membership.ValidateUser(model.UserName, model.Password))
    {
        FormsAuthentication.SetAuthCookie(model.UserName, model.RememberMe);
        if (isValidReturnUrl)
        {
            return Redirect(returnUrl);
        }
        else
        {
            return RedirectToAction(defaultAction, defaultController);
        }
    }

    // If we got this far, something failed; redisplay form.
    return View(model);
}
```

As you saw in Chapter 4, "Input Forms," input data can be collected in a handy data structure and processed on the server. That's just what you do with the *LogOnViewModel* class. The *returnUrl* parameter is only set if an original request was redirected to a login view because the user needed be authenticated first. In this case, the redirect response contains the new URL in the *location* header and the URL includes a *returnUrl* query string parameter.

Validation occurs through the ASP.NET membership API. If validation is successful, a valid authentication cookie is created via the *FormsAuthentication* class. Next, the user is redirected to the originally requested page or to the home page.

Integrating with the Membership API

Centered on the *Membership* static class, the ASP.NET membership API shields you from the details of how the credentials and other user information are retrieved and compared. The *Membership* class doesn't directly contain any logic for any of the methods it exposes. The actual logic is supplied by a provider component.

You select the membership in the configuration file. ASP.NET comes with a couple of predefined providers that target MDF files in Microsoft SQL Server Express and Active Directory. However, it

is not unusual that you end up creating your own membership provider so that you can reuse any existing store of users' data and are in total control over the structure of the data store.

Defining a custom membership provider is not difficult at all. All you do is derive a new class from *MembershipProvider* and override all abstract methods. At a minimum, you override a few methods such as *ValidateUser*, *GetUser*, *CreateUser*, and *ChangePassword*:

```
public class PoorManMembershipProvider : MembershipProvider
{
    public override bool ValidateUser(String username, String password)
    {
       ...
    }

    public override MembershipUser GetUser(String username, Boolean userIsOnline)
    {
       ...
    }

    public override MembershipUser CreateUser(String username, String password,
              String email, String passwordQuestion, String passwordAnswer,
              Boolean isApproved, Object providerUserKey,
              out MembershipCreateStatus status)
    {
       ...
    }

    public override Boolean ChangePassword(String username, String oldPassword, String
newPassword)
    {
       ...
    }

    // Remainder of the MembershipProvider interface
    ...
}
```

In particular, in the implementation of *ValidateUser* you pick up the user name and password and check them against your database. As a security measure, it is recommended that you store your passwords in a hashed format. Here's a quick demo of how to check a typed password against a stored hashed password:

```
public override bool ValidateUser(String username, String password)
{
    // Validate user name
    ...

    // Runs a query against the data store to retrieve the
    // password for the specified user. We're assuming that the
    // retrieved password is hashed.
    var storedPassword = GetStoredPasswordForUser(username);

    // No user found
    if (storedPassword == null)
      return false;
```

```
    // Hash the provided password and see if that matches the stored password
    var hashedPassword = Utils.HashPassword(password);
    return hashedPassword == storedPassword;
}
```

The default membership API has been criticized for being cumbersome and in patent violation of the Interface Segregation principle—which is the "I" principle in the popular SOLID acronym. The membership API attempts to cover a reasonable number of situations and is probably too complex and unnecessarily rich for the most common scenarios these days. Creating a custom membership provider helps, but it doesn't solve the issue entirely because it only builds a simpler façade.

If you don't want to create your own membership layer completely from scratch, another middle-way option you can try is represented by the *SimpleMembership* API as available in WebMatrix. You can incorporate the API in your ASP.NET MVC 3 application using a handy NuGet package.

Note, though, that the SimpleMembership API is just a wrapper on top of the ASP.NET membership API and data stores. It enables you to work with any data store you have and requires only that you indicate which columns operate as the user name and user id. The *WebSecurity* class offers a simplified API to do your membership chores, where the major difference with the classic Membership API is a radically shorter list of parameters for any methods and provides more freedom with the schema of the storage. Here's how you create a new user:

```
WebSecurity.CreateUserAndAccount(username, password,
    new{ FirstName = fname, LastName = lname, Email = email });
```

For more information on the SimpleMembership API, refer to the following blog of one of the members of the team: *http://blog.osbornm.com/archive/2010/07/21/using-simplemembership-with-asp.net-webpages.aspx*.

Integrating with the Role API

Roles in ASP.NET simplify the implementation of applications that require authorization. A *role* is just a logical attribute assigned to a user. An ASP.NET role is a plain string that refers to the logical role the user plays in the context of the application. In terms of configuration, each user can be assigned one or more roles. ASP.NET looks up the roles for the current user and binds that information to the *User* object. ASP.NET uses a role provider component to manage role information for a given user.

The role provider is a class that inherits the *RoleProvider* class. The schema of a role provider is not much different from that of a membership provider and shares the same complexity issues as well. To keep things simpler, or to wrap existing data storage, you might want to create a custom role provider or borrow the SimpleRoles API from WebMatrix.

The following code snippet shows how to programmatically correlate roles and users. You use the *Roles* class, which works on top of the concrete role provider:

```
// Create Admin role
Roles.CreateRole("Admin");
Roles.AddUsersToRole("DinoE", "Admin");

// Create Guest role
Roles.CreateRole("Guest");
var guests = new String[2];
guests[0] = "Joe";
guests[1] = "Godzilla";
Roles.AddUsersToRole(guests, "Guest")
```

At run time, information about the logged-in user is available through the HTTP context *User* object. The following code demonstrates how to determine whether the current user is in a certain role and subsequently enable specific functions:

```
if (User.IsInRole("Admin"))
{
      // Enable functions specific to the role
      ...
}
```

The Remember-Me Feature and Ajax

Nowadays nearly any login view features a check box labeled "Remember me" or "Keep me logged in," as Facebook does. Selecting the box usually results in an authentication cookie that lasts longer; how much longer depends on the code behind the page. The flag just indicates the user's preference of getting a more *persistent* cookie that keeps her connected to the site for a longer time without having to retype credentials.

If the cookie expires, on the next access the user will be automatically redirected to the login page and politely asked to reenter her user name and password. This pattern is not new, and every developer is used to it. However, this pattern might give you a few problems in an Ajax scenario.

Reproducing the Problem

Imagine that a user clicks somewhere and places an Ajax call to the server. Imagine also that the authentication cookie has expired. Subsequently, the server returns an HTTP 302 status code, which redirects the user to the login page. This is just what one would expect, isn't it? What's the issue, then?

In an Ajax scenario, it's the *XMLHttpRequest* object—not the browser—that handles the request. *XMLHttpRequest* correctly handles the redirect and goes to the login page. Unfortunately, the original issuer of the Ajax call will get back the markup of the login page instead of the data it was expecting. As a result, the login page will likely be inserted in any DOM location where the original response was expected. (See Figure 6-2.)

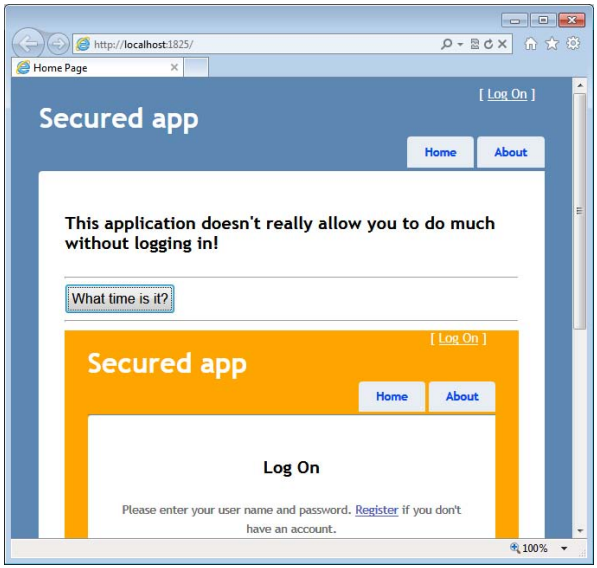

FIGURE 6-2 An Ajax request to a restricted URL injects the login view.

Solving the Problem

To work around the issue, you must intercept the request during the authorization stage and verify it is an Ajax request. If it is and if the request is being rejected, you hook up the status code and change it to 401. The client-side script you have then takes that into account and displays the proper HTML:

```
<script type="text/javascript">
    function failed(xhr, textStatus, errorThrown) {
        if (textStatus == "error") {
            if (xhr.status === 401) {
                $("#divOutput").html("You must be logged in.");
            }
        }
    }
</script>
```

The *failed* JavaScript function is the callback used by the Ajax infrastructure (as discussed in Chapter 4) when a form post or a link request fail. Here's the code for the form in Figure 6-2:

```
@using (Ajax.BeginForm("Now", "Home",
             new AjaxOptions { UpdateTargetId = "divOutput", OnFailure="failed" }))
{
   <input type="submit" value="What time is it?" />
}
<hr />
<div id="divOutput">
</div>
```

The real work is done by a slightly revised version of the *Authorize* attribute. All in all, you can just extend the *AuthorizedOnly* attribute as shown next and use it in any case in which restricted

access to a method is necessary. Replace the previous implementation of the internal method *CheckIfUserIsAuthenticated* with this one:

```
private void CheckIfUserIsAuthenticated(AuthorizationContext filterContext)
{
    // If Result is null, we're OK
    if (filterContext.Result == null)
        return;

    // Is this an Ajax request?
    if (filterContext.HttpContext.Request.IsAjaxRequest())
    {
        // For an Ajax request, just end the request
        filterContext.HttpContext.Response.StatusCode = 401;
        filterContext.HttpContext.Response.End();
    }

    // If here, you're getting an HTTP 401 status code
    if (filterContext.HttpContext.User.Identity.IsAuthenticated)
    {
        var result = new ViewResult {ViewName = View, MasterName = Master};
        filterContext.Result = result;
    }
}
```

With this final update, the *AuthorizedOnly* attribute has now become the definitive replacement for the system's *Authorize* attribute. Here's a sample restricted controller method that can be invoked both via Ajax and regular posts. Figure 6-3 shows the desired effect.

```
[AuthorizedOnly]
public ActionResult Now()
{
    ViewBag.Now = DateTime.Now.ToString("hh:mm:ss");
    if (Request.IsAjaxRequest())
        return PartialView("aNow");
    return View();
}
```

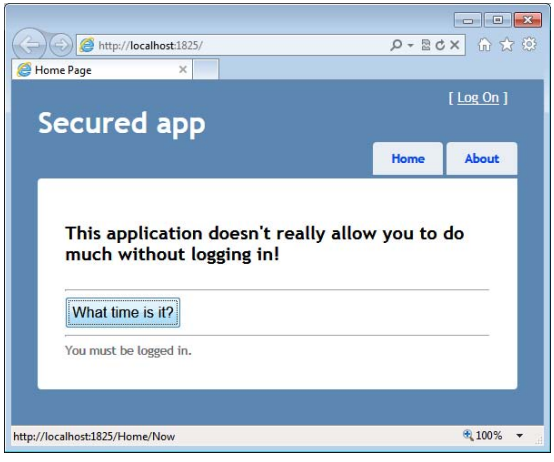

FIGURE 6-3 No login view shows up in case of Ajax requests.

Implementing your own authentication layer in a site is definitely an option. These days, however, it is becoming just one option and probably not even the most compelling one for users. By implementing your own authentication layer, you make yourself responsible for storing passwords safely and charge your team with the extra work required to fully manage an account. From the perspective of users, any new sites they're interested in might add a new user name/password pair to the list. For a user, a forgotten password is really frustrating. Years ago, the Microsoft Passport initiative was an early attempt to make users' lives easier when they moved across a few related sites. With Passport, users just needed to do a single logon and, if they were successfully authenticated, they could freely navigate through all the associated sites.

The Passport initiative, and its related API, is now officially considered obsolete because it was superseded by OpenID (*http://openid.net*). For websites that depend on lots of traffic and a large audience, the OpenID authentication is an interesting feature to have onboard that can really help attract and retain more and more visitors to the site.

The OpenID Protocol

The main purpose of OpenID is to make access to a website easier, quicker, and especially not annoying for end users. The visitors of any sites that support OpenID can sign in using an existing identity token that has been issued by another site. An OpenID-enabled website authenticates its users against an existing (external) identity provider and doesn't need to store passwords and implement a membership layer.

Although offering the standard, site-specific membership solution remains an option, many sites today also allow visitors to use an OpenID they got from some provider for authenticating. One form or authentication doesn't exclude the other.

Figure 6-4 provides an overall view of the authentication logic employed by a site that supports OpenID. When the user clicks to sign in with one of the supported OpenIDs, the site connects to the specified provider and gets an access token about the user. The user might be requested to type in credentials for the OpenID provider site. A user who is already logged in with the provider is automatically logged in also with the site of interest. In this case, the few redirects taking place under the hood never show intermediate pages and the transition is as smooth as within two distinct pages of the same application.

Your site doesn't have to be an OpenID identity provider, but it can easily become a consumer of identity tokens supplied by a few of the OpenID providers available today. Yahoo!, Flickr, and Google are popular OpenID providers. Overall, OpenID is not different from the original Windows Live ID, except that it relies on a number of service providers and doesn't force people to get yet another account from a specific provider. By supporting OpenID, you enable your users to log in to your site using whatever credentials they already have. Users can essentially choose where they log in from.

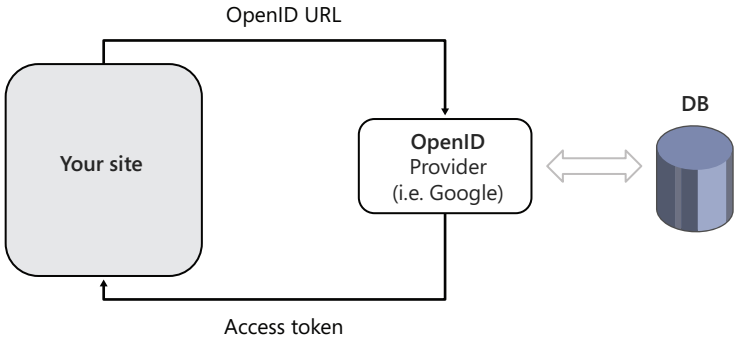

FIGURE 6-4 Authenticating via OpenID.

Identifying Users Through an OpenID Provider

There are quite a few libraries that help integrate OpenID into websites. A very popular one for .NET developers is DotNetOpenAuth (DNOA), available from *http://www.dotnetopenauth.net*. Figure 6-5 shows a screen shot from a sample application using the DNOA library to perform authentication against any valid OpenID URL the user can provide.

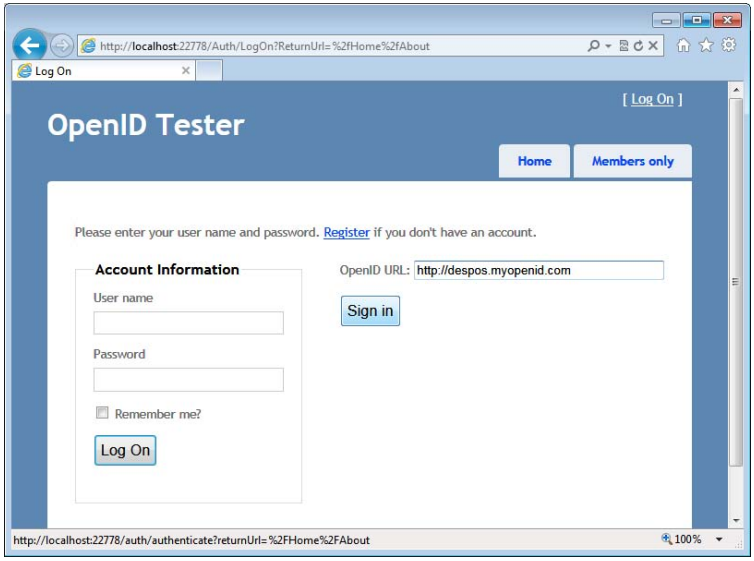

FIGURE 6-5 A sample application for connecting to any OpenID provider you might know of.

As a developer, the first piece of information you should grab is the OpenID URL of the service (or services) you intend to support. This information might be the same for each user, or it can differ for each user. For example, Google and Yahoo! always use the same URL. Here are the URLs for Google and Yahoo!, respectively:

```
https://www.google.com/accounts/o8/id
http://yahoo.com
```

The provider in this case will figure out from the details of the request (for example, a cookie) the account name of the user to authenticate. If no details can be figured out, or the user is not currently logged on to the service, the service displays a login page to collect credentials and redirects back if all is fine (possibly with a cookie for further access). Other providers such as myOpenID (*http://www.myopenid.com*) require a different URL for each user of the form *http://name.myopenid.com*. The service recognizes the account name from the URL and then proceeds as shown earlier.

As a developer, if you intend to support an OpenID provider that uses a fixed URL, you can avoid the text box shown in Figure 6-5 and replace it with a link or a button. In general, you might want to reduce typing on the user's part to a minimum. For example, for myOpenID all you really need is the first part of the URL. Here's the HTML for the small form shown in Figure 6-5:

```
@using (Html.BeginForm("authenticate", "auth", new { returnUrl = Request.
QueryString["ReturnUrl"] }))
{
    <label for="openid_identifier">OpenID URL: </label>
    <input id="openid_identifier" name="openid_identifier" size="40" /><br />
    @Html.ValidationMessage("openid_identifier")
    <br />
    <input type="submit" value="Sign in" />
}
```

The sign-in button posts to a method like the one shown here:

```
public ActionResult Authenticate(String returnUrl, [Bind(Prefix="openid_identifier")]String url)
{
    // First step: issuing the request and returning here
    var response = RelyingParty.GetResponse();
    if (response == null)
    {
        if (!RelyingParty.IsValid(url))
            return View("LogOn");

        try
        {
            return RelyingParty.CreateRequest(url).RedirectingResponse.AsActionResult();
        }
        catch (ProtocolException ex)
        {
            ModelState.AddModelError("openid_identifier", ex.Message);
            return View("LogOn");
        }
    }

    // Second step: redirected here by the provider
    switch (response.Status)
    {
        case AuthenticationStatus.Authenticated:
            FormsAuthentication.SetAuthCookie(response.ClaimedIdentifier, true);
            return Redirect(returnUrl);
```

```
        case AuthenticationStatus.Canceled:
            return View("LogOn");

        case AuthenticationStatus.Failed:
            return View("LogOn");
    }
    return new EmptyResult();
}
```

The method belongs to a controller class that has a *RelyingParty* property defined as follows:

```
protected static OpenIdRelyingParty RelyingParty = new OpenIdRelyingParty();
```

The type *OpenIdRelyingParty* is defined in the DNOA library. The authentication develops in two phases within the same controller method. The method *CreateRequest* prepares the proper HTTP request and associates it with the specified OpenID URL. The provider receives instructions to redirect back to the same URL and the same controller method. The second time, though, the response is not null and you can create a regular ASP.NET authentication cookie where the user name is the name of account as returned by the OpenID provider.

Note that you have no control over this aspect—the provider decides what to return as the friendly name of the authenticated user. For security reasons, for example, Google doesn't return any significant friendly name; the *FriendlyIdentifierForDisplay* property is set to the generic OpenID URL. When you use myOpenID, it sets the *FriendlyIdentifierForDisplay* of the response to the URL, simply removing the scheme information and any trailing slash. To finalize the integration with the ASP.NET authentication infrastructure, you need to create a regular authentication cookie so that the user name can be displayed in the login area. Figure 6-6 shows the page that is displayed when using myOpenID.

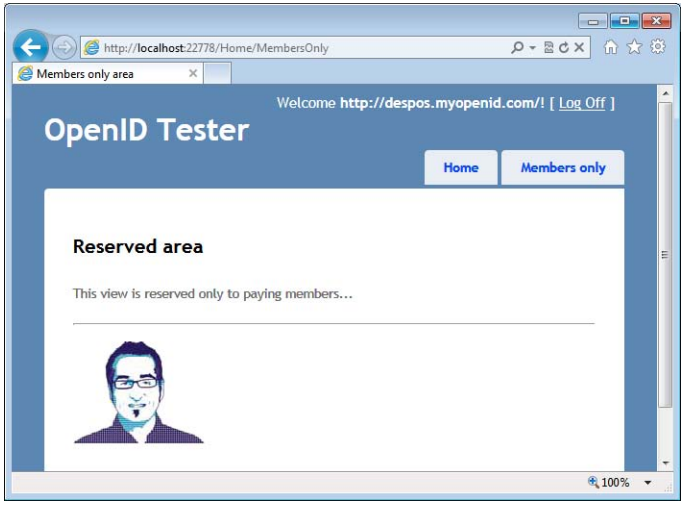

FIGURE 6-6 User authenticated via myOpenID.

The user name depends on the first parameter you pass when you create the cookie. (The second argument is whether you want a persistent cookie to keep the user logged in.)

```
FormsAuthentication.SetAuthCookie(response.ClaimedIdentifier, true);
```

The name you use here is up to you. If you still maintain a list of application-specific nicknames, you can map the claimed identifier to the nickname and display the nickname. Or you can simply update the user interface to notify the user of the successful sign-in without displaying any user name. Yet another option is to create a slightly customized authentication cookie where you use the *UserData* property of the authentication ticket (the actual content of the cookie) to persistently store the friendly name to display. The benefit of this approach is that you save both the claimed identifier and your own adapted friendly name. Here's an extension method that creates a custom authentication cookie:

```
public delegate String UserNameAdapterDelegate(String userName);
public static HttpCookie CreateAuthCookie(this IAuthenticationResponse response,
                                          Boolean persistent = true,
                                          UserNameAdapterDelegate fnAdapter = null)
{
    var userName = response.ClaimedIdentifier;
    var userDisplayName = response.FriendlyIdentifierForDisplay;
    if (fnAdapter != null)
        userDisplayName = fnAdapter(userDisplayName);

    return CreateAuthCookie(userName, userDisplayName, persistent);
}

private static HttpCookie CreateAuthCookie(String username,
                                           String userDisplayName,
                                           Boolean persistent)
{
    // Let ASP.NET create a regular authentication cookie
    var cookie = FormsAuthentication.GetAuthCookie(username, persistent);

    // Modify the cookie to add friendly name
    var ticket = FormsAuthentication.Decrypt(cookie.Value);
    var newTicket = new FormsAuthenticationTicket(ticket.Version,
           ticket.Name, ticket.IssueDate, ticket.Expiration, ticket.IsPersistent,
userDisplayName);
    cookie.Value = FormsAuthentication.Encrypt(newTicket);

    // This modified cookie MUST be re-added to the Response.Cookies collection
    return cookie;
}
```

Here's a slightly modified version of the *Authenticated* method in the controller that takes this extension method into account:

```
switch (response.Status)
{
    case AuthenticationStatus.Authenticated:
        var cookie = response.CreateAuthCookie(true, StopAtFirstToken);
        Response.Cookies.Add(cookie);
```

```
        if (isValidReturnUrl)
            return Redirect(returnUrl);
        return RedirectToAction("Index", "Home");
    ...
}
```

The delegate *UserNameAdapterDelegate* indicates the template of a function you can inject to determine the friendly name to be displayed:

```
public String StopAtFirstToken(String name)
{
    var tokens = name.Split('.');
    return tokens[0];
}
```

Finally, you need to edit the log in view to display the content of the *UserData* field in the cookie instead of the canonical user name:

```
@if(Request.IsAuthenticated) {
    <text>Welcome <strong>@(((FormsIdentity)User.Identity).Ticket.UserData)</strong>!
    [ @Html.ActionLink("Log Off", "LogOff", "Auth") ]</text>
}
```

Figure 6-7 shows the result.

FIGURE 6-7 The user is logged in, and a custom nickname is shown.

Note When you need to add more data to the authentication cookie, the *UserData* property of the internal ticket structure is the first option to consider. The *UserData* property exists exactly for this purpose. However, you can always create an additional and entirely custom cookie or just add values to the authentication cookie. The name of the authentication cookie results from the value of the property *FormsAuthentication.FormsCookieName*.

OpenID vs. OAuth

OpenID is a single sign-on scheme and, as such, it just aims to uniquely identify users in the simplest possible way. OpenID is not related to granting users access to resources managed by the service provider. Or, put another way, the only resources that an OpenID provider manages are the identities of registered users.

The advent of social networks such as Facebook, Twitter, and LinkedIn put the classic single sign-on problem under a different light. Not only do users want to use a single (popular) identity to log on to multiple sites without registering every time, but they also want to be granted some permissions to a site to access information and resources they have on the site—such as posts, tweets, followers, friends, contacts, and so forth.

OAuth (see *http://oauth.net*) is another single sign-on scheme with additional capabilities compared to OpenID. A website that acts as an OAuth provider operates as an identity provider, and when the user logs in the OAuth provider specifies permissions on resources. A website that offers OAuth authentication just acts as the client of a provider using the specific OAuth protocol. Such a website authenticates users and gains an access token it can further use to access resources (for example, merge tweets or contacts with its own user interface). Finally, from the user's perspective, OAuth grants a website (or desktop application) user controlled access to one account without giving away login details.

Popular OAuth providers are Twitter and Facebook.

Important Overall, I don't think that as a website developer you have to make a choice between using OpenID or OAuth for authentication. You decide which external provider you want to authenticate against (for example, Twitter) and then go with the API it requires, whether it's OpenID, OAuth, or a proprietary one. If you're developing a website and want to allow your users to share their account (and related information) with other sites, you make a decision between OpenID, OAuth, or a proprietary protocol. In this case, I recommend you consider OAuth as the first option.

Authenticating via Twitter

Let's see what it takes to authenticate users via Twitter. In the next example, we'll use the same DNOA library to encapsulate the details of the protocols. I'll limit the code to authentication (OpenID and OAuth kind of overlap in the examples of this chapter) but will call out the points where OAuth extends the OpenID scheme.

Registering Your Application

Dealing with an OAuth provider requires some extra steps. First and foremost, you must register your application with the site and get two strings: the consumer key and consumer secret. You'll be embedding these strings in any further programmatic request to the provider. The whole process begins at *http://dev.twitter.com*. Figure 6-8 shows the configuration page for an existing application.

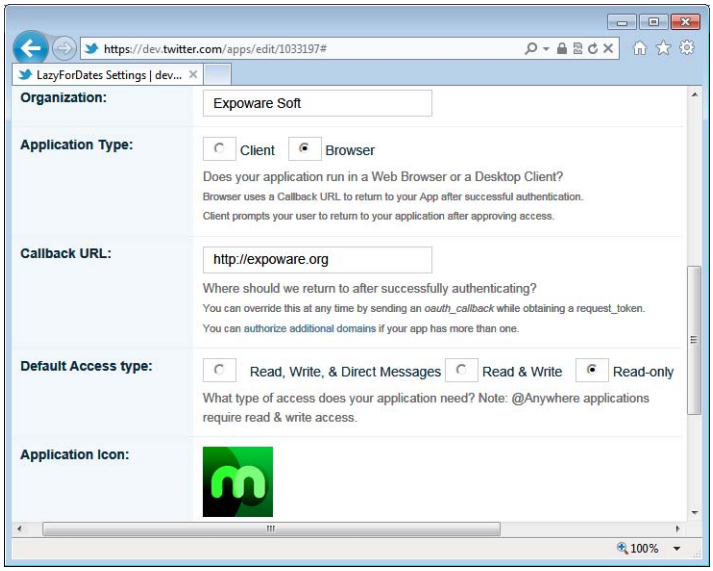

FIGURE 6-8 Registering an application with Twitter.

In particular, you set the application type and the default permission you require on the user's resources. The Callback URL field deserves some attention. The URL is where Twitter returns after successfully authenticating a user. You don't likely want this URL to be constant. However, Twitter requires that you don't leave the field blank if your application is a web application. If you leave the field blank because you intend to specify the callback URL on a per-call basis, Twitter overwrites the application type settings, forcing it to a desktop application. The net effect is that any attempts to authenticate any users return unauthorized. Assigning any URL, including one that results in a 404 message (as shown in the example), will work.

Starting the Authentication Process

The controller method that starts the Twitter authentication is shown next. The class belongs to a project where DNOA is used, in addition to the DNOA Application Block. (*WebConsumer* and *TwitterConsumer* classes are part of this framework.)

```
public ActionResult TwitterLogOn()
{
    var twitterClient = new WebConsumer(TwitterConsumer.ServiceDescription, TokenManager);

    // Create the callback URL for Twitter
    var callbackUri = Request.ReplacePath("Auth/TwitterAuthenticated").AsUri();

    // Place the request
    twitterClient.Channel.Send(
           twitterClient.PrepareRequestUserAuthorization(callbackUri, null, null));

    // You should be redirected to Twitter at this point ...
    return new EmptyResult();
}
```

The *ReplacePath* and *AsUri* methods are plain extension methods you find defined in the source code of the book. (Their purpose is just to make the code cleaner to read.) The two *null*s in the call to *PrepareRequestUserAuthorization* are for dictionaries of parameters for request and redirect, respectively.

Important The *Send* method places the request only if no valid authentication cookie can be found on the client machine.

When the request hits the Twitter site, the user is redirected to the authorization page, as shown in Figure 6-9.

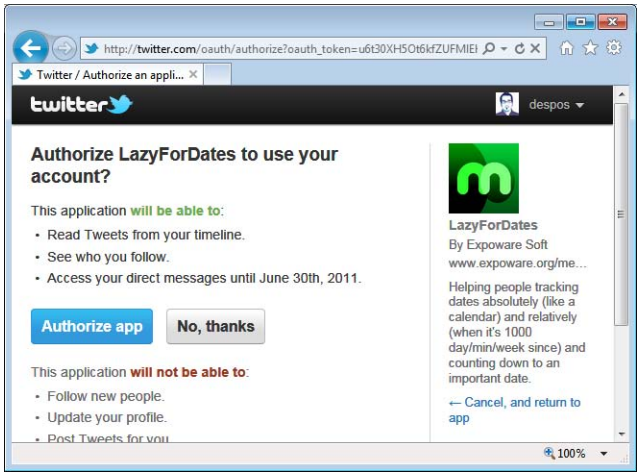

FIGURE 6-9 The user authenticates on Twitter and authorizes the requesting application to access account information.

If the user (me in this case) is already logged in to Twitter, he's simply asked to authorize the requesting application. Otherwise, he first needs to sign in with Twitter and then authorize. Next, Twitter redirects back to the specified URL.

Dealing with the Twitter Response

In the case of an ASP.NET MVC application, the callback URL invoked by Twitter is just another controller method, as shown here:

```
public ActionResult TwitterAuthenticated()
{
    // Process the response
    var twitterClient = new WebConsumer(TwitterConsumer.ServiceDescription, TokenManager);
    var accessTokenResponse = twitterClient.ProcessUserAuthorization();

    // Is Twitter calling back with authorization?
    if (accessTokenResponse != null)
    {
```

```
        // Extract the access token and username for use throughout the site
        var cookie = accessTokenResponse.CreateAuthCookie();
        Response.Cookies.Add(cookie);
    }
    else
    {
        // If the request doesn't come with an authentication token, redirect to the login page
        return RedirectToAction("LogOn", "Auth");
    }

    // If the authentication is successful, redirect to the home page
    return RedirectToAction("Index", "Home");
}
```

The posted response is processed by the DNOA layer and exposed to your application through an access response object. From this piece of information, you create an authentication cookie. *CreateAuthCookie* is another extension method, which wraps up all the details of the cookie creation:

```
public static HttpCookie CreateAuthCookie(this AuthorizedTokenResponse accessTokenResponse,
                                          Boolean persistent = true)
{
    var accessToken = accessTokenResponse.AccessToken;
    var username = accessTokenResponse.ExtraData["screen_name"];
    return CreateAuthCookie(username, accessToken, persistent);
}
```

The internal *CreateAuthCookie* method is nearly the same the method we considered earlier for the OpenID example.

Important To test the Twitter authentication, you need a real *consumer key/consumer secret* pair, but you don't strictly need to test it from a public web server. You can do everything comfortably from your Visual Studio environment and use *localhost:port* as the root of the callback URL.

The Token Manager

Where are the consumer key and consumer secret stored? In DNOA, you need a *token manager* object. A token manager is an object that implements the *IConsumerTokenManager* interface. The interface supplies read-only properties for the consumer key and secret. It also provides other members though. The full interface is listed here:

```
public interface IConsumerTokenManager
{
    String ConsumerKey {get;}
    String ConsumerSecret {get;}
    void ExpireRequestTokenAndStoreNewAccessToken(String consumerKey,
                String requestToken, String accessToken, String accessTokenSecret);
    String GetTokenSecret(String token);
    TokenType GetTokenType(String token);
    void StoreNewRequestToken(UnauthorizedTokenRequest request,
                ITokenSecretContainingMessage response)
}
```

The order in which members are invoked gives you a glimpse of how the entire OAuth stack works.

An OAuth request revolves around two types of tokens: a request token and an access token. Each token has an associated secret string. Both types of tokens are completely different from the consumer key and secret. The client application acquires the request token for the new request through the *WebConsumer* class. The request token represents a still-unauthorized request.

It is essential that the client application stores the request token (and secret) safely until authorization is complete. Method *StoreNewRequestToken* in the token manager object is responsible for this. At this stage, a request token has no significant relationship with a user.

```
public void StoreNewRequestToken(UnauthorizedTokenRequest request,
                                 ITokenSecretContainingMessage response)
{
   // Invoked before going to Twitter; stores a newly generated request token for later use.
   TokensAndSecrets[response.Token] = response.TokenSecret;
}
```

In this sample code, the token is saved to a static in-memory dictionary:

```
protected readonly Dictionary<String, String> TokensAndSecrets = new Dictionary<String,
String>();
```

On the way back from Twitter, your original request token is now an authorized request token because the service provider (Twitter) had the user approve it. The consumer replaces the authorized request token (and secret) with an access token (and secret). The *WebConsumer* class first reads the request secret back using the *GetTokenSecret* method:

```
public String GetTokenSecret(String token)
{
   // Invoked past authorization, this method gets the secret if given a request or access
token.
   return TokensAndSecrets[token];
}
```

Next, the *WebConsumer* class exchanges the request token for an access token. The method involved is shown here:

```
public void ExpireRequestTokenAndStoreNewAccessToken(String consumerKey, String requestToken,
                          String accessToken, String accessTokenSecret)
{
     // Invoked past authorization, this method exchanges a request token
     // (now authenticated) with an access token.
     TokensAndSecrets.Remove(requestToken);
     TokensAndSecrets[accessToken] = accessTokenSecret;
}
```

The access token and secret are just what the client application needs to be able to place further calls to Twitter. The access token is also the piece of information you want to associate with the user and store in the ASP.NET authentication cookie.

Most of the difference between OpenID and OAuth lives in the token manager object. If a website wants to use an OAuth provider for authentication purposes, only then does it have no need to store the access token in a permanent store. If among the functionalities of the website there's one that requires reading, say, the tweets of the authenticated user, the access token is just what you need to be able to place a request to Twitter on behalf of the user. If you don't have it stored somewhere, Twitter will ask the user to re-authorize over and over again.

Note The source code that comes with the book provides an authentication-only implementation of the token manager that also reads the consumer key and secret from the *web.config* file. To test the sample TwitterAuth application, you must first register an application of yours with Twitter and grab your own consumer key and secret.

Summary

Security is always perceived as a hot topic for web applications. So nearly any class or book on a web technology is expected to host a section on how to write secure applications. Assuming that one knows the basics of security and forms authentication for classic ASP.NET (for example, Web Forms), there's not much else left to cover that is specific to ASP.NET MVC.

The run-time pipeline is the same as in Web Forms, and trust levels and process identities are established in exactly the same manner. Also, Forms authentication works in the same way through an HTTP module and a highly configurable cookie. Deep coverage of these topics can be found in Chapter 19 of my latest ASP.NET book *Programming ASP.NET 4* (Microsoft Press, 2011).

Specific to ASP.NET MVC is the way in which one restricts access to action methods and controls authorization. I covered this in the first half of the chapter. The second half touched on features that more and more applications are incorporating these days—authenticating users through external services. This can be done via a couple of single sign-on schemes and underlying protocols—OpenID and OAuth. Whereas OpenID is essentially for uniquely identifying users, OAuth can do that and more. OAuth can get permissions from the user about resources held by the service provider that the application can use. I presented an OAuth sample application that authenticates users using the popular Twitter social network, thus gaining for itself the permission to read the tweets and connect to followers of logged-on users.

CHAPTER 7

Design Considerations for ASP.NET MVC Controllers

> *Part of the inhumanity of the computer is that, once it is competently programmed and working smoothly, it is completely honest.*
>
> *—Isaac Asimov*

The controller is the central element of any operation you perform in ASP.NET MVC. The controller is responsible for getting posted data, executing the related action and then preparing and requesting the view. More often than not, these apparently simple steps originate a lot of code. Worse yet, similar code ends up being used in similar methods, and similar helper classes sprout up from nowhere.

ASP.NET MVC comes with the promise that it makes it easier for you to write cleaner and more testable code. For sure, ASP.NET MVC is based on some infrastructure that makes this possible and easier than in Web Forms. A lot, however, is left to you—the developer—and to your programming discipline and design vision.

Architecturally speaking, the controller is just the same as the code-behind class in Web Forms. It is part of the presentation layer, and in some way it exists to forward requests to the back end of the application. Without development discipline, the controller can easily grow as messy and inextricable as an old-fashioned code-behind class. So it isn't just choosing ASP.NET MVC that determines whether you're safe with regard to code cleanliness and quality.

In this chapter, we'll explore an approach to ASP.NET MVC design that simplifies the steps you need to mechanize the implementation of the controller classes. The idea is to make the controller an extremely lean and mean class that delegates responsibility rather than orchestrating tasks. This design has an impact on other layers of the application and also on some portions of the ASP.NET MVC infrastructure.

Shaping Up Your Controller

Microsoft Visual Studio makes it easy to create your own controller class. It requires you to right-click on the Controllers folder in the current ASP.NET MVC project and add a new controller class. In a controller class, you'll have one method per user action that falls under the responsibility of the controller. How do you code an action method?

An action method should collect input data and use it to prepare one or multiple calls to some endpoint exposed by the middle tier of the application. Next, it receives output and ensures that output is in the format that the view needs to receive. Finally, the action method calls out the view engine to render a specific template.

Well, all this work might add up to several lines of code, making even a controller class with just a few methods quite a messy class. The first point—getting input data—is mostly solved for you by the model-binder class. Invoking the view is just one call to a method that triggers the processing of the action result. The core of the action method is in the code that performs the task and prepares data for the view.

Choosing the Right Stereotype

Generally speaking, an action method has two possible roles. It can play the role of a *controller*, or it can be a *coordinator*. Where do words like "controller" and "coordinator" come from? Obviously, in this context the word "controller" has nothing to do with an ASP.NET MVC controller class.

These words refer to object stereotypes, a concept that comes from a methodology known as Responsibility-Driven Design (RDD). Normally, RDD applies to the design of an object model in the context of a system, but some of its concepts also apply neatly to the relatively simpler problem of modeling the behavior of an action method.

Note For more information about RDD, check out *Object Design: Roles, Responsibilities, and Collaborations* by Rebecca Wirfs-Brock and Alan McKean (Addison-Wesley, 2002).

RDD at a Glance

The essence of RDD consists of breaking down a system feature into a number of actions that the system must perform. Next, each of these actions is mapped to an object in the system being designed. Executing the action becomes a specific responsibility of the object. The role of the object depends on the responsibilities it takes on. Table 7-1 describes the key concepts of RDD and defines some of the terms associated with its use.

TABLE 7-1 Standard RDD concepts and terms

Concept	Description
Application	Refers to a collection of interconnected and interacting objects.
Collaboration	Refers to an established relation between two objects that work together to provide some meaningful behavior. The terms of collaboration are defined through an explicit contract.
Object	Refers to a software component that implements one or multiple roles.
Role	Refers to the nature of a software component that takes on a collection of (related) responsibilities.
Responsibility	Refers to the expected behavior of an object. Stereotypes are used to classify responsibilities.

Table 7-2 summarizes the main classes of responsibility for an object. These are referred to as *object role stereotypes*.

TABLE 7-2 Standard RDD stereotypes

Stereotype	Description
Controller	Orchestrates the behavior of other objects, and decides what other objects should do.
Coordinator	Solicited by events, it delegates work to other objects.
Information holder	Holds (or knows how to get) information and provides information.
Interfacer	Represents a façade to implement communication between objects.
Service provider	Performs a particular action upon request.
Structurer	Manages relations between objects.

In RDD, every software component has a role to play in a specific scenario. When using RDD, you employ stereotypes to assign each object its own role. Let's see how RDD stereotypes can be applied to an action method.

Breaking Down the Execution of a Request

I've described some common steps that all action methods should implement. The responsibility of an action method can be broken down as follows:

- Getting input data sent with the request
- Performing the task associated with the request
- Preparing the view model for the response
- Invoking the next view

Both the Controller and Coordinator RDD stereotypes can be used to implement an action method—but they won't produce the same effects.

Acting as a "Controller"

Let's consider an action method in the apparently simple place-order use-case. In the real-world, placing an order is never a simple matter of adding a record to the Orders table. It's an action that involves several steps and objects. It requires querying the databases to find out about the availability of the ordered goods. It might also require an order to be placed to a provider to refill the inventory. Placing an order typically requires checking the credit status of the customer and syncing up with the bank of the customer and the shipping company. Finally, it also involves doing some updates on some database tables. The following pseudo-code gives you an idea of the concrete steps you need to take:

```
[HttpPost]
public ActionResult PlaceOrder(OrderInfo order)
{
    // Input data already mapped thanks to the model binder

    // Step 1-Check goods availability
    ...
    // Step 2-Check credit status of the customer
    ...
    // Step 3-Sync up with the shipping company
    ...
    // Step 4-Update databases
    ...
    // Step 5-Notify the customer
    ...

    // Prepare the view model
    var model = PlaceOrderViewModel { ... };
    ...

    // Invoke next view
    return View(model);
}
```

Having all these steps coded in the controller at a minimum means that you end up with calls made to the data access layer from the presentation. For simple CRUD (Create, Read, Update, Delete) applications, this is acceptable; but it's not acceptable for more complex applications.

Even when each of the steps outlined resolves in one or two lines of code, you have quite a long and soon unmanageable method. The RDD Controller stereotype applied to ASP.NET MVC controller classes suggests you should use the previous layout of the code. This is not ideal even for moderately complex applications.

Acting as a "Coordinator"

The RDD Coordinator stereotype suggests that you group all of the steps that form the implementation of the action within a single worker object. From within the action method, you place a single call to the worker and use its output to feed the view-model object. The layout follows.

```
[HttpPost]
public ActionResult PlaceOrder(OrderInfo order)
{
    // Input data already mapped thanks to the model binder

    // Perform the task invoking a worker service
    var workerService = new WorkerService();
    var response = workerService.PerformSomeTask();

    // Prepare the view model
    var model = PlaceOrderViewModel(response);
    ...

    // Invoke next view
    return View(model);
}
```

The overall structure of the controller method is much simpler now. Solicited by an incoming HTTP request, the action method relays most of the job to other components. I call these components *worker services*; in RDD jargon, they look a lot like *Controller* objects and, in some simple cases, they're just service providers.

Fat-Free Controllers

ASP.NET MVC is a framework that is designed to be testable and promotes important principles such as separation of concerns (SoC) and Dependency Injection (DI). ASP.NET MVC tells you that an application is separated in a part known as the *controller* and a part referred to as the *view* (not to mention the model discussed here). Being forced to create a controller class doesn't mean you'll automatically achieve the right level of SoC, and it certainly doesn't mean that you're writing testable code. As mentioned in Chapter 1, "ASP.NET MVC Controllers," ASP.NET MVC gets you off to a good start, but any further (required) layering is up to you.

What I haven't probably stated clearly enough is that if you don't pay close attention, you end up with a fat and messy controller class, which certainly isn't any better than a messy (and justifiably despised) code-behind class. So you should aim to create controller classes as lean and mean collections of endpoints and remove any fat from them.

Note According to my standards, I wasn't precise earlier when I called Dependency Injection a *principle*. More specifically, DI is just the most popular pattern used to implement the *Dependency Inversion Principle*, according to which the surface of contact between dependent classes should always be an interface instead of an implementation. Much less known (and understood) than DI in the wild, the Dependency Inversion Principle is the "D" in the popular SOLID acronym that summarizes the five key design principles for writing clean, high-quality code.

Short Is Always Better

If you have a method that is about 100 logical lines long, that code probably includes 10 to 15 lines of comments. Generally, 10 percent is considered to be a fair ratio of code to comments; I'd even go as high as a comment every three logical lines if you want to make sure that you explain clearly the whys and wherefores of what you're doing and really want to help whomever deals with that piece of code after you.

However, regardless of what you decide the ideal ratio is, my point is that a method that's 100 lines long makes little sense. You can probably break it into three or four smaller methods, and get rid of some comments too.

I don't call myself an expert in software metrics, but I usually try to keep my methods below 20 lines—which more or less matches the real estate available in the Visual Studio editor on a normal laptop.

How can you manage to keep the code of action methods as short as possible? Surprisingly enough, applying the RDD Coordinator stereotype is what you must do, but even that's not always sufficient.

Action Methods Coded as View Model Builders

A method designed to be a coordinator invokes a method on a worker object, has some work done, and gets some data back. This data should simply be packed into a dictionary, or a strongly typed class, and then passed down to the view engine.

The worker class, though, is attempting to bridge the gap between the data model you have on the middle tier—the domain model—and the data model you have in the presentation layer—the view model, or the data being worked on in the view. (By the way, "the data being worked on in the view" is the wording originally used in the MVC paper to define the role of the model.)

If the business objects you invoke on your middle tier return collections or aggregates of domain objects, you probably need to massage this data into view-model objects that faithfully represent the contracted user interface. If you move this work into the controller class, you're back to square one. The lines of code you cut off by using worker services and the RDD Coordinator stereotype are replaced by just as many lines for building a view model.

To support your efforts in getting fat-free controllers, I recommend a strategy based on the following points:

- Relay any action to a controller-specific worker service class.
- Make methods of the worker service class accept data as it comes from the model binder.
- Make methods of the worker service class return data expressed as view-model objects that are ready to be passed down to the view engine.
- Grab exceptions via attributes.

- Use .NET Code Contracts for checking preconditions and, where applicable, ensure postconditions.
- For anything else that requires a bit of logic, consider using custom action filters.

Let's see how I envision a worker service class.

Worker Services

A worker service is a helper class that goes hand in hand with the controller. You might reasonably expect to have a distinct worker service class for each controller. On the other hand, the worker service is just an extension of a controller and results from the logical split of the controller behavior pushed by the RDD Coordinator role.

I'm using the word *service* here to indicate that this class provides a service to callers—it has nothing to do with any technology for implementing services. Figure 7-1 shows an architectural perspective of worker services in ASP.NET MVC.

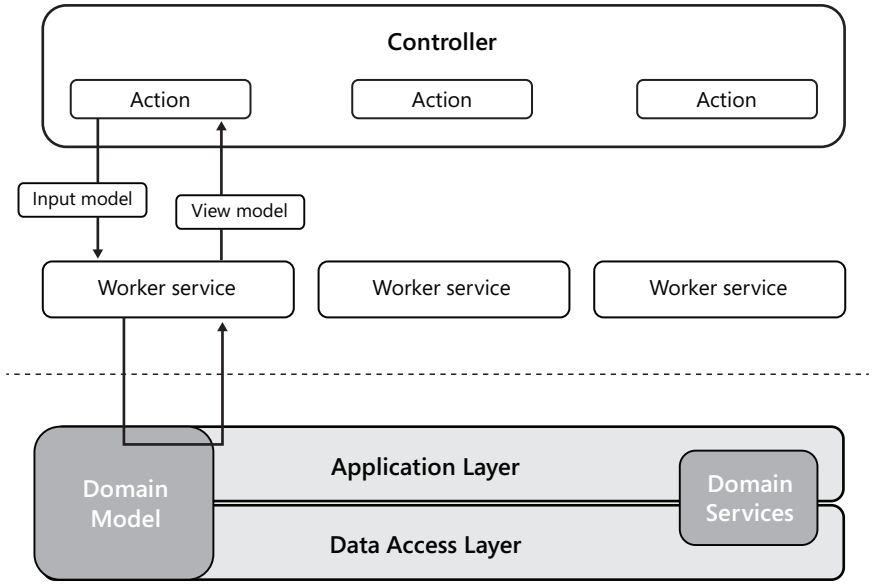

FIGURE 7-1 Worker services and controllers.

A worker service is just a matter of design, and design is design regardless of its complexity. So you don't have to wait for a giant project to experiment with these features. Let's go through a simple example that shows the power of the worker service approach. Admittedly, it might sound like a lot of work to do for a simple demo, but in the end it costs you just an extra interface—and it scales exceptionally well with complexity of the domain.

Implementing a Worker Service

You can start by creating a WorkerServices folder in your ASP.NET MVC project. Which folders you create under it is entirely your responsibility. I usually go with one folder for each controller plus an extra folder for interfaces. Figure 7-2 shows a glimpse of a project using this approach.

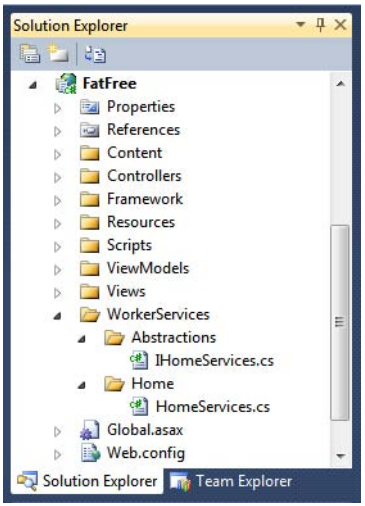

FIGURE 7-2 Worker services in an ASP.NET MVC project.

If you prefer, you can move the WorkerServices section to a separate assembly—it's your call. As mentioned, you create one worker service for each controller. For the *Home* controller, you can create the *IHomeServices* interface and the *HomeServices* class:

```
public interface IHomeServices
{
    HomeViewModel GetHomeViewModel();
}
public class HomeServices : IHomeServices
{
    private IHomeServices _workerService;
    public HomeViewModel GetHomeViewModel()
    {
       ...
    }
    ...
}
```

In the sample application we're considering, the home page picks up a list of featured dates and renders the time span in days between those days and the current day. On the middle tier, you have a repository that returns information about featured dates such as the date, whether it is absolute

or relative (for example, February 8, regardless of the year), and a description for the date. Here's an example for a featured date object for a domain model:

```
namespace FatFree.Framework.DomainModel
{
    public class MementoDate
    {
        public DateTime Date { get; set; }
        public String Description { get; set; }
        public Boolean IsRelative { get; set; }
    }
}
```

The repository will likely fill up a collection of these objects when querying some database. At any rate, the worker service gets a collection of *MementoDate* objects and processes them up to the point of obtaining a collection of *FeaturedDate* objects—a type that belongs to another object model, the view model:

```
namespace FatFree.ViewModels
{
    public class FeaturedDate
    {
        public DateTime Date { get; set; }
        public Int32 DaysToGo { get; set; }
        public String Description { get; set; }
    }
}
```

There are two operations that need be done. First, any relative date must be transformed into an absolute date. Second, the time span between the given date and the current day must be calculated. For example, suppose you want to calculate the distance to the next occurrence of February 8. The target date is different if you're computing January 2 or March 5.

Here's a portion of the code in the worker service:

```
private IDateRepository _repository;
...
public HomeViewModel GetHomeViewModel()
{
    // Get featured dates from the middle tier
    var dates = _repository.GetFeaturedDates();

    // Adjust featured dates for the view
    //  for example, calculate distance from now to specified dates
    var featuredDates = new List<FeaturedDate>();
    foreach(var mementoDate in dates)
    {
        var fd = new FeaturedDate
                        {
                            Description = mementoDate.Description,
                            Date = mementoDate.IsRelative
                                       ? DateTime.Now.Next(mementoDate.Date.Month,
                                                           mementoDate.Date.Day)
```

```
                                        : mementoDate.Date
                    };
        fd.DaysToGo = (Int32)(DateTime.Now - fd.Date).TotalDays;
        featuredDates.Add(fd);
    }

    // Package data into the view model as the view engine expects
    var model = new HomeViewModel
                {
                    Title = "Memento (BETA)",
                    MessageFormat = "Today is <span class='dateEmphasis'>{0}</span>",
                    Today = DateTime.Now.ToString("dddd, dd MMMM yyyy"),
                    FeaturedDates = featuredDates
                };
    return model;
}
```

What about the controller? Here is the code you need:

```
public ActionResult Index()
{
    var model = _workerService.GetHomeViewModel();
    return View(model);
}
```

Figure 7-3 shows the sample page in action.

FIGURE 7-3 Worker services processing dates.

As you can see, there's no magic behind worker services. As the name suggests, they are worker classes that just break up the code that would logically belong to the processor of the request.

Do We Really Need Controllers?

The code of each controller method will hardly be as simple as what I've shown here, which was just one logical line. In real-world scenarios, you might need to pass some input data to the worker service—perhaps use an *if* statement to quickly rule out some cases, or even further edit the view-model object. This latter scenario might occur when you attempt to gain some reusability and get one worker service method to serve the needs of two or more controllers' action methods.

To flesh out the code in action methods, use exception handling, null checks, and preconditions. In the end, to keep action methods lean and mean you need to push the RDD Coordinator role to the limit and move any processing logic out of the controller.

Does this mean that you don't need controllers anymore? Each HTTP request maps to an action method, but you need some plumbing to make the connection. In ASP.NET MVC, the controller is just part of the infrastructure, it shouldn't contain much of your code and, big surprise, there should be no need to test it. If you consider controllers to be part of the infrastructure, then you take their basic behavior for granted; you need to test your worker services instead.

The Ideal Action Method Code

Let's top off this discussion by analyzing an ideal fragment of code you should find in your action methods. It uses attributes to handle exceptions and Code Contracts to determine preconditions:

```
[HandleError(...)]
public class DateController : Controller
{
    private readonly IDateServices _workerService;
    public DateController() : this(new DateServices())
    {
    }
    public DateController(IDateServices service)
    {
        _workerService = service;
    }

    [MementoInvalidDateException]
    [MementoDateExistsException]
    [HttpPost]
    public ActionResult Add(DateTime date, String description)
    {
       Contract.Requires<ArgumentException>(date > DateTime.MinValue);
       Contract.Requires<ArgumentException>(!String.IsNullOrEmpty(description);

       var model = _workerService.AddNewDate(date, description);
       return View(model);
    }
}
```

In this example, custom exception attributes are used to catch specific exceptions that might be raised by the worker service. In this case, you don't need to spoil your code with *if*s and null checks. (I have nothing against using *if* statements, but if I can save myself and my peers a few lines of code and still keep code highly readable, well, by all means I do that.)

Important As an attentive reader, you might have noticed that I completely ignored an important point—how to get ahold of an instance of the worker service. And how does the worker service, in turn, get ahold of an instance of the repository? Techniques and tools to inject dependencies in your code is exactly the next topic. (And in the next chapter, I'll say more about injection points and related techniques in the entire ASP.NET MVC framework.)

Connecting the Presentation and Back End

Assuming we've agreed that you don't want to orchestrate the entire flow of logic for a given request within the context of a controller's action method, the next problem to address is where and how you cross the invisible border between the presentation layer and the back end of the application. In Figure 7-1, a subtly dashed line separates worker service blocks from the application layer—the topmost layer in an application back end.

Implementing action methods as RDD coordinators forces you to relay requests to other layers, but at some point you need to cross the border and invoke enterprise services, databases, workflows, and whatever else you might have. So choosing the coordinator route does have an impact on how you organize the downward layers and tiers of your application.

Note Terms such as *layer* and *tier* are often used interchangeably, and sometimes with reason. Generally, though, a layer and a tier are neatly distinct things. A *layer* refers to a logical separation, such as introducing a different assembly in the same process space. A *tier* refers to something physical, such as a software module that, although reachable, lives in a different process space and perhaps is hosted on a different hardware/software platform. To call a tier, you need data serialization, contracts, and likely a service technology such as Windows Communication Foundation (WCF) in the .NET space.

The iPODD Pattern

Everybody agrees that a multilayer system has a number of benefits in terms of maintainability, ease of implementation, extensibility, and testability. Most of the time, you arrange a three-level architecture with some flavors of service orientation just to make each layer ready for a possible move to a different physical tier. There are various reasons to move a layer onto its own tier: a quest for increased scalability, the need for stricter security measures, and also increased reliability in case the layers become decoupled because of machine failure.

In a three-level scenario, you typically have a presentation segment where you first take care of processing any user input and then arrange responses, a business logic segment that includes all the functional algorithms and calculations that make the system work and interact with other components, and the data access segment where you find all the logic required to read and write from storage.

Although this layout is still rock-solid in general terms, it probably needs to be refreshed in light of technologies available and findings and progress made in the industry with patterns and solutions.

Beyond Classic Layers

Terms like *presentation*, *business*, and *data access* mean everything and nothing today and are becoming quite blurred indeed. How do you really design and implement them? Too many variables apply, and too many choices, patterns, and practices can be adopted. The iPODD pattern described next attempts to expand each of these segments into something more specific and provides guidance on how to do things.

The name iPODD is an acronym for *Idiomatic Presentation, Orchestration, Domain and Data*. Figure 7-4 provides an overall view of the iPODD architecture. Surrounding frames also help to map iPODD blocks to the blocks of a classic three-level architecture. This is just to restate that iPODD is nothing really new, but just a more modern and pragmatic way of applying some old, but still rock-solid theory.

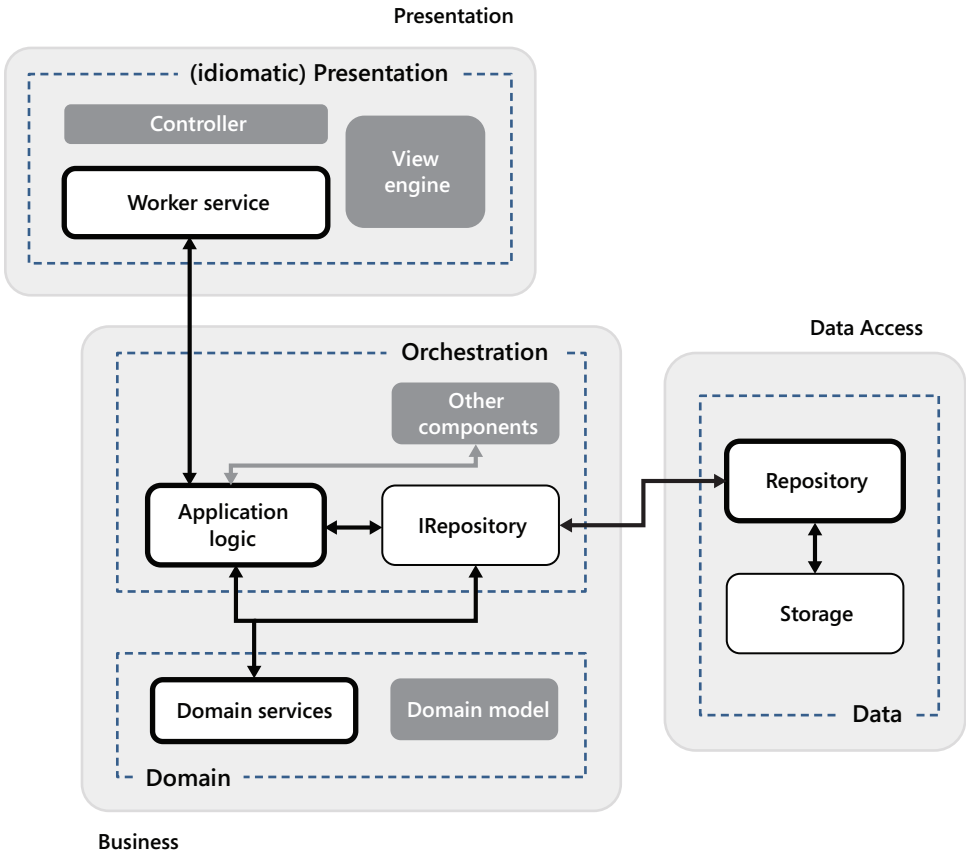

FIGURE 7-4 An example of the iPODD architecture.

The iPODD architecture can be summarized as follows. The presentation layer intercepts a request and relays it to the application layer. The application layer (or service layer) is the segment of the application that implements use-cases. In this regard, it is specific to each application and not reusable. The application layer exposes endpoints to the presentation layer and decouples it from the rest of the system. The application layer orchestrates domain services and the data access layer, as well as external services and business-specific enterprise components you might have. Finally, the data access layer simply reads from and stores to storage.

Note, finally, that storage is not necessarily a relational database. These days, it can be a document database (NoSQL), a cloud data store, or perhaps an enterprise CRM system. It varies to the point that perhaps "data access" is no longer the best term to describe it. I'm leaning toward thinking that "infrastructure layer" is a better term, but then would you really be confident in presenting to your boss architecture devoid of an explicit data access layer?

The (Idiomatic) Presentation Layer

The key lesson that software architects have learned in the past decade is that no application can be built successfully without deeply involving the customer. Already mentioned as one of the key points in the Agile manifesto back in 2001, the collaboration between customers and development team, in reality, is often left to the good will of the involved parties and is kind of neglected. Because of that, it's often limited by time constraints. More often than not, collaboration remains just a good intention that's never entirely pursued.

Today software is used to guide users to do their everyday job in the best and simplest possible way. The software must morph into what users expect, not the other way around—as it was the case for too many years. I have quite a few memories of discussions I had with customers some 25 years ago in which I felt no shame in saying, "No, this feature is impossible to program the way you suggest; the language we're using doesn't support this." Such an answer is hard to imagine today.

Regardless of the technology you employ to build the client side of an application, the presentation layer is the part of the code that collects input from the user and triggers the expected behavior. If the application is distributed, the presentation layer is the segment of code responsible for preparing and executing the remote call and for arranging the new user interface after results are back. The important aspect is that the presentation logic invokes methods designed according to the UI needs. These methods receive and return data formatted for the UI. The rule these days is that the UI receives exactly what it needs, in the form and shape that it likes it. This approach makes you a winner regardless of the various flavors of UI technologies you might encounter—Microsoft Windows, browser, Ajax, Microsoft Silverlight, mobile, and so forth.

When it comes to implementation, the presentation layer is necessarily *idiomatic* in the sense that its actual code depends on the framework you're using. Although the overall idea remains the same, the presentation layer is based on code-behind pages in Web Forms, controllers (plus, optionally, worker services) in ASP.NET MVC, MVVM classes in Silverlight and Windows Presentation Foundation (WPF), and so forth.

As far as ASP.NET MVC is concerned, applying iPODD means delegating the production of a response for a request to a worker service, which in turn will contact the back end to get a response.

The Orchestration Layer

The term *orchestration* refers to the implementation of any algorithm that serves a given use-case. For the use-case "Place the order", the orchestrator is the method that arranges the expected flow of data and coordinates domain services (for example, checking the credit status, refilling the inventory), external services (for example, syncing up with the shipping company), business components (for example, calculating prices), and storage (for example, updating internal databases).

In relatively simple cases, or where you just don't have specific scalability requirements, the controller's worker service might coincide with the orchestration layer. When it comes to this, keep in mind a popular SOA (Service-Oriented Architecture) warning—be chunky, not chatty. (See Figure 7-5.)

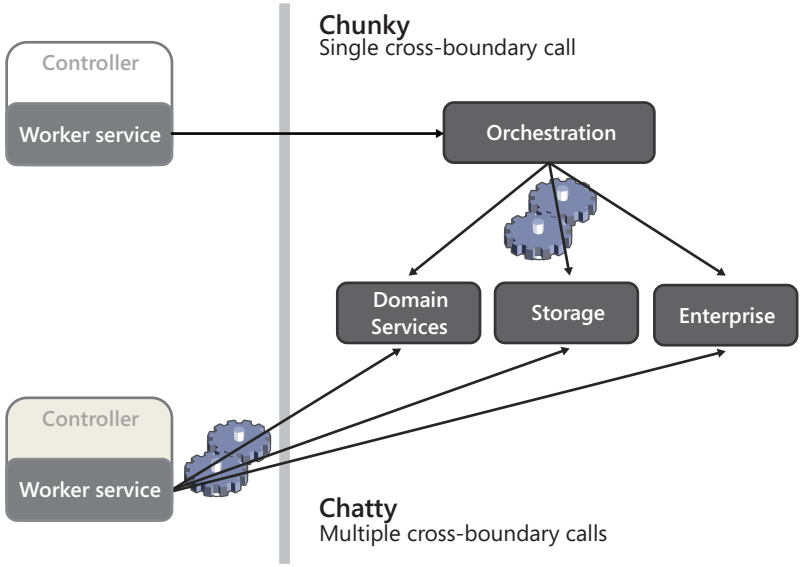

FIGURE 7-5 Chunky vs. chatty orchestration in ASP.NET MVC.

If services you orchestrate live (for the most part) in the same process space as the worker service, you probably don't need to introduce yet another layer. In this case, orchestration coincides with worker services. You might want to compile worker services as a distinct assembly and upgrade their logical rank from a plain helper class to a constituent architectural block.

If services to orchestrate are remote or remotable, you might want to introduce an extra orchestration layer in the same space as the services to orchestrate. In this case, you go from the worker service directly to this additional layer with a single call that wires all the data required for the various steps to take. Orchestrating from another tier—the presentation layer—would cost you a lot in terms of distributed computing, serialization, and network latency. This is the Chatty anti-pattern of SOA design.

The Domain Layer

Although it hasn't been officially declared obsolete, the *DataSet* is a thing of the past for most developers. Not that there's something wrong with the *DataSet* design, it's just that a decade in software is really a long time. The pace of technology is such that a compelling solution devised 10 years ago can hardly have the same effectiveness in the next decade. Today, Entity Framework—not to mention NHibernate and other commercial solutions—makes it really quick to create a basic domain model to design your application around. You can use Entity Framework and its Visual Studio tools to create entities and relationships and make these appear to be the actual database to the application's eyes. You don't need be a Domain-Driven Design (DDD) master to keep database details distinct from the objects you use to implement business operations.

A lot of developers today find it easy and effective to create an entity model and persist it with an Object/Relational Mapper (O/RM) tool, such as Entity Framework or NHibernate.

This leads you to having an assembly with plain-old CLR (POCO) classes padded with data and behavior. This object model represents the model of data you have in the back end of the application. You have classes such as *Customer* with descriptive properties logically equivalent to some database columns: *Name*, *Address*, *Website*, *Email*, *Contact*, *Orders*. In addition, the *Customer* class will likely have a bunch of methods that validate the state of the object and implement object-specific operations on the property values associated with the instance. For example, you might have a method that calculates the total of *Orders* associated with the customer. Or, perhaps, for an *Invoice* class with properties like *Date* and *PaymentMode*, you can have a method that returns the estimated date of payment. In this object model, classes know nothing about persistence and things like connection strings.

Note Object model, domain model, and entity model are all similar terms often used interchangeably. Each term, however, has its own specific meaning. Sometimes, you don't need to go into the level of detail that requires using the precise meaning, so using them interchangeably is just fine. This casual usage, however, doesn't cancel the real meaning that each term holds. An *object model* is a plain, generic collection of objects. A *domain model* is special type of object model where each class is POCO, aggregate roots are identified, factories are used over constructors, and value types tend to replace primitive types most of the time. Finally, an *entity model* is mostly an Entity Framework term that indicates a collection of (mostly anemic) classes, which might or might not be POCO. An anemic class has data but nearly no behavior.

What about more business-oriented behaviors, such as checking the credit status of a customer or perhaps verifying that a customer has placed enough orders to qualify for an elevated level of service or rewards? And, more generally, what about functionalities that produce or manipulate aggregated data that span multiple entities and require database access? These are special forms of orchestration limited to domain entities and their persistence. You implement them through another collection of classes called *domain services*. You likely have a domain service component for each significant entity

or, to use DDD terminology, for each *aggregate root*. The domain model and domain services form the domain layer.

Exposing Entities of the Domain

What's the visibility of the classes in the domain layer? Should they emerge in the presentation layer, or are they destined to live and thrive in the folds of the back-end system? The world would be a better place if one could use domain objects everywhere. If your particular scenario makes it possible to move data around using just one object model—the domain model—by all means call yourself lucky and go ahead. The fact is, however, that this is almost never an option except for conference demos and tutorials.

The presentation is built around use-cases, and each use-case might have a different definition of an entity. The order might have a different representation in use-cases, such as "The user places an order" and "The user reviews pending orders." Sometimes, it is affordable to use the same domain-based representation of the *Order* entity because the various use-cases need only a subset of the original *Order*. More often than not, however, each use-case needs objects that select some information from the original *Order* and some from other entities such as *Products*. In the domain model, you just don't have such aggregates. Hence, a view-focused object model must be created, and data must be transferred to and from it.

The view-focused object model is based on data-transfer objects (DTOs). A DTO is a plain container class (only data, no behavior) that is used to pass data around layers, tiers, and even within the same layer. With DTOs, you can definitely work any place with the data you need. The devil, however, is in the details. A DTO-based solution is expensive and painful to code, period.

To deal with the extra complexity of DTOs, you can leverage tools such as AutoMapper (*http://automapper.codeplex.com*), which saves you from writing repetitive (and boring) code, or you can leverage T4 templates for saving some common code and just write the custom parts yourself.

The Data Layer

How would you get a reference to a domain object? In general, a domain object can be *transient* or *persistent*. It is said to be transient if a new instance is created in memory and populated with run-time data. It is said to be persistent if the instance contains data read from storage. You typically deal with transient entities when you're about to insert a new order; you deal with persistent entities when you fetch an order from storage for display or processing reasons.

In iPODD, the data layer deals with persistence and consists of repository classes—one per significant entity (or, if you prefer, aggregate root). The repository class uses a given storage API to implement persistence. Repositories are classes whose implementation logically belongs to the data access layer. The repository class collects multiple methods that serve the data access needs of that entity. In a repository, you typically find methods to fetch, add, delete, and update data.

The repository exposes an interface to the application layer and uses a storage API internally. So you can have a repository that uses Entity Framework and a POCO model or one that uses NHibernate. You can have a repository that persists the domain through a plain ADO.NET layer.

You can also make the repository point to some cloud storage, Dynamics CRM, or a NoSQL service. Created on a per-entity basis (precisely, major entities only), the repository is the gateway to the actual persistence layer.

Using repositories is important also for another reason—making your business services testable by mocking the persistence layer.

What's the typical structure of a repository class? There are two main schools of thought. Some prefer to have a generic repository that provides basic CRUD methods for each entity:

```
public abstract class Repository<T> where T : IAggregateRoot
{
   internal YourContext ActiveContext { get; set; }
   public Repository()
   {
      ActiveContext = new YourContext();
   }
   public void Add(T item)
   {
      this.AddObject(item);
      this.ActiveContext.SaveChanges();
   }
   public bool Remove(T item)
   {
       try {
          this.DeleteObject(item);
          this.ActiveContext.SaveChanges();
          return true;
       } catch {
          return false;
       }
   }
   public void Update(T item)
   {
       this.ActiveContext.SaveChanges();
   }
   :
}
```

As far as queries are concerned, here's a list of methods you might have in a generic repository class:

```
T[] GetAll<T>();
T[] GetAll<T>(Expression<Func<T, bool>> filter);
T GetSingle<T>(Expression<Func<T, bool>> filter);
```

You pass the details of the query to be executed as a function via either *GetAll* or *GetSingle*.

Others prefer to just have a regular class with as many methods as required by the logic to be implemented. In this case, you end up with a class that is tailor-made to the entity. You can easily

treat special cases for deletions and insertions appropriately and have a specific *get* method to call for each necessary query. In the end, the choice is up to you because none of these approaches is clearly better than the other. Personally, I like to have specific (nongeneric) repositories.

Note Speaking of repositories, there is another point I should mention. It is related to the lifetime of the context object if an O/RM is used to persist data. In the previous listing, you see an *ActiveContext* property instantiated by the constructor. In this way, the context lives as long as the instance is live. Multiple calls you make to the same instance will have access to the same identity map and can track changes. An alternative is to use locally scoped instances that are dismissed at the end of each repository operation.

Injecting Data and Services in Layers

The primary reason for using layers (and tiers) is separation of concerns. As an architect, you determine which layer talks to which layer and you have testing, code inspection, and perhaps check-in policies enforce these rules. Even when two layers are expected to collaborate, however, you don't want them to be tightly coupled. In this regard, the Dependency Inversion Principle—the "D" in SOLID—helps a lot. I'd even say that the Dependency Inversion Principle is much more important than the Dependency Injection pattern, which seems to be on everybody's mind these days.

The Dependency Inversion Principle

Defined, the Dependency Inversion Principle (DIP) states that high-level classes should not depend on lower level classes. High-level classes, instead, should always depend on abstractions of their required lower level classes. In a way, this principle is a specialization of one of the pillars of object-oriented design—program to an interface, not to an implementation.

DIP is the formalization of a top-down approach to defining the behavior of any significant class method. In using this top-down approach, you focus on the work flow that happens at the method level rather than focusing on the implementation of its particular dependencies. At some point, though, lower level classes should be linked to the mainstream code. DIP suggests that this should happen via injection.

In a way, DIP indicates an inversion of the control flow whenever a dependency is met—the main flow doesn't care about details of the dependency as long as it has access to an abstraction of it. The dependency is then injected in some way when necessary. Figure 7-6 shows a personalized version of the classic diagram for the canonical example of DIP as originally presented by Robert Martin in the paper you can find at the following URL: *http://www.objectmentor.com/resources/articles/dip.pdf*.

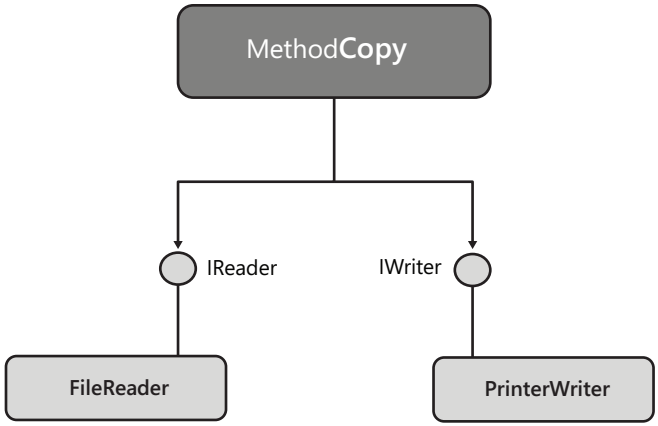

FIGURE 7-6 The DIP diagram.

The paper describes a sample Copy function that reads from a source and writes to a target stream. The Copy function ideally doesn't care about the details of the reader and writer components. It should care only about the interface of the reader and writer. The reader and writer are then injected or resolved in some way around the implementation of the Copy function. How this point is approached depends on the actual pattern you intend to use.

To address DIP, you commonly use either of two patterns: the Service Locator pattern or the Dependency Injection (DI) pattern.

The Service Locator Pattern

The Service Locator pattern defines a component that knows how to retrieve the services an application might need. The caller has no need to specify the concrete type; the caller normally indicates an interface, a base type, or even a nickname of the service in the form of a string or a numeric code.

The Service Locator pattern hides the complexity of component lookup, handles the caching or pooling of instances and, in general, offers a common façade for component lookup and creation. Here's the typical implementation of a service locator:

```
public class ServiceLocator
{
   private static const String SERVICE_QUOTEPROVIDER = "quoteprovider";

   // You might also want to have a generic method GetService<T>()...
   public static Object GetService(Type t)
   {
     if (t == typeof(IQuoteProvider))
     {
        return new SomeQuoteProvider();
     }
```

```
      ...
   }

   public static Object GetService(String serviceName)
   {
      switch(serviceName)
      {
          case SERVICE_QUOTEPROVIDER:
              return new SomeQuoteProvider();
          ...
      }
   }
}
```

As you can see, the locator is merely a wrapper around a *Factory* object that knows how to get an instance of a given (or indirectly referenced) type. Let's have a look now at the code that calls the locator. The following code illustrates a class that first gets quotes for the specified list of symbols and then renders values out to an HTML string:

```
public class FinanceInfoService
{
  public String GetQuotesAsHtml(String symbols)
  {
    // Get dependencies
    var renderer = ServiceLocator.GetService("quoterenderer");
    var provider = ServiceLocator.GetService("quoteprovider");

    // Use dependencies
    var stocks = provider.FindQuoteInfo(symbols);
    var html = renderer.RenderQuoteInfo(stocks);

    return html;
  }
}
```

The locator code lives inside the method that manages the abstraction, and the factory is part of the deal. By simply looking at the signature of the *FinanceInfoService* class, you can't say whether or not it has dependencies on external components. You have to inspect the code of the *GetQuotesAsHtml* method to find it out.

The main Service Locator focus is to achieve the lowest possible amount of coupling between components. The locator represents a centralized console that an application uses to obtain all the external dependencies it needs. In doing so, the Service Locator pattern also produces the pleasant side effect of making your code more flexible and extensible.

Using the Service Locator pattern is not a bad thing from a purely functional perspective. However, in more practical terms a likely better option exists: the DI pattern.

The Dependency Injection Pattern

The biggest difference between Service Locator and DI is that with dependency injection, the factory code lives outside of the class being worked on. The pattern suggests that you design the class in such a way that it receives all of its dependencies from the outside. Here's how to rewrite the *FinanceInfoService* class for making use of DI:

```
public class FinanceInfoService
{
  private IQuoteProvider _provider;
  private IRenderer _renderer;

  public FinanceInfoService(IQuoteProvider provider, IRenderer renderer)
  {
      _provider = provider;
      _renderer = renderer;
  }

  public string GetQuotesAsHtml(string symbols)
  {
    var stocks = _provider.FindQuoteInfo(symbols);
    string html = _renderer.RenderQuoteInfo(stocks);
    return html;
  }
}
```

When it comes to using DI in classes, a critical decision for the developer is about how and where to allow for code injection. There are three ways to inject dependencies into a class—using the constructor, a settable property, or the parameters of a method. All techniques are valid, and the choice is ultimately up to you. In general terms, the consensus is to use constructors for necessary dependencies and setters for optional dependencies. However, some considerations apply.

What if you have many dependencies? In this case, your constructor would look dangerously messy. Even though a long list of parameters in the constructor is often the sign of some design issues, this isn't a hard-and-fast rule. You might encounter situations where you have complex constructors with many parameters. In this case, grouping dependencies in a compound object is a solution. In a nutshell, your goal should be to reveal dependencies and intentions right at construction time. This can be done in two ways: via a set of classic constructors you manage to keep as simple as possible or via factories.

Factories are the preferred approach in the Domain-Driven Design (DDD) methodology. Using a factory, you can express more clearly the context in which you need an instance of the type. You can also deal with dependencies inside the factory code and ensure you return valid objects from the beginning. In addition, your classes end up having only the default constructor (probably implemented as a protected member).

Using constructors also hinders inheritance because derived classes might have the need to receive dependencies as well. When you add a new dependency, this design scheme might require more refactoring work.

When the dependency is optional, however, there's no strict need to make it show up at the constructor level. In this case, using a setter property is fine and probably the recommended approach because it helps keep the constructor (or factory code) leaner and cleaner.

In summary, there are good reasons for using the constructor and good reasons for going with setter properties. As with many other architectural questions, the right answer is, "It depends." And it depends also on your personal taste.

Using Tools for Inversion-of-Control

Dependency injection takes any code that relates to the setup of dependencies out of the class. When dependencies are nested, this code might be quite a few lines long; furthermore, for the most part it is boilerplate code. For this reason, ad hoc frameworks have been created by developers—they're known as Inversion of Control (IoC) frameworks. An IoC container is a framework specifically created to support DI. It can be considered a productivity tool for implementing DI quickly and effectively. From the perspective of an application, a container is a rich factory that provides access to external objects to be retrieved and consumed later.

All IoC frameworks are built around a container object that, when bound to some configuration information, resolves dependencies. The caller code instantiates the container and passes the desired interface as an argument. In response, the IoC framework returns a concrete object that implements that interface. An IoC container holds a dictionary of type mappings where typically an abstract type (for example, an interface) is mapped to a concrete type or an instance of a given concrete type. Table 7-3 lists some of the most popular IoC frameworks available today.

TABLE 7-3 Popular IoC frameworks

Framework	URL
Autofac	*http://code.google.com/p/autofac*
Castle Windsor	*http://www.castleproject.org*
Ninject	*http://www.ninject.org*
Spring.NET	*http://www.springframework.net*
StructureMap	*http://structuremap.net/structuremap/index.html*
Unity	*http://unity.codeplex.com*

After it is configured, an IoC container gives you the ability to resolve the whole chain of dependencies between your types with a single call. And you save yourself all the intricacies of inner dependencies. For example, if you have some *ISomeService* parameter in a class constructor or

property, you can be sure you'll get it at run time as long as you tell the IoC container to resolve it. The beauty of this approach is that if the constructor of the concrete type mapped to *ISomeService* has its own dependencies, these are resolved as well and automatically.

Take this further and you see the point: with an IoC container, you stop caring about the cloud of dependencies. Furthermore, all you do is design the graph of dependencies using the syntax supported by the IoC of choice. Everything else happens free of charge.

IoC containers differ in terms of the syntax they support (for example, lambda expressions), the configuration policies used (for example, the external XML scheme), plus additional features that are available. Two features are gaining a lot of importance today: aspect-orientation capabilities (specifically, interception) and specialized modules that facilitate integration with WCF services.

Note With regard to Unity—the Microsoft's IoC library—you can find coverage of advanced features, including interception, in my Cutting Edge column in MSDN Magazine, specifically, in the January/February 2011 issues. An excellent article that can help you understand how to use Unity to inject dependencies during the initialization of a WCF service can be found at *http://blogs.microsoft.co.il/blogs/gadib/archive/2010/11/30/wcf-and-unity-2-0.aspx*.

Poor Man's Dependency Injection

Today, many tend to confuse the Dependency Injection pattern with using an IoC framework. Using an IoC framework is a matter of productivity; as such, it often requires a minimum critical mass of complexity to become really effective. In simpler cases, you can opt for what many like to call the poor man's dependency injection. You find an example of this technique in the source code of the FatFree example you saw back in Figure 7-3.

How would you inject a worker service in a controller class and a repository in a worker service? The most obvious way is by letting controllers and services create a fresh new instance of the dependent object. This route, however, creates a tight dependency between objects that hinders extensibility and testability. Here's a better approach:

```
public class HomeController : Controller
{
    private readonly IHomeServices _workerService;
    public HomeController() : this(new HomeServices())
    {
    }
    public HomeController(IHomeServices service)
    {
        _workerService = service;
    }
    ...
}
```

When the default constructor is used to instantiate a controller, the worker service member points to a freshly created instance of a default worker service class. A second constructor is available to manually inject any instance you like, at least for testability reasons. Likewise, you do the same for injecting the repository dependency into the worker service.

ASP.NET MVC, however, always uses the default constructor for each controller class, unless you gain control of the controller factory.

Gaining Control of the Controller Factory

In ASP.NET MVC, the instantiation of the controller class is a topical moment. The ASP.NET MVC infrastructure includes a factory that uses the default constructor of the selected controller class. What if you have parameterized constructors on your controller class and need to pass in some data? This scenario is not supported out of the box, but the extremely extensible design of ASP.NET MVC offers a hook for you to replace the default controller factory with your own.

A common way to replace the default controller factory is to integrate an IoC container in it so that any parameter can be resolved brilliantly by looking at the table of registered types. The following sections explain how to do it.

Registering a Custom Controller Factory

It all starts in *Application_Start*, where you register your own controller factory. A controller factory is a class that implements the *IControllerFactory* interface. To register the factory, you pass an instance of the *SetControllerFactory* method to the current instance of the controller builder:

```
protected void Application_Start()
{
    // Register a custom controller factory
    RegisterControllerFactory();
    ...
}
public static void RegisterControllerFactory()
{
    var factory = new UnityControllerFactory();
    ControllerBuilder.Current.SetControllerFactory(factory);
}
```

Let's see a controller factory from the inside.

Building a Customer Controller Factory

A typical controller factory class inherits from *DefaultControllerFactory* and overrides a few methods as detailed in Table 7-4.

TABLE 7-4 Popular IoC frameworks

Overridable methods	Description
CreateController	Governs the creation of a controller instance. It takes the name of the controller (for example, home) and returns a controller instance. The default implementation first invokes *GetControllerType* to map the controller name to a type and then calls *GetControllerInstance* to actually create an instance.
GetControllerInstance	Gets a controller type and the request context, and returns a newly created instance of the controller.
GetControllerSessionBehavior	Gets the session state behavior for the controller type. Note: *You override this method if you want to control programmatically how the session state behavior is determined. By default, it is controlled by the* SessionState *attribute.*
GetControllerType	Gets the controller name and the request context, and returns the expected type of the controller. Note: *You override this method if you don't like the convention that the controller type is always given by* name *followed by "Controller".*
ReleaseController	Performs any cleanup task for when the controller instance is dismissed.

Note that the *CreateController* and *ReleaseController* methods are public; all of the other methods are protected. Creating your own factory also allows you to perform any sort of custom work on freshly created instances of controllers. For example, you might want to centralize the initialization of some custom properties (if you're deriving your controllers from a base class). More likely, you might want to use this hook to give a controller instance a handmade action invoker. (I'll discuss action invokers in the next chapter.)

A Unity-Based Controller Factory

When you introduce a custom factory based on Unity, at a minimum you want to override *GetControllerInstance* to use the Unity infrastructure to resolve the controller type:

```
public class UnityControllerFactory : DefaultControllerFactory
{
    public static IUnityContainer Container { get; private set; }

    public UnityControllerFactory()
    {
        Container = new UnityContainer();
        Container.LoadConfiguration();
    }

    protected override IController GetControllerInstance(
                        RequestContext requestContext, Type controllerType)
    {
        if (controllerType == null)
            return null;
        return Container.Resolve(controllerType) as IController;
    }
}
```

Using the Unity engine guarantees that further dependencies rooted in the controller type are identified and resolved. With reference to the FatFree example, this is just what happens with worker service and date repository dependencies.

The code just shown looks pretty generic—and that's the reason that ultimately made IoC tools so successful. IoC tools enhance productivity because they save you a lot of boilerplate code! Where are the details?

Just like any other IoC framework, Unity needs some configuration data. You can place this data in the *web.config* file or use the Unity API to add it programmatically. In the end, this configuration data consists of specifying which interface type maps to which concrete type and where these types can be found. Here's a modified *web.config* file that contains a Unity section:

```
<configuration>
  <configSections>
    <section name="unity" type="Microsoft.Practices.Unity.Configuration.
UnityConfigurationSection,
                                Microsoft.Practices.Unity.Configuration, ..." />
  </configSections>

  ...
  <unity xmlns="http://schemas.microsoft.com/practices/2010/unity">
    <assembly name="IoCFatFree" />
    <namespace name="IoCFatFree.Framework.DAL" />
    <namespace name="IoCFatFree.WorkerServices.Abstractions" />
    <namespace name="IoCFatFree.WorkerServices.Home" />
    <container>
      <register type="IHomeServices" mapTo="HomeServices">
      </register>
      <register type="IDateRepository" mapTo="DateRepository">
      </register>
    </container>
  </unity>
</configuration>
```

The *namespace* entry registers a namespace that Unity uses to qualify types. The *assembly* entry lists assemblies to be considered to resolve types. Under the *container* group, you register types. The *register* node can contain a bunch of child nodes to provide additional information for injection and interception. (These aspects are not relevant in this book. For more information, refer to the Unity official documentation online at *http://msdn.microsoft.com/en-us/library/ff663144.aspx*.)

Summary

In this chapter, I tried to call each entity involved in a typical ASP.NET MVC application with its own name and role. I wouldn't be surprised if, at first glance, you're tempted to dismiss the material and be a bit annoyed by what seems to be unnecessary complexity. Why on earth should you create a *worker service* that is apparently just a cool way to waste CPU cycles?

If the complexity of your application compares to that of many tutorials available out there (for example, Music Store), good for you. You should go ahead and call the Entity Framework data context right from action methods and forget about layers and layered architecture. Quite a few applications can blissfully and happily be designed in this way. The idea of ignoring entry-level tutorials couldn't be more foreign to me. They do a great job of getting you started and demonstrating things concretely.

It should be clear, however, that it is only the first step. Many common practices just don't scale well with increased complexity of the model and domain. In this chapter, I summarized a few practices that take your ASP.NET MVC application up to the next level of complexity. A key good practice is keeping the controller class lean and mean by moving most of the logic out to worker services. Another good practice is keeping the view as humble and passive as possible by moving code to HTML helpers and controllers—preferably through strongly typed view models, but render actions also are acceptable. Yet another practice is layering the back end of the application to distinguish between domain model, domain services, and repositories. Repositories should be the only way to access data, and they should be invoked by services whenever possible, thus shielding controllers from them.

So are simpler tutorials the wrong way to demonstrate these things?? No, as long as you consider the code in them as a special case of more general patterns discussed in this chapter (and in the rest of the book, to a good extent).

For simple applications, ASP.NET MVC is an easy framework to use; for complex applications, it might get tricky. But if you manage it well, your resulting code will definitely be high-quality code—much more than in Web Forms, assuming a similar level of effort and skills.

CHAPTER 8

Customizing ASP.NET MVC Controllers

We need men who can dream of things that never were.

—John F. Kennedy

The whole ASP.NET MVC stack is full of extensibility points. An extensibility point is a place in the code where the actual behavior to occur can be read from an external and replaceable provider. In the previous chapters, you saw a few examples of extensibility points. You saw, for example, how to replace the controller factory—that is, the component that returns fresh instances of controller classes for each incoming request. (See Chapter 7, "Design Considerations for ASP.NET MVC Controllers.") In Chapter 4, "Input Forms," you also saw how to replace a model metadata provider—that is, the component that reads meta information about classes to be used in HTML forms.

In a programming framework such as ASP.NET MVC, the benefits of an extensible design will ripple across any applications built on top of it. In this chapter, my goal is to help you discover the points of extensibility you find in ASP.NET MVC and to illustrate them with a few examples. I'll start with a brief discussion of the extensibility model in ASP.NET MVC and then proceed with a thorough examination of aspects of controllers you can customize at your leisure and that will result in lean and mean controllers, as well as improved maintainability and readability.

The Extensibility Model of ASP.NET MVC

An extensibility point is tightly coupled with an extensibility model—the programming model according to which a developer can unplug the existing implementation of a component and roll his own. In ASP.NET MVC, you have two functionally equivalent extensibility models that differ in the API they require you to use. One is based on providers you register explicitly using a fluent syntax. The other is based on the classic Service Locator pattern and is introduced to let developers benefit more and more from Inversion of Control (IoC) frameworks.

These two models cover the replacement of internal components, such as view engines, action invokers, metadata providers, and factories. Another segment of the ASP.NET MVC extensibility model is based on special attributes—named *action filters*—that you use to inject custom (and even

optional) code in the execution flow of controller methods. Let's tackle the replacement of internal components first.

The Provider-Based Model

Replaceable internal components of the ASP.NET MVC stack exist for very specific situations. You don't want to take advantage of them in just any application you write. Let's say that these extensibility points are there just in case they're needed. If you, at some point, decide to replace them, be aware that you're interacting with the internal machinery of ASP.NET MVC. Your components, therefore, are the first suspect in the case of performance or functional issues.

Gallery of Extensibility Points

Table 8-1 provides a quick list of extensibility points.

TABLE 8-1 Replaceable components

Provider	Interface or Base Class	Description
Action invoker	*DefaultActionInvoker*	Governs the execution of action methods and the generation of the browser response. It interacts with filters and view engines. (I'll say more about this later.)
Controller factory	*DefaultControllerFactory*	Serves as the factory for controller instances. It can be used to set a custom action invoker for the freshly created controller instance.
Dictionary values	*IValueProvider*	Reads values to be added to the dictionary of input values for a request. Built-in value providers read from query strings, forms, input files, and routes. Custom providers might read from cookies or HTTP headers.
Model binder	*IModelBinder*	Defines a binder type that transforms raw values in some value provider dictionaries into a specific complex type.
Model binder provider	*IModelBinderProvider*	Defines a model binder factory that dynamically selects the right model binder to use for a given type.
Model metadata	*ModelMetadataProvider*	Retrieves meta information about members of a class, and associates that information with any instance of that type. The default provider reads meta information from *DataAnnotations* attributes.
Model validator	*ModelValidatorProvider*	The description is the same as for the preceding model metadata item, except that it focuses on validation aspects. The default provider reads meta information from validation *DataAnnotations* attributes.
TempData	*ITempDataProvider*	Serves as the storage of any data being placed in the *TempData* collection. The default provider uses session state.
View engine	*IViewEngine*	Serves as a component capable of interpreting a view template and producing HTML markup for the browser. Default view engines can translate ASPX and Razor markup to HTML.

As you can see, these are not the components you use every day. I've customized most of them at least once in my years of programming, but usually I haven't customized more than one or two components at a time.

Realistic Scenarios: Culture-Driven View Engine

This section provides a quick list of scenarios in which I used customized versions of some of the replaceable components listed in Table 8-1.

I tend to use a custom view engine to change the location of some views or to introduce localization capabilities. In particular, in Ajax-intensive applications you end up with tons of partial views and you might want to keep them separate from regular views, or even use some custom logic to locate them. In these cases, I derive a custom view engine from one of the default engines (for example, Razor) and modify the location formats as discussed in Chapter 2, "ASP.NET MVC Views." In Chapter 5, "Aspects of ASP.NET MVC Applications," I presented a couple of extension methods to enable the rendering of localized views—for example, *demo.it.cshtml* instead of *demo.cshtml* when the current culture is *it*. This approach has the benefit of giving you granular control over each and every controller method. However, it requires that you specifically select localized viewsfor each method. Another option is integrating the same localization logic into a custom view engine. Here's a sample view engine that automatically checks for partial and regular views with localized names according to the current culture:

```
public class LocalizableRazorViewEngine : RazorViewEngine
{
    protected override IView CreatePartialView(ControllerContext controllerContext,
                                               String partialPath)
    {
        var localizedPath = partialPath;
        if (!String.IsNullOrEmpty(partialPath))
            localizedPath = GetLocalizedViewName(controllerContext.RequestContext, partialPath);
        return base.CreatePartialView(controllerContext, localizedPath);
    }

    protected override IView CreateView(ControllerContext controllerContext,
                                        String viewPath, String masterPath)
    {
        var localizedView = viewPath;
        var localizedMaster = masterPath;
        if (!String.IsNullOrEmpty(viewPath))
            localizedView = GetLocalizedViewName(controllerContext.RequestContext, viewPath);
        if (!String.IsNullOrEmpty(masterPath))
            localizedMaster = GetLocalizedViewName(controllerContext.RequestContext,
masterPath);
        return base.CreateView(controllerContext, localizedView, localizedMaster);
    }

    private static String GetLocalizedViewName(RequestContext context, String view)
    {
        var urlHelper = new UrlHelper(context);
        return UrlExtensions.GetLocalizedUrl(urlHelper, view);  // See companion source code
    }
}
```

In Chapter 2, I briefly described another sample view engine that resolves partial views from a different location. Note that although you can add multiple view engines to your solution, their capabilities don't stack up. The action invoker searches for the first registered engine that can serve a view. After that, the view is resolved according to the capabilities of the selected engine. For this reason, it's preferable that you code in a single custom engine all the features you think you'll need.

Important Keep in mind that if you add a custom view engine that uses a nonstandard root folder (for example, *PartialViews*), it's required that you copy in that folder the *web.config* file that is located by default under Views. That file helps resolve the base class of views.

Realistic Scenarios: Alternate *TempData* Storage

The *TempData* dictionary is used to store data that logically belongs to the current request but must survive across a redirect. We used *TempData* in Chapter 4 while discussing the PRG pattern for input forms. According to the PRG pattern, you should terminate a POST request with a GET redirect to the view that displays results. The entire state of the request (for example, validation messages) might be lost across the redirect. The *TempData* dictionary provides a temporary store for any data (mostly *ModelState*) you need.

By default, the *TempData* dictionary saves its content to the session state. The key difference between using *Session* directly or through *TempData* is that any data stored in *TempData* is automatically cleared up after the successive request terminates. In other words, data stays in memory for just two requests—the current request and the next redirect. *TempData*, in the end, puts much less pressure on memory.

What if your application is not allowed to use session state?

Either you work out an entirely different solution based on the query string or you just provide different storage for *TempData* content. What would be different storage? It can be a cookie (if the overall size of your data matches the limitations on cookie size), or it can be a (distributed) cache. If you choose the latter, storing the data on a per-user basis is entirely your responsibility. Here's what a custom *TempData* provider looks like:

```
public class CookieTempDataProvider : ITempDataProvider
{
   protected override IDictionary<String, Object> LoadTempData(ControllerContext
controllerContext)
   { ... }
   protected override void SaveTempData(ControllerContext controllerContext,
                                        IDictionary<String, Object> values)
   { ... }
}
```

A working example of this class (as well as many other replacements for the components listed in Table 8-1) can be found by simply checking *StackOverflow* or some of the open-source projects that have arisen around ASP.NET MVC. An interesting one is *Mvc Futures*, which contains high-quality

components that Microsoft is considering for inclusion in future versions of ASP.NET MVC. (See *http://aspnet.codeplex.com/releases/view/58781*.)

Using Custom Components in Your Applications

Until ASP.NET MVC 3, there was no standard way to register custom components. Each component listed in Table 8-1 requires its own API to be integrated into a user application. As you'll see in a moment, ASP.NET MVC 3 introduces a new model based on dependency resolvers that is for the most part parallel to the custom model supported until ASP.NET MVC 3. Table 8-2 describes how to register the special replaceable components listed in Table 8-1.

TABLE 8-2 Registering replaceable components

Provider	Description
Action invoker	`// In the constructor of a controller class` `this.ActionInvoker = new YourActionInvoker();`

Controller factory	`// In global.asax, Application_Start`
	`var factory = new YourControllerFactory();`
	`ControllerBuilder.Current.SetControllerFactory(factory);`

Dictionary values	`// In global.asax, Application_Start`
	`var providerFactory = new YourValueProviderFactory();`
	`ValueProviderFactories.Factories.Add(providerFactory);`

Model binder	`// In global.asax, Application_Start` `ModelBinders.Binders.Add(typeof(YourType), new YourTypeBinder());`

Model binder provider	`// In global.asax, Application_Start`
	`var provider = new YourModelBinderProvider();`
	`ModelBinderProviders.BinderProviders.Add(provider);`

Model metadata	`// In global.asax, Application_Start ModelMetadataProviders.` `Current = new YourModelMetadataProvider();`

Model validator	`// In global.asax, Application_Start`
	`var validator = new YourModelValidatorProvider();`
	`ModelValidatorProviders.Providers.Add(validator);`

TempData	`// In the constructor of a controller class this.TempDataProvider` `= new YourTempDataProvider();`

View engine	`// In global.asax, Application_Start`
	`ViewEngines.Engines.Clear();`
	`ViewEngines.Engines.Add(new YourViewEngine());`

A few more words should be spent on value providers. As the code snippets in Table 8-2 show, you don't manage a collection of value providers, but a collection of value provider factories. This means that for each custom value provider you intend to add, you should write two distinct classes: the value provider and its factory. Only the factory class, though, should be registered with the system at startup. The value provider factory is a thin wrapper, as shown here:

```
public class HttpCookieValueProviderFactory : ValueProviderFactory
{
   public override IValueProvider GetValueProvider(ControllerContext controllerContext)
   {
      return new HttpCookieValueProvider(controllerContext);
   }
}
```

The class represents a factory for a custom value provider; the full source code for such a cookie value provider can be found in MVC Futures.

The Service Locator Model

Service Locator is a popular pattern used in the design of loosely coupled systems. The core of the pattern is a globally accessible factory class responsible for serving instances of classes that implement a given contract. The basic interaction taking place between a service locator and its clients can be summarized as a conversation that starts like this: *I need an object that behaves like this type; do you know about some concrete type you can instantiate for me*? The service locator then replies by returning such an instance or null.

The term "Service Locator" (and variations such as "Service Location") is often used to indicate a scenario in which a class gets external dependencies, thus remaining open for extension but closed for modifications. What's the difference, really, between the Service Locator (or Location) pattern and Dependency Injection?

Service Locator vs. Dependency Injection

Functionally speaking, the Service Locator (SL) pattern is nearly identical to Dependency Injection (DI), and both are concrete implementations of the Dependency Inversion Principle—the "D" in the popular SOLID acronym. Service Locator is historically the first pattern that was widely employed in the building of loosely coupled, testable systems. Later on, DI was introduced as a slightly better way of doing the same things. The difference between SL and DI is more or less blurred, depending on the level of abstraction from which you're looking at them. For sure, there's a difference at the source code level. With SL, a class queries an external component for getting its dependencies. With DI, a class is given its dependencies through the constructor (or public properties).

Here's what a class that uses SL looks like:

```
public class FinanceInfoService
{
  private IFinder _finder;
  private IRenderer _renderer;

  public FinanceInfoService()
  {
     _finder = ServiceLocator.GetService<IFinder>();
    _renderer = ServiceLocator.GetService<IRenderer>();
  }

  public String GetQuotesAsHtml(String symbols)
  {
    var stocks = _finder.FindQuoteInfo(symbols);
    return _renderer.RenderQuoteInfo(stocks);
  }
}
```

The same class refactored to use DI looks like this:

```
public class FinanceInfoService
{
  private IFinder _finder;
  private IRenderer _renderer;

  public FinanceInfoService(IFinder f, IRenderer r)
  {
      _finder = f;
      _renderer = r;
  }

  public string GetQuotesAsHtml(string symbols)
  {
    var stocks = _finder.FindQuoteInfo(symbols);
    return _renderer.RenderQuoteInfo(stocks);
  }
}
```

The difference is all in the constructor and in how dependencies are retrieved. Which approach should you use?

For new systems built from scratch, DI is preferable—it keeps your code cleaner and makes it easier to read and test. With DI, dependencies are explicit in the class signature. The surrounding framework, however, is entirely responsible for preparing dependencies and injecting them.

Service Locator in ASP.NET MVC

ASP.NET MVC is designed with several extensibility points but lacks comprehensive support for Dependency Injection. The various APIs listed in Table 8-2 prove this statement beyond any reasonable doubt. To make an existing framework more loosely coupled through the addition of new extensibility points, a service locator is probably the most effective because it's the least intrusive solution. A service locator acts as a black box that you install in a specific point and then let it figure out what contract is required and how to get it.

Parallel to the APIs listed in Table 8-2, ASP.NET MVC offers you the ability to register your own service locator that the framework will use any time it needs to resolve a dependency. The service locator is optional, and using it or sticking with the APIs listed in Table 8-2 is equivalent in terms of functionality.

When resolving a dependency, ASP.NET MVC always uses the internal service locator first. The actual behavior changes slightly if the dependency can be registered singly or multiple times. From a developer's perspective, however, all that matters is that if a custom component is registered in two ways (using the service locator or any of the APIs in Table 8-2), the service locator is given precedence.

Note A singly registered component is a component that must be unique to the application. An example is the controller factory—you can't have two factories in the same application. Examples of components registered multiple times are the view engine and the model validator providers. In both cases, you can have multiple instances registered and exposed to the rest of the system.

Defining Your Dependency Resolver

The implementation of the ASP.NET MVC service locator consists of a thin wrapper around a user-defined object known as the *dependency resolver*. A dependency resolver—which is unique to each application—is an object that implements the following interface:

```
public interface IDependencyResolver
{
   Object GetService(Type serviceType);
   IEnumerable<Object> GetServices(Type serviceType);
}
```

The logic you put in the resolver is entirely up to you. It can be as simple as a *switch* statement that checks the type and returns a newly created instance of a fixed type. It can be made more sophisticated by reading information from a configuration file and using reflection to create instances. Finally, it can be based on Unity or any other IoC framework. Here's a simple, yet functional, resolver:

```
public class SampleDependencyResolver : IDependencyResolver
{
    public object GetService(Type serviceType)
    {
        if (serviceType == typeof(ISomeClass))
             return new SomeClass();
        ...
    }

    public IEnumerable<object> GetServices(Type serviceType)
    {
        return Enumerable.Empty<Object>();
    }
}
```

The code shown next is an example of a resolver that uses Unity (and its configuration section) to resolve dependencies:

```
public class UnityDependencyResolver : IDependencyResolver
{
  private readonly IUnityContainer _container;
```

```
  public UnityDependencyResolver() : this(new UnityContainer().LoadConfiguration())
  {
  }
  public UnityDependencyResolver(IUnityContainer container)
  {
    _container = container;
  }

  public Object GetService(Type serviceType)
  {
    return _container.Resolve(serviceType);
  }

  public IEnumerable<Object> GetServices(Type serviceType)
  {
    return _container.ResolveAll(serviceType);
  }
}
```

You register your own resolver with the ASP.NET MVC framework through the *SetResolver* method of the *DependencyResolver* class, as shown here:

```
protected void Application_Start()
{
   // Prepare and configure the IoC container
   var container = new UnityContainer();
   ...

   // Create and register the resolver
   var resolver = new UnityDependencyResolver(container);
   DependencyResolver.SetResolver(resolver);
}
```

If you use an IoC framework from within the resolver, you need to figure out the best way to provide it with the list of registered types. If you prefer to pass this information via fluent code, you need to fully configure the IoC container object before you create the resolver. If you intend to configure the IoC using the *web.config* file, you can use the default constructor of the resolver—as far as Unity is concerned—which includes a call to load configuration data. Note, however, that you might need to change this code if you target a different IoC framework.

Important The dependency resolver is an internal tool that developers can optionally use to roll their own customized components in lieu of system components. The power of dependency resolvers is limited by the use that ASP.NET MVC makes of them. In other words, if ASP.NET MVC doesn't invoke the resolver before creating, say, the controller cache, there's not much you can do to replace the built-in cache with your own.

Adding Aspects to Controllers

An entirely different form of customization you find in ASP.NET MVC is based on attributes you can attach to controller classes and methods. These attributes are generally known as *action filters*. An action filter is a piece of code that runs around the execution of an action method and can be used to modify and extend the behavior hardcoded in the method itself.

Action Filters

An action filter is fully represented by the following interface:

```
public interface IActionFilter
{
    void OnActionExecuted(ActionExecutedContext filterContext);
    void OnActionExecuting(ActionExecutingContext filterContext);
}
```

As you can see, it offers a hook for you to run code before and after the execution of the action. From within the filter, you have access to the request and controller context and can read and modify parameters.

Embedded and External Filters

Each user-defined controller inherits from the class *Controller*. This class implements *IActionFilter* and exposes *OnActionExecuting* and *OnActionExecuted* methods as protected overridable members. This means that each controller class gives you the chance to decide what to do before, after, or both before and after a given method is invoked. Let's see some code that adds an ad hoc response header any time the method *Index* is invoked:

```
protected DateTime StartTime;
protected override void OnActionExecuting(ActionExecutingContext filterContext)
{
    var action = filterContext.ActionDescriptor.ActionName;
    if (String.Equals(action, "index", StringComparison.CurrentCultureIgnoreCase))
    {
        StartTime = DateTime.Now;
    }

    base.OnActionExecuting(filterContext);
}

protected override void OnActionExecuted(ActionExecutedContext filterContext)
{
    var action = filterContext.ActionDescriptor.ActionName;
    if (String.Equals(action, "index", StringComparison.CurrentCultureIgnoreCase))
    {
        var timeSpan = DateTime.Now - StartTime;
```

```
            filterContext.RequestContext.HttpContext.Response.AddHeader(
                  "MvcGallery3", timeSpan.Milliseconds.ToString());
        }

        base.OnActionExecuted(filterContext);
    }
```

The method counts how many milliseconds it takes to execute and writes that number to a new response header. (See Figure 8-1.)

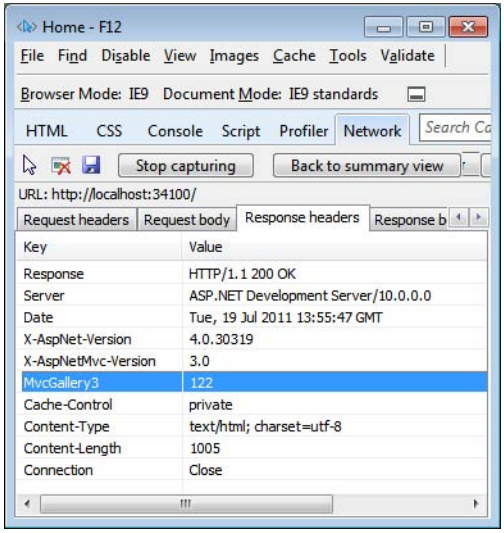

FIGURE 8-1 A custom response header added for the method *Index*.

If you override these methods in a controller class, you end up having a single repository of before/after code for all controller methods. This approach works as long as you want to perform the same operations for each method. If you need to distinguish operations on a per-method basis, using action filters as attributes is probably a better trick to try.

Implemented as an attribute, an action filter provides a declarative means to attach some behavior to a controller's action method. By writing an action filter, you can hook up the execution pipeline of an action method and adapt it to your needs. In this way, you also take out of the controller class any logic that doesn't strictly belong to the controller. In doing so, you make this particular behavior reusable and, more importantly, optional. Action filters are ideal for implementing solutions to cross-cutting concerns that affect the life of your controllers.

Classification of Action Filters

Action filters are classified in different types according to the tasks they actually accomplish. An action filter is characterized by an interface; you have a different interface for each type of filter. Special action filters are exception filters, authorization filters, and result filters. Table 8-3 lists the types of action filters in ASP.NET MVC.

TABLE 8-3 Types of action filters in ASP.NET MVC

Filter Interfaces	Description
IActionFilter	Defines two methods, which execute before and after the controller action
IAuthorizationFilter	Defines a method that executes early in the action pipeline, giving you a chance to verify whether the user is authorized to perform the action
IExceptionFilter	Defines a method that runs whenever an exception is thrown during the execution of the controller action
IResultFilter	Defines two methods, which execute before and after the processing of the action result

The implementation of all the interfaces in Table 8-3 results in a few additional methods on the *Controller* class. Table 8-4 lists them and comments on them all.

TABLE 8-4 Filter methods in the class *Controller*

Method	Description
OnActionExecuting	Invoked just before an action method is executed
OnActionExecuted	Invoked right after the execution of an action method is completed
OnAuthorization	Invoked when authorizing the execution of an action method
OnException	Invoked when an exception occurs in an action method
OnResultExecuting	Invoked just before an action result is executed
OnResultExecuted	Invoked right after the execution of an action result is completed

All these methods are protected and virtual, and they can therefore be overridden in your controller classes to achieve more specialized behavior.

Let's take a closer look at some predefined action filters.

Built-in Action Filters

ASP.NET MVC comes with a few predefined filters, some of which you met already in past chapters: *HandleError*, *Authorize*, and *OutputCache* to name just a few. Table 8-5 lists the built-in filters available in ASP.NET MVC.

TABLE 8-5 Predefined filters in ASP.NET MVC

Filter	Description
AsyncTimeout	Marks an action method as one that will execute asynchronously and terminate in the specified number of milliseconds. A companion attribute also exists for asynchronous methods that do not set a timeout. This companion attribute is *NoAsyncTimeout*.
Authorize	Marks an action method as one that can be accessed only by specified users, roles, or both.
ChildActionOnly	Marks an action method as one that can be executed only as a child action during a render-action operation.
HandleError	Marks an action method as one that requires automatic handling of any exceptions thrown during its execution.

Filter	Description
OutputCache	Marks an action method as one whose output needs to be cached.
RequireHttps	Marks an action method as one that requires a secure request. If the method is invoked over HTTP, the attribute forces a redirect to the same URL but over an HTTPS connection, if that's ever possible.
ValidateAntiForgeryToken	Marks an action method as one that requires validation against the antiforgery token in the page for each POST request.
ValidateInput	Marks an action method as one whose posted input data might (or might not) need validation.

Each controller method can be decorated with multiple filters. The order in which filters are processed is therefore important. All the attributes listed in Table 8-5 derive from the base class *FilterAttribute*, which defines a base property—*Order*. The *Order* property indicates the order in which multiple attributes will be applied. Note that by default the *Order* property is assigned a value of –1, which means that the order is unspecified. However, any filter with an unspecified order is always executed before a filter with a fixed order. Finally, note that if you explicitly set the same order on two or more action filters on a method, an exception will be thrown.

Global Filters

Filters can be applied to individual methods or to the entire controller class. If filters are applied to the controller class, they will have an effect on all action methods exposed by the controller. Global filters, instead, are filters that when registered at application startup are automatically applied to any action of any controller class.

By default, the *HandleError* filter is globally registered in *global.asax*, meaning that it provides some exception-handling capabilities to any action methods. Global filters are plain action filters that are just registered in a different way. Here's how:

```
GlobalFilters.Filters.Add(new HandleError());
```

The *GlobalFilters* collection is checked by the current action invoker before each action is invoked, and all found filters are added to the list of filters enabled to pre-process and post-process the action. I'll return to action invokers and the list of filters for each action later in the chapter.

Gallery of Action Filters

Overall, action filters form an embedded aspect-oriented framework within ASP.NET MVC. ASP.NET MVC provides a bunch of predefined filters, but makes it possible to write your own filters. When it comes to writing an action filter, you typically inherit from *FilterAttribute* and then implement one or more of the interfaces defined in Table 8-3. Most of the time, however, you take a shorter route—deriving from *ActionFilterAttribute*. The *ActionFilterAttribute* class is another, richer, base class for creating your custom action filters. It inherits from *FilterAttribute* and provides a default implementation for all the interfaces listed in Table 8-3. Let's find out what it takes to write a few sample action filters.

Important Action filters are custom components that encapsulate a specific behavior. You write an action filter whenever you want to isolate this behavior and replicate it with ease. Reusability of the behavior is one of the factors for deciding whether to write action filters, but it's not the only one. Action filters also serve the purpose of keeping the controller's code lean and mean. As a general rule, whenever your controller's method code is padded with branches and conditional statements, stop and consider whether some of those branches (or repetitive code) can be moved to an action filter. The readability of the code will be largely improved.

Adding a Response Header

The classic scenario for having a custom action filter is to encapsulate any repetitive behavior you want to apply to many (but not necessarily all) action methods. A canonical example is adding a custom header to a method response. Earlier in the chapter, you saw how to achieve this using the native implementation of *IActionFilter* in the controller class. Let's see how to move that code out of the controller to a distinct class:

```
public class AddHeaderAttribute : ActionFilterAttribute
{
    public String Name { get; set; }
    public String Value { get; set; }

    public override void OnActionExecuted(ActionExecutedContext filterContext)
    {
        if (!String.IsNullOrEmpty(Name) && !String.IsNullOrEmpty(Value))
            filterContext.RequestContext.HttpContext.Response.AddHeader(Name, Value);
        return;
    }
}
```

You now have an easily manageable piece of code. You can attach it to any number of controller actions, to all actions of a controller, and even globally to all controllers. All you do is add an attribute as shown here:

```
[AddHeader(Name="Action", Value="About")]
public ActionResult About()
{ ... }
```

You might argue that this example is not completely equivalent to the previous one, where we calculated the time elapsed between the start and the completion of the action. The *AddHeader* filter,

indeed, just adds a fixed header. You can derive a new class—say, *ReportDuration*—and apply all the logic you need there:

```
public class ReportDurationAttribute : AddHeaderAttribute
{
    protected DateTime StartTime;

    public override void OnActionExecuting(ActionExecutingContext filterContext)
    {
        StartTime = DateTime.Now;
        base.OnActionExecuting(filterContext);
    }

    public override void OnActionExecuted(ActionExecutedContext filterContext)
    {
        var timeSpan = DateTime.Now - StartTime;
        Value = timeSpan.Milliseconds.ToString();
        base.OnActionExecuted(filterContext)
    }
}
```

Let's see a slightly more sophisticated example—compressing the method response, which is a useful feature, especially when large HTML or binary chunks are returned.

Compressing the Response

These days, HTTP compression is a feature that nearly every website can afford because the number of browsers that have trouble with that is approaching zero. (Any browser released in the past 10 years recognizes the most popular compression schemes.)

In ASP.NET Web Forms, compression is commonly achieved through HTTP modules that intercept any request and compress the response. You can also enable compression at the Internet Information Services (IIS) level. Both options work well in ASP.NET MVC, so the decision is up to you. You typically make your decision based on the parameters you need to control, including the MIME type of the resource to compress, level of compression, files to compress, and so forth.

ASP.NET MVC makes it particularly easy to implement a third option—an action-specific filter that sets things up for compression. In this way, you can control a specific URL without the need to write an HTTP module. Let's go through another example of an action filter that adds compression to the response stream for a particular method.

In general, HTTP compression is controlled by two parameters: the *Accept-Encoding* header sent by the browser with each request, and the *Content-Encoding* header sent by the web server with each response. The *Accept-Encoding* header indicates that the browser is able to handle only the specified encodings—typically, *gzip* and *deflate*. The *Content-Encoding* header indicates the compression

format of the response. Note that the *Accept-Encoding* header is just a request header sent by the browser; in no way should the server feel obliged to return compressed content.

When it comes to writing a compression filter, the hardest part is gaining a full understanding of what the browser is requesting. Here's some code that works:

```
public class CompressAttribute : ActionFilterAttribute
{
    public override void OnActionExecuting(ActionExecutingContext filterContext)
    {
        // Analyze the list of acceptable encodings
        var preferredEncoding = GetPreferredEncoding(filterContext.HttpContext.Request);

        // Compress the response accordingly
        var response = filterContext.HttpContext.Response;
        response.AppendHeader("Content-encoding", preferredEncoding.ToString());

        if (preferredEncoding == CompressionScheme.Gzip)
            response.Filter = new GZipStream(response.Filter, CompressionMode.Compress);
        if (preferredEncoding == CompressionScheme.Deflate)
            response.Filter = new DeflateStream(response.Filter, CompressionMode.Compress);

        return;
    }


    private CompressionScheme GetPreferredEncoding(HttpRequestBase request)
    {
       var acceptableEncoding = request.Headers["Accept-Encoding"].ToLower();

       if (acceptableEncoding.Contains("gzip"))
           return CompressionScheme.Gzip;
       if (acceptableEncoding.Contains("deflate"))
           return CompressionScheme.Deflate;

       return CompressionScheme.Identity;
    }

    enum CompressionScheme
    {
        Gzip = 0,
        Deflate = 1,
        Identity = 2
    }
}
```

You apply the *Compress* attribute to the method as follows:

```
[Compress]
public ActionResult Index()
{ ... }
```

Figure 8-2 demonstrates that the *Content-Encoding* response header is set correctly and the response is understood and decompressed within the browser.

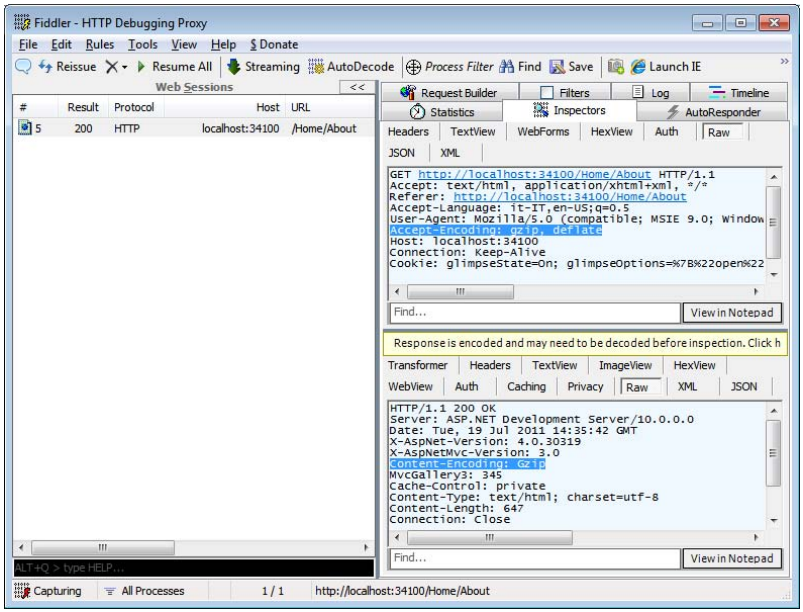

FIGURE 8-2 Fiddler—an HTTP inspection tool—shows the *Content-Encoding* response header.

Almost any browser sets the *Accept-Encoding* header to the string "gzip, deflate", which is not the only possibility. As you can read in RFC 2616 (available at *http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html*), an *Accept* header field supports the *q* parameter as a way to assign a priority to an acceptable value. The following strings are acceptable values for an encoding:

```
gzip, deflate
gzip;q=.7,deflate
gzip;q=.5,deflate;q=.5,identity
```

Even though *gzip* appears in all strings, only in the first one is it the preferred choice. If a value is not specified, the *q* parameter is set to 1; this assigns to *deflate* in the second string and to *identity* in the third string a higher rank than *gzip*. So simply checking whether *gzip* appears in the encoding string still sends back something the browser can accept, but it doesn't take the browser's preference into full account. To write a *Compress* attribute that takes into account the priority (if any) expressed through the *q* parameter, you need to refine the *GetPreferredEncoding* method, as shown here:

```
private CompressionScheme GetPreferredEncoding(HttpRequestBase request)
{
   var acceptableEncoding = request.Headers["Accept-Encoding"].ToLower();
   acceptableEncoding = SortEncodings(acceptableEncoding);

   if (acceptableEncoding.Contains("gzip"))
       return CompressionScheme.Gzip;
   if (acceptableEncoding.Contains("deflate"))
       return CompressionScheme.Deflate;

   return CompressionScheme.Identity;
}
```

The *SortEncodings* method will parse the header string and extract the segment of it that corresponds to the choice with the highest priority.

View Selector

In Chapter 5, I discussed the *CultureAttribute* filter, which is used to force a specific culture for a given action method, thus obtaining the effect of enabling localization for that request. The *CultureAttribute* filter works great if it's installed as a global filter. I won't be discussing the use of an action filter to enable localization any further. My next filter, however, still belongs to the class of filters that can select a different view for a given action. Let's suppose you want to switch to a different view template following the capabilities of the requesting browser.

Creating a multiserving application is not easy and, more importantly, is always a project-specific solution. What I'm going to do is illustrate the role that filters can play in helping you arrange a multiserving solution. To clarify, a multiserving application is an application that can serve the same request in different ways depending on the requesting browser.

Up until a few years ago, a classic example of multiserving was having pages for rich browsers and having pages for less rich desktop browsers. Today, Ajax and JavaScript libraries have significantly mitigated the impact of this problem because JavaScript libraries (for example, jQuery) can now hide most of the differences to the developer. Another emerging dichotomy, however, is between desktop and mobile browsers. The boundaries of the two categories are not as clear as it might seem. In the end, defining boundaries is up to you. Defining boundaries means deciding how to treat smartphones and tablets, as well as deciding what to do with less powerful browsers embedded in mobile devices and cell phones. For the purpose of this chapter, however, I'll assume that you know exactly where your boundaries are. So I'll proceed with an action filter that knows how to switch views based on the user agent string and your rules.

The idea is to write a filter that kicks in just after the action executed and before the action invoker begins processing the results. According to the classification introduced with Table 8-3, this technically is a *result filter*. Let's have a look at the source code:

```
public class BrowserSpecificAttribute : ActionFilterAttribute
{
   public override void OnResultExecuting(ResultExecutingContext filterContext)
   {
       ...
   }
}
```

The filter inherits from *ActionFilterAttribute* and overrides the method *OnResultExecuting*. The method is invoked after the execution of the action method but before the result of the action is processed to generate the response for the browser:

```
public override void OnResultExecuting(ResultExecutingContext filterContext)
{
    var viewResult = filterContext.Result as ViewResult;
    if (viewResult == null)
        return;
```

```
    // Figure out the view name
    var context = filterContext.Controller.ControllerContext;
    var viewName = viewResult.ViewName;
    if (String.IsNullOrEmpty(viewName))
        viewName = context.RouteData.GetRequiredString("action");
    if (String.IsNullOrEmpty(viewName))
        return;

    // Resolve the view selector
    var viewSelector = DependencyResolver.Current.GetService(typeof (IViewSelector))
                       as IViewSelector;
    if (viewSelector == null)
        viewSelector = new DefaultViewSelector();

    // Figure out the browser name
    var isMobileDevice = context.HttpContext.Request.Browser.IsMobileDevice;
    var browserName = (isMobileDevice ?"mobile" :context.HttpContext.Request.Browser.Browser);

    // Get the name of the browser-specific view to use
    var newViewName = viewSelector.GetViewName(viewName, browserName);
    if (String.IsNullOrEmpty(newViewName))
        return;

    // Is there such a view?
    var result = System.Web.Mvc.ViewEngines.Engines.FindView(context, newViewName,
                                                             viewResult.MasterName);
    if (result.View != null)
        viewResult.ViewName = newViewName;
}
```

The algorithm employed is simple. Using the *ControllerContext* object, the filter retrieves the *Request* object from the request context; from there, it gets to know the capabilities of the current browser. The browser name is used as a discriminator to decide about the next view to select.

An object of type *IViewSelector* resolves the name of the view given the browser name. A default implementation of the view selector is shown here:

```
public class DefaultViewSelector : IViewSelector
{
    public String GetViewName(String viewName, String browserName)
    {
        return String.Format("{0}_{1}", viewName, browserName);
    }

    public String GetMasterName(String masterName, String browserName)
    {
        return String.Format("{0}_{1}", masterName, browserName);
    }
}
```

The code assumes that given a view named *Index*, an Internet Explorer–specific version of the view is named *Index_IE*, a version for Firefox is named *Index_Firefox*, and so forth. After the filter has determined the name of the candidate view to show, it also checks with the current view engine to see whether such a view is supported. If so, the *ViewName* property of the *ViewResult* to render is set to

the browser-specific view. If no browser-specific view is found, you need to do nothing else because the generic view invoked by the action method remains in place. (See Figure 8-3.)

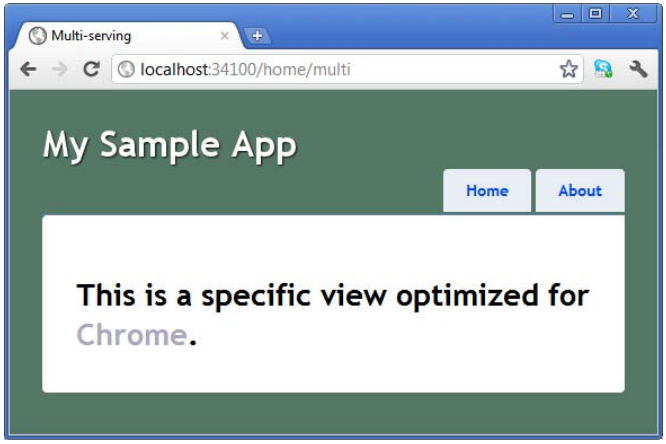

FIGURE 8-3 A view optimized for Chrome.

If the requesting browser turns out to be a mobile device, a view named *xxx_mobile* is selected.

Using the attribute couldn't be easier. All you need to do is decorate the controller method with the attribute, as shown here:

```
[BrowserSpecific]
public virtual ActionResult Index()
{ ... }
```

An action filter like this saves you from adding a bunch of *if* statements to each controller method to return a different *ViewResult* object for each supported browser:

```
public virtual ActionResult Index()
{
    if(GetCurrentBrowser() == "IE")
       return View("Index_IE");
    ...
}
```

You might still regard this code as necessary in some corner cases you hit, but an action filter gives you a great chance to take it from the controller class, thus simplifying the entire design.

Important From this example, you can see quite clearly why and how ASP.NET MVC is different from Web Forms. Switching views is definitely possible in Web Forms also, but it requires a bit of hacking—switching the master page, loading user controls programmatically, and changing templates programmatically. This is an aspect of programming that was not given a high priority when Web Forms was designed because it addresses an issue that people were not really sensitive to. Today is different, and ASP.NET MVC makes it simpler to address issues that are relevant to developers today.

Filling Up the View Model

In Web Forms, a developer coming to grips with a master/detail view has one main concern: getting the ID of the selected item and the details of the child record. The code that populates the drill-down page still contains the list of master records, but no line of code is required to refill the list. The page life cycle and, more importantly, the view state and server control architecture, do the trick.

A huge difference that exists between Web Forms and ASP.NET MVC is the lack of view state and server controls.

The net effect of this is that the controller is now responsible for more than just putting in the view model any data that it specifically obtained out of processing. The controller also needs to retrieve any other data that is required to rebuild the view from scratch—typically, master records, nonstatic menus, breadcrumbs, sitemaps, ads, and the like.

You don't want this code to belong to the controller. Render actions are a possible way to solve the issue. In some cases, an action filter that automatically stuffs global data into the view model also can be a valid solution. Rest assured that you can completely bypass the problem by using a lot of Ajax.

Here's some code for the filter:

```
public class FillUpViewModelAttribute : ActionFilterAttribute
{
    public override void OnActionExecuted(ActionExecutedContext filterContext)
    {
        // Retrieve the view model object
        var model = filterContext.Controller.ViewData.Model as CustomerModelBase;
        if (model == null)
            return;

        // Fill up the view model object by adding globally accessible data
        // not specific to the current action
        model.Countries = ...;
    }
}
```

After the controller action has executed, the filter retrieves the current view-model object and casts it to a base class that you aptly created to store all global information around the view. Any view model you happen to use in your controller method will just inherit from *CustomerModelBase* in the example and add only members specific to the action.

This approach might work very well and produce nifty code, but it requires a full understanding of the trick and a bit of extra care in the design of view-model classes. Here's a sample of the controller's code:

```
[FillUpViewModel]
public virtual ActionResult Details(String customerId)
{
   // Find details of that particular customer
   ...
```

```
    // Prepare the view-model object by adding only details and ignoring any
    // other global data
    var model = new CustomerDetailsViewModel();
    model.Customer = ...;

    // Return the view
    return View(model);
}
```

This implementation is a bit simplistic because the filter is tightly coupled to a base view-model type. I've used this approach a couple of times and created a few similar-looking filter attributes for a few scenarios. In the end, it worked nicely, but I do understand that not everybody will like it. I'd like to end with a couple of extra notes to top off the argument.

First, the main purpose of such filters is not to make any code dissolve. The goal is less ambitious: making this code disappear from controllers and reappear somewhere else where it's more easily manageable. Second, if you resolve to use *ViewData* or *ViewBag* dictionaries at least for common and global view data, your filter won't be tightly coupled to a type; but it still requires some custom logic about what to add to the view model that might be hard to write. However, you can write it only once and reuse it forever.

Special Filters

The action filters considered so far are components aimed at intercepting a few stages of the execution of action methods. What if you want to add some code to help decide whether a given method is fit to serve a given action? For this type of customization, another category of filters are required—*action selectors*.

Action selectors come in two distinct flavors: *action name selectors* and *action method selectors*. Name selectors decide whether the method they decorate can be used to serve a given action name. Method selectors decide whether a method with a matching name can be used to serve a given action. Method selectors typically give their response based on other run-time conditions.

Action Name Selectors

The base class for action name selectors is *ActionNameSelectorAttribute*. The class has a simple structure, as the code here demonstrates:

```
public abstract class ActionNameSelectorAttribute : Attribute
{
    public abstract Boolean IsValidName(ControllerContext controllerContext,
                                        String actionName, MethodInfo methodInfo);
}
```

The purpose of the selector is simple: checking whether the specified action name is a valid action name for the method.

In ASP.NET MVC, there's just one action name selector: the *ActionName* attribute that you can use to alias a controller method. Here's an example:

```
[ActionName("Edit")]
public ActionResult EditViaPost(String customerId)
{
   ...
}
```

The implementation of the *ActionName* attribute is trivial, as the following code demonstrates:

```
public sealed class ActionNameAttribute : ActionNameSelectorAttribute
{
    public ActionNameAttribute(String name)
    {
        if (String.IsNullOrEmpty(name))
            throw new ArgumentException();
        Name = name;
    }

    public override Boolean IsValidName(ControllerContext controllerContext,
                                        String actionName, MethodInfo methodInfo)
    {
        // Check that the action name matches the specified name
        return String.Equals(actionName, Name, StringComparison.OrdinalIgnoreCase);
    }

    public String Name { get; set; }
}
```

The net effect of the attribute is that it logically renames the controller method it's applied to. For example, in the previous example the method is named *EditViaPost*, but it won't be invoked unless the action name that results from the routing process is *Edit*.

Action Method Selectors

Action method selectors are a more powerful and interesting tool for developers. During the preliminary stage in which the system is looking for a controller method that can serve the request, a method selector just indicates whether a given method is valid. Obviously, such a selector determines its response based on certain run-time conditions. Here's the definition of the base class:

```
public abstract class ActionMethodSelectorAttribute : Attribute
{
    public abstract Boolean IsValidForRequest(
         ControllerContext controllerContext, MethodInfo methodInfo);
}
```

In ASP.NET MVC, quite a few predefined method selectors exist. They are *AcceptVerbs*, *NonAction*, plus a bunch of HTTP-specific selectors introduced to simplify coding (*HttpDelete*, *HttpGet*, *HttpPost*, and *HttpPut*). Let's have a look at some of them.

The *NonAction* attribute just prevents the decorated method from processing the current action. Here's how it's implemented:

```
public override Boolean IsValidForRequest(
     ControllerContext controllerContext, MethodInfo methodInfo)
{
    return false;
}
```

The *AcceptVerbs* attribute receives the list of supported HTTP verbs as an argument and checks the current verb against the list. Here are some details:

```
public override Boolean IsValidForRequest(
    ControllerContext controllerContext, MethodInfo methodInfo)
{
    if (controllerContext == null)
        throw new ArgumentNullException("controllerContext");

    // Get the (overridden) HTTP method
    var method = controllerContext.HttpContext.Request.GetHttpMethodOverride();

    // Verbs is an internal member of the AcceptVerbsAttribute class
    return Verbs.Contains<String>(method, StringComparer.OrdinalIgnoreCase);
}
```

Note the use of the *GetHttpMethodOverride* method to retrieve the actual verb intended by the client. The method reads the value in a header field or parameter named *X-HTTP-Method-Override*. (See *http://code.google.com/apis/gdata/docs/2.0/basics.html#UpdatingEntry* for more information about *X-HTTP-Method-Override*.) This is a common protocol for letting browsers place any HTTP verbs even if the physical request is either GET or POST. The method is not defined natively on the *HttpRequest* object, but it was added in ASP.NET MVC as an extension method on *HttpRequestBase*.

The other selectors are simply implemented in terms of *AcceptVerbs*, as shown here for *HttpPost*:

```
public sealed class HttpPostAttribute : ActionMethodSelectorAttribute
{
    private static readonly AcceptVerbsAttribute _innerAttribute;

    public override bool IsValidForRequest(
         ControllerContext controllerContext, MethodInfo methodInfo)
    {
       return _innerAttribute.IsValidForRequest(controllerContext, methodInfo);
    }
}
```

Let's see how to write a custom method selector.

Restricting a Method to Ajax Calls Only

All you need is a class that inherits from *ActionMethodSelectorAttribute* and overrides the *IsValidForRequest* method:

```
public class AjaxOnlyAttribute : ActionMethodSelectorAttribute
{
    public override Boolean IsValidForRequest(
               ControllerContext controllerContext, MethodInfo methodInfo)
    {
        return controllerContext.HttpContext.Request.IsAjaxRequest();
    }
}
```

Any method marked with this attribute is only enabled to serve calls placed via the browser's *XMLHttpRequest* object:

```
[AjaxOnly]
public ActionResult Details(Int32 customerId)
{
    var model = ...;
    return PartialView(model);
}
```

If you try to invoke a URL that, according to routes, should be mapped to an Ajax-only method, well, you'll get a not-found exception.

Restricting a Method to a Given Submit Button

In Web Forms, every page has a single HTML form and nearly every form contains multiple submit buttons. Each button, however, has its own click handler that determines the action to take in case of a click. In ASP.NET MVC, instead, you can have as many HTML forms as you want, with each posting to a different and fixed URL. How can you determine which button was clicked in an HTML form with two or more submit buttons?

At the HTML level, the posting form uploads the content of input fields plus a name/value pair inherent to the clicked button. The name token refers to the *name* attribute of the button; the value token refers to the *value* attribute of the button. Unfortunately, the *value* attribute of the button is the caption. You can't reliably figure out which button was clicked by examining the caption. At a minimum, it's subject to localization, or it can be missing altogether if an image button is used.

The simplest way to solve the problem in ASP.NET MVC is to add a bit of JavaScript to the form so that each button, when clicked, changes the *action* attribute of the form to reflect its name. Here's a brief example of an HTML form with a couple of input fields and two submit buttons—to update or delete some content:

```
<form name="myForm" id="myForm" method="post">
   <input type="text" />
   <input type="text" />
   <hr />
   <input type="submit" value="Update" name="updateAction"
          onclick="setAction('update')" />
```

```
   <input type="submit" value="Delete" name="deleteAction"
          onclick="setAction('delete')" />
</form>
```

The JavaScript function named *setAction* does the following:

```
<script type="text/javascript">
    function setAction(action) {
        document.getElementById("myForm").action = action;
    }
</script>
```

This trick works, but it has the annoyance that you must repeat this code for each form you write in each ASP.NET MVC application. An ad hoc method selector (and the built-in *ActionName* attribute) can do the trick better:

```
public class OnlyIfPostedFromButtonAttribute : ActionMethodSelectorAttribute
{
    public String SubmitButton { get; set; }
    public override Boolean IsValidForRequest(ControllerContext controllerContext,
                                              MethodInfo methodInfo)
    {
        // Check if this request is coming through the specified submit button
        var o = controllerContext.HttpContext.Request[SubmitButton];
        return o != null;
    }
}
```

The selector exposes a public property through which you indicate the name of the button the method is associated with. When evaluated, the selector simply looks for the name of its companion button in the form data collection being posted. By the design of HTML, a match can be found only if the form was posted by clicking that button. Let's see how you use this selector in the controller code:

```
[HttpPost]
[ActionName("Demo")]
[OnlyIfPostedFromButton(SubmitButton = "updateAction")]
public ActionResult Demo_Update()
{
   ViewData["ActionName"] = "Update";
   return View("demo");
}
```

The code reads like this: the invoker can select this method (*Demo_Update*) only if the request is a POST, the action name is *demo*, and the form was posted using the submit button with the name of *updateAction*.

Building a Dynamic Loader Filter

Action filters are definitely a powerful mechanism for developers to use to decide exactly how a given action method executes. From what you've seen so far, however, action filters are also a static mechanism that requires a new compile and deploy step to be modified. Let's explore an approach to loading filters dynamically from an external source.

Interception Points for Filters

Filters are resolved for each action method within the action invoker. There are two main points of interception: the *GetFilters* and *InvokeActionMethodWithFilters* methods. Both methods are marked as protected and virtual. The signatures of both methods are shown here:

```
protected virtual ActionExecutedContext InvokeActionMethodWithFilters(
     ControllerContext controllerContext,
     IList<IActionFilter> filters,
     ActionDescriptor actionDescriptor,
     IDictionary<string, object> parameters);

protected virtual FilterInfo GetFilters(
     ControllerContext controllerContext,
     ActionDescriptor actionDescriptor)
```

The *GetFilters* method is invoked earlier and is expected to return the list of all filters for a given action. After invoking the base method of *GetFilters* in your custom invoker, you have available the full list of filters for each category—that is, a list including exception, result, authorization, and action filters. Note that the *FilterInfo* class—a public class in *System.Web.Mvc*—offers specific collections of filters for each category:

```
public class FilterInfo
{
    // Private members
    ...

    public IList<IActionFilter> ActionFilters { get; }
    public IList<IAuthorizationFilter> AuthorizationFilters { get; }
    public IList<IExceptionFilter> ExceptionFilters { get; }
    public IList<IResultFilter> ResultFilters { get; }
}
```

The *InvokeActionMethodWithFilters* method is invoked during the process related to the performance of the action method. In this case, the method receives only the list of action filters—that is, those filters that are to execute before or after the code for the method.

To build a mechanism that allows developers to change filters that apply to a method on the fly, our interest focuses on the *InvokeActionMethodWithFilters* method.

Adding an Action Filter Using Fluent Code

By overriding the *InvokeActionMethodWithFilters* method in a custom action invoker class, you can use fluent code to configure controllers and controller methods with action filters. The following code shows how to add the *Compress* attribute on the fly to the *Index* method of the *Home* controller:

```
protected override ActionExecutedContext InvokeActionMethodWithFilters(
     ControllerContext controllerContext,
     IList<IActionFilter> filters,
     ActionDescriptor actionDescriptor,
     IDictionary<string, object> parameters)
{
```

```
   // Add the Compress action filter to the Index method of the Home controller
   if (actionDescriptor.ControllerDescriptor.ControllerName == "Home" &&
       actionDescriptor.ActionName == "Index")
   {
        // Configure the filter and add to the list
        var compressFilter = new CompressAttribute();
        filters.Add(compressFilter);
   }

   // Go with the usual behavior and execute the action
   return base.InvokeActionMethodWithFilters(
        controllerContext, filters, actionDescriptor, parameters);
}
```

This code can be refined in a number of aspects. For example, you can support areas and check the controller type rather than the name. In addition, you can read the filters to add from a configuration file and also use an IoC container to resolve them all. More in general, this approach gives you a chance to dynamically configure the filters, and it also lets you keep attributes out of the controller code.

Customizing the Action Invoker

Let's generalize the idea a tiny bit. Suppose you create a custom action invoker and override its *InvokeActionMethodWithFilters* method as shown here:

```
protected override ActionExecutedContext InvokeActionMethodWithFilters(
    ControllerContext controllerContext,
    IList<IActionFilter> filters,
    ActionDescriptor actionDescriptor,
    IDictionary<String, Object> parameters)
{
   // Load (dynamic-loading) filters for this action
   var methodFilters = LoadFiltersForAction(actionDescriptor);
   if (methodFilters.Count == 0)
      return base.InvokeActionMethodWithFilters(controllerContext,
                                                filters, actionDescriptor, parameters);

   // Apply filter(s)
   foreach (var filter in methodFilters)
   {
       var filterInstance = GetFilterInstance(filter);
       if (filterInstance == null)
           continue;

       // Initialize filter (if params are specified)
       InitializeFilter(filter, filterInstance);

       // Add the filter
       filters.Add(filterInstance);
   }
```

```
    // Exit
    return base.InvokeActionMethodWithFilters(controllerContext,
                                              filters, actionDescriptor, parameters);
}
```

The code contains a number of placeholder methods that essentially read about dynamically defined filters as they might be found in the *web.config* file. The provided action descriptor is used to figure out information about the current action. Action details are used to read about defined filters. Finally, filters are instantiated and initialized via reflection and added to the filters collection for the overridden method. You're responsible for the structure of the *web.config* section. A nice-looking section might be the following:

```
<dynamicFilters>
  <action name="Home.Test">
    <filter type="MvcGallery3.Extensions.Filters.CompressAttribute, MvcGallery3.Extensions" />
    <filter type="MvcGallery3.Extensions.Filters.AddHeaderAttribute, MvcGallery3.Extensions">
      <param Name="Name" Value="X-MvcAspectFX" />
      <param Name="Value" Value="3222" />
    </filter>
  </action>
</dynamicFilters>
```

You'll find an implementation of this dynamic-loading filter framework in the companion source code of the book.

Registering the Custom Invoker

Each controller has its own action invoker exposed through a public property. You can replace the action invoker in the constructor of each controller. If you intend to apply the custom invoker to just any controller instance, you can consider moving the code to the controller factory. I usually employ a custom controller base class, defined as shown here, and derive from there any controller that requires dynamic loading of filters:

```
public class AxpectController : Controller
{
    public AxpectController()
    {
        ActionInvoker = new AxpectActionInvoker();
    }
}

public class HomeController : AxpectController
{
    // Attributes for this method come from global filters defined in global.asax
    // and dynamic-loading filters defined in web.config.
    public ActionResult Index()
    {
       return View();
    }
}
```

Using a custom base class for controllers is not a programming technique that all developers like. There might be several reasons for this, including personal preferences, but in general it's because single-inheritance languages (for example, Microsoft C# and Visual Basic .NET) offer just one spot for inheritance and it's always smart to reserve it for later use.

If you don't like using a base controller class to enable the dynamic loading of filters, an alternative can be built on top of a feature introduced with ASP.NET MVC 3—*filter providers*.

Enabling Dynamic Loading via a Filter Provider

The *OnActionExecuting* method is the right place to execute some custom code defined in a previously registered action filter before the action method runs. On the other hand, it's too late at this point to add a new filter to execute some custom code. In addition to using a custom action invoker, filter providers are a system-defined hook for you to dynamically register filters based on run-time conditions.

A filter provider is a class that implements the following interface:

```
public interface IFilterProvider
{
    IEnumerable<Filter> GetFilters(
          ControllerContext controllerContext, ActionDescriptor actionDescriptor);
}
```

The *GetFilters* method is invoked by the action invoker to load filters dynamically for a given action. A custom filter provider is therefore exactly the tool that removes the need to explicitly tweak the action invoker and controller class. You register your own filter provider in *global.asax*, as shown here.

```
protected void Application_Start()
{
    ...
    RegisterFilterProviders(FilterProviders.Providers);
}
public static void RegisterFilterProviders(FilterProviderCollection providers)
{
    providers.Add(new DynamicLoadingFilterProvider());
}
```

That's all the code you need to have in place as far as configuration is concerned. Next, you add information to the configuration file (as done in the preceding code sample) and run the application. Here's the code for the filter provider:

```
public class DynamicLoadingFilterProvider : IFilterProvider
{
    public IEnumerable<Filter> GetFilters(
          ControllerContext controllerContext, ActionDescriptor actionDescriptor)
    {
```

```
        // The method reads from web.config and returns a collection of Filter objects
        return LoadFiltersFromConfiguration(actionDescriptor);
    }
}

public static IList<Filter> LoadFiltersFromConfiguration(ActionDescriptor actionDescriptor)
{
    var methodFilters = LoadFiltersForAction(actionDescriptor);

    var filters = new List<Filter>();
    foreach (var filterName in methodFilters)
    {
        // Instantiate and initialize the filter (if params are specified)
        var filterInstance = GetFilterInstance(filterName);
        if (filterInstance == null)
            continue;
       InitializeFilter(filterName, filterInstance);

        // Add to the list to return
        var filter = new Filter(filterInstance, FilterScope.Action, -1);
        filters.Add(filter);
    }

    return filters;
}
```

A filter provider manages filters through a wrapper class—the *Filter* class. The class wraps up the actual action filter instance plus a value for the order (–1 in the example) and the filter scope. The scope is defined in the *FilterScope* enumeration type. The type is a system type defined as follows:

```
public enum FilterScope
{
    First = 0,        // First to run
    Global = 10,      // Applies to all controllers/actions
    Controller = 20,  // Applies at controller level
    Action = 30,      // Applies at action level
    Last = 100        // Last to run
}
```

With the filter provider in place, the following code, plus the *web.config* content shown earlier, produces the output shown in Figure 8-4 when the method *Test* is invoked:

```
// Dynamically bound to action filters via web.config
// (Adding a custom header and Gzip compression.)
public ActionResult Test()
{
    return View();
}
```

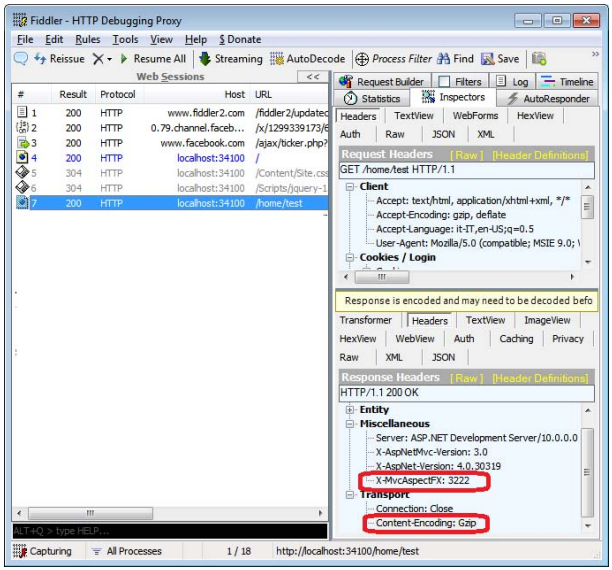

FIGURE 8-4 Response for the request as modified by dynamically added filters.

In this example, we registered the filter provider using code in *global.asax*. Alternatively, you can register a filter provider using your own dependency resolver.

Action Result Types

In most of the examples you've seen thus far in the book, a controller method always returns an *ActionResult* object. This is just the base class that ASP.NET MVC provides to represent the result of an action. Depending on the method used to return the action result—for example, methods *View*, *PartialView*, and the like—the actual type takes a different form. For example, when a method returns HTML markup, the actual action result type is *ViewResult*.

Action result types are also an aspect of ASP.NET MVC that can be customized to a large extent. So let's move to the next stage and consider the tools you have to customize the action result.

Built-in Action Result Types

The response for the browser is generated and written to the output stream when the *ActionResult* object, as returned by the controller action method, is further processed by the action invoker. In other words, the *ActionResult* type—as well as any type directly derived from it—does not represent the real response being sent to the browser. It's merely a wrapper class that contains response-specific data (for example, headers, cookies, status code, and content type, plus any data that can be used to generate the response), and it knows how to process this data to generate the actual response for the browser.

The *ActionResult* object is defined as follows:

```
public abstract class ActionResult
{
    protected ActionResult()
    {
    }

    public abstract void ExecuteResult(ControllerContext context);
}
```

The method *ExecuteResult* is the method that provides the logic to render results to the browser. To understand the mechanics of an action result object, it's useful to look at a couple of action result classes built into ASP.NET MVC.

Returning a Custom Status Code

One of the simplest action result classes is the *HttpStatusCodeResult* class. The class represents an action response that sets a specific HTTP status code and description:

```
public class HttpStatusCodeResult : ActionResult
{
    public HttpStatusCodeResult(Int32 statusCode) : this(statusCode, null)
    {
    }
    public HttpStatusCodeResult(Int32 statusCode, String statusDescription)
    {
        StatusCode = statusCode;
        StatusDescription = statusDescription;
    }
    public override void ExecuteResult(ControllerContext context)
    {
        if (context == null)
            throw new ArgumentNullException("context");

        // Prepare the response for the browser
        context.HttpContext.Response.StatusCode = StatusCode;
        if (StatusDescription != null)
            context.HttpContext.Response.StatusDescription = StatusDescription;
    }
}
```

As you can see, all it does is set the status code and description of the response. Here is how you use this class to return an HTTP 403 code (Forbidden), meaning that the server understood the request but has good reasons to refuse to fulfill it. (See Figure 8-5.)

```
public ActionResult Forbidden()
{
    return new HttpStatusCodeResult(403);
}
```

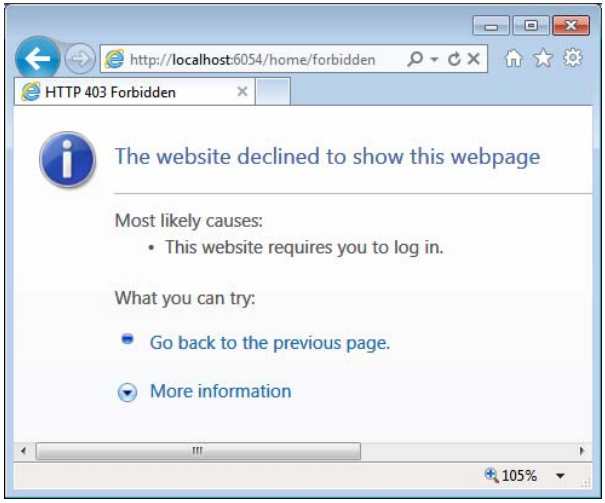

FIGURE 8-5 The web server refuses to fulfill a request.

When you need to return the same status code from several places in your application, it might be a better idea to encapsulate the status code in a custom action result class. It won't really change things that much, but it definitely increases readability. ASP.NET MVC follows this pattern for the 401 code (unauthorized) and offers you the *HttpUnauthorizedResult* class, which is used, among other things, by the *Authorize* action filter. Here's another example for the similarly popular 404 code:

```
public class HttpNotFoundResult : HttpStatusCodeResult
{
    public HttpNotFoundResult() : this(null)
    {}
    public HttpNotFoundResult(String statusDescription) : base(404, statusDescription)
    {}
}
```

Even in their overall simplicity, these classes don't fail to show the core scaffolding of action result classes.

Returning JavaScript Code

A slightly more sophisticated example is the *JavaScriptResult* class. This class supplies a public property—the *Script* property—that contains the script code (as a string) to write to the output stream:

```
public class JavaScriptResult : ActionResult
{
    public String Script { get; set; }

    public override void ExecuteResult(ControllerContext context)
    {
        if (context == null)
            throw new ArgumentNullException("context");
```

```
        // Prepare the response
        var response = context.HttpContext.Response;
        response.ContentType = "application/x-javascript";
        if (Script != null)
            response.Write(Script);
    }
}
```

You use the *JavaScriptResult* class from the action method as shown here:

```
public ActionResult Javascript()
{
   var script = "function helloWorld() {alert('hello, world!');}";
   var result = new JavaScriptResult {Script = script};
   return result;
}
```

For some predefined action results, ASP.NET MVC provides helper methods that hide the creation of the action result object. The previous code can be rewritten in a more compact way, as shown here:

```
public ActionResult Javascript()
{
   var script = "function helloWorld() {alert('hello, world!');}";
   return JavaScript(script);
}
```

When the user invokes an action that returns JavaScript, the classic download panel is displayed as shown in Figure 8-6.

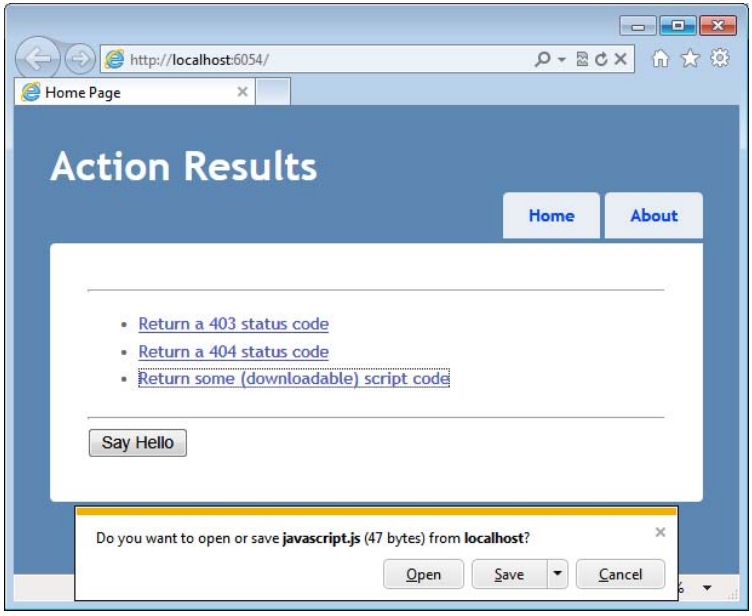

FIGURE 8-6 Script code returned by a JavaScript action result.

It turns out that the right place for action methods returning JavaScript is the *<script>* tag:

```
<script type="text/javascript" src="/home/javascript"></script>
```

After you place a line like the one just shown in the HTML, you can then programmatically invoke any script function being downloaded.

Returning JSON Data

To return JSON data from within an ASP.NET MVC controller class, all you need is an action method that returns a *JsonResult* object. The *JsonResult* class gets any .NET object and attempts to serialize it to JSON using the *JavaScriptSerializer* system class. Here's a possible definition for an action method that serves JSON data:

```
public JsonResult GetCountries(String area)
{
    // Grab some data to serialize and return
    var countries = CountryRepository.GetAll(area);
    return Json(countries);
}
```

Defined on the *Controller* class, the *Json* method internally creates a *JsonResult* object. The purpose of the *JsonResult* object is to serialize the specified .NET object—a list of countries in the example—to the JSON format. The *Json* method has a few overloads through which you can specify the desired content type string (with the default being *application/json*) and request behavior. The request behavior consists of allowing or denying JSON content over an HTTP GET request.

By default, ASP.NET MVC doesn't deliver JSON content through an HTTP GET request. This means that the previous code will fail if it's invoked in the context of a GET—the most obvious way to make a JSON call. If you consider your method to be secure and not at risk of potentially revealing sensitive data, you modify the code as shown here:

```
public JsonResult GetCountries()
{
    // Grab some data to serialize and return
    var countries = CountryRepository.GetAll();
    return Json(countries, JsonRequestBehavior.AllowGet);
}
```

Just like script, JSON data returned from an action method will be downloaded if the method is invoked interactively via the browser. If embedded in a *<script>* tag, or invoked via Ajax, content is made available as raw data. The following script shows an Ajax call that downloads JSON data through an ASP.NET MVC controller. The downloaded data is then used to populate a drop-down list named *listOfCountries*:

```
function downloadCountries(area) {
    $("#listOfCountries").empty();

    $.getJSON("/home/getcountries", { area: "EMEA" },
            function (data) { _displayCountries(data); });
}
```

```
function _displayCountries(data) {
    // Get the reference to the drop-down list
    var listbox = $("#listOfCountries")[0];

    // Fill the list
    for (var i = 0; i < data.length; i++) {
        var country = data[i];
        var option = new Option(country.Name, country.Code);
        listbox.add(option);
    };
}
```

The HTML in the view looks like Figure 8-7.

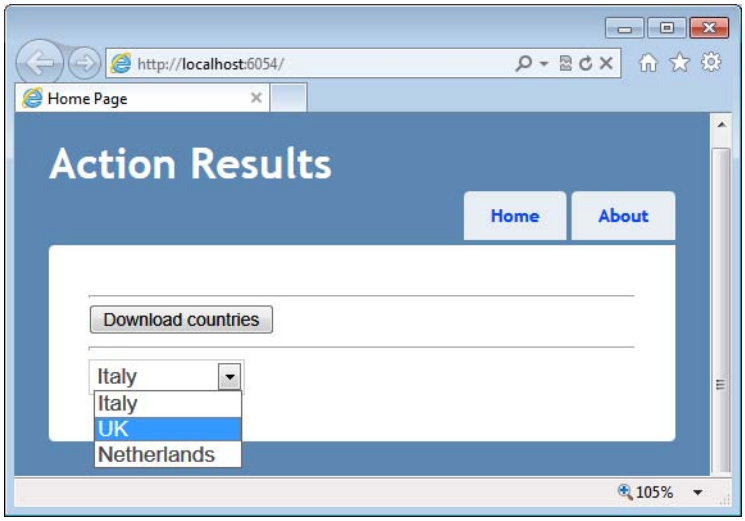

FIGURE 8-7 A drop-down list populated with Ajax-retrieved JSON data.

Note Strictly speaking, a controller action method is not forced to return an *ActionResult* object. However, be aware that whatever type you return will be wrapped up in a *ContentResult* object by the ASP.NET MVC framework. If the method is void, on the other hand, the action result will be an *EmptyResult* object. By using action filters, you can modify the result object, and its parameters, at will. So in the end, you can still have a controller method declared to return nothing but tailor that to return a value with an action filter attached that programmatically returns a given result object.

Custom Result Types

Ultimately, the action result object is a way to encapsulate all the tasks you need to accomplish in particular situations, such as when a requested resource is missing or redirected or when some special response must be served to the browser. Let's examine a couple of interesting scenarios for having custom action result objects.

Returning JSONP Strings

As you might know, browsers don't usually allow pages to place Ajax calls to sites from a different domain. This rule is relatively recent and has been added to reduce to nearly zero the surface attack area for malicious users. The net effect of such restrictions is that you're prevented from making Ajax calls to a website (hosted on a different domain than the requesting page) to download JSON data. More interestingly, this restriction is unilaterally applied by browsers and ignores whether the remote site can serve data or not.

That's much ado for Ajax calls and JSON data, but we still have no control over images and script files, which can be blissfully downloaded from any site that's reachable. It's precisely this feature of browsers that can be exploited to safely download JSON data via Ajax from sites that explicitly allow for it. This is what the JSONP protocol (*JSON with Padding*) is all about.

JSONP is a convention used by some sites to expose their JSON content in a way that makes it easier for callers to consume data via script even from an external domain. The trick consists of returning the JSON content wrapped up in a script function call. In other words, the site would return the following string instead of a string that contains plain JSON data:

```
yourFunction("{...}");   // instead of plain {...} JSON data
```

After the site URL is invoked from within a <*script*> tag, the browser receives what looks like a plain function call with a fixed input string. At this point, whether the input is text or JSON text is irrelevant to the browser.

A website that supports JSONP exposes a public way for callers to indicate the name of the wrapper JavaScript function to be used when returning JSON data. Needless to say, the JavaScript function must be local to the requesting site and is expected to contain the logic that processes JSON data. Let's see how to add JSONP capabilities to an ASP.NET MVC controller. Let's briefly consider an example:

```
public ActionResult GetCountriesJsonp(String area)
{
    var result = new JsonpResult
                     {
                         JsonRequestBehavior = JsonRequestBehavior.AllowGet,
                         Data = GetCountries(area)
                     };
    return result;
}
```

The *JsonpResult* class extends the native *JsonResult* and wraps up the JSON string being returned into a call to the specified JavaScript function:

```
public class JsonpResult : JsonResult
{
    // The callback name here is the parameter name to be added to the URL to specify the
    // name of the JavaScript function padding the JSON response. This name is arbitrary and
    // is part of your site's SDK.
    private const String JsonpCallbackName = "callback";
```

```
    public override void ExecuteResult(ControllerContext context)
    {
        if (context == null)
            throw new ArgumentNullException("context");

        if ((JsonRequestBehavior == JsonRequestBehavior.DenyGet) &&
             String.Equals(context.HttpContext.Request.HttpMethod, "GET"))
            throw new InvalidOperationException();

        var response = context.HttpContext.Response;
        if (!String.IsNullOrEmpty(ContentType))
            response.ContentType = ContentType;
        else
            response.ContentType = "application/json";
        if (ContentEncoding != null)
            response.ContentEncoding = this.ContentEncoding;

        if (Data != null)
        {
            String buffer;
            var request = context.HttpContext.Request;
            var serializer = new JavaScriptSerializer();
            if (request[JsonpCallbackName] != null)
                buffer = String.Format("{0}({1})", request[JsonpCallbackName],
                                                   serializer.Serialize(Data));
            else
                buffer = serializer.Serialize(Data);

            response.Write(buffer);
        }
    }
}
```

The class is nearly the same as *JsonResult* except for a small change in the *ExecuteResult* method. Before serializing to JavaScript, the code checks whether the conventional JSONP parameter has been passed with the request and fixes the JSON string accordingly. Figure 8-8 shows the response body for the following URL:

```
/home/getcountriesjsonp?callback=_cacheForLater&area=asia
```

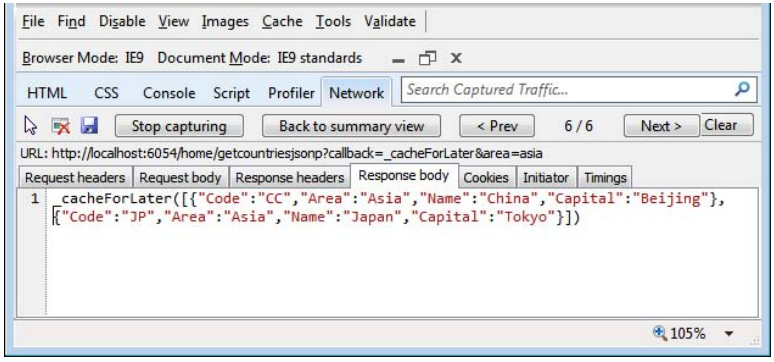

FIGURE 8-8 A JSONP response viewed through the Internet Explorer 9 developer toolbar.

To place JSONP calls, you can follow a couple of routes. You can, for example, just manually arrange a *<script>* tag to make JSON data globally available to the page:

```
<script type="text/javascript"
        src="/home/getcountriesjsonp?callback=_cacheForLater&area=asia"></script>
```

Alternatively, you can place an Ajax call via jQuery or plain JavaScript. In particular, the jQuery library offers specific support through the *getJSON* function. When the URL you pass to *getJSON* contains an *xxx=?* segment, jQuery interprets it as a JSONP call and processes it in a special way. More precisely, jQuery creates a *<script>* tag on the fly and downloads the response through it. The name of the padding function is generated dynamically by jQuery and made to point to the *getJSON* callback code you provide. Here's an example:

```
$.getJSON("/home/getcountriesjsonp?callback=?", { area: "NA" },
            function (data) { _displayCountries(data); });
```

The name *callback* in the URL must match the public JSONP name as defined by the site you're calling. I'm using *callback* in the example because that's the name recognized by the previously presented implementation of *JsonpResult*.

Returning Syndication Feed

If you search the Web for a nontrivial example of an action result, you'll likely find a syndication action result object at the top of the list. Let's briefly go through this popular example.

The class *SyndicationResult* supports both RSS 2.0 and ATOM 1.0, and it offers a handy property for you to choose one or the other programmatically. By default, the class produces an RSS 2.0 feed. To compile this example, you need to reference the *System.ServiceModel.Web* assembly and import the *System.ServiceModel.Syndication* namespace:

```
public class SyndicationResult : ActionResult
{
    public SyndicationFeed Feed { get; set; }
    public FeedType Type { get; set; }

    public SyndicationResult()
    {
        Type = FeedType.Rss;
    }
    public SyndicationResult(
        string title, string description, Uri uri, IEnumerable<SyndicationItem> items)
    {
        Type = FeedType.Rss;
        Feed = new SyndicationFeed(title, description, uri, items);
    }
    public SyndicationResult(SyndicationFeed feed)
    {
        Type = FeedType.Rss;
        Feed = feed;
    }
```

```
    public override void ExecuteResult(ControllerContext context)
    {
        // Set the content type
        context.HttpContext.Response.ContentType = GetContentType();

        // Create the feed, and write it to the output stream
        var feedFormatter = GetFeedFormatter();
        var writer = XmlWriter.Create(context.HttpContext.Response.Output);
        if (writer == null)
            return;
        feedFormatter.WriteTo(writer);
        writer.Close();
    }

    private String GetContentType()
    {
        if(Type == FeedType.Atom)
            return "application/atom+xml";
        return "application/rss+xml";
    }

    private SyndicationFeedFormatter GetFeedFormatter()
    {
        if (Type == FeedType.Atom)
            return new Atom10FeedFormatter(Feed);
        return new Rss20FeedFormatter(Feed);
    }
}

public enum FeedType
{
    Rss = 0,
    Atom = 1
}
```

The class gets a syndication feed and serializes it to the client using either the RSS 2.0 or ATOM 1.0 format. Creating a consumable feed is another story, but it's also a concern that relates to the controller rather than to the infrastructure. Here's how to write a controller method that returns a feed:

```
public SyndicationResult Feed()
{
    var items = new List<SyndicationItem>();
    items.Add(new SyndicationItem(
        "Controller descriptors",
        "This post shows how to customize controller descriptors",
        null));
    items.Add(new SyndicationItem(
        "Action filters",
        "Using a fluent API to define action filters",
        null));
    items.Add(new SyndicationItem(
        "Custom action results",
```

```
            "Create a custom action result for syndication data",
            null));
        var result = new SyndicationResult(
            "Programming ASP.NET MVC",
            "Dino's latest book",
             Request.Url,
             items);

        result.Type = FeedType.Atom;
        return result;
}
```

You create a list of *SyndicationItem* objects and provide a title, some content, and an alternate link (*null* in the code snippet) for each. You typically retrieve these items from some repository you might have in your application. Finally, you pass items to the *SyndicationResult* object along with a title and description for the feed to be created and serialized. Figure 8-9 shows an ATOM feed in Internet Explorer.

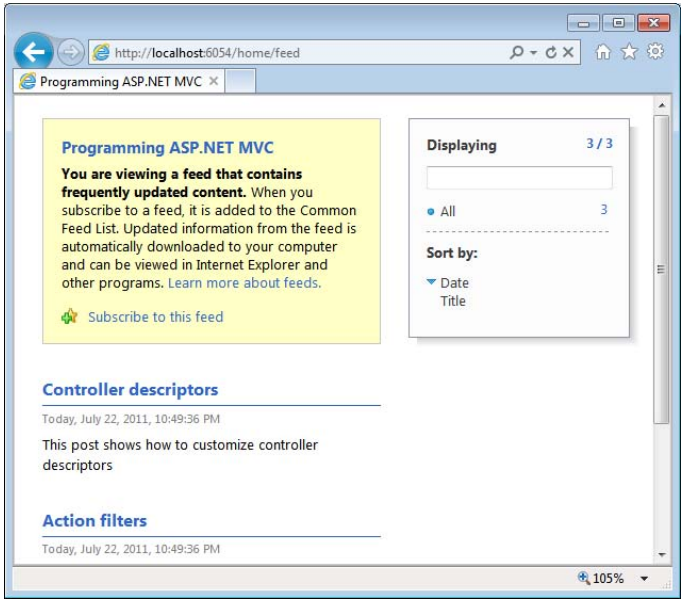

FIGURE 8-9 An ATOM feed displayed in Internet Explorer.

Dealing with Binary Content

A common developer need is returning binary data from a request. Under the umbrella of binary data fall many different types of data, such as the pixels of an image, the content of a PDF file, or even a Microsoft Silverlight package.

You don't really need an ad hoc action result object to deal with binary data. Among the built-in action result objects, you can certainly find one that helps you when working with binary data. If the

content you want to transfer is stored within a disk file, you can use the *FilePathResult* object. If the content is available through a stream, you use *FileStreamResult* and opt for *FileContentResult* if you have it available as a byte array.

All these objects derive from *FileResult* and differ from one another only in how they write out data to the response stream. Let's briefly review how *ExecuteResult* works within *FileResult*:

```
public override void ExecuteResult(ControllerContext context)
{
    if (context == null)
        throw new ArgumentNullException("context");

    var response = context.HttpContext.Response;
    response.ContentType = this.ContentType;
    if (!String.IsNullOrEmpty(this.FileDownloadName))
    {
        var headerValue = ContentDispositionUtil.GetHeaderValue(FileDownloadName);
        context.HttpContext.Response.AddHeader("Content-Disposition", headerValue);
    }

    // Write content to the output stream
    WriteFile(response);
}
```

The class has a public property named *ContentType* through which you communicate the MIME type of the response and which does all of its work via an abstract method—*WriteFile*—that derived classes must override.

The base class *FileResult* also supports the Save As dialog box within the client browser through the *Content-Disposition* header. The property *FileDownloadName* specifies the default name the file will be given in the browser's Save As dialog box. The *Content-Disposition* header has the following format, where *XXX* stands for the value of the *FileDownloadName* property:

```
Content-Disposition: attachment; filename=XXX
```

Note that the file name should be in the US-ASCII character set and no directory path information is allowed. Finally, the MIME type must be unknown to the browser; otherwise, the registered handler will be used to process the content.

The delta between the base class *FileResult* and derived classes is mostly related to the implementation of the *WriteFile* method. In particular, *FileContentResult* writes an array of bytes straight to the output stream, as shown here:

```
// FileContents is a property on FileContentResult that points to the bytes
protected override void WriteFile(HttpResponseBase response)
{
   response.OutputStream.Write(FileContents, 0, FileContents.Length);
}
```

FileStreamResult offers a different implementation. It has a *FileStream* property that provides the data to read, and the code in *WriteFile* reads and writes in a buffered way:

```
protected override void WriteFile(HttpResponseBase response)
{
    Stream outputStream = response.OutputStream;
    using (FileStream)
    {
        byte[] buffer = new byte[0x1000];
        while (true)
        {
            var count = FileStream.Read(buffer, 0, 0x1000);
            if (count == 0)
                return;
            outputStream.Write(buffer, 0, count);
        }
    }
}
```

Finally, *FilePathResult* copies an existing file to the output stream. The implementation of *WriteFile* is quite minimal in this case:

```
// FileName is the name of the file to read and transmit
protected override void WriteFile(HttpResponseBase response)
{
    response.TransmitFile(FileName);
}
```

With these classes available, you can deal with any sort of binary data you need to serve programmatically from a URL.

Returning PDF Files

As a final example, let's see what it takes to return some PDF data. To be honest, when it comes to this the biggest problem to solve is how you get the PDF content. If your PDF content is a static resource such as a server file, all you need to do is use *FilePathResult*. Better yet, you can use the ad hoc *File* method, as shown here:

```
public ActionResult About()
{
    ...
    return File(fileName, "application/pdf");
}
```

To create PDF content on the fly, you can use a bunch of libraries, such as *iTextSharp* (*http://sourceforge.net/projects/itextsharp*). Some commercial products and various open-source projects also let you create PDF content from HTML content. This option is particularly interesting for an ASP.NET MVC application because you can grab a partial view and turn it into downloadable PDF content.

More commonly, you need to create and return PDF documents arranged from a template. There are two main routes you can consider—using Office automation and creating PDFs from Microsoft Word or Microsoft Excel documents, or using Reporting Services. Of the two, using Microsoft Office is perhaps easier to arrange; the other is more reliable and free of hidden costs and subtle issues. By the way, Microsoft itself discourages using Office automation in server applications. (See *http://support.microsoft.com/?id=257757*.) I've used it in a couple of applications without encountering any significant trouble. But my experience could be the exception rather than the rule.

The companion code for this book provides a sample ASP.NET MVC project that serves a PDF file when you invoke a given action method. The code uses Office automation to create new Word documents from a DOTX template. The template includes a few bookmarks that are replaced programmatically with user-defined values. Figure 8-10 shows the sample PDF document created by previous code.

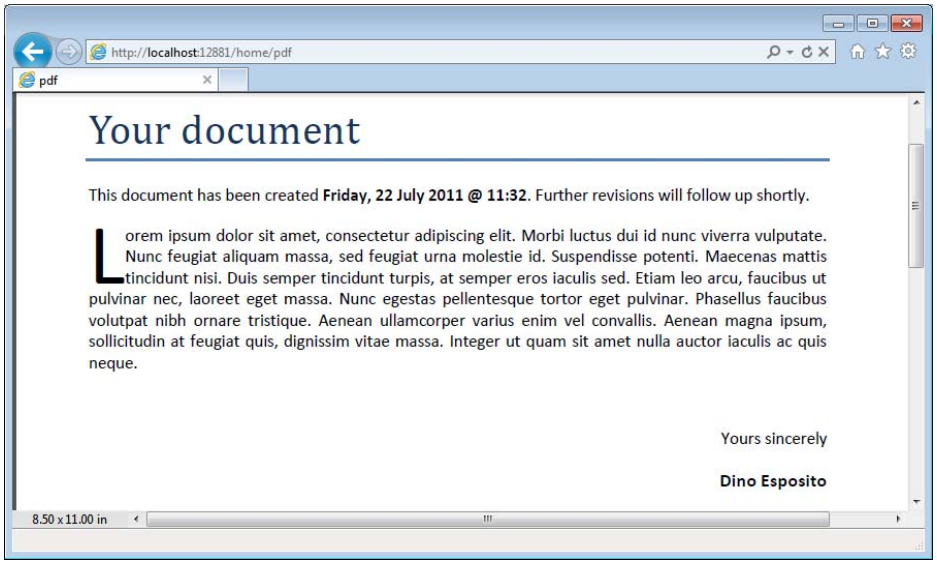

FIGURE 8-10 The sample application creating a PDF document on the fly.

Note The sample PDF application works by creating a local file on the web server disk. As long as you test the application using the Microsoft Visual Studio environment (and the embedded Visual Studio Web server), all goes well. When you start testing it under a real IIS web server, saving the file locally might raise a few problems because of default security permissions. The ASP.NET default account doesn't have write permission on the folder where the file is created. You have to fix this by raising the writing permissions for the ASP.NET account in the folder where you intend to create temporary or persistent PDF files.

Summary

You might or might not like ASP.NET MVC, but you can't honestly deny that it's a highly customizable and extensible framework. In ASP.NET MVC, you can take full control over the execution of each action and intervene before the request has been processed, after it has been processed, or both. Likewise, you can gain control over nearly all aspects of the process that emits the response for the client browser.

Even though nearly all aspects of ASP.NET MVC are customizable, you don't want to rewrite all of them all the time. The aspects of customization discussed in this chapter are those I see as being more frequently customized and, more importantly, those that deliver the greatest benefits if they're properly customized.

CHAPTER 9

Testing and Testability in ASP.NET MVC

> *In preparing for battle I have always found that plans are useless, but planning is indispensable.*
>
> *—Dwight D. Eisenhower*

In the early days of .NET, the average complexity of applications wasn't quite high and the Microsoft Visual Studio debugging tools were quite powerful. Rapid Application Development (RAD) was the paradigm of choice for most; subsequently, few developers really cared about writing test programs.

The success of .NET as a platform resulted in many companies across the full spectrum of the industry needing to acquire new line-of-business applications. In doing so, they dumped an incredible amount of complexity and business rules on the various development teams. Being productive became harder and harder with the sole support of the RAD paradigm. It was ultimately a complete change of priorities: in addition to having to be concerned with time to market, developers had to pay much more attention to maintainability and extensibility. And maintainability brought with it the need to write readable code that could deal with a growing requirement churn.

The ability to test software, and in particular to test software *automatically*, is an aspect of extraordinary importance because automated tests give you a mechanical way to figure out quickly and reliably whether certain features that worked at some point still work after you make some required changes. In addition, tests allow you to calculate metrics and take the pulse of a project as well. In the end, the big change that has come about is that we can no longer spend money on software projects that do not complete successfully. Testing is an important part of the change.

Productivity is still important, but focusing on productivity alone costs too much because it can lead to low-quality code that is difficult and expensive to maintain. And if it's hard to maintain, where's the benefit?

The necessity of testing software in an automated way—we could call it the necessity of applying the RAD paradigm to tests—raised another key point: the need to have software that is *easy* to test. In this chapter, I'll first try to nail down the technical characteristics that a piece of software needs to have to be testable. Next, I'll briefly introduce the basics of unit testing—fixtures, assertions, test doubles, and code coverage—and finish up with some ASP.NET MVC–specific examples of unit tests.

Note For quite a long time, many .NET developers considered terms like "testing" and "debugging" to be nearly synonyms. The action of *debugging* is related to catching and fixing bugs and anomalies in an application. Debugging is carried out by a developer and is mostly an interactive, multistep process. The action of *testing* is related to ensuring that certain parts of your code behave as expected. Testing is carried out by ad hoc programs and is essentially an unattended task that can be automated and integrated in the build process. The two processes are orthogonal, and one doesn't certainly exclude the other. A test can be an effective way to try to reproduce a bug; tests are, on the other hand, a great way to try to prevent bugs.

Testability and Design

In the context of software architecture, a broadly accepted definition for testability is "The ease of performing testing." Testing, of course, is the process of checking software to ensure that it behaves as expected, contains no errors, and satisfies its requirements.

Testing software is conceptually simple: just force the program to work on correct, incorrect, missing, or incomplete data, and see whether the results you get are in line with any set expectations. How would you force the program to work on your input data? How would you measure the correctness of results? In cases of failure, how would you track the specific module that failed?

These questions are the foundation of a paradigm known as Design for Testability (DfT). Any software built in full respect of DfT principles is inherently testable and, as a very pleasant side effect, it is also easy to read, understand and, subsequently, maintain.

Important As I see things, testability is much more important than testing. Testability is an attribute of software that represents a (great) statement about its quality. Testing is a process aimed at verifying whether the code meets expectations. Applying testability (for example, making your code easily testable) is like learning to fish; writing unit tests is like eating a fish.

Design for Testability

Design for Testability was developed as a general concept a few decades ago in a field that was not software. The goal of DfT, in fact, was to improve the process of building low-level circuits within boards and chips.

DfT pioneers employed a number of design techniques and practices with the purpose of enabling effective testing in an automated way. What pioneers called "automated testing equipment" was nothing more than a collection of ad hoc software programs written to test some well-known functions of a board and report results for diagnostic purposes.

DfT was adapted to software engineering and applied to test units of code through tailor-made programs. Ultimately, writing unit tests is like writing software. When you write regular code, you call classes and functions, but you focus more on the overall behavior of the program and the actual implementation of use-cases. When you write unit tests, on the other hand, you need to focus on the input and output of individual methods and classes—a different level of granularity.

DfT defines three attributes that any unit of software must have to be easily testable: control, visibility, and simplicity. You'll be surprised to see that these attributes address exactly the questions I outlined earlier when discussing the foundation of DfT.

The Attribute of *Control*

The attribute of *control* refers to the degree to which the code allows testers to apply fixed input data to the software under test. Any piece of software should be written in a way that makes it clear what parameters are required and what return values are generated. In addition, any piece of software should abstract its dependencies—both parameters and low-level modules—and provide a way for external callers to inject them at will.

The canonical example of the control attribute applied to software is a method that requires a parameter instead of using its knowledge of the system to figure out the parameter's value from another publicly accessible component. In DfT, control is all about defining a virtual contract for a software component that includes preconditions. The easier you can configure preconditions, the easier you can write effective tests.

The Attribute of *Visibility*

The attribute of *visibility* is defined as the ability to observe the current state of the software under test and any output it can produce. After you've implemented the ability to impose ad hoc input values on a method, the next step is being able to verify whether the method behaved as expected. Visibility is all about this aspect—postconditions to be verified past the execution of a method.

The main assumption related to visibility is that if testers have a way to programmatically observe a given behavior, they can easily test it against expected or incorrect values. Postconditions are a way to formalize the expected behavior of a software module.

The Attribute of *Simplicity*

Simplicity is always a positive attribute for any system and in every context. Testing is clearly no exception. Simple and extremely cohesive components are preferable for testing because the less you have to test, the more reliably and quickly you can do that.

In the end, design for testability is a driving factor when writing the source code—preferably right from the beginning of the project—so that attributes such as visibility, control, and simplicity are maximized. When design for testability is successfully applied, writing unit tests is highly effective and easier overall. In addition, your code maximizes maintainability and is easier to read overall.

Note Many would agree that maintainability is the aspect of software to focus upon because of the long-term benefits it can deliver. However, readability is strictly related to and, to a good extent, also part of any maintainability effort. Readability is related to writing code that is easy to read and, subsequently, easy to understand and safer to update and evolve. Readability passes through companywide naming and coding conventions and, better yet, implements ways to effectively convey these conventions to the development teams. In this regard, custom policies in Microsoft Visual Studio Team Foundation Server are a great help.

Loosen Up Your Design

Testable software is inherently better software from a design perspective. When you apply control, visibility, and simplicity to the software development process, you end up with relatively small building blocks that interact only via contracted interfaces. Testable software is software written for someone else to use it programmatically. The typical programmatic user of testable software is the test harness—the program used to run unit tests. In any case, we are talking about software that uses other software. Low coupling, therefore, is the universal principle to apply systematically, and interface-based programming is the best practice to follow for creating software that's easier to test.

Interface-Based Programming

Tight coupling makes software development much simpler and faster. Tight coupling results from an obvious point: if you need to use a component, just get an instance of it. This leads to code like that in the following listing:

```
public class MyComponent
{
     private DefaultLogger _logger;
     public MyComponent()
     {
         _logger = new DefaultLogger();
     }
     public bool PerformTask()
     {
         // Some work here
         bool success = true;
         <...>

         // Log activity
         _logger.Log(...);

         // Return success or failure
         return success;
     }
}
```

The *MyComponent* class is strictly dependent on *DefaultLogger*. You can't reuse the *MyComponent* class in an environment where *DefaultLogger* isn't available. Moreover, you can't reuse *MyComponent* in a runtime environment that prevents *DefaultLogger* from working properly. This is an example of where tight coupling between classes can take you. From a testing perspective, the *MyComponent* class can't be tested without reproducing a runtime environment that is perfectly compatible with the production environment. For example, if *DefaultLogger* logs to Microsoft Internet Information Services (IIS), your test environment must have IIS properly configured and working.

The beauty of unit testing, on the other hand, is that you run your tests quickly and punctually, focusing on the behavior of a small piece of software and ignoring or controlling dependencies. This is clearly impossible when you program your classes to use a concrete implementation of a dependency. Here's how to rewrite the *MyComponent* class so that it depends on an interface, thus resulting in more maintainable and testable code:

```
public class MyComponent
{
     private ILogger _logger;
     public MyComponent()
     {
         _logger = new DefaultLogger();
     }
     public MyComponent(ILogger logger)
     {
         _logger = logger;
     }
     public bool PerformTask()
     {
         // Some work here
         bool success = true;
         ...

         // Log activity
         _logger.Log(...);

         // Return success or failure
         return success;
     }
}
```

The class *MyComponent* is now dependent on the *ILogger* interface that abstracts the dependency on the logging module. The *MyComponent* class now knows how to deal with any objects that implement the *ILogger* interface, including any objects you might inject programmatically.

The solution just shown is acceptable from a testing perspective, even though it is far from perfect. In the preceding implementation, the class is still dependent on *DefaultLogger* and you can't really reuse it without having available the assembly where *DefaultLogger* is defined. At a minimum,

however, it allows you to test the behavior of the class in isolation, bypassing the default logger, as shown here:

```
// Arrange the call
var fakeLogger = new FakeLogger();
var component = new MyComponent(fakeLogger);

// Perform the call and check against expectations
Assert(component.PerformTask());
```

Instructing your classes to work against interfaces rather than implementations is one of five pillars of modern software development. The five principles of development are often summarized with the acronym SOLID, formed from the initials of the five principles:

- Single Responsibility Principle
- Open/Closed Principle
- Liskov's Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

For more information on these principles, check out my book *Microsoft .NET: Architecting Applications for the Enterprise* (coauthored with Andrea Saltarello and published by Microsoft Press, 2008).

In modern software, the idea of writing code against interfaces rather than implementations is widely accepted and applied, but it is also often shadowed by another, more specific, concept—*dependency injection*.

We could say that the whole concept of interface-based programming is hardcoded in the Dependency Inversion Principle (DIP) and that dependency injection is a popular design pattern used to apply the principle. I discussed the Dependency Inversion Principle and related patterns such as Dependency Injection and Service Locator in Chapter 7, "Design Considerations for ASP.NET MVC Controllers."

Relativity of Software Testability

Is design for testability important because it leads to software that is easy to test? Or rather, is it so important because it leads to inherently better designed software? I definitely favor the second option (even though a strong argument can be made for the first option too).

You probably won't go to a customer and use the argument of testability to sell a product of yours. You would likely focus on other characteristics, such as the features, overall quality, user-friendliness, and ease of use. Testability is important mostly to developers, because it is an excellent barometer of the quality of design and coding. From the customer's perspective, there might be no difference between "testable code that works" and "untestable code that works."

On the other hand, a piece of software that is easy to test is necessarily loosely coupled, provides a great separation of concerns between core parts, and is easy to maintain because it can have a battery of tests to promptly catch any regression. In addition, it is inherently simpler in its structure and typically lends itself well to future extensions.

In the end, pursuing testability is a great excuse to have well-designed software. And once you get it, you can also easily test it!

Note I just said that from the customer's perspective there might be no difference between "testable code that works" and "untestable code that works." Well, whether or not there is a difference really depends on the customer. If the customer engaged you on a long-term project, she might be very interested in the maintainability of the code. In situations like this, whether the code is testable or untestable might make a huge difference. But again, it's more a design point than a testability point.

Testability and Coupling

There's a strict relationship between coupling and testability. A class that can't be easily instantiated in a test has some serious coupling problems. This doesn't mean you can't test it automatically, but you'll probably have to configure some database or external connection also in a test environment, which will definitely produce slower tests and higher maintenance costs.

To be effective, a test has to be quick and execute in memory. A project that has good test coverage will likely have a few simple tests per class, which likely amount to a few thousand test calls. It is a manageable problem if each test is quick enough and has no latency caused by synchronization and connections. It is a serious issue otherwise.

If the problem of coupling between components is not properly addressed in the design, you end up testing components that interact with others, producing something that looks more like an integration test than a unit test. Integration tests are still necessary, but they ideally should run on individual units of code (for example, classes) that already have been thoroughly tested in isolation. Integration tests are not run as often as unit tests because of their slow speed and higher setup costs.

In addition, if you end up using integration tests to test a class and a failure occurs, how easily can you identify the problem? Was it in the class you intended to test, or was it the result of a problem in some of the dependencies? Finding the right problem gets significantly more expensive and time consuming. And even when you've found it, fixing it can have an impact on components in the upper layers.

By keeping coupling under control at the design level (for example, by systematically applying dependency injection), you enforce testability. On the other hand, by pursuing testability you keep coupling under control and get a better design for your software.

Testability and Object Orientation

A largely debated point is whether or not it is acceptable to sacrifice (and if it is, to what degree) some design principles (specifically, object-oriented principles) to testability. As mentioned, testability is a driver for better design, but you can have a great design without unit tests and also have great software that is almost impossible to test automatically.

The point here is slightly different. If you pursue good object-oriented design, you probably have a policy that limits the use of virtual members and inheritable classes to situations where it is only strictly necessary. Nonvirtual methods and sealed classes, however, can be hard to test because most test environments need to mock up classes and override members. Furthermore, why should you have an additional constructor that you won't use other than for testing? What should you do?

It is clearly mostly a matter of considering the tradeoffs.

However, consider that commercial tools exist that let you mock and test classes regardless of their design, including sealed classes and nonvirtual methods. An excellent example is Typemock. (See *http://www.typemock.com*.) Recently, Microsoft also released a Visual Studio 2010 add-in that provides similar capabilities. It's called *Moles* and is included in the Visual Studio 2010 Power Tools. (See *http://tinyurl.com/b4zuqj*.)

Before we come to the grips with testing in ASP.NET MVC, let's briefly review the basics of unit testing. If you're already familiar with concepts such as fakes, mocks, and testing in isolation, feel free to jump directly to the "Testing Your ASP.NET MVC Code" section.

Basics of Unit Testing

Unit testing verifies that individual units of code are working properly according to their expected behavior. A *unit* is the smallest part of an application that is testable—typically, a method on a class.

Unit testing consists of writing and running a small program (referred to as a *test harness*) that instantiates test classes and invokes test methods in an automatic way. A test class is a container of test methods, whereas a test method is simply a helper method that invokes the method to test using a specific set of input values. In the end, running a battery of tests is much like compiling. You click a button in the programming environment of choice (for example, Visual Studio), you run the test harness and, at the end of it, you know what went wrong, if anything. (See Figure 9-1.)

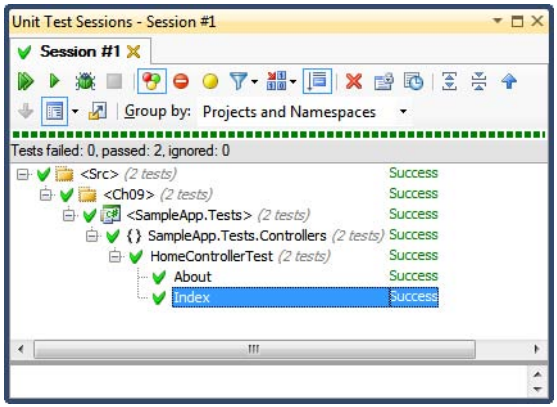

FIGURE 9-1 Running a test project in Visual Studio.

Working with a Test Harness

In its simplest form, a test harness is a manually written program that reads test-case input values and the corresponding expected results from some external files. Then the test harness calls methods using input values and compares the results with the expected values. Obviously, writing such a test harness entirely from scratch is, at a minimum, time consuming and error prone. More importantly, it is restrictive in terms of the testing capabilities you can take advantage of.

The most effective and common way to conduct unit testing entails using an automated test framework. An automated test framework is a developer tool that normally includes a run-time engine and a framework of classes for simplifying the creation of test programs.

Choosing a Test Environment

Popular testing tools are MSTest, NUnit, and its successor, xUnit.net. MSTest is the testing tool incorporated into all versions of Visual Studio 2010. Figure 9-1 shows the user interface of MSTest within Visual Studio.

NUnit (which you can find at *http://www.nunit.org*) is an open-source product that has been around for quite a few years. NUnit is created to be a standalone tool and doesn't *natively* integrate with Visual Studio, which can be either good or bad news—it depends on your perspective of things and your needs and expectations. However, a few tricks exist that enable you to use NUnit from inside Visual Studio. You can configure it as an external executable or, better yet, you can get a plug-in such as ReSharper or TestDriven.NET.

xUnit.net (which you can read more about at *http://xunit.codeplex.com*) is the latest framework for unit testing .NET projects. It comes with an installer that allows you to add it as a new test framework in the default wizard that creates a new ASP.NET MVC application. (See Figure 9-2.) xUnit.net also works well with ReSharper, CodeRush, and TestDriven.NET.

FIGURE 9-2 Picking up your favorite test framework.

At the end of the day, picking a testing framework is really a matter of preference. Regardless of which one you choose, you are hardly *objectively* losing anything really important. The testing matters much more than the framework you use. In my opinion, as of Visual Studio 2010 no significant technical differences exist between MSTest and NUnit. What about xUnit.net, then? As mentioned, xUnit.net is the latest one, and it has been created by the author of NUnit—James Newkirk. For these reasons, xUnit.net is likely to incorporate in its features years of experience and feedback and you might find it closer to being your ideal tool.

Overall, just because test frameworks are nearly the same functionally speaking doesn't mean you can't make an argument for preferring one over another. However, whatever the argument is, it would likely be more of a personal preference than an argument regarding the capabilities of the tools themselves. I'll use MSTest in this book, but I'll briefly point out differences especially with xUnit.net.

Test Fixtures

You start by grouping related tests in a *test fixture*. Test fixtures are just test-specific classes where methods typically represent tests to run. In a test fixture, you might also have code that executes at the start and end of the test run. Here's the skeleton of a test fixture with MSTest:

```
using Microsoft.VisualStudio.TestTools.UnitTesting;
...

[TestClass]
public class CustomerTestCase
{
  private Customer customer;

  [TestInitialize]
  public void SetUp()
  {
    customer = new Customer();
  }
```

```
  [TestCleanup]
  public void TearDown()
  {
    customer = null;
  }

  // Your tests go here
  [TestMethod]
  public void ShouldComplainInCaseOfInvalidId()
  {
    ...
  }
  ...
}
```

Test fixtures are grouped in an ad hoc Visual Studio project. When you create a new ASP.NET MVC project, Visual Studio offers to create a test project for you.

You transform a plain .NET class into a test fixture by simply adding the *TestClass* attribute. You turn a method of this class into a test method by using the *TestMethod* attribute instead. Attributes such as *TestInitialize* and *TestCleanup* have a special meaning and indicate code that runs before and after, respectively, each test in that class. By using attributes such as *ClassInitialize* and *ClassCleanup*, you can define, instead, code that runs only once before and after all tests you have in a class.

Note Some differences exist between xUnit.net and the others regarding test classes (or fixtures). In xUnit.net, you don't need to decorate a test class with a special attribute. The framework will look for all test methods in all public classes available. As far as initialization and cleanup is concerned, xUnit.net requires you to use the class constructor and *Dispose* method for any per-test operations. You implement *IUseFixture* in your test class for one-off, class-level initialization and teardown.

Arrange, Act, Assert

The typical layout of a test method is summarized by the triple "A" acronym: arrange, act, assert. You start arranging the execution context in which you'll test the class by initializing the state of the class and providing any necessary dependencies.

Next, you put in the code that acts on the class under test and performs any required work. Finally, you deal with results and verify that the received output is correct. You do this by verifying assertions based on your expectations.

You write your assertions using the ad hoc assertion API provided by the test harness. At a minimum, the test framework will let you check whether the result equals an expected value:

```
[TestMethod]
public void AssignPropertyId()
{
   // Define the input data for the test
   var customer = new Customer();
   string id = "IDS";
   string expected = id;

   // Execute the action to test.
   customer.ID = id;

   // Test the results
   Assert.AreEqual(expected, customer.ID);
}
```

A test doesn't necessarily have to check whether results are correct. A valid test is also the test aimed at verifying whether under certain conditions a method throws an exception. Here's an example where the setter of the *ID* property in the *Customer* class is expected to raise an *ArgumentException* if the empty string is assigned:

```
[TestMethod]
[ExpectedException(typeof(ArgumentException))]
public void AssignPropertyId()
{
   // Define the input data for the test
   var customer = new Customer();
   var id = String.Empty;

   // Execute the action to test.
   customer.ID = id;

}
```

When writing tests, you can decide to temporarily ignore one because you know it doesn't work but you have no time to fix it at present. You use the *Ignore* attribute for this:

```
[Ignore]
[TestMethod]
public void AssignPropertyId()
{
   ...
}
```

Likewise, you can decide to mark the test as temporarily inconclusive because you are currently unable to determine under which conditions the test will succeed or fail:

```
[TestMethod]
public void AssignPropertyId()
{
   ...
   Assert.Inconclusive("Unable to determine success or failure");
}
```

You might think that ignoring a test, or marking it as inconclusive, is an unnecessary task because you can more simply comment out tests that for some reason just don't work. This is certainly true, but experience teaches us that testing is a delicate task that is always on the borderline between normal priority and low priority. And it is so easy to forget about a test after it has been commented out. It's not by chance that all test frameworks offer a programmatic way to ignore tests while keeping the code active in the project. Test-harness authors know project schedules and budgets are always tight, but they also know maintaining tests in an executable state is important. Whenever you run the tests, you'll be reminded that some tests were ignored or inconclusive. This is preferable overall to just commenting out (and forgetting) tests.

Note xUnit.net replaces the *ExpectedException* attribute with a new method on *Assert*—the *Assert.Throws* method. Ignore is replaced by the *Skip* parameter on the *Fact* attribute—the equivalent for the *TestMethod* attribute. Finally, *Assert.Inconclusive* is not supported in xUnit.net. For more details about differences between xUnit.net and other frameworks, have a look at *http://xunit.codeplex.com/wikipage?title=Comparisons&referringTitle=Home*.

Data-Driven Tests

When you arrange a test for a class method, you might sometimes need to try it with a range of possible values, including correct and incorrect values and values that represent edge conditions. In this case, a data-driven test is a great help.

MSTest supports two possible data sources: a Microsoft Office Excel *.csv* file or any valid ADO.NET data source. The test must be bound to the data source using the *DataSource* attribute, and an instance of the test will be run for each value in the data source. The data source will contain input values and expected values:

```
var id = TestContext.DataRow["ID"].ToString();
var expected = TestContext.DataRow["Result"].ToString();
...
Assert.AreEqual(id, expected);
```

You use the *TestContext* variable to read input values. In MSTest, the *TestContext* variable is automatically defined when you add a new unit test:

```
private TestContext testContextInstance;
public TestContext TestContext
{
   get { return testContextInstance; }
   set { testContextInstance = value; }
}
```

Among other things, the *DataSource* attribute also lets you specify whether test input values are to be processed randomly or sequentially.

Aspects of Testing

Writing unit tests is still a form of programming and has the same need for good practices and techniques as software programming aimed at production code. Writing unit tests, however, has its own set of patterns and characteristics that you might want to keep an eye on.

Very Limited Scope

When introducing DfT at the beginning of the chapter, I made the following point quite clearly: simplicity is a fundamental aspect of software that is key in enabling testability. When applied to unit testing, simplicity is related to giving a very limited scope to the code under test.

A limited scope makes the test self-explanatory and reveals its purpose clearly. This is beneficial for at least two reasons. First, any developers looking into it, including the same author a few weeks later, can quickly and unambiguously understand what the expected behavior of the method under test is.

Second, a test that fails poses the additional problem of you needing to figure out why it failed in order to fix the class under test. The simpler the test method is, the simpler it will be to isolate problems within the class being tested. Furthermore, the more layered the class under test is, the easier it will be to apply changes without the risk of breaking the code somewhere else. Finally, writing tests with a very limited scope is significantly easier for classes that control their dependencies on other components.

Unit testing is like a circle: making it virtuous or vicious is up to you, and it mostly depends on the quality of your design.

Testing in Isolation

An aspect of unit tests that is tightly related to having a limited scope is testing in isolation. When you test a method, you want to focus only on the code within *that* method. All that you want to know is whether *that* code provides the expected results in the tested scenarios. To get this, you need to get rid of all dependencies the method might have.

If the method, say, invokes another class, you assume that the invoked class will *always* return correct results. In this way, you eliminate at the root the risk that the method fails under test because a failure occurred down the call stack. If you test method A and it fails, the reason has to be found *exclusively* in the source code of method A and not in any of its dependencies.

It is highly recommended that the class being tested be *isolated* from its dependencies. Note, though, that this can happen only if the class is designed in a loosely coupled manner. In an object-oriented scenario, class A depends on class B when any of the following conditions are verified:

- Class A derives from class B.
- Class A includes a member of class B.
- One of the methods of class A invokes a method of class B.
- One of the methods of class A receives or returns a parameter of class B.
- Class A depends on a class that, in turn, depends on class B.

How can you neutralize dependencies when testing a method? You use *test doubles*.

Fakes and Mocks

A *test double* is an object that you use in lieu of another. A test double is an object that pretends to be the real one expected in a given scenario. A class written to consume an object that implements the *ILogger* interface can accept a real logger object that logs to IIS or some database table. At the same time, it also can accept an object that pretends to be a logger but just does nothing. There are two main types of test doubles: fakes and mocks.

The simplest option is to use *fake* objects. A fake object is a relatively simple clone of an object that offers the same interface as the original object, but returns hardcoded or programmatically determined values. Here's a sample fake object for the *ILogger* type:

```
public class FakeLogger : ILogger
{
    public void Log(String message)
    {
        return;
    }
}
```

As you can see, the behavior of a fake object is hardcoded; the fake object has no state and no significant behavior. From the fake object's perspective, it makes no difference how many times you invoke a fake method and when in the flow the call occurs. You use fakes when you just want to ignore a dependency.

A more sophisticated option is using *mock* objects. A mock object does all that a fake does, plus something more. In a way, a mock is an object with its own personality that mimics the behavior and interface of another object. What more does a mock provide to testers?

Essentially, a mock allows for verification of the context of the method call. With a mock, you can verify that a method call happens with the right preconditions and in the correct order with respect to other methods in the class.

Writing a fake manually is not usually a big issue—all the logic you need is, for the most part, simple and doesn't need to change frequently. When you use fakes, you're mostly interested in the state that a fake object might represent; you are not interested in interacting with it. Conversely, use a mock when you need to interact with dependent objects during tests. For example, you might want to know whether the mock has been invoked or not, and you might decide within the test what the mock object has to return for a given method.

Writing mocks manually is certainly a possibility, but it is rarely an option you want to consider. For the level of flexibility you expect from a mock, you need an ad hoc mocking framework. Table 9-1 lists a few popular mocking frameworks.

TABLE 9-1 Some popular mocking frameworks

Product	URL
Moq	*http://code.google.com/p/moq*
NMock2	*http://sourceforge.net/projects/nmock2*
Typemock	*http://www.typemock.com*
Rhino Mocks	*http://hibernatingrhinos.com/open-source/rhino-mocks*

Note that no mocking framework is currently incorporated in Visual Studio 2010 and earlier versions.

With the notable exception of Typemock, all frameworks in the table are open-source software. Typemock is a commercial product with unique capabilities that basically don't require you to (re)design your code for testability. Typemock enables you to test code that was previously considered untestable, such as static methods, nonvirtual methods, and sealed classes.

Here's a quick example of how to use a mocking framework such as Moq:

```
[TestMethod]
public void Test_If_Method_Works()
{
    // Arrange
    var logger = new Mock<ILogger>();
    logger.Setup(l => l.Log(It.IsAny<String>()))
    var controller = new HomeController(logger);

    // Act
    ...

    // Assert
    ...
}
```

The class under test—the *HomeController* class—has a dependency on an object that implements the *ILogger* interface:

```
public interface ILogger
{
   void Log(String msg);
}
```

The mock repository supplies a dynamically created object that mocks up the interface for what the test is going to use. The mock object implements the method *Log* in such a way that it does nothing for whatever string argument it receives. You are not really testing the logger here; you are focusing on the controller class and providing a quick and functional mock for the logger component the controller uses internally.

There's no need for you to create an entire fake class; you just specify the code you need a given method to run when invoked. That's the power of mocks compared to fakes.

Number of Assertions per Test

How many assertions should you have per test? Should you force yourself to have just one assertion per test in full homage to the principle of narrowly scoped tests? This is a controversial point.

Many people in the industry seem to think so. Arguments used in support of this opinion are good ones, indeed. One assertion per test leads you to write more focused tests and keep your scope limited. One assertion per test makes it obvious what each test is testing.

The need for multiple assertions often hides the fact that you are testing many features within a single test. And this is clearly a thing to avoid. If you can choose only one rule to follow, one assertion per test is probably the best. If you're testing the state of an object after a given operation, you probably need to check multiple values and need multiple assertions. You can certainly find a way to express this through a bunch of tests, each with a single assertion. In my opinion, though, that would be a lot of refactoring for little gain.

I don't mind having multiple assertions per test as long as the code in the test is testing just one very specific behavior. Most frameworks stop at the first failed assertion, so you theoretically risk that other assertions in the same test will fail on the next run. If you hold to the principle that you test just one behavior and use multiple assertions to verify multiple aspects of the class related to that behavior, all assertions are related—and if the first one fails, the chances are great that by fixing it you won't get more failures in that test.

Testing Inner Members

In some situations, a protected method or property needs to be accessed within a test. In general, a class member doesn't have to be public to deserve some tests. However, testing a nonpublic member poses additional issues.

A common approach to testing a nonpublic member consists of creating a new class that extends the class under test. The derived class then adds a public method that calls the protected method. This class is added only to the test project, without spoiling the class design.

As mentioned earlier, in the .NET Framework an even better approach consists of adding a partial class to the class under test. For this to happen, though, the original class needs to be marked as partial itself. However, this is not a big deal design-wise.

In .NET, you can also easily make internal members of a class visible to another assembly (for example, the test assembly) by using the *InternalsVisibleTo* attribute:

```
[assembly: InternalsVisibleTo("MyTests")]
```

You can add the preceding line to the *assemblyinfo.cs* file of the project that contains the class with *internal* members to make available. Note that you can use the attribute multiple times so that you make visible internal members of classes to multiple external executables.

As I see things, using this attribute is a little more obtrusive than using partial classes. To take advantage of the attribute, in fact, you must mark as *internal* any members you want to recall from tests. Internal members are still not publicly available, but the level of visibility they have is higher

than private or protected. In other words, you should use *internal* and *InternalsToVisible* sparingly and only where a specific need justifies its use.

Finally, MSTest also has a nice programming feature—the *PrivateObject* class—that offers to call nonpublic members via reflection:

```
var resourceId = "WelcomeMessage";
var resourceFile = "MyRes.it.resx";
var expected = "...";
var po = new PrivateObject(controller);
var text = po.Invoke("GetLocalizedText", new object[] { resourceId, resourceFile });
Assert.AreEqual(text, expected);
```

You wrap the object that contains the hidden member in a new instance of the *PrivateObject* class. Next, you call the *Invoke* method to indirectly invoke the method with an array of objects as its parameter list. The method *Invoke* returns an object that represents the return value of the private member.

Code Coverage

The primary purpose of unit and integration tests is to make the development team confident about the quality of the software they're producing. Basically, unit testing tells the team whether they are doing well and are on the right track. But how reliable are the results of unit tests?

Any measure of reliability you want to consider also depends to some extent on the number of unit tests and the percentage of code covered by tests. On the other hand, no realistic correlation has ever been proved to exist between code coverage and the quality of the software.

Typically, unit tests cover only a subset of the code base, but no common agreement has ever been reached on what a "good" percentage of code coverage is. Some say 80 percent is good; some do not even bother quoting a figure. For sure, forms of full code coverage are actually impractical to achieve, if ever possible.

All versions of Visual Studio 2010 have code-coverage tools.

In addition, code coverage is a rather generic term that can refer to quite a few different calculation criteria, such as function, statement, decision, and path coverage. *Function* coverage measures whether each function in the program has been executed in some tests. *Statement* coverage looks more granularly at individual lines of the source code. *Decision* coverage measures the branches (such as an *if* statement) evaluated, whereas *path* coverage checks whether every possible route through a given part of the code has been executed.

Each criterion provides a viewpoint into the code, but what you get back are only numbers to be interpreted. So it might seem that testing all the lines of code (that is, getting 100 percent statement coverage) is a great thing; however, a higher value for path coverage is probably more desirable. Code coverage is certainly useful because it helps you identify which code hasn't been touched by tests. However, code coverage doesn't tell you much about how well tests have exercised the code. Want a nice example?

Imagine a method that processes an integer. You can have 100 percent statement coverage for it, but if you lack a test in which the method gets an out-of-range, invalid value, you might get an exception at run time in spite of all the successful tests you have run.

In the end, code coverage is a number subject to specific measurement. Relevance of tests is what really matters. Blindly increasing the code coverage or, worse yet, requiring that developers reach a given threshold of coverage is no guarantee of anything. It is probably still way better than having no tests, but it says nothing about the relevance and effectiveness of tests. Focusing on expected behavior and expected input is the most reasonable way to approach testing. A well-tested application is an application that has a high coverage of relevant scenarios.

Note The Microsoft Pex add-in for Visual Studio aims to understand the logic of your code and suggests relevant tests you need to have. Internally, Pex employs static analysis techniques to build knowledge about the behavior of your application. You can download Pex from *http://tinyurl.com/b4zuqj*.

Testing Your ASP.NET MVC Code

Testability is often presented as an inalienable feature that makes ASP.NET MVC the first option to consider when it comes to web development for the Microsoft platform. For sure, ASP.NET MVC helps developers write more solid and well-designed software with due separation of concerns between view and behavior. The ASP.NET MVC runtime also offers an API that abstracts away any dependencies your code can have on ASP.NET intrinsic objects. This change marks a huge difference from Web Forms as far as testing is concerned. The bottom line is that ASP.NET MVC is definitely a framework that enables unit testing.

Which Part of Your Code Should You Test?

As mentioned, ASP.NET MVC provides neat separation between the pillars of an application—controllers, views, and models. In addition, it loosens the dependencies of your code on intrinsic components of the ASP.NET runtime such as *Request*, *Response*, and *Session*. The template project also serves a *global.asax* file where all the initialization work is written in a test-oriented way—little details that help.

ASP.NET MVC does its best to enable and support testing. ASP.NET MVC, however, doesn't write your tests and knows nothing about the real structure of your application and the layers it is made of. You should aim to write tests that are relevant; you should not simply aim for getting a high score in code coverage.

How Do I Find Relevant Code to Test?

The location of the relevant code to test mostly depends on the layers you have in the code. In light of what we discussed in Chapter 7, I'd say, "Do not put it in the controller." Too many examples emphasizing the support for unit testing that ASP.NET MVC offers are limited to testing the controller. Consider the following code:

```
[TestClass]
public class HomeControllerTest
{
    [TestMethod]
    public void Index()
    {
        var controller = new HomeController();
        var result = controller.Index() as ViewResult;
        Assert.AreEqual("Welcome to ASP.NET MVC!", result.ViewBag.Message);
    }
}
```

The test creates a new instance of the controller class and invokes the method *Index*. The method returns a *ViewResult* object. The assertion then checks whether the *Message* property in the *ViewResult* instance equals a given string. Let's review the controller's code:

```
public ActionResult Index()
{
    ViewBag.Message = "Welcome to ASP.NET MVC!";
    return View();
}
```

The relevant part of this code is the assignment. To keep the controller lean and mean, you should consider moving this code to a worker service as discussed in Chapter 7. Here's how to rewrite the method to isolate the core logic:

```
public ActionResult Index()
{
    _service.GetIndexViewModel(ViewBag);
    return View();
}
```

At this point, you no longer need to test the controller. You might want to test the service class. There might be situations in which the body of the controller method is a bit fleshier; however, it will be mostly glue code made of conditional statements and trivial assignments—nothing you really need to test.

The same reasoning can be applied to the worker service, leading you to write tests for just highly specialized components with extremely clear and limited assertions.

Note When writing a unit test, you should know a lot of details about the internals of the unit you're testing. Unit testing is, in fact, a form of white-box testing, as opposed to black-box testing in which the tester needs no knowledge of the internals and limits testing to entering given input values and expecting given output values.

The Domain Layer

In Chapter 7, I defined the context of the *domain layer*, which refers to the invariant objects of your application's business context—the data and behavior. If you're architecting an e-commerce application, your domain is made of entities like invoice, customer, order, shipper, and offer. Each of these entities has properties and methods. For example, an invoice will have properties such as *Number*, *Date*, *Payment*, and *Items* and methods such as *GetEstimatedDayOfPayment*, *GetTotal*, and *CalculateTaxes*. For some of these entities (for example, aggregate roots), you might also need special services that perform ad hoc operations in a cross-entity manner. For example, you might want to have a method that gets a customer and figures out if she has placed enough orders to qualify as a gold-level customer.

This is a portion of the code you absolutely want to test thoroughly—namely, be sure it's extensively covered by relevant unit tests. Because this is the core of your application, you want to make sure that all corner cases are properly hit and that inconsistent values/states are properly detected. Finally, you want to be sure that a proper battery of tests can alert you to any regression being introduced in later stages of development. If you can cover with tests just one segment of the application, I recommend that segment be the domain layer—if you have one.

The Orchestration Layer

Depending on how many layers and tiers you really implement, the orchestration layer (which was discussed in Chapter 7) either can be fully identified with the worker services of an ASP.NET MVC controller or can form a layer of its own.

The need to test this layer thoroughly depends on the amount of logic you have in it. In a CRUD system, this layer is mostly thin enough to just test samples of it. However, if the presentation layer offers a significantly different representation of the data than the storage maintains, the orchestration layer is responsible for arranging data in a particular format for the view. In this case, the layer becomes a more critical part of your application and deserves more attention.

If you reduce the controller to be a mere passthrough layer—one that gets view models from the orchestration layer and passes it to the ASP.NET MVC infrastructure—you have no need to test it. Or, better yet, you have other portions of the application that you might to focus on first.

The Data Access Layer

What service is providing your data access layer? Is it simply running SQL statements for you? If that is the case, after you ascertain the code works at development time (for example, your SQL statements are correct), you're all set.

If the data access layer sums up additional capabilities and incorporates some logic that adapts data into different data structures than storage, you might want to consider some tests. But in this case why not separate the CRUD wheat from the adapter chaff and test the adapters only?

Important Unit testing is not really a matter of numbers; it is a matter of quality of numbers. Not only do you need tests, but you need tests that cover relevant aspects of your code. You won't sell an application because of unit tests; you sell an application if the application passes the acceptance tests. And acceptance tests indicate what behavior is relevant for the end user. Finding what behavior is relevant for the units of code your application is made of is exactly the effort one expects from a great developer.

Unit Testing ASP.NET MVC Code

Beyond the theory of unit testing—some would even call it the *art* of unit testing—there are some concrete and pragmatic aspects to deal with. Specifically, there a few practices and techniques you want to understand in order to write unit tests for ASP.NET MVC applications.

Writing a unit test is equivalent to calling a method simulating the particular context you're interested in. A unit test is just software—a method in a class—and, as such, it allows you to use most of the tricks and techniques you normally use in a regular code class. In this chapter, I'm just discussing what's relevant to know and not every aspect of writing a unit test for ASP.NET MVC.

Note If you're interested in the art of unit testing, make sure you get a copy of *The Art of Unit Testing* by Roy Osherove (Manning Publications, 2009). See *http://www.manning.com/osherove*.

Testing If the Returned View Is Correct

There might be situations in which the controller decides on the fly about the view to render. This happens when the view to render is based on some conditions known only at run time. An example is a controller method that has to switch view templates based on the locale, user account, day of the week, or anything else your users might ask you.

In a test, you can catch the view being rendered using the *ViewName* property of the *ActionResult* object:

```
[TestMethod]
public void Should_Render_Italian_View()
{
   // Simulate ad hoc runtime conditions here
   ...

   // Parameters
   var productId = 42;
   var expectedViewName = "index_it";

   // Go
   var controller = new ProductController();
   var result = controller.Find(productId) as ViewResult;
   if (result == null)
       Assert.Fail("Invalid result");
   Assert.AreEqual(result.ViewName, expectedViewName);
}
```

The assumption is that the *ProductController* class returns localized views for the selected product. In this case, a good example of the run-time conditions to simulate for the sake of the test is setting the current locale to *it*.

By checking the public properties of the specific *ActionResult* object returned by the controller method, you can also perform ad hoc checks when a particular response is generated, such as JSON, JavaScript, binaries, files, and so forth.

Testing Localization

Sometimes, you'll find it useful to have some tests that quickly check whether certain parts of the user interface are going to receive proper localized resources when a given language is selected. Here's how to proceed with a unit test:

```
[TestMethod]
public void Test_If_Localizated_Strings_Are_Used()
{
    // Simulate ad hoc runtime conditions here
    const String culture = "it-IT";
    var cultureInfo = CultureInfo.CreateSpecificCulture(culture);
    Thread.CurrentThread.CurrentCulture = cultureInfo;
    Thread.CurrentThread.CurrentUICulture = cultureInfo;

    // Ensure resources are being returned in the correct language
    var showMeMoreDetails = MyText.Product.ShowMeDetails;

    // Assert
    Assert.AreEqual(showMeMoreDetails, "Maggiori informazioni");
}
```

In the unit test, you first set the culture on the current thread, and then you attempt to retrieve the value for the resource and assert against expected values.

You can use this technique to test nearly everything that's related to localization, including localized views and resources. Here's a unit test written for the *UrlHelper* extension method I discussed in Chapter 5, "Aspects of ASP.NET MVC Applications":

```
[TestMethod]
public void Test_If_Url_Extensions_Work()
{
    // Data
    var url = "sample.css";
    var expectedUrl = "sample.it.css";

    // Set culture to IT
    const String culture = "it-IT";
    var cultureInfo = CultureInfo.CreateSpecificCulture(culture);
    Thread.CurrentThread.CurrentCulture = cultureInfo;
    Thread.CurrentThread.CurrentUICulture = cultureInfo;

    // Act & Assert
    var localizedUrl = UrlExtensions.GetLocalizedUrl(url);
    Assert.AreEqual(localizedUrl, expectedUrl);
}
```

This quick demo hides a very interesting story. In Chapter 5, I showed code that defined the *GetLocalizedUrl* extension method as shown here:

```
public static String GetLocalizedUrl(UrlHelper helper, String resourceUrl)
```

To test this method, you need to provide an instance of the *UrlHelper* class. Unfortunately, the constructor of the *UrlHelper* class is coupled with the ASP.NET MVC infrastructure:

```
public UrlHelper(RequestContext context)
```

How can you get a valid *RequestContext* in a testing environment? You need to mock up the HTTP context. It's definitely a doable thing, as you'll see in a moment, but it requires too much work in this scenario. A simple refactoring will help you focus on what's really relevant to test.

What you want ultimately is to check the ability of the code to return *sample.it.css* instead of *sample.css* when the culture is Italian. You don't need to test whether the resource really exists on the web server. So you don't strictly need the request context. Let's rewrite the *GetLocalizedUrl* method as shown here:

```
public static String GetLocalizedUrl(UrlHelper helper, String resourceUrl)
{
    var url = GetLocalizedUrl(resourceUrl);
    return VirtualFileExists(helper, url) ? url : resourceUrl;
}
```

```
public static String GetLocalizedUrl(String resourceUrl)
{
    var cultureExt = String.Format("{0}{1}",
        Thread.CurrentThread.CurrentUICulture.TwoLetterISOLanguageName,
        Path.GetExtension(resourceUrl));
    return Path.ChangeExtension(resourceUrl, cultureExt);
}
```

The effect is the same, but the test is quicker and more focused.

Testing Redirections

A controller action might also redirect to another URL or route. Testing a redirection, however, is no harder than testing a context-specific view. A controller method that redirects will return a *RedirectResult* object if it redirects to a specific URL; it will, instead, return a *RedirectToRouteResult* object if it redirects to a named route.

The *RedirectResult* class has a familiar *Url* property you can check to verify whether the action completed successfully. The *RedirectToRouteResult* class has properties such as *RouteName* and *RouteValues* that you can check to ensure the redirection worked correctly.

Testing Routes

Especially if you make extensive use of custom routes, you might want to test them carefully. In particular, you're interested in checking whether a given URL is matched to the right route and if route data is extracted properly.

To test routes, you must reproduce the *global.asax* environment and begin by invoking the *RegisterRoutes* method. The *RegisterRoutes* method populates the collection with available routes:

```
[TestMethod]
public void Test_If_Product_Routes_Work()
{
    // Arrange
    var routes = new RouteCollection();
    MvcApplication.RegisterRoutes(routes);
    RouteData routeData = null;
    // Act & Assert whether the right route was found
    var expectedRoute = "{controller}/{action}/{id}";
    routeData = GetRouteDataForUrl("~/product/id/123", routes);
    Assert.AreEqual(((Route) routeData.Route).Url, expectedRoute);
}
```

The *GetRouteDataForUrl* method in the test is a local helper defined as follows:

```
private static RouteData GetRouteDataForUrl(String url, RouteCollection routes)
{
    var httpContextMock = new Mock<HttpContextBase>();
    httpContextMock.Setup(c => c.Request.AppRelativeCurrentExecutionFilePath).Returns(url);
    var routeData = routes.GetRouteData(httpContextMock);
    Assert.IsNotNull(routeData, "Should have found the route");
    return routeData;
}
```

The method is expected to invoke *GetRouteData* to get information about the requested route. Unfortunately, *GetRouteData* needs a reference to *HttpContextBase*, where it places all inquiries about the request. In particular, *GetRouteData* needs to invoke *AppRelativeCurrentExecutionFilePath* to know about the virtual path to process. By mocking *HttpContextBase* to provide an ad hoc URL, you completely decouple the route from the run-time environment and can proceed with assertions.

The sample code shown earlier uses the Moq framework to create test doubles. Let's find out more about mocking and how to use mocks to neutralize or replace dependencies.

Dealing with Dependencies

As far as testing is concerned, you could say that there are two main types of dependencies: those you want to ignore, and those you want to interact with, but in a controlled way. In both cases, you need to provide a test-double object—namely, an object that behaves like the expected one while providing an expected behavior. If the classes under test support dependency injection, providing a test double is a piece of cake.

Regardless of the names being used to indicate test doubles (fakes, mocks, stubs), the raw truth is that you need an object that implements a given contract. So how do you write this test-double object?

About Mock and Fake Objects

A test double is a class you write and add to the test project. This class implements a given interface or inherits from a given base class. After you have the instance, you inject it inside the object under test using the public interface of the object being tested. (Clearly, I'm assuming the object under test was designed with testability in mind.)

You might need a different test-double class for each test you write. These classes look nearly the same and differ perhaps in terms of the value they return. Should you really write and maintain hundreds of similar looking classes? You shouldn't, and that's why mocking frameworks exist. A mocking framework provides infrastructure for you to quickly create classes that implement a contract. Additionally, a mocking framework provides some facilities to let you configure the interaction with the methods of this dynamically created class. In particular, you can instruct the mock to get you an instance of a class that exposes the *ISomething* contract and returns 1 when method *ExecuteTask* is invoked with a given argument.

You probably need to write your own class when in the implementation of one or more methods you need to maintain some state or just execute some custom logic. In the context of this book, I call *fakes* the classes you specifically write in the test project to neutralize dependencies. I call *mocks* the classes that you create—for the same purpose—using a mocking framework.

Do you still think that a relevant difference exists between fakes and mocks? If any exists, it's purely about the label you want to attach to each.

Testing Code that Performs Data Access

The canonical example of dependency injection in testing is when you have a worker service class (in simple scenarios, it can even be the controller class) that needs to perform data access operations. In Chapter 7, I defined a worker service class name *HomeServices* that gets a list of dates from a repository. The service class is required to do some extra work on the list of dates before packing them into a view model for display. In particular, the worker service calculates the time span between the current day and the specified date. The repository will likely run a query against some database to return dates. Here's how to inject a fake dependency in the service class so that you can test the service class without dealing with queries and connection strings:

```
[TestClass]
public class DateRepositoryTests
{
    [TestMethod]
    public void Test_If_Dates_Are_Processed()
    {
        var inputDate = new DateTime(2012, 2, 8);
        var fakeRepository = new FakeDateRepository();
        var service = new HomeServices(fakeRepository);
        var model = service.GetHomeViewModel();

        var expectedResult = (Int32) (DateTime.Now - inputDate).TotalDays;
        Assert.AreEqual(model.FeaturedDates[0].DaysToGo, expectedResult);
    }
}
```

The *FakeDateRepository* class will look like this:

```
public class FakeDateRepository : IDateRepository
{
   public override IList<MementoDate> GetFeaturedDates()
   {
      return List<MementoDate>
                    {
                       new MementoDate {Date = new DateTime(...)}
                    });
   }
}
```

You need a new *FakeDateRepository* class for each test you plan to write. For example, suppose you want to test the behavior of the service both when the date falls before and after the current date. You need two tests and two slightly different versions of *FakeDateRepository*—the only difference being in the returned date. Here's where a mocking framework can help:

```
[TestClass]
public class DateRepositoryTests
{
    [TestMethod]
    public void Test_If_Dates_Are_Processed()
    {
        var inputDate = new DateTime(2012, 2, 8);
```

```
        var fakeRepository = new Mock<IDateRepository>();
        fakeRepository.Setup(d => d.GetFeaturedDates()).Returns(new List<MementoDate>
                                                        {
                                                            new MementoDate {Date = inputDate}
                                                        });

        var service = new HomeServices(fakeRepository.Object);
        var model = service.GetHomeViewModel();

        var expectedResult = (Int32) (DateTime.Now - inputDate).TotalDays;
        Assert.AreEqual(model.FeaturedDates[0].DaysToGo, expectedResult);
    }
}
```

The fake repository is created using Moq and is injected in the *HomeServices* class via the constructor. The *Mock<IDateRepository>* object is a dynamically created class (Moq uses Castle DynamicProxy internally to dynamically generate code) that implements the *IDateRepository* interface and returns the input date whenever the *GetFeaturedDates* method is invoked. To write a test against another input, you simply duplicate the test method without explicitly dealing with source classes.

Testing Asynchronous Methods

A great example that shows how refactoring the code can make testing easier comes from the asynchronous methods you met in Chapter 1, "ASP.NET MVC Controls." As you might recall, asynchronous methods on asynchronous controllers are executed in two distinct steps. The first step triggers the long-running operation and yields to an operating system thread outside of the ASP.NET thread pool for any subsequent wait for results. The second step uses computed results to prepare the view.

An asynchronous method is made of two actual methods, as shown in the following example:

```
public void PerformTaskAsync(SomeData data)
{
    ...
}
public ActionResult PerformTaskCompleted(SomeResponse data)
{
    ...
}
```

How about testing? Testing the *Completed* branch is a non-issue: you test it as you would test any other synchronous method. Most of the time, however, you have no need to test the *Completed* method because it's too trivial. It's more interesting to unit-test the *Async* branch:

```
[TestMethod]
public void Should_Run_Async()
{
    var controller = new MyAsyncController();
    var waitHandle = new AutoResetEvent(false);
```

```
    // Create and attach event handler for the "Finished" event
    var eventHandler = delegate(object sender, EventArgs e)
                           {
                              // Signal that the finished event was raised
                              waitHandle.Set();
                           };
    controller.AsyncManager.Finished += eventHandler;
    String expected = ...;

    var data = new SomeData() {Id = 1};
    controller.PerformTaskAsync(data);
    if (!waitHandle.WaitOne(5000, false))
    {
        // Wait until the event handler is invoked or times out
        Assert.Fail("Test timed out.");
    }

    // data is the entry name used by PerformTaskAsync to forward information
    var response = controller.AsyncManager.Parameters["data"] as SomeResponse;
    Assert.IsNotNull(response);
    Assert.AreEqual(response.Data, expected);
}
```

The test consists of invoking the *Async* method using a synchronization tool to prevent the method from terminating. The *waitHandle* synchronization object waits, at most, for five seconds before it fails. If the method completes before the timeout, an event handler associated with the *Finished* event on the *AsyncManager* object fires so that you can signal the lock. The programming paradigm of asynchronous controllers requires that an *Async* method use the *Parameters* dictionary on the *AsyncManager* object to pass information to the *completed* method. The same dictionary can be used in testing to assert expectations.

Any thoughts?

First off, tests should be quick and focused on a specific and small segment of code. Furthermore, tests should have no dependencies. Asynchronous methods, on the other hand, find their natural fit in scenarios when your code is I/O-bound to some external source. Clearly, the two aspects are in conflict.

The idea is to rewrite the asynchronous code so that it doesn't directly talk to the I/O source. You might want it to talk to an intermediate object—a mediator—that encapsulates the details of the asynchronous communication. From the perspective of the unit test, the mediator is mockable and gets mocked to make the test synchronous and not depend on the network latency of a typical asynchronous controller operation. Here's how to rewrite the asynchronous controller to employ a mediator. Sometimes this is referred to as the *Humble Object pattern*.

Refactoring to the Humble Object Pattern

The following code shows how to implement the Humble Object pattern in an asynchronous controller that gets a news feed from a remote site. Here's the interface of the mediator:

```
public interface INewsHumbleObject
{
    void GetFeedItems(String url, AsyncManager asyncManager);
    void Prepare(AsyncManager asyncManager);
}
```

And here's the default implementation of the mediator:

```
public class NewsHumbleObject : INewsHumbleObject
{
    public void Prepare(AsyncManager asyncManager)
    {
        asyncManager.OutstandingOperations.Increment();
    }

    public void GetFeedItems(String url, AsyncManager asyncManager)
    {
        var uri = new Uri(url);
        var client = new WebClient();
        client.DownloadStringCompleted += (sender, e) =>
                                                {
                                                    var feed = ProcessToFeed(e.Result);
                                                    asyncManager.Parameters["feedItems"] = feed;
                                                    asyncManager.OutstandingOperations.
Decrement();
                                                };
        client.DownloadStringAsync(uri);
    }

    private static IList<SyndicationItem> ProcessToFeed(String xml)
    {
        var textReader = new StringReader(xml);
        var xmlReader = XmlReader.Create(textReader);
        var feed = SyndicationFeed.Load(xmlReader);
        return feed == null ? null : new List<SyndicationItem>(feed.Items);
    }
}
```

The *Prepare* method simply increases the reference count of pending operations. *GetFeedItems* manages the retrieval of the data, hiding the details of how it happens. More in general, in the definition of a mediator for wrapping asynchronous operations, you should consider one initialize method and one execute method.

The controller looks like the code shown here:

```
public class NewsController : AsyncController
{
    private readonly INewsHumbleObject _humbleObject;
    public NewsController() : this(new NewsHumbleObject())
    {
    }
    public NewsController(INewsHumbleObject humbleObject)
    {
        _humbleObject = humbleObject;
    }

    public void TennisAsync()
    {
        var url = "...";
        _humbleObject.Prepare(AsyncManager);
        _humbleObject.GetFeedItems(url, AsyncManager);
    }

    public ActionResult TennisCompleted(IList<SyndicationItem> feedItems)
    {
        return View(feedItems);
    }
}
```

Testing this method is much simpler now:

```
[TestMethod]
public void Test_If_News_Are_Being_Downloaded_Async()
{
    // Arrange
    var humbleObject = new FakeNewsHumbleObject();
    var controller = new NewsController(humbleObject);
    controller.TennisAsync();

    // Act
    var response = controller.AsyncManager.Parameters["feedItems"]
                as IList<SyndicationItem>;

    // Assert
    Assert.IsNotNull(response);
    Assert.AreEqual(response.Count, 3);
}
```

And finally here's the fake humble object:

```
public class FakeNewsHumbleObject : INewsHumbleObject
{
    public void GetFeedItems(String url, AsyncManager asyncManager)
    {
        // Place canned data inside the Parameters collection of AsyncManager
        var items = new List<SyndicationItem>
                    {
                        new SyndicationItem(
                            "News #1",
                            "This is the first news",
                            null),
                        new SyndicationItem(
                            "News #2",
                            "This is the second news",
                            null),
                        new SyndicationItem(
                            "News #3",
                            "This is the third news",
                            null)
                    };
        asyncManager.Parameters["feedItems"] = items;
        asyncManager.OutstandingOperations.Decrement();
    }

    public void Prepare(AsyncManager asyncManager)
    {
        // Same as production code
        asyncManager.OutstandingOperations.Increment();
    }
 }
```

Even though the *GetFeedItems* method in the fake humble object just returns canned values, it's hard to replace it with a mock object. The method also needs to deal with some state, and that requires code. That's why I prefer to go with my own fake in similar situations.

Mocking the HTTP Context

I'm not really happy when I have to mock the HTTP context to write some ASP.NET MVC unit tests. Sometimes you really need it. More often than you might think, however, a bit of refactoring on your code makes mocking the HTTP context unnecessary.

When it comes to unit testing, there's a mistake (or at least I call it this) I see too often. Some developers seem unable to look beyond the controller level. These developers put large chunks of logic in the controller. These developers make little use of specific, highly specialized classes. These developers seem to think that beyond the controller you can have only a Microsoft SQL Server database or an Entity Framework model. And just having a layer of repositories around LINQ-to-Entities queries is already a cutting-edge solution. In Chapter 7, I tried to illustrate a different pattern for plain code.

If the granularity of controller methods is fairly coarse, you inevitably need to mock a long list of objects to run your tests. A controller method that deals directly with query strings and session state needs to have a valid HTTP context arranged for a test to run. This is extra work for you. Also, refactoring the controller method to add encapsulation and wrappers is extra work for you—but it's work of a different type.

You see immediately the benefits of a cleaner design: testing is much easier for one thing. Efforts expended to make tests work result in more wasted time, instead. When the test finally runs, you feel more relieved than satisfied. Mocking the HTTP context is a constant requirement if you end up with coarse-grained controller methods. It's not just the ASP.NET MVC framework; it's also you.

Having said that, however, I have to add that sometimes you really need to mock the HTTP context. Let's see how to do it.

Mocking the *HttpContext* Object

The *HttpContext* object inherits from *HttpContextBase*, so all you need to do is create a mock for it. The *HttpContext* object is a plain aggregator of object references; it hardly needs to contain extra code. So a mock is just fine most of the time. Here's how to build a fake HTTP context using Moq:

```
public void BuildHttpContextForController(Controller controller)
{
    var contextBase = new Mock<HttpContextBase>();
    var request = new Mock<HttpRequestBase>();
    var response = new Mock<HttpResponseBase>();
    var server = new Mock<HttpServerUtilityBase>();
    ...

    contextBase.Setup(c => c.Request).Returns(request);
    contextBase.Setup(c => c.Response).Returns(response);
    contextBase.Setup(c => c.Server).Returns(server);

    // Pass the fake context down to the controller instance
    var context = new ControllerContext(
                new RequestContext(contextBase.Object, new RouteData()), controller);
    controller.ControllerContext = context;
    return;
}
```

This is only the first level of mocking for ASP.NET intrinsic objects. Whenever the application queries for, say, *Request* in a unit test, it gets the mocked object. The mocked object, however, is a dummy object and needs its own setup. Let's see a couple of examples.

Mocking the *Request* Object

You probably want to extend the mocked *Request* object with expectations regarding some specific members. For example, here's how to simulate a GET or POST request in a test:

```
var method = "get";
contextBase.Setup(c => Request.HttpMethod).Return(method);
```

Earlier in the chapter, while discussing the testing of routes, we also ran into similar code:

```
var url = ...;
contextBase.Expect(c => Request.AppRelativeCurrentExecutionFilePath).Return(url);
```

You probably don't want to use the *Request.Form* object to read about posted data from within a controller because you might find model binders to be more effective. However, if you have a legacy call to *Request.Form["MyParam"]* in one of your controller's methods, how would you test it?

```
// Prepare the fake Form collection
var formCollection = new NameValueCollection();
formCollection["MyParam"] = ...;

// Fake the HTTP context and bind Request.Form to the fake collection
var contextBase = new Mock<HttpContextBase>();
contextBase.Setup(c => c.Request.Form).Returns(formCollection);

// Assert
...
```

In this way, every time your code reads anything through *Request.Form* it actually ends up reading from the name/value collection provided for testing purposes.

Mocking the *Response* Object

Let's see a few examples that touch on the *Response* object. For example, you might want to mock up *Response.Write* calls by forcing a fake *HttpResponse* object to write to a text writer object:

```
var writer = new StringWriter();
var contextBase = new Mock<HttpContextBase>();
contextBase.Setup(c => c.Response).Return(new FakeResponse(writer));
```

In this case, the *FakeResponse* class is used as shown here:

```
public class FakeResponse : HttpResponseBase
{
    private readonly TextWriter _writer;
    public FakeResponse(TextWriter writer)
    {
        _writer = writer;
    }

    public override void Write(string msg)
    {
        _writer.Write(msg);
    }
}
```

This code will let you test a controller method that has calls to *Response.Write* like the one shown here:

```
public ActionResult Output()
{
    HttpContext.Response.Write("Hello");
    return View();
}
```

Here's the test:

```
[TestMethod]
public void Should_Response_Write()
{
    // Arrange
    var writer = new StringWriter();
    var contextBase = new Mock<HttpContextBase>();
    contextBase.Setup(c => c.Response).Returns(new FakeResponse(writer));
    var controller = new HomeController();
    controller.ControllerContext = new ControllerContext(
                contextBase.Object, new RouteData(), controller);

    // Act
    var result = controller.Output() as ViewResult;
    if (result == null)
        Assert.Fail("Result is null");

    // Assert
    Assert.AreEqual("Hello", writer.ToString());
}
```

Similarly, you can configure a dynamically generated mock if you need to make certain properties or methods just return a specific value. Here are a couple of examples:

```
var contextBase = new Mock<HttpContextBase>();

// Mock up the Output property
contextBase.Setup(c => Response.Output).Returns(new StringWriter());

// Mock up the Content type of the response
contextBase.Setup(c => Response.ContentType).Returns("application/json");
```

For cookies, on the other hand, you might want to mock the *Cookies* collection on both *Request* and *Response* to return a new instance of the *HttpCookieCollection* class, which will act as your cookie container for the scope of the unit test. I'll show more details related to mocking the *Response* object later in the chapter while discussing how to test controller methods with action filters.

Mocking the *Session* Object

A mock is easier to use, but sometimes you need to assign a behavior to the various methods of the mocked object. This is easy to do when the behavior is as simple as returning a given value. To effectively test whether the method correctly updates the session state, though, you need to provide an in-memory object that simulates the behavior of the original object and has the ability to store information—not exactly an easy task to mock. Using a fake session class, however, makes it straightforward. Here's a minimal yet effective fake for the session state:

```
public class FakeSession : HttpSessionStateBase
{
    private readonly Dictionary<String, Object> _sessionItems =
                                            new Dictionary<String, Object>();

    public override void Add(String name, Object value)
    {
        _sessionItems.Add(name, value);
    }

    public override Object this[String name]
    {
        get { return _sessionItems.ContainsKey(name) ? _sessionItems[name] : null; }
        set { _sessionItems[name] = value; }
    }
}
```

And here's how to arrange a test:

```
[TestMethod]
public void Should_Write_To_Session_State()
{
    // Arrange
    var contextBase = new Mock<HttpContextBase>();
    contextBase.Setup(c => c.Session).Returns(new FakeSession());
    var controller = new HomeController();
    controller.ControllerContext = new ControllerContext(
                      contextBase.Object, new RouteData(), controller);

    // Act
    var expectedResult = "green";
    controller.SetColor();    // Runs Session["PreferredColor"] = "green";

    // Assert
    var result = controller.HttpContext.Session["PreferredColor"];
    Assert.AreEqual(result, expectedResult);
}
```

If your controller method only reads from *Session*, your test can be even simpler and you can avoid faking the *Session* entirely. Here's a sample controller action:

```
public ActionResult GetColor()
{
    var o = Session["PreferredColor"];
    if (o == null)
        ViewData["Color"] = "No preferred color";
    else
        ViewData["Color"] = o as String;

    return View("Color");
}
```

The following code snippet shows a possible way to test the method just shown:

```
// Arrange
var contextBase = MockRepository.GenerateMock<HttpContextBase>();
contextBase.Expect(s => s.Session["PreferredColor"]).Return("Blue");
var controller = new HomeController();
controller.ControllerContext = new ControllerContext(
                                    contextBase.Object, new RouteData(), controller);

// Act
var result = controller.GetColor() as ViewResult;
if (result == null)
   Assert.Fail("Result is null");

// Assert
Assert.AreEqual(result.ViewData["Color"].ToString(), "Blue");
```

In this case, you instruct the HTTP context mock to return the string "Blue" when its *Session* property is requested to provide a value for the entry "*PreferredColor*".

In what is likely the much more common scenario, where a controller method needs to read *and* write the session state, you need to use the test solution based on some *FakeSession* class.

Mocking the *Cache* Object

Mocking the ASP.NET *Cache* object is a task that deserves a bit more attention, even though mocking a caching layer doesn't require a new approach. The *HttpContextBase* class has a *Cache* property, but you can't mock it up because the property doesn't represent an abstraction of the ASP.NET caching systems—it's a concrete implementation of a particular class instead. Here's how the *Cache* property is declared on the *HttpContextBase* class:

```
public abstract class HttpContextBase : IServiceProvider
{
    public virtual Cache Cache { get; }
    ...
}
```

The type of the *Cache* property is actually *System.Web.Caching.Cache*—the real cache object, not an abstraction. Even more unfortunately, the *Cache* type is sealed and therefore is not mockable and is unusable in unit tests.

What can you do? There are two options. One entails using testing tools that can deal with sealed classes. One is Typemock Isolator (a commercial product); another is Microsoft Moles. The other possibility is to use a wrapper class to perform any access to the *Cache* from within any code you intend to test. We examined this approach in Chapter 5.

Based on what you've seen, you create a cache service object that implements a given interface—say *ICacheService*. Next, you register this class with the application in *global.asax* and add a public static property to read/write the cache:

```
protected void Application_Start()
{
    ...

    // Inject a global caching service(for example, one based on ASP.NET Cache)
    RegisterCacheService(new AspNetCacheService());
}

private static ICacheService _internalCacheObject;
public void RegisterCacheService(ICacheService cacheService)
{
    _internalCacheObject = cacheService;
}

public static ICacheService CacheService
{
    get { return _internalCacheObject;  }
}
```

In controller methods, you stop using the *HttpContext.Cache* entirely. Your controller will have the following layout instead:

```
public partial class HomeController : Controller
{
    public ActionResult SetCache()
    {
        // MvcApplication is the name of the global.asax class; change it at will
        MvcApplication.CacheService["PreferredColor"] = "Blue";
        return View();
    }
    ...
 }
```

How would you test this? Here's an example:

```
[TestMethod]
public void Should_Write_To_Cache()
{
    // Arrange
    var fakeCache = new FakeCache();
    MvcApplication.RegisterCacheService(fakeCache);

    // Act
    controller.SetCache();

    // Assert
    Assert.AreEqual("Blue", fakeCache["PreferredColor"].ToString());
}
```

The *FakeCache* class can be something like this:

```
public class FakeCache : ICacheService
{
    private readonly Dictionary<String, Object> _cacheItems =
                 new Dictionary<String, Object>();

    public object this[String name]
    {
        get
        {
            if (_cacheItems.ContainsKey(name))
                return _cacheItems[name];
            else
                return null;
        }
        set { _cacheItems[name] = value; }
    }
}
```

In this way, you can test controller methods and services, making use of cached data even when the cache is a distributed cache. The cache service hides all details and makes the application more extensible and testable.

Testing Controller Methods with Filters

When you write a unit test for a controller method, you start from a freshly created instance of the controller class. This is not exactly what happens in a real application. In a real application, the controller instance is returned from a controller factory and is padded with some context information about the controller itself, routes, and the request. Furthermore, the method is invoked by the action invoker, which takes care of loading any filters as appropriate.

The bottom line is that the execution of a controller method is a process tightly coupled with the ASP.NET MVC infrastructure. It's one thing to test the behavior of a controller class method; it's quite another to do that when taking filters (for example, infrastructure) into account.

My suggestion is to test methods in isolation, ignoring any filters they might have attached. Next, if the filters are really critical to your application, you can try to test the filter in isolation. Testing a filter in isolation might not be a walk in the park, because you need to create mocks for the filter context—which filter context depends on the type of filter you're writing. (See Chapter 8, "Customizing ASP.NET MVC Controllers.")

The following code shows an approach that I successfully employed a few times, though I don't dare say that it will work each and every time. The skeleton of the code goes in the right direction; be aware that you might need to apply some changes here and there depending on what the filter is really doing.

Let's assume you have the following controller method:

```
[AddHeader(Name="Author", Value="Dino")]
public ActionResult About()
{
   return View();
}
```

The *AddHeader* attribute is expected to add a response header with given values. Here's an approach to testing this method along with the associated filter:

```
[TestMethod]
public void Test_Method_With_Filters()
{
    var actionName = "about";
    var controllerName = "home";

    // Mock controller context
    var httpContext = GetHttpContext();
    var routeData = GetRouteData(controllerName, actionName);
    var controller = new HomeController {ActionInvoker = new UnitTestActionInvoker()};
    var controllerContext = new ControllerContext(httpContext, routeData, controller);

    // It is known that About is decorated with [AddHeader(Name="Author", Value="Dino")]
    controller.ActionInvoker.InvokeAction(controllerContext, actionName);

    var expectedHeader = "Dino";
    Assert.AreEqual(controllerContext.HttpContext.Response.Headers["Author"], expectedHeader);
}
```

The idea is that you use the controller's action invoker to call the method instead of calling it directly from the controller's instance. The action invoker knows about filters and ensures they'll be called in the right order as in a real request. The action invoker class has a public *InvokeAction* method, which requires a valid controller context.

The first issue is getting a valid controller context where critical parts of the HTTP context are properly mocked up. Critical parts are just those parts involved in the operation you're testing. For the controller context, you need to set up *Response* and *RouteData*, as shown here:

```
private static HttpContextBase GetHttpContext()
{
    var response = new FakeResponse();
    var mockHttpContext = new Mock<HttpContextBase>();
    mockHttpContext.Setup(c => c.Response).Returns(response);
    return mockHttpContext.Object;
}
private static RouteData GetRouteData(String controllerName, String actionName)
{
    var routeData = new RouteData();
    routeData.Values["controller"] = controllerName;
    routeData.Values["action"] = actionName;
    return routeData;
}
```

The second issue to take into account is the action invoker. By design, the action invoker is tightly coupled to the infrastructure and knows several internal ASP.NET MVC objects—from *Request* to view engines. You can't just use the default action invoker. At some point, in fact, it will throw a null reference exception. On the other hand, you can't reasonably catch up with the invoker's requirements and mock up every possible mockable object in the entire ASP.NET MVC code base. A smarter (and shorter) solution is required. Mine consists of using a unit-test-specific version of the default action invoker class.

The unit-test action invoker differs from the default one in one aspect—it doesn't execute the action result. Clearly, this makes the solution a partial solution that doesn't work in every scenario. However, as you'll see it's probably the best compromise you can reach.

An action invoker that doesn't execute the action result is problematic only for a small subset of action filters—those that need to run *after* the action result has been processed. Personally, for what little that it means, in three years of extensive ASP.NET MVC programming I never had the need to write such a filter. You can find the *UnitTestActionInvoker* class in the companion source code for this book.

Let's finalize the example.

The *AddHeader* action filter (discussed in Chapter 8) adds a response header and, of course, it does that via the *Response* object. So you need a mock for the ASP.NET MVC *Response* object. This mock will expose the *Headers* collection and provide facilities to track the headers being added:

```
public class FakeResponse : HttpResponseBase
{
    private readonly NameValueCollection _headers;
    public FakeResponse()
    {
        _headers = new NameValueCollection();
    }
```

```
    public override void AddHeader(String name, String value)
    {
        Headers.Add(name, value);
    }

    public override NameValueCollection Headers
    {
        get { return _headers; }
    }
}
```

It is important that in this case you also override *AddHeader*—the implementation on the native *HttpResponse* object does a lot of things that involve objects missing in a unit-test scenario. As a result, your test will fail miserably. On the other hand, all you need to do is check whether the action filter called *AddHeader* with correct values.

To finish off, it would be nice to reflect a bit more on the reasons that drove me to cut off the execution of the action result in the *UnitTestActionInvoker* class. Especially when the action result is *ViewResult*, meaning that HTML must be generated, the invoker should connect to one of the selected view engines, which in turn needs to walk the file system to retrieve the template from a virtual path. This is infrastructure, not your code. This code is not really supposed to be mockable at this level. So from my perspective, it reasonably fails, returning some null reference exception.

What else can you do?

You can test methods and (post) result filters separately. To test a filter, you need to mock its filter context and add the references required case by case, which is not really different from mocking the HTTP context.

Note This example tested just one filter. Testing multiple filters at the same time is not impossible, but it requires more care and therefore more time. The point here is not whether testing filters and methods together is possible, but whether it's effective. Sometimes you might find it preferable to test them together (as discussed here); sometimes you might find separate tests more effective. Finally, sometimes you might even find that having unit tests for filters is not necessary.

Summary

Testability is a fundamental aspect of software, as the ISO/IEC 9126 paper recognized back in 1991. With ASP.NET MVC, designing your code for testability is easier and encouraged. But you also can write testable code in ASP.NET Web Forms and test it to a good extent. Testability is an excellent excuse to pursue good design. And design makes a difference under the hood.

We all agree that writing tests is helpful, recommended, and even sexy. However, unless you associate it with some test-driven design methodology, writing tests risks being an end in itself. On one hand, you have the classes to test; on the other hand, you have classes that promise to test other classes. And you have nothing in the middle that guarantees that tests are appropriate and meaningful.

An approach that creates the environment for a much more effective refactoring, while not excluding unit testing, is based on software contracts. A software contract defines the terms and conditions for each method in each class. That gives you an optional run-time checking mechanism as well as concrete guidance on how to refactor when you have to. Software contracts are not a new concept in software, but they have found widespread implementation only in .NET 4. Check out the documentation for code contracts and the articles I wrote on the subject in the 2011 installments of the Cutting Edge column in MSDN Magazine. (See *http://msdn.microsoft.com/en-us/magazine/default.aspx*.)

PART III

Client-Side

CHAPTER 10

More Effective JavaScript

It matters not what someone is born, but what they grow to be.

—J. K. Rowling

A statement you might hear frequently these days is that JavaScript is the language of the future—the primary tool to deal with the wonders being delivered by HTML5. So far, I haven't jumped onto the HTML5 bandwagon, although I do realize the key role that HTML5 plays when it comes to planning multiplatform mobile applications.

Many people see HTML5 as the lingua franca of future applications—write it once and run it everywhere. I can't see what the future has in store, but looking at the past I can conclude that we never had any real lingua franca even though we've been told quite a few times that one has been found. Anyway, wherever you expect to find HTML5 in the near future (be it Microsoft Windows 8, mobile devices, or websites), you should have JavaScript to code for it.

Regardless of whether or not HTML5 is the lingua franca of applications for this decade, JavaScript has regained a lot of importance even beyond the realm of web applications. All web applications these days are required to add more and more client-side features. I'm not simply talking about client-side validation and input facilities—I'm talking about full Ajax applications. A full Ajax application is a web application that reduces page swaps to a bare minimum. A full Ajax application is centered on very few core pages whose user interface changes interactively following the user's action. Obviously, you do a lot of work on the HTML Document Object Model (DOM), and you do that work using JavaScript.

As an ASP.NET MVC developer, you should be ready to write more JavaScript code. More importantly, you should be ready to import more JavaScript code written by others. Either of these two ways of using JavaScript is OK because they are not mutually exclusive options. Currently, the most effective approach to adding more power to the client is by using ad hoc libraries full of DOM facilities and adding new features to the existing JavaScript language.

The ideal mix of JavaScript code is often obtained by stacking up and composing together bits and pieces of existing libraries (for example, jQuery, jQuery UI, and various plug-ins) in a custom recipe that suits a particular application. The amount of JavaScript code required for each view is growing; you need to stop for a while, refresh your JavaScript skills, and develop new practices to incorporate JavaScript in highly dynamic views, such as Ajax views.

This chapter revisits the basics of the JavaScript language and discusses some patterns to organize and reuse JavaScript code from within an ASP.NET MVC application.

Revisiting the JavaScript Language

Developed for the web in the 1990s, JavaScript is widely used today well outside of the web. JavaScript is used to write application extensions (for example, Photoshop or Firefox); it's used to create native mobile applications (for example, PhoneGap or Appcelerator Titanium); it's used for some flavors of server-side applications (for example, Node.js).

Overall, JavaScript is an unusual language because it was created for non-developers; it has remarkably low barriers to entry, and it's flexible enough that experts can use it to do nearly everything they need to. As I see things, the challenge today is for average developers to create effective content with some good design while keeping readability and maintainability at a decent level.

Language Basics

Over the years, several descriptions have been applied to the JavaScript language: *object-based but not object-oriented*, *just too simple*, *too flexible for being simple*, and maybe more. That fact is that some of its native features are now everywhere. Anonymous functions are the most prominent example because you find them in nearly every language today, and closures are appearing in Java and PHP as well.

JavaScript code is interpreted, meaning that JavaScript programs need an environment in which to run. Their natural habitat is the web browser. The syntax is driven by ECMAScript. The latest standard is ECMAScript 5 (December 2009). The current de facto standard, however, dates back to the first release in 1999.

Let's briefly navigate through the basics of the language and refresh concepts that, frankly, most ASP.NET developers just picked up but never studied thoroughly.

Type System

The JavaScript type system is made of primitive types and a few built-in objects. When you write JavaScript code, however, the range of types you can work with is actually larger. In addition to using built-in objects, you can also rely on objects provided by the host. The canonical example is the *window* object that the most common JavaScript host—the web browser—publishes into the JavaScript engine.

Primitive types are number, string, Boolean, *null*, *undefined*, *Object*, and *Function*. The built-in objects are *Array*, *Math*, *Date*, *Error*, *RegExp*, plus wrapper objects for a few primitive types: *String*, *Boolean*, and *Number*.

The number type represents floating-point numbers with 0 or more decimal places. There are no separate types for integers, long, or bytes. The range of numbers is between -10^{308} and 10^{308}. You can write numbers in decimal, octal, or hexadecimal format. The special number NaN represents the result of a math operation that makes no sense (such as division by zero).

The string type represents a row of 0 or more characters; it doesn't represent an array. Individual characters are represented as strings of 1 character. Special characters in a string begin with a back slash (\), such as *n* to indicate a carriage return. The content of a string is bracketed in matching pairs of single or double quotes. The primitive value is wrapped in a *String* object that adds a few methods, including *split* and *substring*.

Null vs. Undefined

A JavaScript variable that doesn't contain a meaningful value can be assigned *null* as well as *undefined*. What's the difference? When it comes to nullness, JavaScript introduces a subtle difference that many higher level languages such as C# and Java miss.

In particular, *undefined* is the default value that the runtime environment assigns to any variables being used. An unassigned variable contains *undefined*, not *null* or some default value as in C#. On the other hand, *null* is a value that still represents a null, empty, or nonexisting reference, but it has been explicitly assigned by the developer. In a nutshell, a variable set to *undefined* has never been touched by any code; a variable that holds *null* got that value via some path in your code.

If you run *typeof* on an unassigned variable, you get *undefined*—it's a whole type by itself. If you run *typeof* on a variable assigned with *null*, you get *Object*. Pay attention to the following code:

```
var x;  // hence, undefined
var y = null;
```

What happens if you compare *x* and *y*? If you use the == operator, you get *true*, meaning that *undefined* ultimately evaluates to *null*. If you compare using the === operator, you get *false*: the two variables hold the same value but are of different types.

Local and Global Variables

In JavaScript, a variable is a storage location that is not restricted to always storing values of a fixed type. When assigned a value, variables take on the type of the data being stored. For this reason, a JavaScript variable might change its type quite a few times in its lifespan.

```
var data = "dino";   // now data is of type string
data = 123;          // now data is of type number
```

This behavior is different from the typical behavior of C# variables, unless C# variables are defined of type *dynamic* in .NET 4. Variables spring into existence the first time they're used; until then, they hold a value of *undefined*.

When defining variables, you should use the *var* keyword as a hint to the parser and yourself. The *var* keyword is not strictly required, but it's highly recommended. Variables defined within a function are scoped to the function if declared with *var*. If not, variables are treated as global, but they remain undefined until the function executes once. Variables declared in the global scope are always global regardless of whether *var* is used.

```
<script type="text/javascript">
  var rootServer = "http://www.expoware.org/";   // global
  section = "mobile";                            // global
</script>

<script>
  function doSomething() {
     var temp = 1;                               // local
     mode = 0;                                   // global, but undefined until called
  }
</script>
```

The JavaScript runtime environment stores global variables as properties of hidden objects referenced through the *this* keyword. Browsers often mirror this object via the *window* object.

In any programming language, coding is (much) easier if you can use global variables. Globals, however, have downsides too. A critical downside is the risk of name collisions between variables defined in different parts of your code, third-party libraries, advertising partners, and analytics libraries. A name collision combined with the dynamic typing of JavaScript variables might lead you to inadvertently modify the state of the application with unpleasant anomalies at run time.

Consider that creating globals unwillingly is easy too: miss a *var* and you get a global; mistype a variable name in an assignment and you get a fresh new global. This latter feature is possible because JavaScript allows you to use a variable without declaring it first. When you need to use global variables, a good technique is creating them as properties of a wrapper object. You place the following code in a JavaScript file you link from every page:

```
var Globals = (function() { return this; }());
```

Next, you use *Globals.Xxx*, where *Xxx* is any global variable you might need to have. At least, this ensures your global variables will stand out.

Note JSLint (available at *http://www.jslint.com*)—an online tool for static analysis of JavaScript code—does help catch anti-patterns in your code, including the lack of *var* keywords.

Variables and Hoisting

Hoisting is a JavaScript feature that allows developers to declare variables anywhere in the scope and use them anywhere in that scope. In JavaScript, you're allowed to first use the variable and then declare it (as *var*) later:

```
function() {     // Not allowed in C#
    mode = 1;
    ...
    var mode;
}
```

The overall behavior is as if the *var* statement was placed at the top. Historically, the feature was introduced to keep as low as possible the entry barrier to JavaScript. When you use JavaScript to write significant portions of code, however, hoisting is a clear source of confusion and becomes error prone. A good habit consists of placing all your variables at the top of each function—preferably, in a single *var* statement—as shown here:

```
function() {
   var start = 0,
       total = 10,
       sum = function (x,y) {return x+y;},
       index;
   ...
}
```

JSLint can be instructed to catch the use of multiple *var* statements in a single function and remind you about this pattern.

Objects

The primary reason for not cataloging JavaScript as an object-oriented language is that the definition of an object you get from JavaScript is different from the commonly accepted idea of an object you get out of classic object-oriented languages such as C++ or C#.

In JavaScript, an object is a dictionary of name/value pairs. The blueprint of the object is implicit, and you have no way to access it. A JavaScript object usually has only data, but you can add behavior. The (explicit) structure of an object can change at any time—with new methods and properties. The implicit structure never changes. Because of the implicit blueprint, any apparently blank object in JavaScript still has a few properties, such as *prototype*. (I'll return to *prototype* later.)

Note that variables that store objects don't actually contain the object's dictionary; they just reference the object's bag of properties. The bag of properties is distinct from the variable store, and different variables can reference the same bag of data. The *new* operator creates a new bag of properties. When you pass an object around, you just pass the reference to the bag.

Adding a member to a JavaScript object works only for that particular instance. If you want to add a new member to all instances being created of that type, you have to add the member to the object's prototype:

```
if (typeof Number.prototype.random === "undefined") {
    Number.prototype.random = function() {
        var n = Math.floor(Math.random() * 1000);
        return n;
    };
}
```

Note that augmenting the prototype of native objects is considered a bad practice because it makes the code less predictable and can hurt maintainability.

You use the *Object* type to create aggregates of values and methods, which is the closest you get in JavaScript to C# objects. The direct use of the *Object*'s constructor (as shown next) is usually disregarded:

```
var dog = new Object();
dog.name = "Jerry Lee Esposito";
dog.getName = function() {
     return this.name;
}
```

A better approach entails using an object literal as shown here:

```
var dog = {
   name: "Jerry Lee Esposito",
   getName: function() {
      return this.name;
   }
};
```

If you use the *Object*'s constructor, the interpreter has to resolve the scope of the constructor call. In JavaScript, there's no guarantee that no local object exists in the scope with the same name as a global object. The interpreter, therefore, has to walk up the stack to find the nearest definition of the constructor that applies. In addition to this performance issue, using the constructor directly also doesn't transmit the sense of objects as dictionaries, which is a key point of JavaScript programming.

Functions

In JavaScript, a function is a bit of code bundled up into a block and, optionally, given a name. If a function is not given a name, it's an anonymous function. Functions are treated like objects, they can have properties, and they can be passed around and interacted with.

In JavaScript, anonymous functions are the pillar of functional programming. An anonymous function is a direct offshoot of lambda calculus or, if you prefer, a language adaptation of old-fashioned function pointers. Here's an example of an anonymous function:

```
function(x, y) {
   return x + y;
}
```

The only difference between a regular function and an anonymous function is in the name (or lack thereof).

You use functions for two main reasons: for creating custom objects and, of course, for defining repeatable behavior. Of the two reasons, the former is the most compelling in JavaScript. Consider the following code:

```
// The this object is implicitly returned
var Dog = function(name) {
  this.name = name;
  this.bark = function() {
     return "bau";
  };
};
```

To use the *Dog* object, you need to instantiate it using the classic *new* constructor as shown here:

```
var jerry = new Dog("jerry");    // OK
var hassie = Dog("hassie");      // Doesn't throw but, worse, may alter the application's state
```

The tricky thing is that if you forget the *new* operator, you won't get any exception and your code just runs, but the *this* object being used is now the global *this* object. This means you're potentially altering the state of the application. Here's a safe countermeasure:

```
var Dog = function(name) {
   var that = {};
   that.name = name;
   that.bark = function() {
      return "bau";
   };
   return that;
};
```

The difference is that you now explicitly create and return a new object—named *that*. This is familiarly known as the "Use-That-Not-This" pattern.

Object-Orientation in JavaScript

There was a time when JavaScript code in web pages was limited to a few lines of basic manipulation of the DOM. There was no need to design this code into reusable blocks and attach it unobtrusively to page elements. Although the average level of JavaScript code complexity is nearly the same as in the recent past, the quantity of it you need to have on each page is dramatically rising. Forms of packaging are required, because today JavaScript code is like a small application on its own. You need to make gains in reusability and also (if not especially) in maintainability. Furthermore, you need to make sure your code runs in isolation because in JavaScript it's too easy to miss variables, spoil globals, and mistype names without a clear indication that you have done so. In this regard, JSLint is a great help, but it's not like a compiler.

To top off the discussion about the basics of the JavaScript language, let me introduce *closures* and *prototypes*—two approaches you can take to implement object-orientation in JavaScript.

Making Objects Look Like Classes

Before I get to closures and prototypes, however, let me spend a few more words on the native *Object* type and its usage. As mentioned, you can use the *new* keyword to create a new dictionary-like object. Next, you stuff data into it and add methods by wiring functions to property names. Here's an example:

```
var person = new Object();
person.Name = "Dino";
person.LastName = "Esposito";
person.BirthDate = new Date(1979,10,17)
person.getAge = function() {
  var today = new Date();
  var thisDay = today.getDate();
  var thisMonth = today.getMonth();
  var thisYear = today.getFullYear();
  var age = thisYear-this.BirthDate.getFullYear()-1;
  if (thisMonth > this.BirthDate.getMonth())
      age = age +1;
  else
  if (thisMonth == this.BirthDate.getMonth() &&
      thisDay >= this.BirthDate.getDate())
      age = age +1;
  return age;
}
```

What you have is an object modeled after a person; you don't really have a *Person* object. As you saw earlier, this has both readability and performance issues. In addition, the object is sparsely defined over multiple lines.

Closures and prototypes offer alternate ways to define the layout of a type, and they are the native mechanisms to leverage for doing object-oriented programming (OOP) in JavaScript.

Using Closures

A *closure* is a general concept of programming languages. Applied to JavaScript, a closure is a function that can have variables and methods defined together within the same context. In this way, the outermost (anonymous or named) function "closes" the expression. Here's an example of the closure model for a function that represents a *Person* type:

```
var Person = function(name, lastname, birthdate) {
   this.Name = name;
   this.LastName = lastname;
   this.BirthDate = birthdate;

   this.getAge = function() {
      var today = new Date();
      var thisDay = today.getDate();
      var thisMonth = today.getMonth();
      var thisYear = today.getFullYear();
      var age = thisYear-this.BirthDate.getFullYear()-1;
      if (thisMonth > this.BirthDate.getMonth())
          age = age +1;
```

```
      else
         if (thisMonth == this.BirthDate.getMonth() &&
             thisDay >= this.BirthDate.getDate())
             age = age +1;
      return age;
   }
}
```

As you can see, the closure is nothing more than the constructor of the pseudoclass. In a closure model, the constructor contains the member declarations and members are truly encapsulated and private to the class. In addition, members are instance based, which increases the memory used by the class. Here's how you use the object:

```
var p = new Person("Dino", "Esposito", new Date( ... );
alert(p.Name + " is " + p.getAge());
```

The closure model gives full encapsulation, but nothing more. To compose objects, you can only resort to aggregation.

Using Prototypes

The prototype model entails that you define the public structure of the class through the JavaScript *prototype* object. The following code sample shows how to rewrite the preceding *Person* class to avoid a closure:

```
// Pseudo constructor
var Person = function(name, lastname, birthdate) {
    this.initialize(name, lastname, birthdate);
}

// Members
Person.prototype.initialize = function(name, lastname, birthdate) {
    this.Name = name;
    this.LastName = lastname;
    this.BirthDate = birthdate;
}

Person.prototype.getAge = function() {
    var today = new Date();
    var thisDay = today.getDate();
    var thisMonth = today.getMonth();
    var thisYear = today.getFullYear();
    var age = thisYear-this.BirthDate.getFullYear()-1;
    if (thisMonth > this.BirthDate.getMonth())
        age = age +1;
    else
       if (thisMonth == this.BirthDate.getMonth() &&
           thisDay >= this.BirthDate.getDate())
           age = age +1;
    return age;
}
```

In the prototype model, the constructor and members are clearly separated and a constructor is always required. As for private members, well, you just don't have them. The *var* keyword that would keep them local in a closure doesn't apply in the prototype model. So you can define a getter/setter for what you intend to be properties, but the backing field will remain accessible from the outside. You can resort to some internal convention, such as prefixing with an underscore the name of members you intend as private. That's just a convention, however, and nothing prevents developers from accessing what the class author considers private.

By using the prototype feature, you can achieve inheritance by simply setting the *prototype* property of a derived object to an instance of the "parent" object:

```
Developer = function Developer(name, lastname, birthdate) {
   this.initialize(name, lastname, birthdate);
}
Developer.prototype = new Person();
```

Note that you always need to use *this* to refer to members of the prototype from within any related member function.

In the prototype model, members are shared by all instances as they are invoked on the shared *prototype* object. In this way, the amount of memory used by each instance is reduced, which also provides for faster object instantiation. Aside from syntax peculiarities, the prototype model makes defining classes much more similar to the classic OOP model than the closure model.

Plain Custom Objects vs. a Hierarchy of Classes

The choice between closure and prototype should also be guided by performance considerations and browser capabilities. Prototypes have good load times in all browsers. Closures work great in some browsers (for example, Internet Explorer) and worse in others.

Prototypes provide better support for Microsoft IntelliSense, and they allow for tool-based statement completion when used in tools that support this feature, such as Microsoft Visual Studio. Prototypes can also help you obtain type information by simply using reflection. You won't have to create an instance of the type to query for type information, which is unavoidable if closures are used. Finally, prototypes allow you to easily view private class members when debugging.

So you have two basic options for dealing with JavaScript objects that look like classes. Prototypes are the option chosen most often by library designers, including designers of the Microsoft Ajax library. Also, in jQuery, the *prototype* property is used extensively.

Having said that, if I had to write client code for a web front end, I'd probably go with jQuery, use a lot of anonymous functions, and not even bother about having a hierarchy of custom objects. I would certainly create custom objects, but I'd use them as plain and flat containers of data and behavior—with no inheritance or polymorphism. If, on the other hand, I had to write my own framework to support some server-side infrastructure, I'd probably opt for a more classic object-oriented approach. In that case, however, I'd probably consider using an existing library instead of rolling my own. For that, MooTools is an excellent choice. (See *http://mootools.net*.)

jQuery's Executive Summary

Without beating around the bush, I'll say that if you're writing JavaScript code in web views today you're likely using jQuery. If you're not, you should be. The jQuery library is certainly not the only JavaScript library you can pick up to save yourself quite a bit of work around DOM manipulation and event handling. However, it's the world de facto standard. I consider jQuery almost an extension to the JavaScript language, and certainly an extension to the JavaScript skills of any web developers.

This chapter is not the place where you can expect to find some sort of extensive coverage of jQuery. For that, you can pick some good books or just check out the online documentation at *http://docs.jquery.com/Main_Page*. If you're looking for online content in a more readable format than dry documentation, look here: *http://jqfundamentals.com/book*.

In this chapter, I'm providing an overview of the key concepts in jQuery, a strong understanding of which will enable you to quickly grab programming details and features in the library.

DOM Queries and Wrapped Sets

The main reason for the worldwide success of the jQuery library is its unique mix of functional and DOM programming. The library works by selecting DOM elements and applying functions over them, which is just what client web developers need to do most of the time.

The Root Object

The root of the jQuery library is the *jQuery* function. Here's the overall structure of the library:

```
(
    function( window, undefined )
    {
         var jQuery = (function() {
              // Define a local copy of jQuery
              var jQuery = function(selector, context) {
                 ...
              }
              ...
              return jQuery;
         })();

         /* the rest of the library goes here */
         ...
         window.jQuery = window.$ = jQuery;
    }
) (window);
```

The nested *jQuery* function is mapped as an extension to the browser's *window* object and is aliased with the popular *$* function. The function has the following prototype:

```
function(selector, context)
```

The *selector* indicates the query expression to run over the DOM; the *context* indicates the portion of the DOM from which to run the query. If no context is specified, the *jQuery* function looks for DOM elements within the entire page DOM.

The *jQuery* function typically returns a *wrapped set*—namely, a collection of DOM elements. Nicely enough, this wrapped set is still a jQuery object that can be queried using the same syntax, resulting in chained queries.

Running a Query

The word *query* in the library's name says it all (*j* stands for JavaScript)—the jQuery library is primarily designed for running (clever) queries over the DOM and executing operations over the returned items.

The query engine behind the library goes far beyond the simple search capabilities of, say, *document.getElementById* (and related functions) that you find natively in the DOM. The query capabilities of jQuery use the powerful CSS syntax, which gives you a surprising level of expressivity. You find similar query expressivity only in the DOM of HTML 5 where CSS syntax is widely and uniformly supported.

The result of a query is a *wrapped set*. A wrapped set is an object containing a collection of DOM elements. Elements are added to the collection in the order in which they appear in the original document.

A wrapped set is never null, even if no matching elements have been found. You check the actual size of the wrapped set by looking at the *length* property of the jQuery object, as shown here:

```
// Queries for all IMG tags in the page
var wrappedSet = new jQuery("img");
var length = wrappedSet.length;
if (length == 0)
    alert("No IMG tags found.");
```

Note that the expression just shown, through which you get the wrapped set, is fully equivalent to the more commonly used *$("img")*.

The wrapped set is not a special data container. "Wrapped set" is a jQuery-specific term that indicates the results of a query.

Enumerating the Content of a Wrapped Set

To loop through the elements in the wrapped set, you use the *each* function. The *each* function gets a function as a parameter and invokes that on each element:

```
// Prints out names of all images
$("img").each(function(index) {
   alert(this.src);
});
```

The callback function you pass to *each* receives the 0-based index of the current iteration. Nicely enough, you don't need to retrieve the corresponding DOM element yourself; you just use the

keyword *this* to refer to the element currently being processed. If the callback function returns *false*, the iteration is stopped. Note that *each* is a quite generic function made available for any task for which a more specific jQuery function doesn't exist. If you find a jQuery function that already does what you intend to code through *each*, by all means use the native function.

You use the *length* property to read the size of the wrapped set. You can also use the *size* function, but the *length* property is slightly faster:

```
// You better use the length property
alert($("img").size());
```

The *get* function extracts the wrapped set from the jQuery object and returns it as a JavaScript array of DOM elements. If you pass an index argument, instead, it will return the DOM element found at the specified 0-based position in the wrapped set.

```
var firstImage = $("img")[0];
```

Note that the *get* function (as well as indexers) breaks the jQuery chainability because it returns a DOM object or an array of DOM objects. You can't further apply jQuery functions to the results of a *get* call.

Many more operations are available on wrapped sets, and many others can be added through plug-ins.

Selectors

A query is characterized by a *selector*. A selector is simply the expression that, when properly evaluated, selects one or more DOM elements. In jQuery, you have three basic types of selectors—selectors based on IDs, cascading style sheets (CSS), or tag names. In addition, a selector can result from the composition of multiple simpler selectors combined using ad hoc operators. In this case, you have a compound selector.

Basic Selectors

An *ID selector* picks up DOM elements by ID. An ID selector commonly selects only one element unless multiple elements in the page share the same ID—this condition violates the HTML DOM standard, but it's not too unusual in the real world. Here's the syntax of an ID selector:

```
// Select all elements in the context whose ID is Button1
$("#Button1")
```

The leading # symbol just tells jQuery how to interpret the following text.

A *CSS-based selector* picks up all elements that share the given CSS class. The syntax is shown here:

```
// Select all elements in the context styled with the specified CSS class
$(".header")
```

In this case, the leading dot (.) symbol tells jQuery to interpret the following text as a CSS style name.

Finally, a *tag-based selector* picks up all elements with the specified tag, such as all IMG tags, all DIV tags, or whatever else you specify. In this case, the selector consists of the plain tag name—no leading symbol is required:

```
// Select all IMG elements in the context
$("img")
```

As mentioned, you can also concatenate two or more selectors to form a more specific one.

Compound Selectors

Concatenation is possible through a number of operators. For example, the white space picks up all elements that satisfy the second selector and are descendants of those matching the first. Here's an example:

```
// Select all anchors contained within a DIV
$("div a")
```

The selector just shown is functionally equivalent to the following jQuery expression:

```
$("div").find("a");
```

Similar to the white space, the > operator selects elements that are direct child elements (and not just descendants) of the elements matched by the first selector:

```
// All anchors direct child elements of a DIV
$("div > a")
```

The preceding selector is functionally equivalent to the following jQuery expression:

```
$("div").children("a")
```

Plain concatenation of selectors results in a logical AND of conditions. For example, consider the following query:

```
$("div.header.highlight")
```

It selects all DIV elements styled using both the class *header* and class *highlight*.

The + operator—the *adjacent* operator—selects sibling elements in the second selector *immediately* preceded by elements selected by the first selector. Here's an example:

```
// All P immediately preceded by A
$("a + p")
```

The ~ operator—the *next* operator—is similar to + except that it selects sibling elements just preceded by others. Here's an example:

```
// All P preceded by A
$("a ~ p")
```

By using the comma, instead, you return the union of elements queried by multiple selectors. In terms of operations, the comma represents a logical OR of selectors. The next example, in fact, picks up elements that are either A or P:

```
// All A and all P
$("a, p")
```

Beyond simple operators, you have filters. A *filter* is a jQuery-specific expression that contains some custom logic to further restrict the selected elements.

Predefined Filters

Selectors can be further refined by applying filters on position, content, attributes, and visibility. A filter is a sort of built-in function applied to the wrapped set returned by a basic selector. Table 10-1 lists positional filters in jQuery.

TABLE 10-1 Positional filters

Filter	Description
:first	Returns the first DOM element that matches
:last	Returns the last DOM element that matches
:not(selector)	Returns all DOM elements that do not match the specified selector
:even	Returns all DOM elements that occupy an even position in a 0-based indexing
:odd	Returns all DOM elements that occupy an odd position in a 0-based indexing
:eq(index)	Returns the DOM element in the wrapped set that occupies the specified 0-based position
:gt(index)	Returns all DOM elements that occupy a position in a 0-based indexing greater than the specified index
:lt(index)	Returns all DOM elements that occupy a position in a 0-based indexing less than the specified index
:header	Returns all DOM elements that are headers, such as H1, H2, and the like
:animated	Returns all DOM elements that are currently being animated via some functions in the jQuery library

Table 10-2 lists all filters through which you can select elements that are child elements of a parent element.

TABLE 10-2 Child filters

Filter	Description
:nth-child(expression)	Returns all child elements of any parent that match the given expression. The expression can be an index or a math sequence (for example, 3n+1), including standard sequences such as odd and even.
:first-child	Returns all elements that are the first child of their parent.
:last-child	Returns all elements that are the last child of their parent.
:only-child	Returns all elements that are the only child of their parent.

A particularly powerful filter is *nth-child*. It supports a number of input expressions, as shown here:

```
:nth-child(index)
:nth-child(even)
:nth-child(odd)
:nth-child(expression)
```

The first format selects the *n*th child of all HTML elements in the source selector. All child elements placed at any odd or even position in a 0-based indexing are returned if you specify the *odd* or *even* filter instead.

Finally, you can pass the *nth-child* filter a mathematical sequence expression, such as *3n* to indicate all elements in a position that are a multiple of 3. The following selector picks up all rows in a table (labeled *Table1*) that are at the positions determined by the sequence 3n+1—that is, 1, 4, 7, and so forth:

```
#Table1 tr:nth-child(3n+1)
```

Table 10-3 lists expressions used to filter elements by content.

TABLE 10-3 Content filters

Filter	Description
:contains(text)	Returns all elements that contain the specified text
:empty	Returns all elements with no children
:has(selector)	Returns all elements that contain at least one element that matches the given selector
:parent	Returns all elements that have at least one child

As far as content filters are concerned, you should note that any text in an HTML element is considered a child node. So elements selected by the *empty* filter have no child nodes and no text as well. An example is the <*br* /> tag.

A popular and powerful category of filters are attribute filters. Attribute filters allow you to select HTML elements where a given attribute is in a given relationship with a value. Table 10-4 lists all attribute filters supported in jQuery.

TABLE 10-4 Attribute filters

Filter	Description
[attribute]	Returns all elements that have the specified attribute. This filter selects the element regardless of the attribute's value.
[attribute = value]	Returns all elements where the specified attribute is set to the specified value.
[attribute != value]	Returns all elements whose specified attribute has a value different from the given one.
[attribute ^= value]	Returns all elements whose specified attribute has content that starts with the given value.
[attribute $= value]	Returns all elements whose specified attribute has content that ends with the given value.
*[attribute *= value]*	Returns all elements whose specified attribute has content that contains the given value.

Attribute filters can also be concatenated by simply placing two or more of them side by side, as in the following example:

```
var elems = $("td[align=right][valign=top]");
```

The returned set includes all <*td*> elements where the horizontal alignment is *right* and the vertical alignment is *top*.

The next expression, which is much more sophisticated, demonstrates the power and flexibility of jQuery selectors because it combines quite a few of them:

```
#Table1 tr:nth-child(3n+1):has(td[align=right]) td:odd
```

It reads as follows:

> *Within the body of element* Table1, *select all* <tr> *elements at positions 1, 4, 7, and so forth. Next, you keep only table rows where a <td> element exists with the attribute* align *equal to the value of* right. *Furthermore, of the remaining rows, you take only the cells on columns with an odd index.*

The result is a wrapped set made of <*td*> elements.

Finally, a couple more filters exist that are related to the visibility of elements. The *:visible* filter returns all elements that are currently visible. The *:hidden* filter returns all elements that are currently hidden from view. The wrapped set also includes all input elements of whose *type* attribute equals "*hidden*."

Filter vs. Find

To further restrict a query, you can use either the *find* or *filter* function on a wrapped set. They are not the same, of course.

The function *filter* explores the current wrapped set for matching elements and doesn't ever look into DOM for descendants. The function *find*, instead, looks inside of each of the elements in the wrapped set for elements that match the expression. In doing so, however, the function explores the DOM of each element in the wrapped set.

Chaining Operations on a Wrapped Set

The jQuery library offers a wide range of functions you can apply to the content of a wrapped set. (For a complete list, you can only resort to online documentation or some in-depth book.) Function calls can be chained because any wrapped set returned by a query is, in turn, another jQuery object that can be further queried. The following expression, for example, works just fine:

```
$(selector).hide().addClass("hiddenElement");
```

It first hides from view all matching elements and then adds a specific CSS class to each of them.

Operations you can perform on wrapped sets can be classified in a few groups, as described in Table 10-5.

TABLE 10-5 Operations on a wrapped set

Effect	Description
DOM manipulation	Creates DOM trees, adds/removes elements, or modifies existing elements
Event binding	Binds and unbinds handlers to events fired by DOM elements
Styling	Applies, removes, or toggles CSS classes to selected elements and gets or sets individual CSS properties.
Visibility	Shows and hides DOM elements using transition effects (for example, fading) and duration.

In addition, in jQuery you find two other groups of functionalities—cache and Ajax calls—that work with the content of wrapped sets, though they can't be strictly considered operations available on wrapped sets.

Events

Handling events is a common activity in JavaScript programming. The jQuery library provides a bunch of functions to bind and unbind handlers to events fired by DOM elements.

Binding and Unbinding

The *bind* and *unbind* pair of functions are used to attach a callback function to the specified event. Here's an example in which all elements that match the selector will have the same handler attached for the *click* event:

```
$(selector).bind("click", function() {
   ...
});
```

You use the *unbind* function to detach any currently defined handler for the specified event:

```
$(selector).unbind("click");
```

Note that the *unbind* function doesn't remove handlers that have been inserted directly in the markup through any of the *onXXX* attributes.

The jQuery library also defines a number of direct functions to bind specific events. Facilities exist for events such as *click*, *change*, *blur*, *focus*, *dblclick*, *keyup*, and so forth. The following code shows how to bind a handler for the *click* event:

```
$(selector).click(function() {
   ...
});
```

Invoked without a callback, the same event functions produce the effect of invoking the current handler, if any are registered. The following code, for example, simulates the user's clicking on a specific button:

```
$("#Button1").click();
```

You can achieve the same effect in a more generic way using the *trigger* function:

```
$("#Button1").trigger("click");
```

Event handlers receive a jQuery internal object—the *Event* object. This object provides a unified programming interface for events that goes hand in hand with the World Wide Web Consortium (W3C) recommendation, and it resolves discrepancies in the slightly different implementations provided by some browsers:

```
$("#Button1").click(function(evt) {
    // Access information about the event
    ...

    // Return false if you intend to stop propagation
    return false;
});
```

The *Event* object features properties such as mouse coordinates, the JavaScript time of the event, which mouse button was used, and the target element of the event.

Live Event Binding

Live binding is a nice feature of jQuery that allows you to keep track of event bindings for a given subset of DOM elements for the entire page lifetime. In other words, if you opt for live binding instead of plain binding, you are guaranteed that any new dynamically added elements that match the selector will automatically have the same handlers attached. You operate live binding through *live* and *die* functions. Here's an example:

```
$(".specialButton").live("click", function() {
   ...
})
```

All buttons decorated with the *specialButton* CSS style have the given function attached as the handler for the *click* event. To stop live binding for some elements, you use the *die* function:

```
$(".specialButton").die("click");
```

The difference between using *live* and *bind* (or specific event functions such as *click*) is that when function *live* is used, any new DOM elements added to the page and decorated with the *specialButton* style automatically have the handler added. This won't happen if *bind* is used.

Page and DOM Readiness

In the beginning of client-side development, there was just one place where you could put the initialization code of a web page—in the *onload* event on either the *window* object or the <*body*> tag. The *onload* event fires as soon as the page has finished loading—that is, after the download of all linked images, CSS styles, and scripts has terminated. There's no guarantee, however, that at this time the DOM has been fully initialized and is ready to accept instructions.

The document root object in the DOM exposes a read-only *readyState* property just to let you know the current state of the DOM and figure out when it's OK for your page to start scripting it. Using the *readyState* property is an approach that definitely works, but it's a bit cumbersome. For this reason, jQuery offers its own *ready* event that signals when you can start making calls into the framework safely.

```
<script type="text/javascript">
$(document).ready(
   function() {
     alert("I'm ready!");
   });
</script>
```

You can have multiple calls to *ready* in a page or view. When multiple calls to *ready* are specified, jQuery pushes specified functions to an internal stack and serves them sequentially after the DOM is effectively ready.

The ready event is fired only at the document level; you can't have it defined for individual elements or any collection of elements in a wrapped set.

Note The *onload* event is called after the HTML and any auxiliary resources are loaded. The *ready* function is called after the DOM is initialized. The two events can run in any order. The *onload* event won't ensure the page DOM is loaded; the *ready* function won't ensure all resources, such as images, have been loaded.

Aspects of JavaScript Programming

Today, you often use JavaScript for some client-side logic and input validation. You use JavaScript to download data from remote servers, to implement Windows-like effects such as drag-and-drop, for resizing, for templates, for popup and graphic effects, for local data caching, and to manage history and events around the page. It's used for large chunks of code that have a good level of reusability and need to be safely isolated from one another.

In other words, you want your JavaScript code to be maintainable and unobtrusive.

Unobtrusive Code

For years, it has been common to write HTML pages with client buttons explicitly attached to JavaScript event handlers. Here's a typical example:

```
<input type="button" value="Click me" onclick="handleClick()" />
```

From a purely functional perspective, there's nothing wrong with this code—it just works as expected and runs the *handleClick* JavaScript function whenever the user clicks the button. This approach is largely acceptable when JavaScript is just used to spice up web pages; however, it

becomes unwieldy when the amount of JavaScript code represents a significant portion of the page or the view.

Style the View with Code

The expression "unobtrusive JavaScript" is popular these days, and it just means that it would be desirable not to have explicit links between HTML elements and JavaScript code. In a way, unobtrusive JavaScript is the script counterpart of CSS classes.

With CSS, you write plain HTML without inline style information and add style to elements using CSS classes. Likewise, you avoid using event handler attributes (*onclick*, *onchange*, *onblur,* and the like) and use a single JavaScript function to attach handlers when the DOM is ready. Here's a concise but effective example of unobtrusive JavaScript:

```
<script type="text/javascript">
    $(document).ready(function () {
        $("#Button1").bind("click", function () {
            var date = new Date().toDateString();
            alert(date);
        });
    });
</script>

<h2>JavaScript Patterns</h2>
<fieldset>
   <legend>#1 :: Click</legend>
   <input type="button" id="Button1" value="Today" />
</fieldset>
```

You can move the entire <*script*> block to a separate JavaScript file and have your view be clean and readable.

Pragmatic Rules of Unobtrusive JavaScript

Unobtrusive JavaScript establishes a fundamental principle—any behavior in any web page has to be an injectable dependency and not a building block. There will be more and more situations in which you need to download or configure the script code on the fly based on run-time conditions, such as the capabilities of the browser (or mobile device). To address these scenarios properly, start thinking about JavaScript as an injectable dependency.

Likewise, the script code should not make assumptions about the structure of the DOM and the browser capabilities. You should avoid making direct access to HTML elements, members of DOM elements, and browser objects. You should check for any of these before you access them. Suppose you replace the previous *click* handler with the following code. Beyond the details of the sample code, the key difference is that now the handler is accessing the DOM—directly.

```
$("#Button1").bind("click", function () {
     var date = new Date().toDateString();
     $("#Message").html(date);
});
```

I have conflicting feelings about this code. One is a positive feeling because it takes me exactly where I want, quickly and without effort. For the same reason, however, I also have a negative feeling. When code gets complex, or simply large, you need to exercise some forethought.

This code is logically equivalent to setting server controls from the *Button1_Click* handler of an ASP.NET Web Forms page. It does not have the benefits of neatly separating views from the presentation logic (controllers)—one of the key changes introduced by ASP.NET MVC. And there is a second reason I don't like this code.

The other reason is that it uses anonymous functions. As I see things, anonymous functions (in any language) are plain shortcuts—yes, much like old-fashioned function pointers of 20 years ago. They can be a great thing when you need a shortcut or where a shortcut is applicable. However, they can't become a wide-ranging implementation facility. If so, your code loses isolation, modularity, and readability. In one word, they cause your code to lose *maintainability*. And I regard limited maintainability to be the primary deadly sin of modern software. Let's see how you can possibly rewrite the previous code.

Important As you saw in Chapter 4, "Input Forms," ASP.NET MVC 3 makes a point of enabling unobtrusive JavaScript in its Ajax HTML helpers. It couldn't be simpler—just enable a Boolean flag and the framework will do the rest: no more messy script tags with endless function calls. Instead, you have clean attributes interpreted by external scripts. Well done! But this is just about making one small area of your code JavaScript-unobtrusive.

Unobtrusive JavaScript is a wider and deeper concept. It aims at getting a neat separation between content (HTML document), presentation (CSS style sheets), and behavior (script code). Ideally, a page should come back to a plain sequence of document tags—headers, *P*, and maybe the new tags coming up in HTML 5 such as *section*, *footer*, and *article*. The page should be able to present its content in a readable way as plain HTML. Next, you add CSS and the choice of different CSS styles on, say, a per-device basis. Next up, you add JavaScript. Both CSS and JavaScript should be linked in a single place, and it should be easy to switch them on and off.

As a software developer, you probably use forms of dependency injection every day. Well, just think of CSS and JavaScript code as dependencies you inject into an HTML page.

Using jQuery Plug-ins to Increase Unobtrusiveness

Imagine you have a page with a button; upon clicking, you trigger some local or remote action and then want to refresh some portions of the user interface. Because it seems like a simple scenario, you code it with anonymous functions and access to DOM elements performed from within event handlers. At some point, you need to call the code that refreshes the user interface from other handlers. What should you do? Duplicate the code? Write a more or less reusable function? Any approach that

produces the desired effect is ultimately a solution. My suggestion here is to use an approach that scales well with the complexity and quantity of events and updates. This approach also forces you to keep script code isolated in self-contained blocks.

The idea is that you write a global jQuery plug-in that knows how to refresh the content of an updatable view. The plug-in is made of two methods: one for registering the update helper and one to actually invoke the update helper.

```
<script src="@Url.Content("~/Scripts/AppPlugins/updater.js")"></script>
<script src="@Url.Content("~/Scripts/AppPlugins/updatehelpers.js")"></script>

<script type="text/javascript">
    function handlerForButton1() {
        var today = new Date().toDateString();
        var data = { Message: today }
        $.updater().refresh("ResultArea", data);
    }
</script>

<script type="text/javascript">
    $(document).ready(function () {
        $.updater().setup(("ResultArea", UpdateHelpers.ResultAreaUpdater);
        $("#Button1").bind("click", handlerForButton1);
    });
</script>

<h2>Unobtrusive #3</h2>
<input type="button" id="Button1" value="Today" />
<hr />
<div id="ResultArea">
    <p id="Message"></p>
</div>
```

In the *ready* event, you call *updater*—the jQuery plug-in—on the *DIV* tag that receives updates. In the call to the plug-in, you pass the reference to a global JavaScript function that only knows how to update a particular DIV. When it's time to refresh the *DIV*, you call the *refresh* method on the plug-in, which takes data and internally invokes the update helper. Here's the update helper:

```
// Global container of update helpers
var UpdateHelpers = UpdateHelpers || {};

// Updates the ResultArea panel
UpdateHelpers.ResultAreaUpdater = function (data) {
    // Prefix #ResultArea ensures only child elements are retrieved.
    $("#ResultArea #Message").html(data.Message);
};
```

The apparently weird syntax used to define the global container deserves a further comment. If you move functions to a separate script file, they belong to the global scope. Grouping them in a super container that you control is a good idea because, at a minimum, it makes globals stand out. *UpdateHelpers* is such a global container. Where do you define and initialize it? It can happen in a file

you import every time. In this case, however, you charge the browser with yet another download. The syntax

```
var UpdateHelpers = UpdateHelpers || {};
```

is equivalent to the following:

```
if (typeof(UpdateHelpers) === "undefined")
     var UpdateHelpers = {};
```

And you can safely place it multiple times at the beginning of each file where you declare globals.

Finally, here's the generic *updater* plug-in:

```
(function ($) {
    var _registeredElements = [];
    $.updater = function () {
        _methods = {
            setup: _setup,
            refresh: _refresh
        };
        return _methods;
    };

    var _setup = function (elementId, methodName) {
        _registeredElements[elementId] = methodName;
        return this;
    };

    var _refresh = function (elementId, data) {
        var func = _registeredElements[elementId];
        if (func != null)
            func(data);
        return this;
    };
})(jQuery);
```

Applying this pattern probably involves too much work for simple views with just one button or two. It becomes a huge help when the richness of views and need of interactivity grows.

Reusable Packages and Dependencies

The solution discussed is clearly jQuery-oriented and benefits from the facilities built around the library. Bringing jQuery into an ASP.NET MVC project is not problematic, but it still represents a constraint and, especially in some corporate environments, it might be a constraint that's hard to overcome. Let's explore another approach for packaging code in JavaScript that has no dependencies on external libraries.

The *Namespace* Pattern

A golden rule of JavaScript programming is grouping related properties—including globals—into containers. When such containers are shared among multiple script files, it might be hard to decide (and enforce) which file has the responsibility of initializing containers and child objects. You just saw an example of a simple syntax that can be used to define a global container.

```
var GLOBALS = GLOBALS || {};
```

This trick works, but it's often too cumbersome to use when you have several nested objects to deal with. The *Namespace pattern* comes to the rescue.

The Namespace pattern consists of a piece of code that iterates over the tokens of a dot-separated string (for example, a C# or Java namespace) and ensures that the proper hierarchy of objects is initialized. The *namespace* function ensures the code is not being destructive and skips over existing instances. Here's some sample code:

```
var GLOBALS = GLOBALS || {};

GLOBALS.namespace = function (ns) {
    var objects = ns.split("."),
            parent = GLOBALS,
            startIndex = 0,
            i;

    // You have one GLOBALS object per app. This object already exists if you
    // can call this function. So you can safely ignore the root of the namespace
    // if it matches the parent string.
    if (objects[0] === "GLOBALS")
        startIndex = 1;

    // Create missing objects in the namespace string
    for (i = startIndex; i < objects.length; i++) {
        var name = objects[startIndex];
        if (typeof parent[name] === "undefined")
            parent[name] = {};
        parent = parent[name];
    }
    return parent;
};
```

After you have referenced the *namespace* function, you can then place the following calls:

```
GLOBALS.namespace("Modules");                    // GLOBALS has a Modules property
GLOBALS.namespace("GLOBALS.Modules");            // GLOBALS has a Modules property
```

These two calls are equivalent, and both guarantee that GLOBALS has an initialized *Modules* property. Consider the following:

```
// GLOBALS has a Modules property, and Modules has an Updaters property
GLOBALS.namespace("GLOBALS.Modules.Updaters");
```

The *namespace* function proceeds iteratively and also ensures that *Modules* has an *Updaters* property.

Important Although the implementation shown here can be considered relatively standard, I feel obliged to credit Stoyan Stefanov for inspiring this code and the section on the Module pattern. Stoyan is the author of the excellent *JavaScript Patterns* (O'Reilly Media, 2010). I recommend this book to anybody who wants to go beyond the content of this chapter.

The *Module* Pattern

The *Module pattern* provides a way to package self-contained blocks of code that can be simply added or removed from a project. The pattern wraps a classic JavaScript function into an immediate function that guarantees privacy of data and ensures that only what you explicitly reveal as public is actually perceived as public by clients. In JavaScript, an immediate function is a function defined inline and immediately executed. The syntax is shown here:

```
(
   function(...) {
     // Body
   } (...)
);
```

Let's rewrite the previous example with the Module pattern:

```
<script type="text/javascript">
    GLOBALS.namespace("Updaters");

    GLOBALS.Updaters.ResultAreaUpdater = (function (updateHelper) {
        var _updater = updateHelper,
            _refresh = function (data) {
                if (_updater != null)
                    _updater(data);
            };

        return { Refresh: _refresh };
    } (GLOBALS.Helpers.ResultAreaUpdateHelper));
</script>
```

The code defines an updater block for a certain area of the view. The updater module is a function that takes a helper function that provides the logic for updating the DOM. The update helper is stored internally as a dependency and used by the *_refresh* internal function. Up to here, it's nearly the same as having a closure.

The difference is that in the Module pattern you return an object that reveals the public API you want to make visible. In particular, the immediate function that surrounds the definition executes

the function and assigns the variable an object with a single member—*Refresh*. Anything else is completely hidden from view to outsiders. Here's how you use a module:

```
<script type="text/javascript">
    function handlerForButton1() {
        var today = new Date().toDateString();
        var data = { Message: today };
        GLOBALS.Updaters.ResultAreaUpdater.Refresh(data);
    }
</script>
```

The Namespace pattern is not necessary for the implementation of the Module pattern, but it helps a lot to have it.

Script and Resource Loading

More and more script in web pages means more and more script files to download. This might soon become a serious issue and needs to be addressed. When a page has several scripts, the degree of parallelism at which the browser can operate is dramatically lowered. So it is for the load time of the page. Let's see why.

The Download Is Always Synchronous

The HTTP/1.1 specification suggests that browsers download no more than two components in parallel per host name. However, that never happens for script files: browsers always download script files synchronously and one at a time. As a result, the total download time is at least the sum of times required to download individual files and, maybe worse, the browser is idle while downloading a script file. Page rendering resumes only after the script files have been downloaded, parsed, and executed.

Browsers implement synchronous downloads mostly to stay on the safe side. In fact, there's always the possibility that script files include instructions such as JavaScript immediate functions or *document.write* that could modify the status of the current DOM.

Scripts at the Bottom

To improve the page-loading performance, you can use a simple trick that consists of moving all script files to the bottom of the page just before the *</body>* tag. When you do this, browsers don't need to interrupt the page rendering process to load scripts. Browsers can then do their best to display an early view of the page.

Although placing manual scripts at the bottom is the safest approach, another option exists that you can set up declaratively and without resorting to writing or importing ad hoc JavaScript code—the *defer* attribute.

```
<script src="..." defer="defer"></script>
```

Introduced with the HTML 4 specification, the *defer* attribute tells the browser whether or not loading the script can be deferred to the end of the page processing. A <*script*> tag decorated with the *defer* attribute implicitly states that it's not doing any direct document writing and it's safe for it to be loaded at the end. The purpose of the *defer* attribute is similar to the *async* attribute you find in the HTML 5 specification. (Currently, only a few browsers support this feature. One is Firefox, which has done so since version 3.6.)

So what's the issue with the *defer* attribute? It's not the attribute itself—it's how browsers support it. When regular and deferred scripts exist, the behavior of browsers is not identical. In practice, the order in which scripts are processed is often erratic and not reproducible. This lack of consistency might not be a problem for your application, but in theory it can create edge conditions that are hard to test and debug. An interesting reading on the topic is the post at *http://hacks.mozilla.org/2009/06/defer*.

Nonblocking Scripts Download

The synchronous download is a behavior that browsers associate only with markup <*script*> tags. To download a script in a nonblocking way, all you need to do is create a <*script*> element and add it to the DOM on the fly:

```
var script = document.createElement("script");
script.src = ...;
document.appendChild(script);
```

Again, this is still far from being a solution that can be largely applied. It works well with standalone files with no dependencies. When the order of download is important, this solution is inadequate and needs more logic to work well.

Two libraries are extremely popular as far as improving the script downloading is concerned. They are LABjs (available at *http://www.labjs.com*) and RequireJS (available at *http://www.requirejs.org*). Both libraries allow the loading of scripts (and other resources) in parallel and take into account declared dependencies between files. Here's a code snippet that shows how to use RequireJS:

```
<script src="scripts/require.js"></script>
<script>
require(["namespace.js", "updaters.js", "misc.js"],
    function() {
       // This optional callback is invoked when all files
       // listed above are loaded.
    });
</script>
```

Libraries start downloading scripts in the order in which files are listed. The order in which files are processed, however, is unpredictable and depends on the actual download time and latency. You can, however, indicate dependencies between files. You do that through an additional plug-in—the *order* plug-in:

```
<script>
   require(["order!namespace.js", "order!updaters.js", "misc.js"]);
</script>
```

Scripts whose name is prefixed by *order!* are loaded asynchronously as usual, but they're evaluated in the order in which they are listed in the source code. With reference to the code snippet, it means that *namespace.js* is always evaluated before *updaters.js*, regardless of which one downloads first. No assumptions can be made, however, about *misc.js*.

Dealing with Static Files

A golden rule of web development states that after you've neutralized the performance hit of static files such as scripts, style sheets, and images, you're pretty much done with optimization. There are two main ways to minimize the download time of static resources, and one doesn't exclude the other.

The most obvious trick is reducing the size of the files being downloaded. The second most obvious trick consists of not downloading them at all. Before I get into the details, let me state up front that these optimization tricks should be played when you're done with development. If they're applied at development time, most of them only add frustration and slow down your progress.

Reducing the size of downloadable files means compressing their content. Browsers inform the web server about the type of compression they support through the *Accept-Encoding* header. As you saw in Chapter 8, "Customizing ASP.NET MVC Controllers," in ASP.NET MVC you can add a proper response header to any controller action and instruct the web server to use any of the supported encodings for the response. Compression can also be enabled on static resources directly at the web-server level. It should be noted that GZIP compression (or perhaps the *deflate* compression) should not be used on files (and responses) that are already compressed on their own. For example, you don't want to use GZIP for a JPEG image. Although GZIP compression can cut the size by 50 percent or so, it gives you much less impressive results on already compressed sources and at the cost of more CPU work.

For script files, GZIP compression can be combined with *minification*. Script files are rich with white spaces and blanks. Removing these characters can cut a significant percentage of text out of the file. A *minifier* parses a script file and rewrites it in a way that is functionally equivalent to the original but devoid of any unnecessary characters. Clearly, a minified file is designed for use by computers only and can't be processed by humans.

In addition, scripts and style sheets can be combined in a smaller number of files, which reduces the number of requests the browser has to process. This is an important point for any web application, but it's becoming especially important for web applications that expected to have a mobile audience also.

The second aspect to consider is browser caching. Static resources are said to be static just because they don't change frequently. So why should you download them over and over again? By assigning your static resources a very long duration (through the *Expires* response header), you save the browser from downloading these resources frequently. Again, you can do that at the web-server level for static resources and programmatically via the *Response* object for dynamically served resources.

In this regard, a content delivery network (CDN) is beneficial because it increases the likelihood that the browser cache already contains a resource that might have been referenced using the same

URL by other sites using the same CDN. Note that you won't benefit much from placing on a CDN files that only one application uses.

Using Sprites

To improve the serving of images, you can consider using *sprites*. A sprite is a single image that results from the composition of multiple images. Constituent images are saved side by side in the process of forming a new image, as shown in Figure 10-1.

FIGURE 10-1 A composed image served as a sprite.

In pages, you use the <*img*> tag to reference the total image and resort to using a specific CSS style to select the portion of it you're interested in. For example, here's the CSS you need to use for the segments of the image just shown to render a clickable button:

```
.signup-link
{
    width: 175px;
    height: 56px;
    background-image: url(/images/loginsprite.png);
    background-position: -0px 0px;
    background-repeat: no-repeat;
}

.signup-link:hover
{
    background-position: -177px 0px;
}

.signup-link:active
{
    background-position: -354px 0px;
}
```

The *background-position* indicates the relative position at which to start rendering the image. For example, when the mouse hovers over the image, the browsers begins rendering from pixel 177 for the specified width. (As you can see in Figure 10-1, for clarity, I added a blank line of pixels, which is why you have to skip one pixel when counting positions.) The net effect is that you have just one image, just one download, a cached image, and a cool effect for users. (See Figure 10-2.)

FIGURE 10-2 Sprites in action.

ASP.NET MVC, Ajax and JavaScript

Like it or not, an Ajax-intensive application requires a lot of JavaScript programming. Whether you use a rich dialect such as jQuery or write your own library, you need to exercise programming discipline to nearly the same extent you do in your server development. With Ajax, on the other hand, you write a share of your presentation layer with JavaScript. Scattered JavaScript code is a thing of the past. It might still work; you might still be able to take the project home, but that's not the way to go.

I like to summarize Ajax development in two main points:

- You need JavaScript, so learn how to best organize your JavaScript code.
- Give your Ajax applications an Ajax-specific service layer.

We already refreshed our JavaScript skills and discussed patterns and practices to better organize your script code. So let's attack the second point of the previous list.

The Ajax Service Layer

In software architecture, the Service Layer pattern is defined as "a layer of services that establishes a set of available operations and coordinates the application's response in each operation." The definition is adapted from Martin Fowler's excellent book *Patterns of Enterprise Application Architecture* (Addison-Wesley, 2002). The service layer is the next stop on the way to the back end of an application after a user action. The worker services I defined in Chapter 7, "Design Considerations for ASP.NET MVC Controllers," are a possible implementation of the Service Layer pattern.

A Tailor-Made HTTP Façade

The main point I want to emphasize is that, in an Ajax context, the service layer is a collection of HTTP endpoints that works out a convenient API for the presentation layer to call. The JavaScript presentation layer doesn't directly call into middle-tier components. It doesn't set up a potentially long conversation over the wire with the back end. Instead, it opts for a chunky (as opposed to chatty) model, and it places one call to a remote HTTP endpoint that orchestrates calls and returns data ready for display. This is mostly a performance reason for introducing a façade. But there's more to it than that.

The architectural relevance of such an HTTP façade lies in the fact that it decouples the middle tier from a very special presentation layer, such as an Ajax presentation layer. An Ajax presentation layer is special essentially because it's a partial-trust web client.

For security reasons, service technologies hosted on the web server require special adjustments to enable JavaScript callers. In addition, it's likely that some of the application services you have in the middle tier run critical procedures. Any piece of code bound to a URL in the HTTP façade, instead, is publicly exposed over the Internet—not an ideal situation for a business-critical service. So decoupling application services from the Ajax presentation layer is a measure of design, but it's also a matter of security. (See Figure 10-3.)

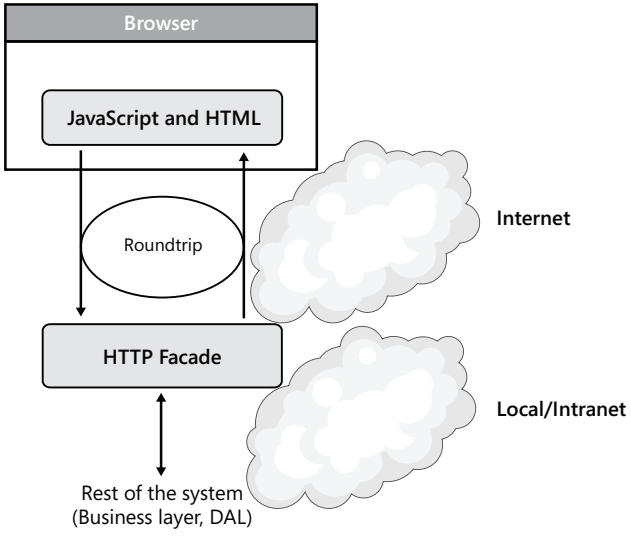

FIGURE 10-3 A typical Ajax architecture.

How would you build the HTTP façade?

The façade contains URLs that are callable from clicks and taps by the end user. An Ajax request doesn't likely need an entire or even partial HTML view; it probably just needs a bunch of data, presumably JSON. You need to have these endpoints ready-made in an Ajax application.

ASP.NET MVC Controllers vs. Web Services

An Ajax client can invoke any public HTTP endpoint as long as it's hosted on the same domain as the originating page. You can write endpoints callable by Ajax clients using a number of technologies. As far as the ASP.NET platform is concerned, you can use ASMX Web Services, Windows Communication Foundation (WCF) services, and plain HTTP handlers. You can even use ISAPI modules, PHP resources, or Java Server Pages. It doesn't really matter—in any case, the Ajax client will get whatever the service returns.

As an ASP.NET MVC developer, however, you can blissfully ignore this discussion—you use a made-to-measure ASP.NET MVC controller. As you saw in past chapters, a controller can have action methods that return JSON data. These action methods can be bound to a route and invoked via a URL that you, as a developer, can control at will. How could you ask for more?

In the context of ASP.NET MVC applications, you should not consider WCF services as an option as long as the Ajax service layer is concerned. Or, put another way, you don't need WCF services to arrange an Ajax front end in an ASP.NET MVC application. WCF is a rich framework with tremendous power and startup costs. WCF is powerful because it can work on top of a number of protocols and bindings—you just need HTTP. Instead of cutting down WCF to operate effectively in an Ajax context, just use ASP.NET MVC controllers—you'll get the same performance at a much lower cost.

ASP.NET MVC is, in this regard, different from ASP.NET Web Forms. In Web Forms, you don't have an easy and quick way to make a call over the wire from the client. Web Forms natural endpoints are pages, and pages return HTML. You can force pages to return JSON, or you can employ custom HTTP handlers—the implementation of JSON (de)serialization is on your shoulders.

Avoid Exposing WCF Services to Ajax Clients

I recommend you avoid using WCF services to expose Ajax endpoints. Although WCF services definitely work for this if you use them, you set up a more complex infrastructure than you really need to achieve your goals. This doesn't mean you don't need WCF services altogether.

WCF services are ideal for designing parts of the architecture: they isolate volatile portions of the system, and they're perfect at protecting critical areas. WCF services offer a number of capabilities, including transactionality, queuing, and multiple protocol support. To me, WCF seems like a powerful tool to use to architect the back end. For Ajax clients, I definitely prefer to have a thin and flexible layer in between the browser and, perhaps, back end WCF services.

Ways to Write Ajax ASP.NET MVC Applications

Although the definition of an Ajax application is well established—an application that provides updates of its views in an incremental way—no common agreement exists on the tools. Every web platform has its own tools and facilities. In addition, you can always get a commercial framework that gives you a rich user interface and that shields you from the details of ever coding Ajax manually.

As far as ASP.NET MVC is concerned, you have two main routes to choose from. I introduced one of these options in Chapter 4 while discussing input forms. You get Ajax capabilities by essentially switching to a different implementation of HTML links and forms. The other option entails that you write a lot of JavaScript yourself.

The two options refers to two distinct Ajax patterns: the HTML Message pattern and the Browser Template pattern. The former exchanges HTML messages between the browser and server; the latter is based on JSON data contracts. The winner—let me say it up front—is not so obvious.

Using Ajax Helpers

Ultimately, what are the most common operations for which a browser page interacts with the web server? The user can follow a link or click to submit a form. Beyond that, a third scenario is when the page contains script code that handles page events. When these events are triggered (for example, when the user does a drag-and-drop, changes a selection, or types text), the script code might start a remote Ajax request.

As you saw in Chapter 4, ASP.NET MVC offers two HTML helpers for emitting Ajax-aware action links and forms. In Chapter 4, you saw plenty of examples related to forms, so let's use action links here. Figure 10-4 shows the sample application I use to demonstrate Ajax capabilities. In the companion source code for this book, you'll find the same sample coded using both the HTML Message pattern and the Browser Template pattern.

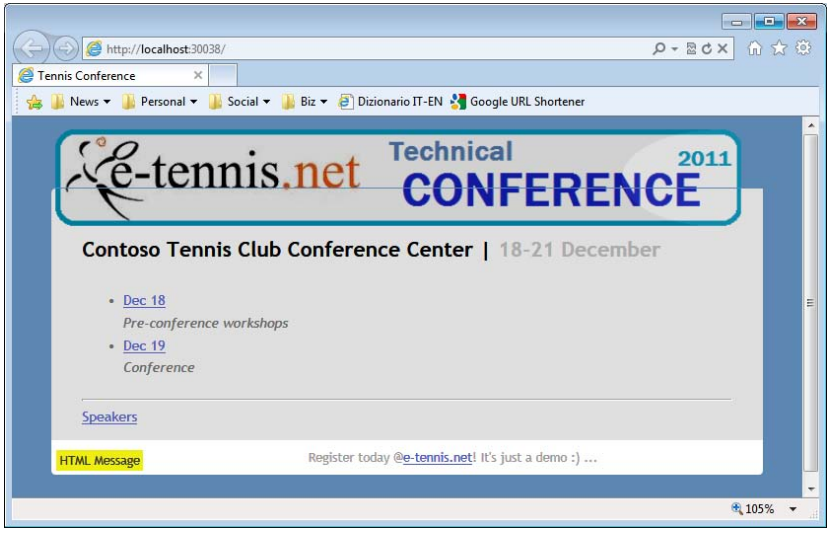

FIGURE 10-4 The Conference sample application.

The main view contains a list of action links that provide the conference schedule for the selected day. Here's an excerpt from the source behind the view:

```
@foreach(var day in Model.Days)
{
    <li>
        @Ajax.ActionLink(day.Date,
                         "day",
                         "schedule",
                         new { id = index++ },
                         new AjaxOptions { UpdateTargetId = "schedulePlaceholder" })
        <br />
        <i>@day.Description</i>
    </li>
}
```

The *ActionLink* helper gets the link text, action name, and controller name, respectively, plus route values to compose the URL. It emits a regular *<a>* tag; the only difference with *Html.ActionLink* is that it captures the *click* event on the link and places the request via Ajax. The code that captures the click event can be unobtrusive or not; it's unobtrusive in the default configuration of the standard ASP.NET MVC Visual Studio project:

```
<a data-ajax="true" data-ajax-mode="replace" data-ajax-update="#schedulePlaceholder"
   href="/schedule/day/0">Dec 18</a>
```

This means you also need to include the following file to really have some Ajax behavior. (If you forget to include the script file, you simply get regular postbacks.)

```
script src="@Url.Content("~/Scripts/jquery.unobtrusive-ajax.min.js")"
       type="text/javascript"></script>
```

The action link invokes the specified method on the controller. If the controller returns a partial view (for example, just delta to the current view without layout), you have a standard Ajax behavior without altering the structure of your application that much.

```
public ActionResult Day(Int32 id)
{
    var model = _service.GetContentForDay(id);
    return PartialView(model);  // instead of View(model)
}
```

The only key change to apply is in the views of your application, which will become, for the most part, partial views (for example, user controls).

Intricacies of Ajax Helpers

This approach is definitely the fastest to develop. Is it effective too? The potential issue here is that the browser and web server are exchanging HTML messages—that is, good chunks of HTML markup. The markup contains whatever you have put in it—including inline style information and inline script code if you have some. (You shouldn't, however; having inline code is debatable, but having style information is just a bad practice.)

Before you jump to the conclusion that using Ajax helpers is a bad thing, think about how much HTML you download in non-Ajax applications. Any implementation of the HTML Message pattern is much faster than non-Ajax solutions; it's also much faster than partial rendering in ASP.NET Web Forms. The implementation of the HTML Message pattern you find in ASP.NET MVC is nearly the best possible implementation of it. By combining this with GZIP compression and output caching, your requests are really served quickly enough.

My personal perspective is that you shouldn't drop Ajax helpers until you have clear evidence that the HTML being returned is too large for the purpose. I think there's just one common scenario where you want to look elsewhere—when you have subviews to update that are based on rich, complex, but relatively static HTML templates. If all that changes is the data displayed (and the HTML is complex), you might want to look to transform your controller into a JSON data provider.

For input forms, however, I wouldn't consider using anything other than Ajax helpers.

Using JSON Data Providers

The idea is simple: you call an HTTP endpoint, get the data you need, and refresh the user interface with some ad hoc script code. Let's proceed step by step.

Calling the HTTP endpoint and getting the data you need are not really an issue. If you use jQuery, it reduces to the following code:

```
var url = "ajax/day/1";
$.getJSON(url, function (data) {
    // Process downloaded data
    $("#Description").html(data.CurrentDay.Description);
    ...
});
```

The JSON data the controller serves is often just modeled after some server-side data type you already have. In particular, you can serve your view model objects to the client script and be able to reuse JavaScript objects with the same structure. Here's an excerpt from the Ajax controller used in the sample conference application:

```
public class AjaxController : Controller
{
    private readonly IScheduleServices _service;

    public AjaxController() : this(new ScheduleServices())
    {
    }
    public AjaxController(IScheduleServices service)
    {
        _service = service;
    }

    [AjaxOnly]
```

```
    public ActionResult Day(Int32 id)
    {
        var model = _service.GetContentForDay(id);
        return Json(model, JsonRequestBehavior.AllowGet);
    }
    ...
}
```

As you can see, bringing any data you need down to the client is relatively easy and, more importantly, it doesn't have a significant impact on your MVC programming habits. The real pain comes with the rendering of the view.

In a full Ajax application, you have just one page for each significant functionality. This page, however, might have different subviews to show at different times. Using the HTML Message pattern, with the view template being handled on the server, you can even have a single placeholder in the page where you append downloaded responses. In a full Ajax solution, on the other hand, all the subview templates are on the client and switching between them is up to your script code.

Another aspect to consider is that all of your links should point to JavaScript code. (If you use *Ajax.ActionLink* to generate links, you still need to add a JavaScript *OnSuccess* callback.) This adds up to more and more pieces of script code.

Organizing JavaScript Code (and Reorganizing Views)

Overall, the biggest issue is organizing the view templates and their containers and arranging some script code to manage those views. I divide this JavaScript code into two categories: updaters and controllers. An *updater* is a plain piece of JavaScript code that updates a specific portion of the user interface. You typically need an updater for each changing segment of your client user interface. A *controller* is the JavaScript component that manages a group of related views. An updater is mostly associated with a DIV or any other block element; a controller is also associated with a block element, except that, in this case, the block element has a few children.

How would you organize this script code?

The most obvious solution—writing a collection of global JavaScript functions—works, but it's quite fragile in the long run. You run into naming-convention issues with too many functions with too similar names and behavior. The overall solution tends to be error prone and a bit too time consuming. What else?

Having said that updaters and controllers go hand in hand with root DOM elements, you might think that jQuery plug-ins are an option to consider. It's definitely an option to consider, but a plug-in is a reusable component whereas here we're explicitly talking about components that manage just one specific DOM tree. In addition, your jQuery plug-in would be a pretty complex one with several messages to handle. In the end, it will look more like a jQuery UI widget than a classic jQuery plug-in. To minimize programming efforts, you might be tempted to extend the jQuery object with too many entries—a bad programming practice. (See *http://docs.jquery.com/Plugins/Authoring#Namespacing*.)

It should also be said that this "bad" programming practice is just a bad programming practice and not a source of errors. The worst effect it produces is that it pollutes the jQuery global namespace. This would be a deadly sin if you were writing a reusable plug-in. As mentioned, however, this plug-in is not reusable and is clearly scoped to one page and one HTML block element. So, in the end, should you really care?

Anyway, global self-contained functions are probably the best way to go. You can define them in the global JavaScript space, group them in your own global object, or hang them off the jQuery object. In this case, you can also have chained calls with no extra effort.

A Look at JavaScript Updaters

Based on Figure 10-4, here's an example of how to restructure the HTML template of the host page to implement navigation within the schedule of a conference:

```
<div id="mainPanel">
    <div id="header">
        <h2>@Model.Location | <span class="hilite">@Model.Date</span></h2>
    </div>
    <div id="navigationBarPanel">
        <input type="button" id="buttonBack" value="Back" />
        <hr />
    </div>
    <div id="contentPanel">
        <div style="display:none;" id="panelDays">@Html.Partial("uc_days", Model)</div>
        <div style="display:none;" id="panelDay">@Html.Partial("uc_day")</div>
        <div style="display:none;" id="panelSession">@Html.Partial("uc_session")</div>
        <div style="display:none;" id="panelSpeaker">@Html.Partial("uc_speaker")</div>
        <div style="display:none;" id="panelSpeakers">@Html.Partial("uc_speakers")</div>
    </div>
    <div id="applicationBarPanel">
        <hr />
        <input type="button" id="buttonSpeakers" value="Speakers" />
    </div>
</div>
```

You can have three main panels: navigation, content, and application. The content panel is articulated in one panel for each view template—list of conference days, sessions per day, session details, list of speakers, and speaker details. For each of these DIVs, you need an updater, and you need a controller to govern the whole thing, including back navigation.

The following code runs on page readiness and sets up JavaScript components:

```
<script type="text/javascript">
    $(document).ready(function () {
        $("#buttonSpeakers").bind("click", buttonSpeakersHandler);
        $("#buttonBack").bind("click", buttonBackHandler);
        $.mainPanelController().setView(ViewTypes.Days);
    });
</script>
```

The jQuery object is extended with the *mainPanelController* plug-in, which exposes a *setView* method to select the initial view to display. The difference between the *mainPanelController* plug-in and a classic plug-in is that this one is global and not associated with the content of a wrapped set:

```
(function ($) {
    var _methods = {},
        _currentViewType = null,
        _registeredUpdaters = null,
    ...

    $.mainPanelController = function () {
       ...

        _panelDay = $("#panelDay");
        _panelDayId = _panelDay.attr("id");
        _registeredUpdaters[_panelDayId] = UpdateHelpers.DayPanelUpdater;
       ...
       _methods = {
            setView : _setView,
            refreshView : _refreshView,
            back : _back
        };
        return _methods;
    };

    var _setView = function (viewType) {
        _currentViewType = viewType;
        _contentPanelSetView(viewType);
        if (viewType === ViewTypes.Days) {
            _navigationBarSetView(ViewTypes.Hidden);
            _applicationBarSetView(ViewTypes.Display);
            return this;
        };
        if (viewType !== ViewTypes.Days) {
            _navigationBarSetView(ViewTypes.Display);
            _applicationBarSetView(ViewTypes.Hidden);
            return this;
        };
        return this;
    };

    var _refreshView = function (viewType, data) {
        var func = null;
        if (viewType === ViewTypes.Day) {
            func = _registeredUpdaters[_panelDayId];
        };
        if (viewType === ViewTypes.Session) {
            func = _registeredUpdaters[_panelSessionId];
        };
        if (viewType === ViewTypes.Speakers) {
            func = _registeredUpdaters[_panelSpeakersId];
        };
        if (viewType === ViewTypes.Speaker) {
            func = _registeredUpdaters[_panelSpeakerId];
        };
```

```
        if (func != null)
            func(data);
        return this;
    };

    var _back = function() {
        var viewType = _getReturnViewType();
        _setView(viewType);
    };
})(jQuery)
```

The plug-in returns an object that exposes three methods—one to set a particular view, one to refresh a particular view, and one to move back to the previous view. Each view has an updater—namely, a JavaScript function that simply gets JSON data and updates a portion of the DOM. As an example, here's the updater that refreshes the view based on the list of sessions:

```
UpdateHelpers.DayPanelUpdater = function (data) {
    // Display the date of the selected day
    $("#panelDay #Header").html(data.CurrentDay.Date);

    // Display the list of sessions in each time slot
    $("#panelDay #ScheduleArea").empty();
    $("#panelDay #tmplTimeSlots")
    .tmpl(data.CurrentDay.TimeSlots)
    .appendTo("#panelDay #ScheduleArea");
};
```

The updater receives downloaded JSON data as returned by the controller method and uses the jQuery Template plug-in to create a list of sessions on the fly. You can get the jQuery Template plug-in here: *http://api.jquery.com/category/plugins/templates*.

Finally, here's the code that renders the list of sessions:

```
<script type="text/javascript">
    function _selectSession(id) {
        var url = "/ajax/session/" + id;
        $.getJSON(url, function (data) {
            $.mainPanelController().refreshView(ViewTypes.Session, data)
                                   .setView(ViewTypes.Session);
        });
    }
</script>

<!-- Template markup -->
<script id="tmplTimeSlots" type="text/x-jquery-tmpl">
    <tr>
        <td>${StartTime}</td>
        <td>${Description}</td>
    </tr>
```

```
    {{each Sessions}}
        <tr>
            <td> </td>
            <td><a href="javascript:_selectSession(${Id})">${Title}</a></td>
        </tr>
    {{/each}}
</script>

<h2><span id="Header"></span></h2>
<table id="ScheduleArea">
</table>
```

When the user clicks on a session, a JSON call is placed and the user interface is updated with the following surprisingly fluent code:

```
$.mainPanelController().refreshView(ViewTypes.Session, data)
                       .setView(ViewTypes.Session);
```

You tell the controller to refresh the view with the list of sessions using the specified data and then display it. (See Figure 10-5.)

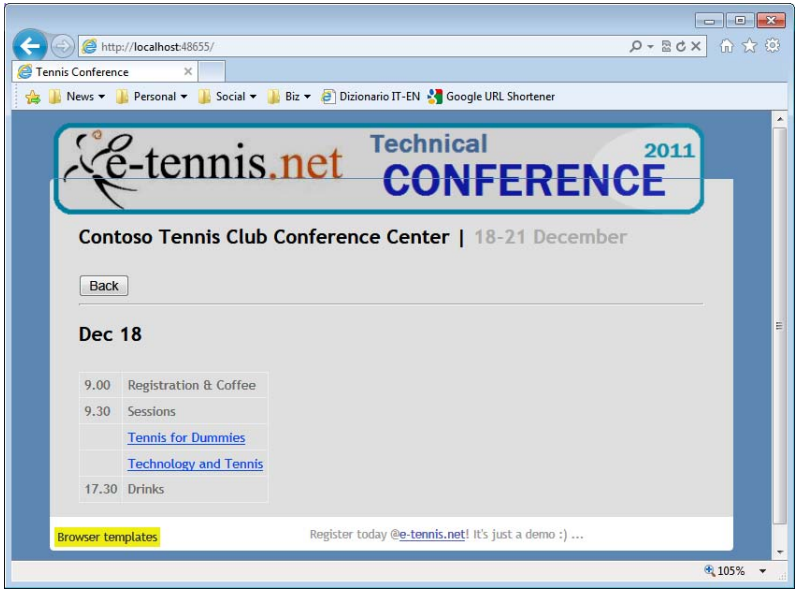

FIGURE 10-5 Displaying a list of sessions.

The full source code for this example can be downloaded from XXX.

Intricacies of JSON-Based Solutions

Any data exchange over the web based on JSON is significantly smaller than a data exchange based on HTML. HTML contains both markup and data; JSON contains only data. Managing the view template, however, is equally problematic because it requires you to design carefully the user interface

and introduce a lot of JavaScript code. More importantly, the JavaScript code must be well organized or you're lost.

A JSON-based solution might be really effective, but the effort required to implement it should never be taken lightly. Don't choose this solution just because you've heard that it's faster. It's faster when it runs, if it ever runs.

Summary

People like to consume interactive applications through the web. For various reasons, the most common way of writing these applications is still by using JavaScript. JavaScript, in fact, happily survived the advent of Adobe Flash and Microsoft Silverlight. Although Flash and Silverlight will still be used in web applications, they will no longer cannibalize old-fashioned, vintage JavaScript.

JavaScript was originally introduced to give web authors the ability to incorporate some simple logic and action in HTML pages. JavaScript was not designed to be a classic and cutting-edge programming language. The design of JavaScript was influenced by many languages, but the predominant factor was simplicity. It needs extra facilities to support the development of any functionality that goes beyond changing the attribute of a DOM element. And this is where libraries like the de facto standard jQuery and emerging MooTools fit in.

JavaScript is largely employed to make web applications interactive by downloading data on demand from the server. For an ASP.NET MVC developer, this is challenging because it affects the way you write controllers and the structure of the views. It also requires that you organize your client code properly by making use of patterns and design skills.

Index

Symbols and Numbers

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

About the Author

Dino Esposito is a software architect and trainer living near Rome and working all around the world. Having started as a C/C++ developer, Dino has embraced the ASP.NET world since its beginning and has contributed many books and articles on the subject, helping a generation of developers and architects to grow and thrive.

More recently, Dino shifted his main focus to principles and patterns of software design as the typical level of complexity of applications—most of which were, are, and will be Web applications—increased beyond a critical threshold. Developers and architects won't go far today without creating rock-solid designs and architectures that span from the browser presentation all the way down to the data store, through layers and tiers of services and workflows. Another area of growing interest for Dino is mobile software, specifically cross-platform mobile software that can accommodate Android and iPhone, as well as Microsoft Windows Phone 7.

Every month, at least five different magazines and Web sites in one part of the world or another publish Dino's articles, which cover topics ranging from Web development to data access and from software best practices to Android, Ajax, Silverlight, and JavaScript. A prolific author, Dino writes the monthly "Cutting Edge" column for MSDN Magazine, the "CoreCoder" columns for DevConnectionsPro Magazine, and the Windows newsletter for Dr.Dobb's Journal. He also regularly contributes to popular Web sites such as DotNetSlackers—http://www.dotnetslackers.com.

Dino has written an array of books, most of which are considered state-of-the-art in their respective areas. His more recent books are Programming ASP.NET MVC 3 (Microsoft Press, 2011) and Microsoft .NET: Architecting Applications for the Enterprise (Microsoft Press, 2008), which is slated for an update in 2011.

Dino regularly speaks at industry conferences worldwide (such as Microsoft TechEd, Microsoft DevDays, DevConnections, DevWeek, and Basta) and local technical conferences and meetings in Europe and the United States.

In his spare time (so to speak), Dino manages software development and training activities at **Crionet** and is the brains behind some software applications for live scores and sporting clubs.

If you would like to get in touch with Dino for whatever reason (for example, you're running a user group, company, community, portal, or play tennis), you can tweet him at **@despos** or reach him via Facebook.

What do you think of this book?

We want to hear from you!

To participate in a brief online survey, please visit:

microsoft.com/learning/booksurvey

Tell us how well this book meets your needs—what works effectively, and what we can do better. Your feedback will help us continually improve our books and learning resources for you.

Thank you in advance for your input!

Microsoft® Press