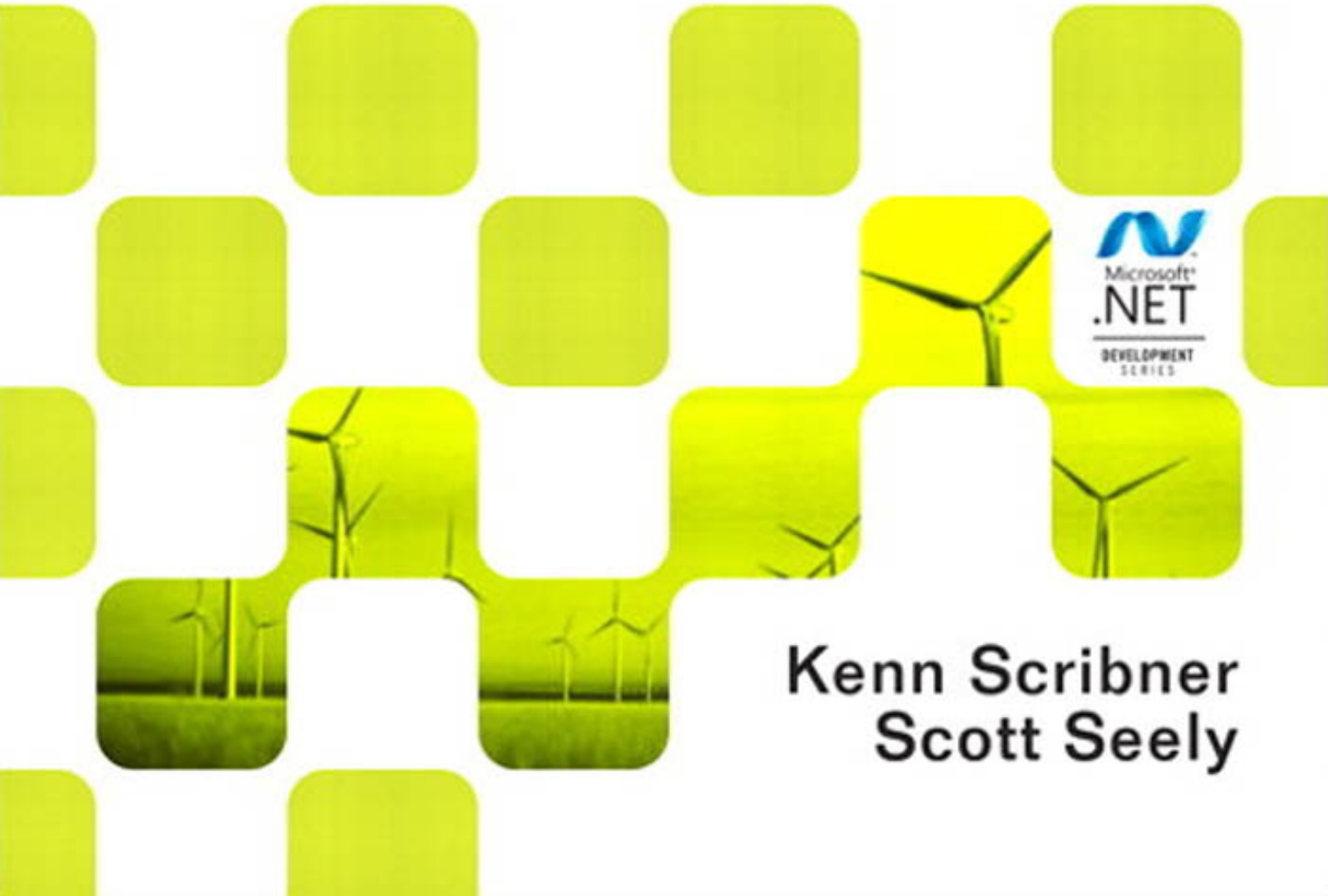
Effective REST Services via .NET

For .NET Framework 3.5



Kenn Scribner
Scott Seely

Library of Congress Cataloging-in-Publication Data:

Scribner, Kenn.
Effective REST services via .NET : for .NET Framework 3.5 / Kenn Scribner, Scott Seely.
p. cm.
ISBN-13: 978-0-321-61325-7 (pbk. : alk. paper)
ISBN-10: 0-321-61325-2 (pbk. : alk. paper) 1. Web services. 2. Representational state transfer (Software architecture) 3. Web site development. 4. Internet programming. 5. Microsoft .NET Framework. I. Seely, Scott, 1972- II. Title.
TK5105.88813.S32 2009
006.7'882—dc22

2009002859

ISBN-13: 978-0-321-61325-7
ISBN-10: 0-321-61325-2

Text printed in the United States on recycled paper at Courier in Westford, Massachusetts.
First printing April 2009

Foreword

SOAP has been with us for the better part of ten years now, but few would argue that it has lived up to its promise. It works, and it has been used to tie together thousands of services and service consumers. But if you believe simpler is better, SOAP falls short. XML is verbose, WS-* is complex, and interoperability is elusive. SOAP may be the right tool for the job when it comes to composing an SOA symphony from disparate enterprise servers, but for the vast majority of today's consumer-oriented applications, SOAP is not only overkill, it's too slow and too complex to merit honest consideration.

Enter REST. Fast becoming the most important Web service protocol on the planet, REST is everything SOAP isn't. Want to dress up an application with real-time weather data? How about firing off an HTTP request to http://www.contoso.com/weather/98052 and parsing a few bytes of XML returned in the response? You don't need elaborate tooling support to consume WSDL contracts and generate Web Service proxies, nor do you need SOAP libraries to generate and digest hundreds of lines of message headers. A few lines of code and an XML parser will do. Moreover, you might not even need the parser; REST services can return data in any format they want. With REST, the wire format is driven by the requirements of the application, not by the protocol that the application uses.

This is why the world seems to have settled on REST as the simplest and most effective means for publishing data and services to consumer-oriented applications. Flickr exposes a REST API for searching its massive store of photographs; Amazon offers a REST API for searching its product

catalog; Google and Yahoo provide REST APIs for search, traffic data, geographical data, and more; and Digg makes news stories, videos, and more available via REST. The list could go on and on, but the upshot is that REST is the language spoken by the most "interesting" public-facing services today and is likely to be used to expose even more interesting stuff in the future. When you publish data the REST way, you're in good company. And understanding REST is the key to building content-rich applications that draw from resources outside your own application domain. You probably don't have access to your own Doppler weather radar, but somebody else does—and if they're willing to share that information through REST, then you, too, can incorporate real-time weather data into your UI.

Microsoft's Web stack offers a number of ways for developers to build RESTful services. You can implement them using IIS, ASP.NET, ASP.NET's MVC framework, WCF, and even Azure and .NET Services. Understanding the programming models is a key first step in architecting and implementing REST services and clients. Should you author services using WCF or ASP.NET? Should data be encoded as XML or JSON? Which model delivers the best balance of performance and ease of implementation, which one provides the best support for unit testing, and what should I know about best practices before I start?

I can't think of anyone better or more qualified to answer these questions than Kenn Scribner and Scott Seely. I've had the privilege of working with both of them in recent years, and besides being passionate about the subject, both are world-class presenters with a knack for breaking down complex information and presenting it in understandable, bite-sized chunks. Both have real-world experience building Web Services and Web Service clients on Microsoft platforms. And both possess the perspective needed to present a fair and balanced view of REST development, sharing with you not only the hows but the whys—and in some cases, the why-nots.

I can't promise you that REST won't be displaced in a year or two by something sexier. In fact, I can guarantee that it *will* be replaced someday, perhaps sooner rather than later; such is the nature of our industry. But for the time being, REST is where the action is, and becoming REST-literate is

one of the smartest things a developer can do to sharpen his or her skill set. Join Kenn and Scott as they take you on a RESTful journey through the Microsoft technology stack; and most of all, sit back and enjoy the ride!

Jeff Prosise
Knoxville, TN
February 2009

Preface

Kenn's Thoughts: The Road to REST, an Engineer's Tale

It was the spring of 2008, and I had just completed some work for Justin Smith, a senior engineer and connected systems expert at Microsoft. The work involved developing two related Web sites that would consume services offered by a third Web application that would be hosted in something known as the "cloud." After six weeks of iterative development, we had a set of Web applications that were designed to demonstrate nearly all the ways Web applications can communicate using .NET technologies.

I'd heard about REST, of course, having been fortunate enough to work with Dino Esposito on several of his ASP.NET and AJAX books. Dino is a huge REST proponent. But I admit my background was more along the lines of the SOAP protocol, and I looked at Web services more as remote method calls than creatures of the Internet ecosystem. It didn't bother me that I needed fancy proxies to communicate with XML-based (and not JSON-based) Web services. I truly hadn't consciously considered the notion that remote procedure call (RPC) style messaging wasn't quite architecturally in harmony with the Internet itself.

But I'd had this nagging concern for some time. SOAP and XML-RPC services were becoming very complex, and it seemed that at every turn we were trying to solve some problem that the basic architecture of the Internet presented. Security, streaming large binary objects, browser-based proxies (for AJAX), and so forth were leading to an ever-increasing

number of new specifications, each designed to layer more complexity on to what had started as a simple concept. And in most cases, we were trying to bypass the basic workings of the Internet rather than using them to our advantage.

After working with Justin, and after building a very detailed and fully functional set of Web applications that were primarily based on RESTful principles, I found I had drunk from the RESTful Kool-Aid pitcher, and I was stunned by what I had overlooked all of these years. I can remember the epiphany…I literally sat up in my chair, stunned by what I had realized.

What I had overlooked was the simplicity and elegance of the Web's architecture and design. I had been overcome by the glamour of XML and serializing binary information for transmission in response to requests for actions. I had lost sight of the Internet's most basic capability of asking for and receiving a resource's representation. The simplicity and elegance of the Internet struck a new chord with me that day. Even though I had been working with Internet-based technologies for nearly ten years, I found I'd suddenly rediscovered programming for the Internet.

And the simple truth is this is not a bad thing, nor is it uncommon. REST as an architectural concept is precisely in line with the architecture of the Web itself. Any tool you have that can build Web applications can be used to build RESTful services. If you have access to the HTTP method—`GET`, `POST`, and so forth—and if you have access to the HTTP headers and entity body, you have all you need to create a RESTful service. Anything else is there only to make creating RESTful services easier by hiding some of the detail. I haven't had the pleasure of meeting Dr. Roy Fielding, but based on his doctoral dissertation that introduces us all to the concept of REST, I'd hazard a guess that he'd prefer you understand REST at its lowest level before using frameworks that mask the underpinnings. When you do, REST makes perfect sense and things become very clear. Or at least I felt so.

Scott's Thoughts: REST Is Best

From 2000 until the middle of 2006, I worked at Microsoft on Web services. For four of those years, I worked on Windows Communication

Foundation (WCF)—that amazing, transport-agnostic, messaging-unification machine. When WCF finally came out, it supported WS-* and some very basic REST/POX messaging. A few folks on the team were hard at work adding first-class REST support in the form of URI templates and extra functionality for HTTP-hosted services that were later released in .NET 3.5. Why this focus on REST? REST was starting to get very popular, thanks to Roy Fielding's dissertation. Like many in the Web service community, I read his dissertation many times, trying to really understand what made the Web scale as well as it did. When the WCF 3.5 bits started coming out as previews, I checked out the greatly improved REST support. I was getting excited by what I was seeing and learning. In the broader community and at work, I was finding that people were getting more and more comfortable using HTTP as a communication medium to create, retrieve, update, and delete resources.

I also started seeing the value in easy-to-type URLs. Furthermore, I found that the architecture and code just makes sense to developers. During 2006, I taught several multiday classes on WCF and gave presentations on WCF at a few conferences across the country. My talks on REST were well received. My talks on WCF internals weren't. People appreciated the elegance of what WCF can do. They just did not see the value in the steep learning curve one had to traverse to master the technology. The thing that pushed me over to REST was the realization of why people were flocking to REST over WS-*. In general, developers used Web browsers and built Web applications long before they ever had to add a service of any kind. REST development builds on what Web developers already know, so there is less to learn. WS-* might be elegant and cover many scenarios, but it does not build on what most developers in most shops across the globe already know.

In the summer of 2008, I joined the development team at MySpace as an architect. Guess what architectural style one of the world's largest .NET sites uses to handle access to Friends, photo albums, and other resources. Yes, it's REST. The platform is, first and foremost, a Web platform. REST holds more value for HTTP base endpoints than a WS-* one ever will. REST integrates well with so many other platforms without a whole lot of effort. It doesn't impose structure on the payload contents—only on the

payload metadata. REST is a model that novice developers understand and that expert-level developers can easily manipulate. I love the fact that it is penetrating so much of service development. My day job involves working on OpenSocial—already one of the most successful REST APIs ever developed. Through my work with OpenSocial, I have seen that HTTP and REST compose well with many different security mechanisms. I find it interesting that WS-* protocols compose well with other XML mechanisms. REST composes with other HTTP mechanisms. After spending so many years working with SOAP and other RPC mechanisms, I like what REST has to offer.

How This Book Approaches REST

Today, we use this stuff. We build solutions based on this stuff. We like this stuff. And we're truly glad to have this book in our hands as architects and developers. Both authors and the entire team behind this book hope you will find it informative and useful as well.

One thing we didn't want was a 1,000-page monster. When you understand REST, the concept is actually simple, and applying .NET technologies to create RESTful solutions becomes a relatively easy task. If it can't be explained in a few pages, something's not right.

The first couple of chapters introduce you to the concepts involved with REST. In a sense, you're taken back to the earliest days of the Internet to rediscover how the Internet works and how the architectural concept known as REST fits into the Internet ecosystem so well. The first chapter, "RESTful Systems: Back to the Future," addresses REST itself, and you learn what it means to be RESTful and how to identify behaviors that are *not* RESTful. Chapter 2, "The HyperText Transfer Protocol and the Universal Resource Identifier," is devoted to HTTP and the URI. These are the two fundamental tools you'll work with when developing .NET-based solutions.

Chapters 3 and 4 dig into the client side of the equation. RESTful services are there to serve a client's needs, and there is no better way to begin to use REST than to consume RESTful services from a client's perspective. There you learn what works and what doesn't, with the lessons you learn

translating to design principles when you create RESTful services yourself. Chapter 3, "Desktop Client Operations," shows you how to access RESTful services from desktop applications (both authors believe that the desktop is not a dead platform but is instead enhanced by Internet data and service access), and Chapter 4, "Web Client Operations," shows you how to access RESTful services from Web-based applications, including Silverlight 2.0. For consistency, both chapters access a single REST service. Later chapters build individual services unique to each chapter to increase the breadth of exposure to different RESTful service implementations.

After you have a feel for how a client might use your service, it's time to dive into server-side programming. Here the book starts with the basics: what Internet Information Services (IIS) is, how it is put together, and how you use it to implement RESTful services. Chapter 5, "IIS and ASP.NET Internals and Instrumentation," leads you through the most foundational server-side technology: Microsoft's premier Web server. Clearly this is one technology that supports nearly all the other .NET Web-based technologies, RESTful or otherwise, and understanding how it works is crucial to building effective REST services.

Chapters 6 though 8 then use higher-order .NET technologies to implement RESTful services. Chapter 6, "Building REST Services Using IIS and ASP.NET," uses what you learned in Chapter 5 to create a Web blog service using only traditional ASP.NET constructs. Chapter 7, "Building REST Services Using ASP.NET MVC Framework," introduces you to the ASP.NET MVC framework and shows how implementing a RESTful service might differ from traditional ASP.NET when you have the MVC framework to rely on. Of course, no .NET book discussing RESTful technologies would be complete without digging into the nuts and bolts of WCF, and Chapter 8, "Building REST Services Using WCF," does just that.

The final chapter, Chapter 9, "Building REST Services Using Azure and.NET Services," shows how you would combine cloud computing with RESTful services to accomplish tasks that otherwise would be nearly impossible. In this case the sample application demonstrates a comment service you can execute from behind your firewall on your private network. Your service will reach out and allow other people over the Internet who are working behind their firewalls to work with your service.

We then provide three appendixes we hope you'll find helpful. The first, Appendix A, ".NET REST Architectural Considerations and Decisions," discusses some of the architectural aspects and why you might choose a particular .NET technology over another. Appendix B, "HTTP Response Codes," discusses each of the possible HTTP response codes and in particular what they mean to RESTful services and clients. And, finally, Appendix C, "REST Best Practices," tries to provide some concise guidance for creating RESTful services.

Let's face it. When it comes to writing effective software, the more you know, the more effective your software will be. Although we don't assume that you're the world's foremost expert on writing ASP.NET applications, we do believe you'll have some real-world .NET experience before reading this book. We're going to be working at some of the lowest levels of HTTP, IIS, and even ASP.NET, so some familiarity with each of these is a plus. But neither is this book a 1,000-page monster, so if there are bits and pieces you're not so familiar with, there should be plenty here to introduce you to concepts and techniques you'll find useful in your daily work.

Finally, we've set up a Web page in addition to the publisher's page where you can send comments and questions directly to us. If anyone (gasp!) finds…inconsistencies…in our sample software, we'll post updated code there for you to download. Interesting and informative tidbits might find their way there as well, time permitting. Both authors earn their living writing software just as you do, and we're every bit as busy as you are making ends meet with the world economy the way it has been in the latter part of 2008. But both authors love this architectural concept and are committed to helping you understand and use it as well. So if interesting and informative things come up, we'll put them on the book's Web page:

www.endurasoft.com/rest.aspx

7

Building REST Services Using ASP.NET MVC Framework

In Chapter 6 we created a basic RESTful service using bare-bones ASP.NET techniques such as implementing an HTTP handler that represents our service. Interestingly, this isn't far from what happens with the ASP.NET MVC framework, but we don't look at it quite that way. Moreover, we gain some niceties revolving around URL routing and controller action invocations.

The ASP.NET MVC Framework

In mid-2007 or so, Microsoft introduced a new way to build ASP.NET applications that is based on the classic model-view-controller (MVC) design pattern. Although we could argue whether it fits the true MVC pattern or the more contemporary front controller pattern, the idea is that the traditional Web Forms method of creating Web pages is replaced by a framework that is actually based on RESTful principles. If you've not tried the ASP.NET MVC framework, it is available for download from this URL: http://www.asp.net/mvc

Traditional Web pages are rooted in disk files, and the representation they offer is the rendered HTML that comes from either the HTML stored in the file or, in the case of ASP.NET, the page offered up by the ASP.NET `PageHandlerFactory`. Consider what happens when you enter a URI such as the following:

```
http://www.contoso.com/Default.aspx
```

Here, ASP.NET receives the incoming request and shuttles it to the `PageHandlerFactory`. `PageHandlerFactory`'s job is to locate the compiled code that represents the requested page. This code, based on `IHttpHandler`, then passes through a series of what amounts to workflow steps to render the HTML that is ultimately returned to the client. In the end, though, whether the client requests an HTML page or an ASP.NET page, the URI they use targets a resource that (typically) resides in a specific file on disk. And if you're using `PageHandlerFactory`, the representation the client will receive is HTML or some dialect of HTML, like XHTML.

The ASP.NET Web Forms model uses two files most of the time. The first file is a markup file that contains basic HTML and ASP.NET-specific markup that indicates which controls the page handler will instantiate and otherwise manipulate. The second file is called the "code-behind" file (or sometimes "code-beside"), and it contains programming logic in your choice of .NET language. As far as it goes, this page mechanism is fine and it works. But there is a tight coupling that exists between the markup and the logic that drives the page since the markup and code-behind pages are closely related. This doesn't separate the view from the logic behind the view, which causes difficulties when considering such things as automated unit testing or test-driven design, or even when trying to inject standard practices like separation of concerns. Although much can be done by developers to mitigate this tight coupling, in practice most development teams don't (or can't) make the investment because it is not self-evident, and it requires planning, training, and code review to ensure consistent implementation. Ironically, these techniques involve separating the data, the user interface, and the manipulation of that user interface—which in itself is a form of MVC.

Moreover, ASP.NET had received negative comments from time to time from some in the developer community due to the way the Web Forms

page is rendered. These developers consider the Web Forms page rendering process to be "heavy," meaning it takes too long and requires too much server resource to render a simple page. Authoring and rendering ASP.NET controls isn't a simple process either, and at times scalability can be impacted. In addition, the Web Forms model is inherently stateful, using information caches such as view state, control state, and even the easy-to-access session state. It's entirely too easy to get yourself into trouble when implementing a Web site with more complexity than simple content pages.

> **NOTE**
>
> It isn't my intention to argue the merits or demerits of either ASP.NET platform here. In my opinion both Web Forms and ASP.NET MVC are good and have beneficial uses. To me it's more a matter of selecting the proper tool for the job. Web Forms are more resource-centric whereas ASP.NET MVC is more action-centric. Web Forms give you some programming niceties, because view state isn't necessarily a bad thing at times, whereas ASP.NET MVC allows you to program "closer to the metal."
>
> I also found the MVC framework to be a wonderfully RESTful platform, but I understand Microsoft doesn't necessarily agree with this sentiment, preferring for developers to instead use Windows Communication Foundation services. The ASP.NET MVC framework is a terrific platform on which to build RESTful solutions using the very constructs the framework itself provides, but I also understand Microsoft's position. In practice, I think you should decide for yourself based on your application's requirements. I wouldn't hesitate to create a RESTful solution based on ASP.NET MVC if that best fit my application's needs.

The ASP.NET MVC framework was created to address these issues, and you can download the framework as well as learn much more about it at http://asp.net/mvc. The ASP.NET MVC framework is built using a modified version of the venerable model-view-controller pattern, the original concept for which is shown in Figure 7.1. Although you won't find this figure in the original source material, the idea Figure 7.1 embodies comes from

the original source, which you can find at http://heim.ifi.uio.no/~trygver/1979/mvc-1/1979-05-MVC.pdf. I used the word "modified" only because when creating a new ASP.NET MVC project, you're given sample views and controllers. However, any model creation is up to you, so implementing the feedback to the view is therefore also up to you to implement should you choose to do so.

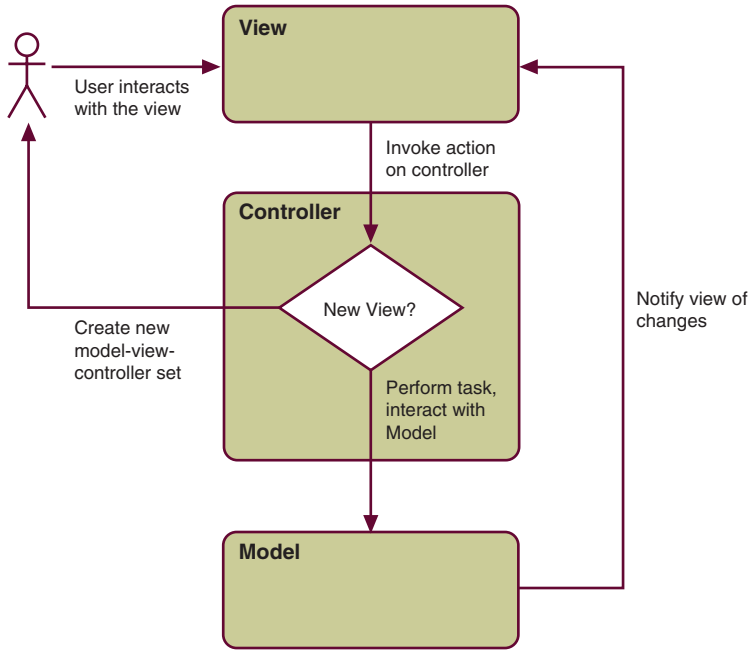


FIGURE 7.1: Original model-view-controller pattern

NOTE

If you're wondering about the feedback line on the right side of the diagram (as I was for a long time), it comes from the definition of the view in the aforementioned reference. Other patterns, like the model-view-presenter (and more contemporary patterns based on MVC, like the front controller pattern of which the ASP.NET MVC framework is built around) often seek to address things such as this direct model-view feedback or provide for other pattern optimizations.

The model-view-controller pattern was revolutionary in the sense that it clearly separated the user interface–specific rendering (the view) from the logic that drives what is shown. User actions, such as button clicks, are passed from the view to the controller, which will either create a new model-view-controller set (such as when redirecting to a different Web page) or interact with the model, which is where both the application logic and the data access reside.

The ASP.NET MVC framework relies on the `UrlRoutingModule` to shuttle Web server requests to the `MvcHttpHandler`, which then interprets the requested URL and activates the appropriate controller. Controller activation is therefore ultimately based on the URI, and a controller action (method) is activated instead of directly targeting a disk-based resource. At its very core, the ASP.NET MVC framework is based on RESTful principles!

In fact, think back to the preceding chapter. Remember the virtual nature of the service I created? The `BlogService` didn't actually exist as a `.aspx` or `.ashx` file but rather was created and registered through the use of `UriTemplateTable`. By adding items to the `UriTemplateTable` and then later checking the incoming URI against the preregistered URIs the service would accept, the service could discern valid URIs, at least from the service's point of view. It could then also dispatch the processing of those URIs along with matched information, such as the parsed `blogID`.

This is very similar to the mechanisms I just described when looking at the ASP.NET MVC framework. The framework provides a more programmer-friendly and standardized way to execute your own code (the `BlogService` is fully custom, after all), but the process for accessing resources is very, very similar in both cases. Let's now look at some MVC framework details.

URL Routing

The URL routing module is driven by "mapped routes," which are URIs you specify and couple to a specific controller. This process is much like setting up the `UriTemplateTable` in the preceding chapter. Here is the route map for the default route when you create a brand-new MVC Web application:

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

`Default` is the name of the route, and you can imagine that this is used as a key in a route table (undoubtedly a dictionary object). The value `{controller}/{action}/{id}` is the "designed" URI, which essentially says this mapped URL will be activated like so:
http://servername/virtualdirectory/controller/action/id
If no controller is specified, it will default to `HomeController`. The MVC framework will take the value placed in the route map, `Home`, and automatically concatenate the word `Controller` to look up the appropriate controller class in the `Controllers` folder, which in this case is `HomeController`. If no action is specified, the MVC framework will examine the `HomeController` class for a method named `Index`. However, if the URI
http://servername/virtualdirectory/Home/About
is used, then the MVC framework will invoke the `About` method contained within the `HomeController`.

In both sample URLs there was no `id` value. None was specified in the URL, and the `Index` and `About` methods contained within the `HomeController` have no parameters to deal with it since none is expected for those actions. However, if you created a blog and provided an action that listed pages in your blog, the `id` value could become the page number:

```
public ActionResult Page(Int32 id)
{
    ...
}
```

In this case, the URL for page 3 of your blog would be this:
http://*servername*/*virtualdirectory*/Home/Page/3
The `HomeController`'s `Page` method would be invoked and passed the value `3` as the `id` parameter. You'd then process the page number using whatever logic makes sense for locating and displaying the desired view.

There is a special case this chapter's service takes advantage of when it registers the RESTful service URI with the URL routing framework, and that is the parameter wildcard. If your application has the specific pattern

the default route maps for you, which is controller, action, and then ID, the default route works fine. But if your application might have a URI that varies, you could map your route using the wildcard and decide what to do when your controller is invoked. Here's an example:

```
routes.MapRoute(
    "VariableURI",
    "{MyController}/{*contentUri}",
    new { controller = "MyController", action = "ServiceRequest" }
);
```

This URI is registered using the `VariableURI` name, but it can be activated using an infinite number of URIs, all of which map to `MyController`'s `ServiceRequest` action method. The remainder of the URI is provided to the `ServiceRequest` method as a string parameter. You'd use this if you simply can't define the URI for all possible client invocations or your URI will vary. Later in the chapter I'll show you how to register a new route in the route map.

Controller Actions

Note that the `Page` method shown previously returns something known as an `ActionResult`. You might imagine an `ActionResult` returning some rendered HTML value, but in fact an `ActionResult` is simply an abstract class defined as such:

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

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

Several concrete `ActionResult` classes are shipped with the ASP.NET MVC framework, including `ViewResult`, which is returned from the controller's `View` method (I'll discuss this in a bit more detail in a following section), `RedirectToRouteResult`, and `PartialViewResult`. Of course, nothing says you can't create your own, and I did exactly that when creating the RESTful service for this chapter. (And of the six cases the service handles, only one of those cases returns HTML to the client.)

As I alluded to earlier in the chapter, URIs are mapped to controller actions (not disk files as with traditional ASP.NET). Controller actions are implemented by methods hosted by your controller classes that return `ActionResult` values. The behavior of the controller action in most cases would be to spin up a `.aspx` page (the view), but though this is common, it isn't required. Your controller can take other actions, depending on your application's needs.

Accepting HTTP Methods

A nice feature of the ASP.NET MVC framework is the capability to separate controller actions based on HTTP method. That is, you can specify one controller action for HTTP `GET` and another for `POST`, `PUT`, `DELETE`, or whatever. Coupled with this concept is the capability to overload the naming of the methods from the framework's point of view. This is a great feature for RESTful services in which you generally support more than HTTP `GET`.

Let's look at an example. Here is a valid URI for this chapter's sample service, `CodeXRC`:

```
https://servername/virtualdirectory/CodeXRC
```

You can imagine `CodeXRC` as a simple-minded source code repository. If a client accessed this URI, to determine what to do, you would need to examine the HTTP method. If it was `HEAD`, you would do one thing. If it was `GET` or `POST`, you would do another. To accomplish this, you would probably write code that used a switch statement using the HTTP Method to decide what to do:

```
public ActionResult Index()
{
    switch (this.HttpContext.Request.HttpMethod.ToLower())
    {
        case "head":
            // Do HTTP HEAD
            return DispatchHead();

        case "get":
            // Do HTTP GET
            return DispatchGet();

        case "put":
            // Do HTTP PUT
            return DispatchPut();
```

```
            case "post":
                // Do HTTP POST
                return DispatchPost();

            case "delete":
                // Do HTTP Delete
                return DispatchDelete();

            default:
                // Unknown method, process error
                break;
        }

        // Process error
        DispatchErrorMethodNotAllowed();
    }
```

In fact, this is precisely what you saw in the preceding chapter. It's another example of a dispatch table in which the appropriate service handler is invoked based on HTTP method. It's also very much "boilerplate" code that could be rolled into a framework, and that's exactly what was done in ASP.NET MVC.

But then we have an issue. We can't have identically named methods with identical method signatures. If the framework can route actions to a controller based on HTTP method, then there has to be some way to overload the name of the method so that we don't have syntactical errors. That is, we can't have this situation:

```
// HTTP HEAD?
public ActionResult Index()
{
    ...
}

// HTTP GET?
public ActionResult Index()
{
    ...
}
...
// HTTP DELETE?
public ActionResult Index()
{
    ...
}
```

Clearly this won't compile, but this is exactly the situation we would have since a single URI serves all HTTP methods (keeping the original service URL in mind: https://servername/virtualdirectory/CodeXRC).

It's for this reason the ASP.NET MVC framework coupled the ability to handle different HTTP methods with different controller actions using an aliased name. The `AcceptVerbs` and `ActionName` attributes cleanly disambiguate the HTTP method and aliased name:

```
// HTTP HEAD
[AcceptVerbs("HEAD")]
[ActionName("Index")]
public ActionResult ProcessHead()
{
    ...
}

// HTTP GET
[AcceptVerbs("GET")]
[ActionName("Index")]
public ActionResult ProcessGet()
{
    ...
}
...
// HTTP DELETE
[AcceptVerbs("DELETE")]
[ActionName("Index")]
public ActionResult ProcessDelete()
{
    ...
}
```

In this case I've rewritten the previous example to show the proper technique. From a URI perspective, the controller action is always `Index`. But the true controller action to be invoked will depend on the HTTP method used to invoke the action. If the HTTP `GET` method is used, the `ProcessGet` action is invoked, and so forth.

This chapter's sample RESTful service makes heavy use of this new feature, and I'd expect that many services will do so over time as well.

Views

Although this book isn't about building Web sites using ASP.NET MVC, I thought a paragraph or two that describes how the views are handled is appropriate since I'm introducing the framework.

> **NOTE**
>
> You'll find that I didn't make use of the view capability in the chapter's sample application. Perhaps I should have. I certainly could have. I just found it more convenient to take the notion of a controller "action" literally and have the controller address the service request. The point was not to render a "view" but to respond to a call for action, even if that action results in rendered HTML. Had I been building Web pages, I would have done things differently. (If you disagree with my implementation, that's fine. When you build your own services, by all means follow your own interpretation of the pattern.)

The ASP.NET MVC framework uses a folder (conventionally) named `Views` to contain all the views, with views associated with a particular controller in a subfolder named after the controller. Views associated with the `HomeController`, for example, are found in the `Home` folder, which is a child folder of `Views` in the main application directory. If all you ever do is invoke the view associated with the action, the MVC framework will automatically select the view named for the action. For example, the default `HomeController` has an `About` action that invokes the `About` view in the `Home` folder of the `Views` Web application directory. This code does that job, with the MVC framework's help:

```
public class HomeController : Controller
{
    ...
    public ActionResult About()
    {
        ViewData["Title"] = "About Page";

        return View();
    }
}
```

The `About` action returns a `ViewResult` from the controller's base `View` method, which implements the algorithm I mentioned for locating the default view for the action. And as you recall, `ViewResult` is derived from `ActionResult`.

However, you might want to use another view, and as it happens the controller's base `View` method is overloaded, allowing you to select other views based on name. One I find myself using a lot is this overloaded version:

```
protected internal ViewResult View(string viewName, object model);
```

With this overloaded version, you provide the name of the view you would rather invoke as well as some page-specific data the view can access through its `ViewData` property (specifically `ViewData.Model`). Perhaps you have one view that's based on a data grid and another based on a chart. Using this overloaded `View` method, you can select the most appropriate view based on your application's needs.

You also can redirect to another page entirely. The simplest way is to use the controller's `Redirect` method, which returns a `RedirectAction` object. But other redirect methods exist, such as `RedirectToAction` and `RedirectToRoute`. Of course, these allow you to redirect to a different controller action or mapped route.

The Model

In MVC terms, the model is where you place your data access layer and any application-specific business logic. When you create a brand-new ASP.NET MVC Web application, the project wizard creates controllers and views for you as starter code. But no model is created—only a subdirectory is created for you within which you place model code. Nearly all the `CodeXRC` service functionality is implemented in classes that are housed in the model folder, and you'll find this is typical for MVC-style applications.

ASP.NET MVC Security

ASP.NET MVC incorporates the notion of an `Authorize` attribute and the `AccountController` class. Imagine a controller that looks something like this:

```
public class MyController : Controller
{
    ...
    [Authorize]
    public ActionResult DoSomething()
    {
        ...
    }
}
```

The `Authorize` attribute causes the ASP.NET MVC handler to look for a controller named `AccountController` and invokes its `Login` action. The `Login` action, and the `AccountController` for that matter, are designed to work with the ASP.NET Forms Authentication module. Therefore, `Login` and `Register` generate an authentication cookie, `Logout` destroys the cookie, and everything else the account controller does favors the Forms authentication process in ASP.NET.

After seeing the additional work ASP.NET MVC offered, which was to wrap access to the Forms authentication services (or at least a couple of them) behind a custom interface, I thought it would be useful to try to work the HTTP Basic Authentication module from the preceding chapter into the MVC framework. But the simple truth is that given the module from the preceding chapter, the entire concept behind the `AccountController`, or at least the Forms authentication parts of it, aren't needed.

With Forms authentication, navigating between secured pages is accomplished using the ASP.NET security cookie. The Forms Authentication module looks for the cookie, and if it's present and valid when accessing secured resources, the Forms Authentication module allows the secured page to render. The `Authorize` attribute controls which actions are secured.

HTTP Basic Authentication, however, doesn't use a cookie. In fact, this is what makes it so appealing to RESTful services. The client needs to cache the credentials and offer them to the Web server each time access to a secured resource is desired. Modern Web browsers all do this for you, and most (if not all) of the sample desktop applications in this book will also cache the credentials for you if you choose so that each service access doesn't force reauthentication.

In the end, I simply copied the module from the preceding chapter into this chapter's sample application, changed the namespaces involved, and

made the necessary adjustments to the `web.config` file, and the HTTP Basic Authentication worked tremendously well. Since I didn't require the `AccountController`, I deleted it and the views associated with it, knowing that the browser itself would query me for my credentials—I don't need a "login" page for that. True, I also then dismissed the registration and password change request pages, but there is some merit for brevity when producing chapter samples.

> **NOTE**
>
> I also can't use the spectacular `Authorize` attribute, but I found I didn't need it with the custom authentication module. If I were producing a traditional MVC Web application, one based on views, I'd embrace these new ASP.NET MVC tools—the account controller and security attribute—but for a truly RESTful service that requires HTTP Basic Authentication, I discovered they're just not the right tools for the job.

Building an MVC RESTful Service—CodeXRC

With this understanding of the basic ASP.NET MVC framework, it's time to actually build a RESTful service that relies on the framework for basic operations. This service, unlike the blog service in the preceding chapter, won't create an `HttpHandler` to service the RESTful requests but will instead be based on controller actions that are activated by the specific service URI mapped in the URL routing table.

To show a service that did more than produce "Hello, World!" I decided to implement the basis of a source code vault, or perhaps a cloud-based file system. The idea behind this is that you select files on your local hard drive and store them in this secured service. This is an aggressive service for the schedule I had to work with, but though it was becoming more complex the further I got into development, I was more convinced it made for a great chapter sample because it addresses many interesting RESTful concepts. These will be evident when I discuss the URI design.

> **NOTE**
>
> Although not provided with this chapter's sample application, the ASP.NET MVC framework is perfectly suited for creating test-driven design (TDD) applications. Creating applications using test-driven techniques tends to reduce the number of latent bugs and increases code maintainability.

I named the service `CodeXRC`, the "XRC" part having no particular meaning except it isn't copyrighted as far as I could tell. (I don't want to upset the publisher's legal staff.) The URIs for the service are designed like so:

- Add/update a project: `PUT` to `/CodeXRC`
- Add/update a project (alternate): `PUT` to `/CodeXRC/{project}`
- Add/update a project folder: `PUT` to `/CodeXRC/{project}/{folder}`
- Add/update a file: `PUT` to `/CodeXRC/{project}/{folder}/{ext}/{file}`
- Delete all projects: `DELETE` to `/CodeXRC`
- Delete a project: `DELETE` to `/CodeXRC/{project}`
- Delete a folder: `DELETE` to `/CodeXRC/{project}/{folder}`
- Delete files by extension: `DELETE` to `/CodeXRC/{project}/{folder}/{ext}`
- Delete a file: `DELETE` to `/CodeXRC/{project}/{folder}/{ext}/{file}`
- Get all projects (high level): `GET` to `/CodeXRC`
- Get a project (all folders/files): `GET` to `/CodeXRC/{project}`
- Get a folder: `GET` to `/CodeXRC/{project}/{folder}`
- Get files by extension: `GET` to `/CodeXRC/{project}/{folder}/{ext}`
- Get a file: `GET` to `/CodeXRC/{project}/{folder}/{ext}/{file}`
- Get statistics/headers: `HEAD` to a valid service URI

This complex-looking set of URIs really amounts to simulating a file system using REST (see Figure 7.2).

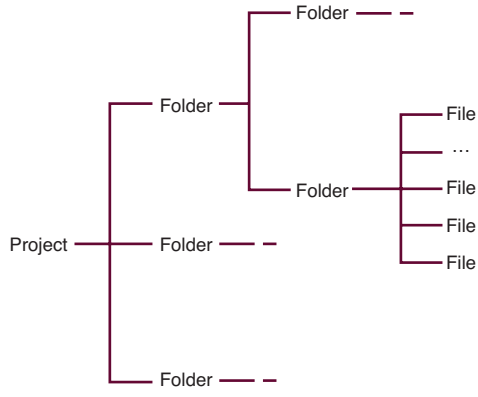


Figure 7.2: CodeXRC **project storage**

The files are stored on the server's file system in the hierarchy shown in Figure 7.2, but there is also a database associated with the service to maintain ownership of the projects and files as well as foreign key relationships for searching and deletion. Users are registered using a process not provided with the service, but when they're registered their account information is stored in the typical ASP.NET tables contained within the database. Projects are owned by the registered users and are identified by their user ID, which is a `Guid`. Roles are also maintained: View, Insert, and Delete. At present the service allows you to access only your own projects, but it would be a relatively easy extension to add View capabilities to projects not your own.

Creating the URL Mapping

With the URI set defined for the service, the place to start is with the URL routing table and mapping. To do this, you open the `Global.asax.cs` file and look for the `RegisterRoutes` method. There, you'll see the default route mapped into the route table (this is created for you by the Visual Studio MVC application wizard). Add this code just before the default route:

```
// Register the RESTful service URIs...note this *MUST*
// come before the default URI mapping or you'll sustain
// 404 errors if the URI doesn't match the default
// mapping.
routes.MapRoute(
    "CodeXRC",
    "CodeXRC/{*contentUri}",
    new { controller = "CodeXRC", action = "ServiceRequest" }
);
```

Of course, this looks a lot like the mapping I discussed previously in the chapter, and given the variable nature of the URI set, you can probably see why I opted for the wildcard approach.

> **NOTE**
>
> You could individually map each URL segment and apply a default, as is done with the Default URL route map, but I chose to simplify the action signatures and parse the URI manually.

When requests come into the server designated for the `CodeXRC` service, the `CodeXRCController` will be called on to take the appropriate action. As is proper for the ASP.NET MVC framework, the `CodeXRCController` is located in the Web application `Controllers` folder.

The `CodeXRCController`

The `CodeXRCController` contains four actions, one for each HTTP method the service handles: HTTP `HEAD`, `GET`, `PUT`, and `DELETE`. Each action is adorned with the appropriate `AcceptVerbs` and `ActionName` attributes. The `HEAD` action is shown in Listing 7.1.

Listing 7.1: The `CodeXRCController` HTTP `HEAD` action

```
[AcceptVerbs("HEAD")]
[ActionName("ServiceRequest")]
public ActionResult HeadRequest(string contentUri)
{
    // Filter the query string and remove any trailing '/' so
    // we don't have an extra array element.
    contentUri = FilterUri(contentUri);
```

continues

Listing 7.1: Continued

```
        // Save the URI parameters
        string[] directives = null;
        if (!String.IsNullOrEmpty(contentUri))
        {
            directives = contentUri.Split('/');
        }

        // Output to a temporary stream
        MemoryStream strm = new MemoryStream();
        this.HttpContext.Items.Add("OutputStream", strm);

        // Handle the request
        CodeXRCGetService result = new CodeXRCGetService(directives, true);
        return result;
    }
```

In Listing 7.1 you can see that the controller action is named `HeadRequest`, but because I applied the `ActionName` attribute, the ASP.NET MVC framework will route the action as if it were named `ServiceRequest`, which matches the name of the action I registered in `Global.asax.cs`. And since the `AcceptVerbs` attribute lists only HTTP `HEAD` as the accepted HTTP method, this action is called only for `HEAD` requests.

The parameter `contentUri` will contain anything on the query string past the controller name. That is, if the client issued a request to the URI http://*servername*/AspNetMvcRestService/CodeXRC/Project1/Folder2/cs/CodeFile3
the `contentUri` parameter would contain the string `Project1/Folder2/cs/CodeFile3`.

To decide what parameters are present, the `contentUri` string is split using the slash as the delimiter. However, I filter the string first to remove a trailing slash. This prevents a phantom entry in the resulting string array. That is, this URI would result in three parameters after the string split:
http://*servername*/AspNetMvcRestService/CodeXRC/Project1/Folder2/
In reality, though, only two parameters are present: `Project1` and `Folder2`. The third entry in the string array would be an empty string, present only because of the trailing slash. By removing the slash, the service doesn't need to check for empty parameter elements in the parameters string array.

In the case of HTTP `HEAD`, the service will perform all the normal `GET` processing. However, the stream the information is written to will differ for

the other HTTP methods. For example, GET will use the actual output stream, thus sending a response to the client. HEAD will use a temporary stream, allowing the service to determine the size of the stream so that the appropriate Content-Length header can be returned to the client. The output stream to use is assigned a slot in the HttpContent Items collection so that it won't need to be passed around as a separate method parameter in all the internal processing methods.

The request is handled by a class that derives from ActionRequest: CodeXRCGetService. As you might imagine, there are corresponding service classes for PUT and DELETE as well. Since GET and HEAD are closely related, a single service class handles both HTTP methods. Something to keep in mind is that the response output stream is not seekable, meaning we can't query its length. It would have been nice to simply write to the appropriate stream and just query the stream length, but unfortunately the response output stream throws an exception when you access its Length property. This means you have to ask the question "Is this a HEAD request?" If so, then (and only then) should you access the stream's Length property and create the Content-Length header. I use the stream's CanSeek property for that.

In contrast, the HTTP PUT method service controller action is shown in Listing 7.2.

Listing 7.2: The CodeXRCController HTTP PUT action

```
[AcceptVerbs("PUT")]
[ActionName("ServiceRequest")]
public ActionResult PutRequest(string contentUri)
{
    // Filter the query string and remove any trailing '/' so
    // we don't have an extra array element.
    contentUri = FilterUri(contentUri);

    // Save the URI parameters
    string[] directives = null;
    if (!String.IsNullOrEmpty(contentUri))
    {
        directives = contentUri.Split('/');
    }

    // Output to the true response stream
    this.HttpContext.Items.Add("OutputStream",
                this.HttpContext.Response.OutputStream);
```

continues

Listing 7.2: Continued

```
        // Handle the request
        CodeXRCPutService result = new CodeXRCPutService(directives);
        return result;
    }
```

I've included the `PutRequest` method here only to highlight the differences: the HTTP method it accepts is `PUT`, the controller action is actually `ServiceRequest` even though the controller method is called `PutRequest`, the stream to be used for the response is the true output stream (versus a temporary one), and the `PUT` behavior is exhibited by the `CodeXRCPutService` class. Otherwise, the request handling for each HTTP method is similar at the controller level.

`CodeXRC` Service Classes

All the `CodeXRC` service classes use `ActionResult` as their base class. In fact, there is a single `CodeXRC` service base class, `CodeXRCServiceBase`, that is used to maintain instances of the URI parameters (which I called "directives" because they direct the service), the data access component, and an auxiliary component that sports helper methods for supporting error responses and such. This base class is shown in Listing 7.3.

Listing 7.3: The `CodeXRCServiceBase` class

```
public class CodeXRCServiceBase : ActionResult
{
    protected CodeXRCAuxServices _aux = new CodeXRCAuxServices();
    protected CodeXRCDataAccess _dal = new CodeXRCDataAccess();
    protected string[] _directives = new string[0];

    public CodeXRCServiceBase()
    {
    }

    public CodeXRCServiceBase(string[] directives)
    {
        // Later logic depends on this not being
        // null even if it has no elements.
        if (directives != null)
        {
            // Assign value
            _directives = directives;
        }
```

```
    }

    public abstract void ExecuteResult(ControllerContext context);
    {
        // Disallow base implementation...note you can't make
        // this abstract since this class is itself derived and
        // this method overridden.
        throw new NotImplementedException(
          "You must override the base ExecuteResult implementation.");
    }
}
```

Since all the derived classes need to use the data access component, the auxiliary helper method component, and the directives, it made sense to collect that in a base class.

Returning HTML, XML, and JSON

The services themselves are broken out into "get," "put," and "delete" versions, each handling the respective HTTP method. Each service method handler also overrides the `ActionResult ExecuteResult` method. The `CodeXRCGetService` implementation is shown in Listing 7.4. The other implementations are similar.

Listing 7.4: `CodeXRCGetService ExecuteResult` **method**

```
public override void ExecuteResult(ControllerContext context)
{
    // Test the user's credentials
    if (context.HttpContext.User.IsInRole("View"))
    {
        try
        {
            if (context.HttpContext.Request.AcceptTypes.Count() > 0)
            {
                // Loop through the collection of accepted types.
                // The first one we hit we understand, take it...
                foreach (string type in
                            context.HttpContext.Request.AcceptTypes)
                {
                    switch (type.ToLower())
                    {
                        case "text/html":
                        case "*/*":
                            RenderHtml(context);
                            return;
```

continues

Listing 7.4: Continued

```
                        case "text/xml":
                            RenderXml(context);
                            return;

                        case "application/json":
                            RenderJson(context);
                            return;

                        default:
                            // Try next one...
                            break;
                    }
                }

                // Couldn't accept any requested type
                _aux.RenderErrorNotAcceptableGet(context);
            }
            else
            {
                // Couldn't accept any requested type
                _aux.RenderErrorNotAcceptableGet(context);
            }
        }
        catch (Exception ex)
        {
            // Couldn't accept any requested type
            _aux.RenderErrorInternalError(context, ex.ToString());
        }
    }
    else
    {
        // User forbidden
        _aux.RenderErrorForbidden(context);
    }
}
```

> **NOTE**
>
> I am a fervent believer in application tracing, and for any Web application I write, I put as much tracing into the code as possible. However, for demonstration purposes here, I've omitted such code to focus instead on the functional nature of the service itself. Tracing is critical to debugging deployed Web applications and I heartily recommend using it.

In Listing 7.4 you can see that the code begins by checking the client's role (the client was authenticated or this method wouldn't be executing). If the client has the appropriate role credentials, the method then checks the desired return type: HTML, XML, or JSON. The default content type, `*/*`, will return HTML. This allows browsers to view the contents of a project, as shown in Figure 7.3. The chapter's sample client will always provide the service with an `Accept` header containing just one accepted content type, but because browsers often include many requested content types, the loop allows the service to check the list of content types for the first one the service has the capability of supporting.

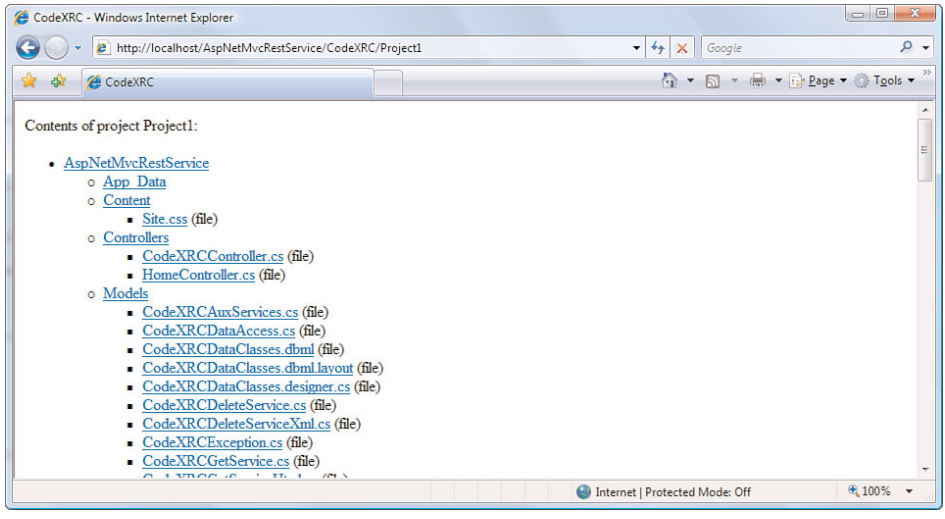


Figure 7.3: Browser REST service access

If you use the chapter's Windows Forms sample client, you can interact with the service in a greater variety of ways, including asking for the project contents as XML, as shown in Figure 7.4.

After the desired return content type is determined, the code in Listing 7.4 then invokes the appropriate rendering method. Listing 7.5 shows you how XML is rendered. I won't show all the supporting methods because there are many, and they often are recursive (and therefore complex) since the methods often need to traverse directory structures. They're also not important to demonstrate how to create RESTful services using the

ASP.NET MVC framework, so I'll leave spelunking their inner workings to you. They're just business logic, so to speak, and as such you'll find them all in the `Models` folder of the `AspNetMvcRestService` Web application.

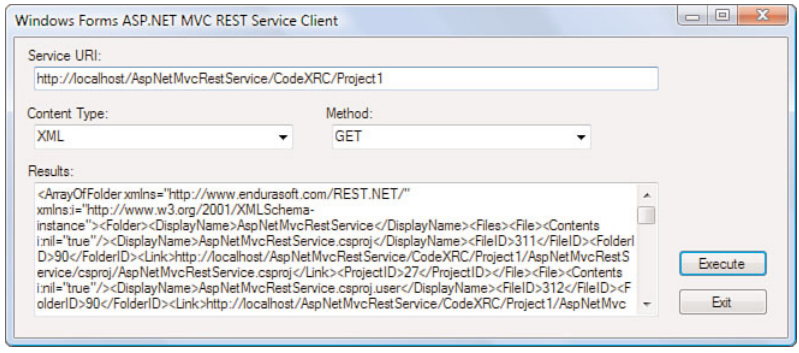


Figure 7.4: CodeXRC project contents as XML

Listing 7.5: The `RenderXML` method

```
private void RenderXml(ControllerContext context)
{
    // Pull the user's ID. This works here since we're using the
    // SQL-based membership provider.
    MembershipUser user =
      Membership.GetUser(context.HttpContext.User.Identity.Name);
    Guid ownerID = (Guid)user.ProviderUserKey;

    context.HttpContext.Response.ContentType = "text/xml";
    switch (_directives.Count())
    {
        case 0:
        default:
            // List all projects for user:
            // http://{host}/{vdir}/CodeXRC
            RenderXmlProjectList(context, ownerID);
            break;

        case 1:
            // List specific project for user:
            // http://{host}/{vdir}/CodeXRC/{project}
            RenderXmlProject(context, ownerID, _directives[0]);
            break;
```

```
            case 2:
                // List specific folder for user:
                // http://{host}/{vdir}/CodeXRC/{project}/{folder}
                RenderXmlFolder(context, ownerID, _directives[0],
                  _directives[1]);
                break;

            case 3:
                // List files by extension for user:
                // http://{host}/{vdir}/CodeXRC/{project}/{folder}/{ext}
                RenderXmlFilesByType(context, ownerID, _directives[0],
                  _directives[1], _directives[2]);
                break;

            case 4:
                // Return specific file to user:
                // http://{host}/{vdir}/CodeXRC/{proj}/{folder}/{ext}/{file}
                RenderXmlFileByName(context, ownerID, _directives[0],
                  _directives[1], _directives[2], _directives[3]);
                break;
        }

        // Assign the content length, but only if HEAD.
        Stream stream = context.HttpContext.Items["OutputStream"] as Stream;
        if (stream != null && stream.CanSeek)
        {
            context.HttpContext.Response.Headers["Content-Length"] =
              GetContentLength(context).ToString();
        }
    }
```

The methods to render HTML and JSON are similar. Actually, rendering XML and JSON is easy using the `DataContractSerializer` and `DataContractJsonSerializer` objects, respectively. But the HTML is rendered by hand (again, one could perhaps use a view for this or augment the XML output with an XSLT reference, but I chose to keep the service logically contained, for right or wrong). After the content is rendered, the stream is examined for its length if the HTTP method was `HEAD`.

An important aspect of returning a resource representation is creating the links (remember hypermedia as the engine of application state). When representations are created, a special class is used to generate the appropriate links, which are shown in Listing 7.6.

Listing 7.6: The CodeXRC link builder class

```
public static class CodeXRCLinkBuilder
{
    public static string BuildProjectLink(ControllerContext context,
                                           string projectName)
    {
        // Generate the link (remember HATEOAS)
        string fullUrl = context.HttpContext.Request.Url.ToString();
        string baseUrl = fullUrl.Substring(0, fullUrl.IndexOf(
          context.HttpContext.Request.ApplicationPath));
        return String.Format("{0}{1}/CodeXRC/{2}", baseUrl,
          context.HttpContext.Request.ApplicationPath, projectName);
    }

    public static string BuildFolderLink(ControllerContext context,
                                         string projectName,
                                         string folderName)
    {
        // Generate the link (remember HATEOAS)
        string fullUrl = context.HttpContext.Request.Url.ToString();
        string baseUrl = fullUrl.Substring(0, fullUrl.IndexOf(
          context.HttpContext.Request.ApplicationPath));
        return String.Format("{0}{1}/CodeXRC/{2}/{3}", baseUrl,
          context.HttpContext.Request.ApplicationPath,
          projectName, folderName);
    }

    public static string BuildFileLink(ControllerContext context,
                                       string projectName,
                                       string folderName,
                                       string fileExtension,
                                       string fileName)
    {
        // Generate the link (remember HATEOAS)
        string fullUrl = context.HttpContext.Request.Url.ToString();
        string baseUrl = fullUrl.Substring(0, fullUrl.IndexOf(
            context.HttpContext.Request.ApplicationPath));
        return String.Format("{0}{1}/CodeXRC/{2}/{3}/{4}/{5}", baseUrl,
          context.HttpContext.Request.ApplicationPath, projectName,
          folderName, fileExtension, fileName);
    }
}
```

The link builder class simply encapsulates the logic in a single place that's necessary to build valid URIs for the various resources.

Creating Resources with XML

The only representation the `CodeXRC` service accepts for creating new resources is XML, and the `CodeXRCPutService` `ExecuteResult` method code is shown in Listing 7.7.

Listing 7.7: `CodeXRCPutService` `ExecuteResult` **method**

```
public override void ExecuteResult(ControllerContext context)
{
    // Test the user's credentials
    if (context.HttpContext.User.IsInRole("Insert"))
    {
        // We'll accept only XML for creating/updating a
        // resource. We could accept JSON easily, but this
        // is how you'd limit inputs to a specific type.
        if (context.HttpContext.Request.ContentType.ToLower() ==
              "text/xml")
        {
            // Create the indicated resource
            CreateResource(context);
        }
        else
        {
            // Wasn't XML...
            _aux.RenderErrorNotAcceptablePut(context);
        }
    }
    else
    {
        // User forbidden
        _aux.RenderErrorForbidden(context);
    }
}
```

The `ExecuteResult` in this case checks for the appropriate role as well as XML as the content type, but only to return an error if the indicated type isn't appropriate. Therefore, there is only one "create resource" method, which is shown in Listing 7.8.

Listing 7.8: `CodeXRCPutService` `CreateResource` **method**

```
private void CreateResource(ControllerContext context)
{
    // Pull the user's ID. This works here since we're using
    // the SQL-based membership provider.
    MembershipUser user =
```

continues

Listing 7.8: Continued

```
        Membership.GetUser(context.HttpContext.User.Identity.Name);
    Guid ownerID = (Guid)user.ProviderUserKey;

    context.HttpContext.Response.ContentType = "text/html";
    switch (_directives.Count())
    {
        case 0:
        default:
            // Create a project for user (will be a specific project):
            // http://{host}/{vdir}/CodeXRC
            CreateProject(context, ownerID);
            break;

        case 1:
            // Create specific project for user (all child folders
            // and files):
            // http://{host}/{vdir}/CodeXRC/{project}
            CreateProject(context, ownerID, _directives[0]);
            break;

        case 2:
            // Delete specific folder (all child folders and files):
            // http://{host}/{vdir}/CodeXRC/{project}/{folder}
            CreateFolder(context, ownerID, _directives[0],
              _directives[1]);
            break;

        case 3:
            // Delete files by extension in specified folder:
            // http://{host}/{vdir}/CodeXRC/{project}/{folder}/{ext}
            CreateFileByType(context, ownerID, _directives[0],
              _directives[1], _directives[2]);
            break;

        case 4:
            // Delete specific file:
            // http://{host}/{vdir}/CodeXRC/{proj}/{folder}/{ext}/{file}
            CreateFile(context, ownerID, _directives[0], _directives[1],
              _directives[3] + "." + _directives[2]);
            break;
    }
}
```

The code in Listing 7.8 is somewhat similar to the code for retrieving a resource representation in Listing 7.5. The URI is examined and the appropriate resource is created. Of course, there is no need to return a content

length, so that code is missing from Listing 7.8. The code to handle HTTP `DELETE` is similar to the code shown in Listing 7.8, so I won't show that here either.

Returning Error Information

In the preceding chapter, error information was returned according to the application's error handling configuration. For this service, I chose to return explicit error information as HTML I create at the point where the exception is caught. The `CodeXRCAuxServices` class provides several error response methods, with an example being the HTTP `403 (Forbidden)` response shown in Listing 7.9.

Listing 7.9: The `CodeXRCAuxServices` `RenderErrorForbidden` method

```
public void RenderErrorForbidden(ControllerContext context)
{
    context.HttpContext.Response.StatusCode =
      (Int32)HttpStatusCode.Forbidden;
    context.HttpContext.Response.ContentType = "text/html";
    using (StreamWriter wtr =
    new StreamWriter((Stream)context.HttpContext.Items["OutputStream"]))
    {
        StringBuilder sb = new StringBuilder();
        sb.Append("<html><head><title>CodeXRC</title></head><body>");
        sb.Append("You are not authorized to access the resource you ");
        sb.Append("requested.<br/></body><html>");
        wtr.WriteLine(sb.ToString());
        wtr.Flush();
        wtr.Close();
    }
}
```

Other errors are handled in a similar manner.

The `CodeXRC` Data Access Layer

The service's data access is provided by the `CodeXRCDataAccess` class. I personally generally prefer to provide data access service through an interface and use a strategy pattern to load the appropriate data access component, but here I keep the example simple and just deal with data access directly.

Listing 7.10 contains the data access code used to read the database and return all the user's projects.

Listing 7.10: The `CodeXRCDataAccess ListProjects` method

```
public List<ProjectDTO>
  ListProjects(ControllerContext context, Guid ownerID)
{
    // Retrieve all projects
    CodeXRCDataClassesDataContext dataContext =
      new CodeXRCDataClassesDataContext();
    var projectQuery = from p in dataContext.Projects
         orderby p.ProjectID
         where p.OwnerID == ownerID
         select p;
    List<ProjectDTO> projects = new List<ProjectDTO>();
    foreach (var project in projectQuery)
    {
        // Create the new project item
        ProjectDTO projectItem = new ProjectDTO(project);
        projectItem.Link = CodeXRCLinkBuilder.BuildProjectLink(
          context, project.DisplayName);
        projects.Add(projectItem);
    }
    return projects;
}
```

As you can see from Listing 7.10, I make heavy use of LINQ to SQL, and you'll find this throughout the data access code. The LINQ to SQL context is shown in Figure 7.5.

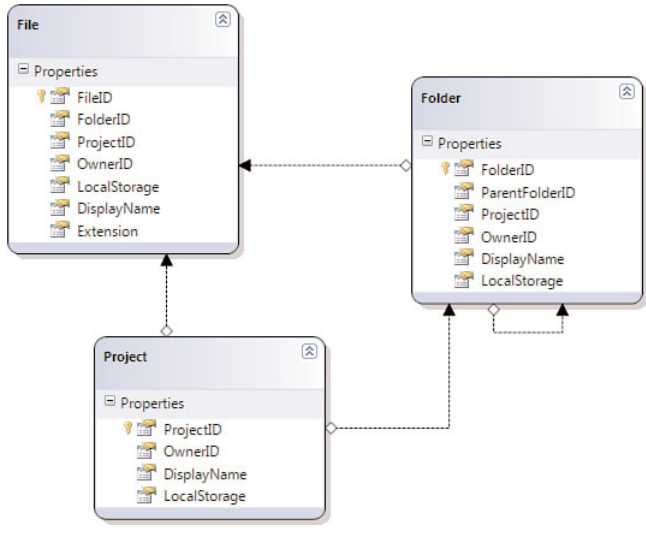


Figure 7.5: The CodeXRC LINQ data context

The foreign keys shown in Figure 7.5 make it easier to query for specific entities. The data access code also creates the folders and files, and when it does, it stores the local directory or file paths in the LocalStorage column for later recall. The service stores all folders and files within the `App_Data` subdirectory, but you could easily change this if you prefer. I selected it simply because the service didn't require any changes to file system permissions to read, write, and delete files in that location.

The data is conveyed to the client using data transfer objects (DTO). I've shown the `ProjectDTO` in Listing 7.11, but the other DTOs are similar.

Listing 7.11: The project data transfer object

```
[DataContract(Namespace = "http://www.endurasoft.com/REST.NET/",
➥ Name="Project")]
public class ProjectDTO
{
    public ProjectDTO()
    {
    }

    public ProjectDTO(Project dbProject) :
        this()
    {
        // Copy properties
        this.ProjectID = dbProject.ProjectID;
        this.DisplayName = dbProject.DisplayName;
    }

    public ProjectDTO(Project dbProject, FolderDTO[] folders) :
        this(dbProject)
    {
        // Copy folders
        this.Folders = folders;
    }

    [DataMember]
    public Int32 ProjectID { get; set; }

    [DataMember]
    public string DisplayName { get; set; }

    [DataMember(IsRequired = false)]
    public string Link { get; set; }

    [DataMember(IsRequired = false)]
    public FolderDTO[] Folders { get; set; }
}
```

Each DTO provides several constructors, including a copy constructor that accepts the corresponding LINQ entity. (Note that another way to approach this is to create extension methods and keep the DTOs in a separate assembly, which has advantages from a code-separation standpoint, and the DTOs wouldn't have to reference LINQ.) The client has similar DTOs I created using the same technique described in the preceding chapter—I queried the service for the appropriate XML and used `xsd.exe` to create a corresponding C# class. I then removed the `XmlSerializer` attributes and replaced them with `DataContractSerializer` attributes. This is cheating a little since I had unfair knowledge as to how the XML was generated, but you could use either serializer as long as you faithfully re-create the XML the service requires.

The `CodeXRC` Service Client

Even though this service will return HTML, to demonstrate creation and deletion of projects, folders, and files, I created a Windows Forms client you can use to experiment with the various aspects of the service. The client won't exercise all aspects of the service. It only adds projects, whereas the service will accept folders and files as well (but keep in mind you'll need to provide correct foreign keys, which is typical of many RESTful services when updating resources).

The basic user interface you've already seen in Figure 7.4. The client also uses a similar authentication dialog box as the preceding chapter's client, so I won't repeat that as well. However, adding a new project does merit some description. The dialog box for this is shown in Figure 7.6.

The Create Project dialog box appears only when XML is selected as the content type, `PUT` is selected as the HTTP method, and you click the Execute button. Other content types for `PUT` display an error dialog box; and other HTTP methods don't involve resource creation, so no error is necessary.

After you provide a project name, you can select folders or files. If you choose to add folders, you'll be presented with the dialog box shown in Figure 7.7.

After you select a folder, the child folders and files will be added to the tree view control shown in the Create Project dialog box. From this tree control you can remove files and folders simply by selecting them and pressing the Delete key.

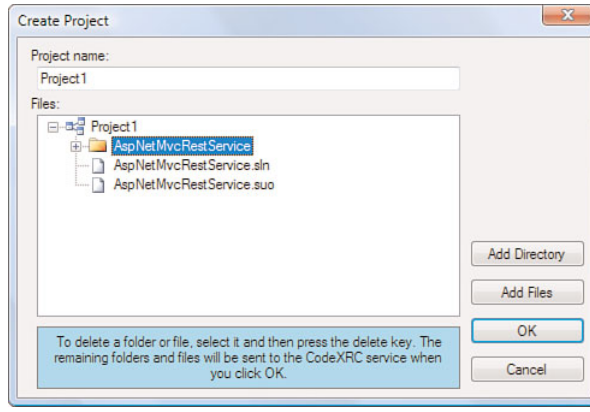


Figure 7.6: The `CodeXRC client` Create Project dialog box

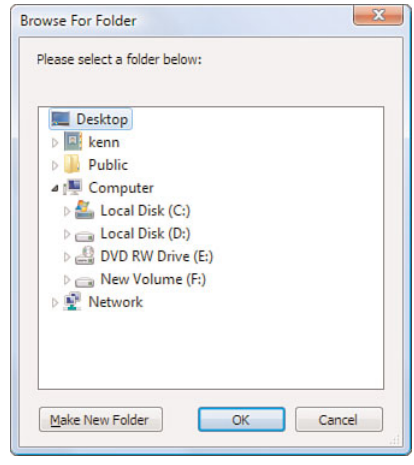


Figure 7.7: The `CodeXRC client` folder browser dialog box

If you instead decide to add files directly to the project, the client creates a virtual folder, with the same name as the project, and adds the files to that. The service doesn't add files directly to a project, but this is simply an implementation detail you could change fairly easily if you wanted. I kept this as a business rule simply to have the rule in play. My reasoning was that projects and folders aren't the same in the service I implemented even though a project is implemented as a folder on disk. Figure 7.8 shows the Create Project dialog box when files are added to the project directly.

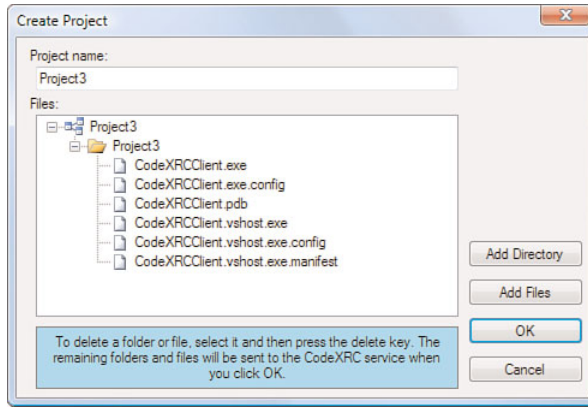


FIGURE 7.8: Adding files directly to a project

In all cases the files are read as binary files and converted into Base64 strings for transmission over the network. Base64, on average, introduces a 30% (or so) increase in size, but it's the only safe way to send binary information over a network using XML. The client will convert the files into Base64 and the service will convert them back into their original binary form for storage; but the client is limited in the sense that it cannot recall the files. I'll leave the implementation of that to you to complete, but the basic conversion logic is already present in the service itself. The service will return to the client the XML representation of a created project, but this is so you could cache the foreign keys if you chose to do so. The contents of the files are wiped from the representation to conserve bandwidth.

Where Are We?

This chapter introduced the ASP.NET MVC framework and how you could create classes based on `ActionResult` to handle RESTful actions and responses. The ASP.NET MVC framework is based on the model-view-controller design pattern, or more accurately on the contemporary front controller pattern. The actions you create are directly mapped to URIs you design and provide to the URL routing subsystem the framework provides. This is a bit different from traditional ASP.NET, in which URLs are mapped directly to disk-based files.

The MVC framework has the capability to differentiate controller actions based on HTTP method by judicious application of the `AcceptVerbs` attribute, and you can disambiguate the action method names using the `ActionName` attribute.

The service logic itself is considered model code. The model in MVC terms is where the data access and business logic for your Web application is properly placed. I elected not to use a view to generate the HTML the service returned, but you certainly could if you chose to do so.

It's at this point you might notice a pattern emerging. When working with pure ASP.NET, you need to provide quite a bit of infrastructure and "glue" logic. With the MVC framework, quite a bit of the infrastructure is provided for you. It's more a matter of understanding what the framework provides and how to best fit your service into that framework.

The next chapter, however, shows you how to use the Windows Communication Foundation (WCF) to implement RESTful services, and by using WCF you implement even less boilerplate infrastructure, allowing you to concentrate nearly entirely on the core service functionality. WCF is also very much more configurable, allowing you to tailor your service's behavior even after it has been deployed.

Index

Symbols

A

B

C

D

E

F

G

H

I

J-K

L

M

N

O

P

Q

R

S

T

U

V

W

X-Y-Z