Microsoft

Programming Microsoft ASP.NET 3.5

Dino Esposito

Table of Contents

Chapter 20. AJAX-Enabled Web Services..... 1
Implementing the AJAX Paradigm..... 2
Web Services for ASP.NET AJAX Applications..... 9
WCF Services for ASP.NET AJAX Applications..... 24
Remote Calls via Page Methods..... 32
Conclusion..... 37

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23

Chapter 20

AJAX-Enabled Web Services

In this chapter:

Implementing the AJAX Paradigm 1006
Web Services for ASP.NET AJAX Applications 1013
WCF Services for ASP.NET AJAX Applications 1028
Remote Calls via Page Methods 1036
Conclusion 1041

The sense of continuity that end users feel when working with AJAX pages, as opposed to the traditional stop-and-go of classic Web pages, is a big step forward for the usability of any application. It is beneficial to users, but it also leads architects and developers to plan more ambitious features and, in the end, to deliver better and richer applications.

At the highest level of abstraction, Web applications are client/server applications that require an Internet connection between the two layers. This connection, though, is incorporated in the special client application—the browser. And the browser clears the user interface before updating the screen with the results of a server operation. To make the usability of Web applications grow as close as possible to that of desktop applications, the overall software platform must fulfill two key requirements. One is a client-side infrastructure that allows for opening and managing the Internet connection with the server. The other requirement is the availability of a public and known programming interface on the server.

The AJAX paradigm is as easy to understand and embrace as it is challenging to implement effectively. The availability of powerful tools and technologies to simplify the development of such applications is a key factor. For the needs of the –server-side of an AJAX application, we do have solid options in the form of ASP.NET Web services and Windows Communication Foundation (WCF) services. As far as the presentation layer is concerned, though, we need to run some client-side code to format data and update the user interface. Because the client is the browser, JavaScript is the only possible programming language. But JavaScript has a number of limitations (performance and leaks in some browser implementations) that make it unfit in scenarios where the client-side is really thick and akin to a smart client. In addition, all that JavaScript can do is target the document object model (DOM). This might not be sufficient to raise the user experience to the level that is required. In a nutshell, this is the big challenge of today's Web platform.

Chapter 20. AJAX-Enabled Web Services
Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23

In this chapter, I'll review the technologies for building an AJAX server architecture for ASP.NET applications. For the presentation layer, I'll stick to JavaScript. In the next chapter, I'll review the opportunities being offered by the Silverlight platform for rich Internet client development.

Implementing the AJAX Paradigm

According to the AJAX paradigm, Web applications work by exchanging data rather than pages with the Web server. An AJAX page sends a request with some input arguments and receives a response with some return values. The code in the client browser orchestrates the operation, gets the data, and then updates the user interface. From a user perspective, this means that faster roundtrips occur and, more importantly, page loading is quicker and the need to refresh the page entirely is significantly reduced.

As a result, a Web application tends to look like a classic desktop Microsoft Windows application. It is allowed to invoke server code from the client and run and control server-side asynchronous tasks, and it features a strongly responsive and nonflickering user interface.

Moving Away from Partial Rendering

As we saw in the previous chapter, partial rendering replaces classic full postbacks with partial postbacks that update only a portion of the requesting page. An AJAX postback is more lightweight than a full postback, but a drawback is that the AJAX postback is still a request that moves view state, event validation data, and any other input fields you might have around the page. Also, the AJAX postback is still a request that goes through the full server-side page life cycle. It differs from a regular ASP.NET postback only because it has a custom rendering phase and, of course, returns only a portion of the whole page markup. Put another way, an AJAX postback is definitely faster and much more beneficial than regular postbacks, but it's still subject to a number of constraints. And, more importantly, it doesn't fit just any scenario. To fully enjoy the benefits of AJAX, we have to move past partial rendering and rewrite our applications to use full-fledged behind-the-scenes asynchronous communication and user interface updates.

The Flip Side of Partial Rendering

This isn't to suggest that partial rendering is inherently bad or wrong. There are two opposite forces that apply to the Web. One is the force of evolution signified by AJAX that is geared toward the adoption of new technologies and patterns. The other is the force of continuity that tends to make things evolve without disrupting the neat flow from past to present. In the case of AJAX, the force of continuity is exemplified by ASP.NET partial rendering.

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23

With ASP.NET partial rendering, there's nothing really new architecturally speaking. It's just the same ASP.NET model revamped through a set of tricky solutions that make page rendering smarter and, more importantly, limited to the fragments of the page that really need a refresh.

Partial rendering has inherent limitations and should be considered a short-term solution for adding AJAX capabilities to legacy applications. This said, we can't ignore that a significant share of today's Web applications just need a bit of facelift to look better and run faster. In this regard, partial rendering is the perfect remedy.

The ASP.NET AJAX Emerging Model

The ASP.NET application model based on postback and view state is, technologically speaking, probably a thing of the past. Of course, this doesn't mean that thousands of pages will be wiped out tomorrow and that hundreds of applications must be rewritten in the upcoming months. More simply, a superior model for Web applications is coming out that is more powerful, both technologically and architecturally, and able to serve today's demand for enhancing and enriching the user experience.

The AJAX emerging model is based on two layers—a client –application layer and a server –application layer. The –client layer sends requests to the –server layer and the server layer sends back responses. Server endpoints are identified through URLs and expose data feeds—mostly JavaScript Object Notation (JSON) data streams—to the client. The –server layer is only a façade that receives calls and forwards them to the business layer of the application. Figure 20-1 depicts the entire model.

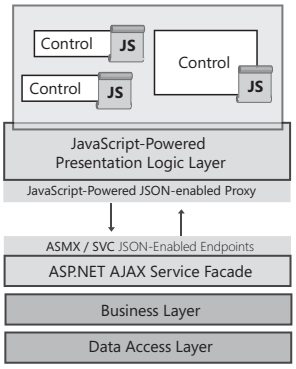

FIGURE 20-1 The ASP.NET AJAX emerging model for Web applications

Let's drill down a bit more in the client layer and –server layer of true AJAX applications for the ASP.NET platform.

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23

Designing the –Client Layer of an ASP.NET AJAX Application

The –client layer manages the user interface and incorporates the presentation logic. How would you code the presentation logic? Using which language or engine? And using which delivery format? For the client layer to really be cross-browser capable, you should use JavaScript and HTML. However, by using JavaScript and HTML you can hardly provide the innovative, immersive, and impressive user experience that many categories of applications loudly demand.

Limits of JavaScript

JavaScript was not designed to back up the presentation logic of Web pages. Originally, it was a small engine added to one of the first versions of Netscape Navigator with the sole purpose of making HTML pages more interactive. Today, it is being used for more ambitious tasks, but it's still nearly the same relatively simple engine created a decade ago.

JavaScript is an interpreted, dynamic-binding, and weakly-typed language with first-class functions. It was influenced by many languages and was designed to look like a simpler form of Java so that it would be easy to use for nonexpert Web page authors.

Worse yet, JavaScript is subject to the browser's implementation of the engine. The result is that the same language feature provides different performance on different browsers and might be flawed on one browser while working efficiently on others. This limitation makes it difficult to write good, cross-browser JavaScript code and justifies the love/hate relationship (well, mostly hate) that many developers have with the language.

Enriching JavaScript

When you move towards AJAX-based architectures, you basically move some of the workload to the client. But on the Web, the client is the browser and JavaScript is currently the sole programming option you have. How can you get a richer and more powerful JavaScript?

At the end of the day, JavaScript has two main drawbacks: it is an interpreted language (significantly slower than a compiled language) and is not fully object oriented. Extending JavaScript is not as easy and affordable as it might seem. Being so popular, any radical change to the language risks breaking a number of applications. But, on the other hand, radical changes are required to meet the upcoming challenges of AJAX.

There exists a proposed standard for JavaScript 2.0 that is discussed in a paper you can download at *http://www.mozilla.org/js/language/evolvingJS.pdf*. And at *http://developer.mozilla.org/presentations/xtech2006/javascript*, you can read Brendan Eich's considerations regarding the feature set in JavaScript 2.0. (Brendan Eich is the inventor of the language.)

It is key to note that the proposed standard is intended, among other things, to achieve better support for programs assembled from components and packaged. Time will tell, however,

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23

if and how JavaScript will undergo a facelift. As a matter of fact, from the AJAX perspective JavaScript is at the same time a pillar of the Web but also one of its key bottlenecks.

Using ad hoc libraries (for example, the Microsoft AJAX library) and widgets (for example, Dojo, Gaia), you can mitigate some of the JavaScript development issues and still deploy richer applications. Honestly, there's not much you can do to improve the performance of heavyweight JavaScript pages.

The real turning point for empowering the Web presentation layer is Silverlight 2.0. Silverlight is a cross-platform browser plug-in that brings a fraction of the power of the common language runtime (CLR) and.NET Framework to the browser, including support for managed languages. I'll cover Silverlight in the next chapter.

Limits of the HTML Markup Language

Today's Web pages use HTML to express their contents. But what's HTML, exactly? Is it a document format? Or is it rather an application delivery format? Or is it none of the above? If you look back at the origins of the Web, you should conclude that HTML is a document format designed to contain information, some images and, more importantly, links to other documents.

Today, we use HTML pages with tables, cascading style sheet (CSS) styles, and lots of images for the pictures they contain and to add compelling separators and rounded borders to otherwise ugly and squared blocks of markup. If you're looking for a document format, HTML is outdated because it lacks a number of composing and packaging capabilities that you find, for example, in the Microsoft Office Open XML formats. If you're looking for an application-delivery format, HTML lacks a rich layout model, built-in graphics, and media capabilities.

Just as for JavaScript, though, getting rid of HTML is not a decision to make with a light heart because HTML is popular and used in a wide variety of applications (not just Web pages). Embedding richer content in a thin HTML wrapper might be a good compromise. And, again, Silverlight with its full support for the XAML language and the Windows Presentation Foundation (WPF) engine is the real turning point.

What About AJAX-Specific Controls?

The programming model of ASP.NET pages based on server controls gained wide acceptance and proved to be quite helpful in the past. How are server controls affected by the aforementioned limitations of JavaScript and HTML, and what's the impact of Silverlight on them?

All in all, server controls are orthogonal to JavaScript, HTML, and even Silverlight. Server controls are the programming tools used to generate the delivery format of the application. They can generate HTML as well as XAML (eXtended Application Markup Language, which is

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23

the language of Silverlight), and they can support JavaScript as well as Silverlight or managed languages.

Today, a number of commercial products exist to take the user interface of Web applications to the next level. They are all made of a collection of rich and smart server controls that generate HTML and JavaScript. This is to say that it's not by using Telerik or ComponentArt or Infragistics that you work around the issues that slow down the implementation of the AJAX paradigm in the today's Web. Rather, more is required.

A set of lower level tools is required to enlarge the browser's capabilities. This can be obtained in two ways: new browser technology or cross-platform browser extensions (for example., plug-ins). The first option is a utopian plan that would take years to be effective. The second option is what you get with Silverlight.

Designing the –Server Layer of ASP.NET AJAX Applications

There are situations in which the partial-rendering model is not appropriate and other situations in which it is just perfect. When the client requires that a specific operation be executed on the server with no frills and in a purely stateless manner, you should consider options other than partial rendering. Enter remote server method calls.

Making a call to a remote server requires that a public, well-known application programming interface (API) be exposed and made accessible from JavaScript or whatever other programming technology you have available in the browser (for example, Silverlight).

As Figure 20-1 shows, the –server layer of an AJAX application is made of services. But which services?

A Service-Oriented –Server-Side Architecture

The –server layer is easy to devise. It is made of services, and on the ASP.NET platform this can only mean XML Web services or WCF services. Hold on, though. You should take the preceding statement literally because the involvement of XML Web services might take you in the wrong direction.

In the context of ASP.NET AJAX, XML Web services are instrumental to the definition of a public, contract-based API that JavaScript (or Silverlight) code can invoke. It doesn't necessarily mean that you can call just any WS-* Web services from an AJAX client. In the context of ASP.NET AJAX, I suggest you think of Web services as a sort of application-specific façade to expose some server-side logic to a JavaScript (or Silverlight) client.

To be invoked from within an ASP.NET AJAX page, the remote service must meet a number of requirements, the strictest of which relate to the location of the endpoint and underlying platform. AJAX-enabled services must be hosted in the same domain from which

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23

the call is made. This means that a Web service must be an ASP.NET XML Web service (an *.asmx* endpoint) and must be hosted in an IIS application on the same Web server as the ASP.NET application.

From the client, you can't just call any Web services on Earth regardless of location and platform. This is a security measure; not a technical limitation.

In summary, there are three ways to define services for the –server layer of an ASP.NET AJAX application:

- ASP.NET XML Web services with an *.asmx* endpoint
- WCF services with an *.svc* endpoint
- Page methods with an *.aspx* endpoint defined on the same page that calls them

In the rest of the chapter, we'll delve deep into these three options.

Note The term "service" is a bit overused and often abused. In AJAX, a service indicates a piece of code that is local to the application (resident on the domain of the application) and exposes functionalities to the client. In the end, services used by AJAX applications tend not to use Simple Object Access Protocol (SOAP) to communicate (they use JSON) and are not necessarily autonomous services in the service-oriented architecture (SOA) sense. They are instead bound to the platform and the domain where they're hosted. Based on this, they can hardly be called WS-* Web services or SOA services.

REST Services

The ideal service for AJAX applications is centered around the idea of data and resources to expose to Web clients. It is reachable over HTTP and requires that clients use URLs (and optionally HTTP headers) to access data and command operations. Clients interact with the service using HTTP verbs such as GET, POST, PUT, and DELETE. Put another way, the URL represents a resource and the HTTP verb describes the action you want to take on the resource. Data exchanged in those interactions is represented in simple formats such as JSON and plain XML, or even in syndication formats such RSS and ATOM.

A service with these characteristics is a Representational State Transfer (REST) service. For more information on the definition of REST, have a look at the original paper that describes the vision behind it. You can find it at the following URL: *http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm*.

Microsoft is currently working on a new generation of data services for the ASP.NET platform that fully embodies REST principles. Such data services are slated to be part of the .NET Framework 3.5 Service Pack 1, which is scheduled to ship sometime in 2008. Meanwhile, you

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23

can get the flavor of it using the latest Community Technology Preview (CTP) of the ASP.NET 3.5 Extensions toolkit.

Data Serialization

Needless to say, the communication between browser and remote services occurs over the HTTP protocol. But what about the content of the packets? An AJAX call consist of some data that is passed as arguments to the invoked service method and some data that is returned as the output. How is this data serialized?

The common serialization format that can be understood on both ends is JavaScript Object Notation (JSON). You can learn more on syntax and purposes of JSON at *http://www.json.org*.

JSON is a text-based format specifically designed to move the state of an object across tiers. It is natively supported by JavaScript in the sense that a JSON-compatible string can be evaluated to a JavaScript object through the JavaScript *eval* function. However, if the JSON string represents the state of a custom object, it's your responsibility to ensure that the definition of the corresponding class is available on the client.

The ASP.NET AJAX network stack takes care of creating JSON strings for each parameter to pass remotely. On the server, ad hoc formatter classes receive the data and use .NET reflection to populate matching managed classes. On the way back, .NET managed objects are serialized to JSON strings and sent over. The script manager is called to guarantee that proper classes referenced in the JSON strings—the Web service proxy class—exist on the client. The nicest thing is that all this machinery is transparent to programmers.

The JSON format describes the state of the object as shown here:

```
{"ID"="ALFKI", "Company":"Alfred Futterkiste"}
```

The string indicates an object with two properties—*ID* and *Company*—and their respective, text-serialized values. If a property is assigned a nonprimitive value—say, a custom object—the value is recursively serialized to JSON.

JSON vs. XML

For years, XML has been touted as the lingua franca of the Web. Now that AJAX has become a key milestone for the entire Web, XML is pushed to the side in favor of JSON as far as data representation over the Web is concerned. Why is that?

Essentially, JSON is slightly simpler and more appropriate for the JavaScript language than XML. Although it might be arguable whether JSON is easier to understand than XML for humans—this is just my thought, by the way—it is certainly easier than XML for a machine to process. No such thing as an XML parser is required for JSON. Everything you need to parse

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23

the text is built into the JavaScript language. JSON is also less verbose than XML and less ambitious too.

JSON is not perfect either. The industrial quantity of commas and quotes it requires makes it a rather quirky format. But can you honestly say that XML is more forgiving?

With JSON, you also gain a key architectural benefit at a relatively low cost. You reason in terms of objects everywhere. On the server, you define your entities and implement them as classes in your favorite managed language. When a service method needs to return an instance of any class, the state of the object is serialized to JSON and travels over the wire. On the client, the JSON string is received and processed, and its contents are loaded into an array, or a kind of mirror JavaScript object, with the same interface as the server class. The interface of the class is inferred from the JSON stream. In this way, both the service and the client page code use the same logical definition of an entity—or, more precisely, of the entity's data transfer object (DTO).

It goes without saying that, from a purely technical standpoint, the preservation of the data contract doesn't strictly require JSON to be implemented. You could get to the same point using XML as well. In that case, though, you need to get yourself an XML parser that can be used from JavaScript.

Parsing some simple XML text in JavaScript might not be an issue, but getting a full-blown parser is another story completely. Performance and functionality issues will likely lead to a proliferation of similar components with little in common. And then you must decide whether such a JavaScript XML parser should support things such as namespaces, schemas, whitespaces, comments, and processing instructions.

As I see it, for the sake of compatibility you will end up with a subset of XML limited to nodes and attributes. At that point, it is merely a matter of choosing between the "angle brackets" of XML and the "curly brackets" of JSON. Additionally, JSON has a free parser already built into the JavaScript engine—the aforementioned function *eval*.

Web Services for ASP.NET AJAX Applications

Let's start examining the steps required to build an ASP.NET AJAX service using an *.asmx* endpoint. The service is part of a server layer that your pages interact with using JavaScript.

Web Services as Application-Specific Services

ASP.NET doesn't just let you call into any SOAP-based Web services from JavaScript. When you take advantage of AJAX extensions for ASP.NET, you use JavaScript to place calls into some server code within the boundaries your own application and domain. In some way, the application server code must be exposed to the client. The way in which this happens

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273 Prepared for Ashwin Srinivas, Safari ID: avsrinivas@gmail.com
Publisher: Microsoft Press
Print Publication Date: 2008/02/23 User number: 300640

depends on the capabilities of the platform. In ASP.NET 2.0 with AJAX Extensions installed, you can rely only on ASP.NET XML Web services local to the application (as modified to return JSON data). In ASP.NET 3.5, you can also employ WCF services. External Web services—those being services outside your application's domain—cannot be invoked directly from the client for security reasons, neither in ASP.NET 2.0 nor ASP.NET 3.5. This is by design.

By default, ASP.NET Web services work by sending and receiving SOAP packets instead of JSON packets and expose their contract using a Web Services Description Language (WSDL) document. What about ASP.NET XML Web services working in the context of an AJAX application?

The *web.config* file of an ASP.NET AJAX application can modify the HTTP handler that receives *.asmx* requests and redirect these calls to an HTTP handler that understands JSON streams. (I'll return to this point with more details later in the chapter.) This means that an ASP.NET XML Web service can be a dual service and can be able to accept and serve both SOAP and JSON requests. Acting at the configuration level, though, you can disable any SOAP support and hide any WSDL file for public discovery of the service functionalities.

And since I'll be referring to JSON-enabled ASP.NET Web services, from this point forward I'll drop the "XML" since we won't be working with SOAP and XML when invoking ASP.NET Web services. ASP.NET Web services for AJAX applications do not use any SOAP messages.

Defining the Remote API

A contract is used to specify what the server-side endpoints expose to callers. If you plan to implement the service as an ASP.NET Web service, an explicit contract is not strictly required. A contract, instead, is mandatory if you opt for a WCF service in ASP.NET 3.5. All in all, designing the public API as an interface produces cleaner code, which is never a bad thing. When you're done with the interface of the server API, you proceed with the creation of a class that implements the interface. Finally, you publish the remote API and let the ASP.NET AJAX runtime manage calls from the client.

For ASP.NET Web services, you define the contract through a plain interface that groups methods and properties for the server API. Here's an example of a simple service that returns the current time on the server:

```
using System;
public interface ITimeService
{
    DateTime GetTime();
    string GetTimeFormat(string format);
}
```

The contract exposes two methods: *GetTime* and *GetTimeFormat*. These methods form the server API that can be called from within the client.

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23

Warning You are on your own when implementing a given interface in an ASP.NET Web service. There's no automatic runtime check to enforce the requirement that exactly those methods are exposed by the server API.

Implementing the Contracted Interface

After you have defined the server API you want to invoke from the client, you implement it in a class and then bind the class to a publicly addressable endpoint.

An ASP.NET Web service is usually implemented through a .NET class that derives from the *WebService* base class. Here's an example:

```
using System.Web.Services;
public class TimeService : WebService, ITimeService
{
    ...
}
```

To direct the Web service to support a given interface, you simply add the interface type to the declaration statement and implement corresponding methods in the body of the class.

Note that deriving from the *WebService* base class is optional and serves primarily to gain the service direct access to common ASP.NET objects, such as *Application* and *Session*. If you don't need direct access to the intrinsic ASP.NET objects, you can still create an ASP.NET Web service without deriving from the *WebService* class. In this case, you can still use ASP.NET intrinsics through the *HttpContext* object.

Publishing the Contract

Now that we have defined the formal contract and implementation of the server API of an ASP.NET AJAX application, one more step is left—publishing the contract. How do you do that?

Publishing the contract means making the server API visible to the JavaScript client page and, subsequently, enabling the JavaScript client page to place calls to the server API. From the client page, you can invoke any object that is visible to the JavaScript engine. In turn, the JavaScript engine sees any class that is linked to the page. In the end, publishing a given server contract means generating a JavaScript proxy class that the script embedded in the page can command.

When the server API is implemented through a Web service, you register the Web service with the script manager control of the ASP.NET AJAX page. In addition, you add a special HTTP handler for *.asmx* requests in the application's *web.config* file.

Let's expand upon the topic of AJAX Web services development by exploring a few examples.

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23

Remote Calls via Web Services

Web services provide a natural environment for hosting server-side code that needs to be called in response to a client action such as clicking a button. The set of Web methods in the service refers to pieces of code specific to the application.

Creating an AJAX Web Service

A Web service made to measure for an ASP.NET AJAX application is similar to any other ASP.NET Web service you might write for whatever purposes. Two peripheral aspects, though, delineate a clear difference between ASP.NET AJAX Web services and traditional ASP.NET XML Web services.

First and foremost, when working with ASP.NET AJAX Web services, you design the contract of an ASP.NET AJAX Web service to fit the needs of a particular application rather than to configure the behavior of a public service. The target application is also the host of the Web service. Second, you must use a new attribute to decorate the class of the Web service that is not allowed on regular ASP.NET XML Web services.

The effect of this is, in the end, that an ASP.NET AJAX Web service might have a double public interface: the JSON-based interface consumed by the hosting ASP.NET AJAX application, and the classic SOAP-based interface exposed to any clients, from any platforms, that can reach the service URL.

The *ScriptService* Attribute

To create an ASP.NET AJAX Web service, you first set up a standard ASP.NET Web service project. Next, you import the *System.Web.Script.Services* namespace:

```
using System.Web.Script.Services;
```

The attribute that establishes a key difference between ASP.NET XML Web services and ASP.NET AJAX Web services is the *ScriptService* attribute. You apply the attribute to the service class declaration, as shown here:

```
namespace Core35.WebServices
{
    [WebService(Namespace = "http://core35.book/")]
    [WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
    [ScriptService]
    public class TimeService : System.Web.Services.WebService, ITimeService
    {
       ...
    }
}
```


Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23

The *ScriptService* attribute indicates that the service is designed to accept calls from JavaScript-based client proxies. If the Web service lacks the attribute, an exception is thrown on the server when you attempt to place calls from your AJAX-enabled client. Figure 20-2 shows the message that is returned when an AJAX page links to a service not flagged with the attribute.

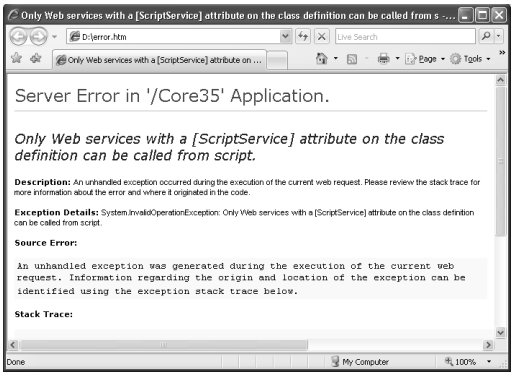

FIGURE 20-2 The result of an ASP.NET AJAX page referencing a nonscriptable service

More precisely, the page shown in the figure is never displayed to any end users. The markup comes with an HTTP 500 error code when access is attempted to a nonscriptable Web service from JavaScript. To get the screen shot in Figure 20-2, I intercepted the HTTP 500 response, saved the body to a local file, and displayed the HTML file in a browser.

The internal ASP.NET machinery refuses to process any calls directed at ASP.NET AJAX Web services that lack the *ScriptService* attribute.

Important You should avoid exposing sensitive pieces of the middle tier to the public without a well-configured security barrier. It is recommended, then, that you add to the Web service only methods that form a sort of user interface–level business logic, where no critical task is accomplished. In addition, you should consider adding a validation layer in the body of Web methods and perhaps using network-level tools to monitor calling IP addresses and, if needed, block some of them. Finally, you can consider SSL/TLS even. That won't preclude unauthorized use, but it helps with snooping when the use is authorized.

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23

Blocking SOAP Clients

Once created, an AJAX Web service is published as an ASMX resource. By default, it's a public URL and can be consumed by AJAX clients, as well as discovered and consumed by SOAP clients and tools. But you can opt to disable SOAP clients and tools altogether. Just enter the following configuration settings to the *web.config* of the ASP.NET application that hosts the service:

```
<webServices>
   <protocols>
      <clear />
   </protocols>
</webServices>
```

This simple setting disables any protocols defined for ASP.NET Web services (in particular, SOAP) and lets the service reply only to JSON requests. Note that with these settings on, you can no longer call the Web service through the browser's address bar for a quick test. Likewise, you can't ask for the WSDL by adding the *?wsdl* suffix to the URL.

Defining Methods for a Web Service

Public methods of the Web service class decorated with the *WebMethod* attribute can be invoked from the client page. Any method is invoked using the HTTP POST verb and return its values as a JSON object. You can change these default settings on a per-method basis by using an optional attribute—*ScriptMethod*.

The *ScriptMethod* attribute features three properties, as described in Table 20-1.

TABLE 20-1 Properties of the *ScriptMethod* Attribute

Property	Description
ResponseFormat	Specifies whether the response will be serialized as JSON or as XML. The default is JSON, but the XML format can come in handy when the return value of the method is an *XmlDocument* object. In this case, because *XMLHttpRequest* has the native ability to expose the response as an XML DOM, using JSON you save unnecessary serialization and deserialization overhead.
UseHttpGet	Indicates whether an HTTP GET verb should be used to invoke the Web service method. The default is *false*, meaning that the POST verb is used. The GET verb poses some security issues, especially when sensitive data is being transmitted. All the data, in fact, is stored in the URL and is visible to everybody.
XmlSerializeString	Indicates whether all return types, including strings, are serialized as XML. The default is *false*. The value of the property is ignored when the response format is JSON.

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23

Because of the repercussions it might have on security and performance, the *ScriptMethod* attribute should be used very carefully. The following code uses the attribute without specifying nondefault settings:

```
[WebMethod]
[ScriptMethod]
public DateTime GetTime()
{
   ...
}
```

The *WebMethod* attribute is required; the *ScriptMethod* attribute is optional. You should use the *ScriptMethod* attribute only when you need to change some of the default settings. In general, you should have very good reasons to use the *ScriptMethod* attribute.

Registering AJAX Web Services

To place calls to an ASP.NET Web service from the client, all that you need is the *XMLHttpRequest* object, the URL of the target Web service, and the ability to manage JSON streams. For convenience, all this functionality is wrapped up in a JavaScript proxy class that mirrors the remote API. The JavaScript proxy is automatically generated by the ASP.NET AJAX framework and injected into the client page.

To trigger the built-in engine that generates any required JavaScript proxy and helper classes, you register the AJAX Web service with the script manager control of each page where the Web service is required. You can achieve this both declaratively and programmatically. Here's how to do it declaratively from page markup:

```
<asp:ScriptManager ID="ScriptManager1" runat="server">
    <Services>
        <asp:ServiceReference Path="~/WebServices/TimeService.asmx" />
    </Services>
</asp:ScriptManager>
```

You add a *ServiceReference* tag for each Web service bound to the page and set the *Path* attribute to a relative URL for the *.asmx* resource. Each service reference automatically produces an extra <*script*> block in the client page. The URL of the script points to a system HTTP handler that, under the hood, invokes the following URL:

```
~/WebServices/TimeService.asmx/js
```

The */js* suffix appended to the Web service URL instructs the ASP.NET AJAX runtime to generate the JavaScript proxy class for the specified Web service. If the page runs in debug mode, the suffix changes to */jsdebug* and a debug version of the proxy class is emitted.

By default, the JavaScript proxy is linked to the page via a <*script*> tag and thus requires a separate download. You can also merge any needed script to the current page by setting the *InlineScript* attribute of the *ServiceReference* object to *true*. The default value of *false* is

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23

helpful if browser caching is enabled and multiple Web pages use the same service reference. In this case, therefore, only one additional request is executed, regardless of how many pages need the proxy class. A value of *true* for the *InlineScript* property reduces the number of network requests at the cost of consuming a bit more bandwidth. This option is preferable when there are many service references in the page and most pages do not link to the same services.

To register AJAX Web services programmatically, you add the following code, preferably in the *Page_Load* event of the page's code-behind class:

```
ServiceReference service = new ServiceReference();
service.Path = "~/WebServices/TimeService.asmx";
ScriptManager1.Services.Add(service);
```

Whatever route you take, to invoke the Web service you need to place a call to the proxy class using JavaScript. The proxy class has the same name as the Web service class and the same set of methods. We'll return to this topic in a moment.

Configuring ASP.NET Applications to Host AJAX Web Services

To enable Web service calls from within ASP.NET AJAX applications, you need to add the following script to the application's *web.config* file and register a special HTTP handler for *.asmx* requests:

```
<httpHandlers>
   <remove verb="*" path="*.asmx" />
   <add verb="*" path="*.asmx"
        type="System.Web.Script.Services.ScriptHandlerFactory" />
   ...
</httpHandlers>
```

This setting is included in the default *web.config* file that Microsoft Visual Studio 2008 creates for you when you create an AJAX-enabled Web project.

A handler factory determines which HTTP handler is in charge of serving a given set of requests. The specialized ASP.NET AJAX Web service handler factory for *.asmx* requests distinguishes JSON calls made by script code from ordinary Web service calls coming from SOAP-based clients, including ASP.NET and Windows Forms applications. JSON-based requests are served by a different HTTP handler, whereas regular SOAP calls take the usual route in the ASP.NET pipeline.

Consuming AJAX Web Services

A referenced ASP.NET AJAX Web service is exposed to the JavaScript code as a class with the same name as the server class, including namespace information. As we'll see in a moment, the proxy class is a singleton and exposes static methods for you to call. No instantiation is

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23

required, which saves time and makes the call trigger more quickly. Let's take a look at the JavaScript proxy class generated from the public interface of an AJAX Web service.

The Proxy Class

To understand the structure of a JavaScript proxy class, we'll consider what the ASP.NET AJAX runtime generates for the aforementioned *timeservice.asmx* Web service. In the following example, the full name of the Web service class is *Core35.WebServices.TimeService*, and therefore it is the name of the JavaScript proxy as well. Here's the first excerpt from the script injected into the client page for the time Web service:

```
Type.registerNamespace('Core35.WebServices');
Core35.WebServices.TimeService = function()
{
   Core35.WebServices.TimeService.initializeBase(this);
   this._timeout = 0;
   this._userContext = null;
   this._succeeded = null;
   this._failed = null;
}
Core35.WebServices.TimeService.prototype =
{
    GetTime : function(succeededCallback, failedCallback, userContext)
    {
        return this._invoke(Core35.WebServices.TimeService.get_path(),
                    'GetTime', false, {}, succeededCallback,
                    failedCallback, userContext);
    },
    GetTimeFormat : function(timeFormat, succeededCallback,
                             failedCallback, userContext)
    {
        return this._invoke(Core35.WebServices.TimeService.get_path(),
                    'GetTimeAsFormat', false, {format:timeFormat},
                    succeededCallback, failedCallback, userContext);
    }
}
Core35.WebServices.TimeService.registerClass(
       Core35.WebServices.TimeService',
       Sys.Net.WebServiceProxy);
Core35.WebServices.TimeService._staticInstance = new Core35.WebServices.TimeService();
```

As you can see from the prototype, the *TimeService* class has two methods—*GetTime* and *GetTimeFormat*—the same two methods defined as Web methods in the server-side Web service class. Both methods have an extended signature that encompasses additional parameters other than the standard set of input arguments (as defined by the server-side methods). In particular, you see two callbacks to call—one for the success of the call, and one for failure—and an object that represents the context of the call. Internally, each method on the proxy class yields to a private member of the parent class—*Sys.Net.WebServiceProxy*—that uses *XMLHttpRequest* to physically send bytes to the server.

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23

The last statement in the preceding code snippet creates a global instance of the proxy class. The methods you invoke from within your JavaScript to execute remote calls are defined around this global instance, as shown here:

```
Core35.WebServices.TimeService.GetTime = function(
          onSuccess,onFailed,userContext)
{
   Core35.WebServices.TimeService._staticInstance.GetTime(
          onSuccess, onFailed, userContext);
}

Core35.WebServices.TimeService.GetTimeFormat = function(
          format, onSuccess, onFailed, userContext)
{
    Core35.WebServices.TimeService._staticInstance.GetTimeFormat(
          format, onSuccess, onFailed, userContext);
}
```

The definition of the proxy class is completed with a few public properties, as described in Table 20-2.

TABLE 20-2 Static Properties on a JavaScript Proxy Class

Property	Description
path	Gets and sets the URL of the underlying Web service
timeout	Gets and sets the duration (in seconds) before the method call times out
defaultSucceededCallback	Gets and sets the default JavaScript callback function to invoke for a successful call
defaultFailedCallback	Gets and sets the default JavaScript callback function, if any, to invoke for a failed or timed-out call
defaultUserContext	Gets and sets the default JavaScript object, if any, to be passed to success and failure callbacks

If you set a "default succeeded" callback, you don't have to specify a "succeeded" callback in any successive call as long as the desired callback function is the same. The same holds true for the failed callback and the user context object. The user context object is any JavaScript object, filled with any information that makes sense to you, that gets automatically passed to any callback that handles the success or failure of the call.

Note The JavaScript code injected for the proxy class uses the *path* property to define the URL to the Web service. You can change the property programmatically to redirect the proxy to a different URL.

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23

Executing Remote Calls

A Web service call is an operation that the page executes in response to a user action such as a button click. Here's the typical way of attaching some JavaScript to a client button click:

```
<input type="button" value="Get Time" onclick="getTime()" />
```

The button, preferably, is a client button, but it can also be a classic server-side *Button* object submit button as long as it sets the *OnClientClick* property to a piece of JavaScript code that returns *false* to prevent the alternative default submit action:

```
<asp:Button ID="Button1" runat="server" Text="Button"

           OnClientClick="getTime();return false;" />
```

The *getTime* function collects any required input data and then calls the desired static method on the proxy class. If you plan to assign default values to callbacks or the user context object, the best place to do it is in the *pageLoad* function. As discussed in Chapter 19, "Partial Rendering: The Easy Way to AJAX," the *pageLoad* function is invoked when the client page ASP.NET AJAX tree has been fully initialized, and precisely because of this it is more reliable than the browser's *onload* event.

```
<script language="javascript" type="text/javascript">
    function pageLoad()
    {
        Core35.WebServices.TimeService.set_defaultFailedCallback(methodFailed);
    }
    function getTime()
    {
        Core35.WebServices.TimeService.GetTimeFormat(
             "ddd, dd MMMM yyyy [hh:mm:ss]", methodComplete);
    }
    function methodComplete(results, context, methodName)
    {
        $get("Label1").innerHTML = results;
    }
    function methodFailed(errorInfo, context, methodName)
    {
        $get("Label1").innerHTML = String.format(
               "Execution of method '{0}' failed because of the
                following:\r\n'{1}'",
               methodName, errorInfo.get_message());
    }
</script>
```

Because the Web service call proceeds asynchronously, you need callbacks to catch up both in the case of success and failure. The signature of the callbacks is similar, but the internal format of the results parameter can change quite a bit:

```
function method(results, context, methodName)
```


Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23

Table 20-3 provides more details about the various arguments.

TABLE 20-3 Arguments for JavaScript Web Service Callback Functions

Argument	Description
results	Indicates the return value from the method in the case of success. In the case of failure, a JavaScript *Error* object mimics the exception that occurred on the server during the execution of the method.
context	The user context object passed to the callback.
methodName	The name of the Web service method that was invoked.

Based on the previous code, if the call is successful the *methodCompleted* callback is invoked to update the page. The result is shown in Figure 20-3.

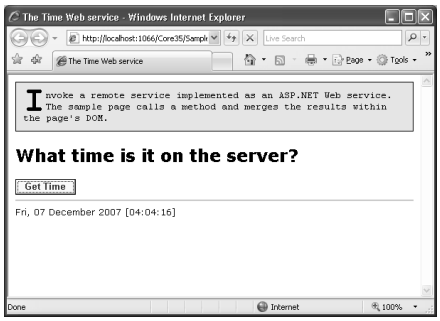

FIGURE 20-3 A remote call made from the client

Error Handling

The "failed" callback kicks in when an exception occurs on the server during the execution of the remote method. In this case, the HTTP response contains an HTTP 500 error code (internal error) and the body of the response looks like the following:

```
{"message":"Exception thrown for testing purposes",
 "stackTrace":"  at Core35.WebServices.MyDataService.Throw() in
              d:\\Core35\\App_Code\\Services\\MyDataService.cs:line
              62","ExceptionType":"System.InvalidOperationException"}
```

On the client, the server exception is exposed through a JavaScript *Error* object dynamically built based on the message and a stack trace received from the server. This *Error* object is exposed to the "failed" callback via the results argument. You can read back the message and stack trace through *message* and *stackTrace* properties on the *Error* object.

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23

You can use a different error handler callback for each remote call, or you can designate a default function to be invoked if one is not otherwise specified. However, ASP.NET AJAX still defines its own default callback, which is invoked when it gets no further information from the client developer. The system-provided error handler callback simply pops up a message box with the message associated with the server exception. If you define your own "failed" callback, you can avoid message boxes and incorporate any error message directly in the body of the page.

Giving User Feedback

A remote call might take a while to complete because the operation to execute is fairly heavy or just because of the network latency. In any case, you might feel the need to show some feedback to the user to let her know that the system is still working. In the previous chapter, we saw that the Microsoft AJAX library has a built-in support for an intermediate progress screen and also a client-side eventing model. Unfortunately, this functionality is limited to calls that originate within updatable panels. For classic remote method calls, you have to personally take care of any user feedback.

You bring up the wait message, the animated GIF, or whatever else you need just before you call the remote method:

```
function takeaWhile()
{
   // In this example, the Feedback element is a <span> tag
   $get("Feedback").innerHTML = "Please, wait ...";
   Core35.WebServices.MySampleService.VeryLengthyTask(
        methodCompletedWithFeedback, methodFailedWithFeedback);
}
```

In the "completed" callback, you reset the user interface first and then proceed:

```
function methodCompletedWithFeedback(results, context, methodName)
{
    $get("Feedback").innerHTML = "";
    ...
}
```

Note that you should also clear the user interface in the case of errors. In addition to showing some sort of wait message to the user, you should also consider that other elements in the page might need to be disabled during the call. If this is the case, you need to disable them before the call and restore them later.

Handling Timeouts

A remote call that takes a while to complete is not necessarily a good thing for the application. Keep in mind that calls that work asynchronously for the client are not necessarily asynchronous for the ASP.NET runtime. In particular, note that when you make a client call to an

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23

.asmx Web service, you are invoking the *.asmx* directly. For this request, only a synchronous handler is available in the ASP.NET runtime. This means that regardless of how the client perceives the ongoing call, an ASP.NET thread is entirely blocked (waiting for results) until the method is done. To mitigate the impact of lengthy AJAX methods on the application scalability, you can set a timeout:

```
Core35.WebServices.MySampleService.set_timeout(3000);
```

The *timeout* attribute is global and applies to all methods of the proxy class. This means that if you want to time out only one method call, you have to reset the timeout for all calls you're making from the page. To reset the timeout, you just set the timeout property to zero:

```
Core35.WebServices.MySampleService.set_timeout(0);
```

When the request times out, there's no response received from the server. It's simply a call that is aborted from the client. After all, you can't control what's going on with the server. The best you can do is abort the request on the client and take other appropriate measures, such as having the user try again later.

Considerations for AJAX-Enabled Web Services

Now that we know how to tackle AJAX-enabled Web services, it would be nice to spend some time reflecting on some other aspects of them—for example, why use local services?

Why Local Web Services?

To make sure you handle AJAX Web services the right way, think of them as just one possible way of exposing a server API to a JavaScript client. You focus on the interface that must be exposed and then choose between ASP.NET Web services, WCF services, and page methods for its actual implementation. Looking at it from this angle, you might find it to be quite natural that the Web service has to be hosted in the same ASP.NET AJAX application that is calling it.

But why can't you just call into any SOAP-based Web services out there? There are two main reasons: security and required support for JSON serialization.

For security reasons, browsers tend to stop script-led cross-site calls. (Not all scripts are benevolent.) Most browsers bind scripted requests to what is often referred to as the "same origin policy." Defined, it claims that no documents can be requested via script that have a different port, server, or protocol than the current page. In light of this, you can use the *XMLHttpRequest* object to place asynchronous calls as long as your request hits the same server endpoint that served the current page.

Because of the cross-site limitations of *XMLHttpRequest* in most browsers, ASP.NET doesn't allow you to directly invoke a Web service that lives on another IIS server or site. Without this

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23

limitation, nothing would prevent you from invoking a Web service that is resident on any platform and Web server environment, but then your users are subject to potential security threats from less scrupulous applications. With this limitation in place, though, an additional issue shows up: the inability of your host ASP.NET AJAX environment to build a JavaScript proxy class for the remote, non–ASP.NET AJAX Web service.

Note Because of the impact that blocked cross-site calls have on general AJAX development, a new standard might emerge in the near future to enable such calls from the browser. It might be desirable that the client sends the request and dictates the invoked server accept or deny cross-site calls made via *XMLHttpRequest*. As of this writing, though, the possibility of direct cross-site calls from AJAX clients (not just ASP.NET AJAX Extensions) remains limited to the use of IFRAMEs and finds no built-in support in ASP.NET 3.5.

Why JSON-Based Web Services?

A call to a Web service hosted by the local ASP.NET AJAX application is not conducted using SOAP as you might expect. SOAP is XML-based, and parsing XML on the client is very expensive in terms of memory and processing resources. It means that an XML parser must be available in JavaScript, and an XML parser is never an easy toy to build and manage, especially using a relatively lightweight tool such as JavaScript. So a different format is required to pack messages to be sent and unpack messages just received. Like SOAP and XML schemas together, though, this new format must be able to serialize an object's public properties and fields to a serial text-based format for transport. The format employed by ASP.NET AJAX Web services is JSON.

The client-side ASP.NET AJAX network stack takes care of creating JSON strings for each parameter to pass remotely. The JavaScript class that does that is called *Sys.Serialization.JavaScriptSerializer*. On the server, ad hoc formatter classes receive the data and use .NET reflection to populate matching managed classes. On the way back, .NET managed objects are serialized to JSON strings and sent over. The script manager is called to guarantee that proper classes referenced in the JSON strings—the Web service proxy class—exist on the client.

Runtime Support for JSON-Based Web Services

As a developer, you don't necessarily need to know much about the JSON format. You normally don't get close enough to the heart of the system to directly manage JSON strings. However, a JSON string represents an object according to the following sample schema:

```
{
 "__type":"IntroAjax.Customer",
 "ID":"ANATR",
 "ContactName":"Ana Trujillo"
 ...
}
```


Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23

You'll find a number of comma-separated tokens wrapped in curly brackets. Each token is, in turn, a colon-separated string. The left part, in quotes, represents the name of the property; the right part, in quotes, represents the serialized version of the property value. If the property value is not a primitive type, it gets recursively serialized via JSON. If the object is an instance of a known type (that is, it is not an untyped JavaScript associative array), the class name is inserted as the first piece of information associated with the *__type* property. Any information being exchanged between an ASP.NET AJAX client and an ASP.NET AJAX Web service is serialized to the JSON format.

To the actual Web service, the transport format is totally transparent—be it SOAP, JSON, plain-old XML (POX), or whatever else. The runtime infrastructure takes care of deserializing the content of the message and transforms it into valid input for the service method. The ASP.NET AJAX runtime recognizes a call directed to an AJAX Web service because of the particular value of the *Content-Type* request header. Here's an excerpt from the Microsoft AJAX client library where the header is set:

```
request.get_headers()['Content-Type'] = 'application/json; charset=utf-8';
```

The value of this header is used to filter incoming requests and direct them to the standard ASP.NET XML Web service HTTP handler or to the made-to-measure ASP.NET AJAX Web service handler that will do all the work with the JSON string.

WCF Services for ASP.NET AJAX Applications

Starting with ASP.NET 3.5, you can use Windows Communication Foundation to build AJAX-callable services. Overall, the developer's experience is mostly the same whether you use ASP.NET Web services or WCF services. Instead, the richness of the WCF platform—specifically designed to generate and support software services—is a no-brainer. A good question, then, is, "Why weren't WCF services available in ASP.NET AJAX pages before ASP.NET 3.5?"

Before the availability of the .NET Framework 3.5, the WCF platform had no built-in support for taking JSON as input and returning it as output. So what's does the .NET Framework 3.5 really do in the area of WCF? It basically empowers WCF to support JSON serialization. Now WCF services can optionally output JSON, and not always SOAP envelopes, over HTTP. All that you have to do is configure an endpoint to use the *webHttpBinding* binding model and enable Web scripting through a new attribute. I'll provide more detail about this in a moment.

Building a Simple WCF Service

Having always been a huge fan of the bottom-up approach to things, I just can't learn anything new without first testing it in a very simple scenario that then evolves as quickly as possible into a more realistic one. So let's simply create a new Web site in Visual Studio 2008, click to add a new WCF service item, and name the item *TimeService*.

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273 Prepared for Ashwin Srinivas, Safari ID: avsrinivas@gmail.com
Publisher: Microsoft Press
Print Publication Date: 2008/02/23 User number: 300640

Rewriting *TimeService* as a WCF Service

After confirming the operation, you find your project extended with a service endpoint (say, *timeservice.svc*) and its related code-behind file placed in the *App_Code* folder—say, *wcftimeservice.cs*. In addition, the *web.config* file is modified, too, to provide registration and discovery information for the service being created.

You might want to define the contract for the service by using an interface. This is not strictly required, but it is helpful and also gives you the possibility of implementing multiple contracts in the same actual class.

```
namespace Core35.Services.Wcf
{
    // Contract
    [ServiceContract(Namespace = “Core35.Services”, Name=”WcfTimeService”)]
    public interface ITimeService
    {
        [OperationContract]
        DateTime GetTime();

        [OperationContract]
        string GetTimeFormat(string format);
    }
}
```

In the example, the *ITimeService* interface represents the contract of the service. The *ServiceContract* attribute marks the contract, whereas the *OperationContract* attribute indicates methods. In simpler cases, you can just define a class that is both the contract and implementation. If you do so, you use the *ServiceContract* and *OperationContract* attributes directly in the class.

Pay attention to the *Namespace* and *Name* properties of the *ServiceContract* attribute. They gain additional importance in AJAX-enabled WCF services, as we'll see in a moment. The following code shows a class that implements the *ITimeService* contract:

```
using System;
using System.Runtime.Serialization;
using System.ServiceModel;
using System.ServiceModel.Activation;
using System.ServiceModel.Web;

namespace Core35.Services.Wcf
{
   [AspNetCompatibilityRequirements(
          RequirementsMode=AspNetCompatibilityRequirementsMode.Allowed)]
   public class TimeService : ITimeService
   {
      public DateTime GetTime()
      {
            return DateTime.Now;
      }
```


Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23

```
        public string GetTimeFormat(string format)
        {
            return DateTime.Now.ToString(format);
        }
    }
}
```

In the end, the *TimeService* class exposes a couple of public endpoints—named *GetTime* and *GetTimeFormat*.

Registering the Service

The endpoint to reach the methods on this interface are defined in an SVC file, like the *timeservice.svc* file shown next:

```
<%@ ServiceHost Debug="true"
                Service="Core35.Services.Wcf.TimeService"
                CodeBehind="~/App_Code/WcfTimeService.cs" %>
```

The service host, whether it is running in debug or release mode, indicates the location of the source files and the type that implements the service. If the code for the service is placed inline in the SVC file, you must indicate an additional *Language* attribute.

The final step before you can test the service is registering its usage in the *web.config* file of the host ASP.NET application. Here's what you need to have:

```
<system.serviceModel>
   <behaviors>
     <endpointBehaviors>
        <behavior name="TimeServiceAspNetAjaxBehavior">
           <enableWebScript />
        </behavior>
     </endpointBehaviors>
   </behaviors>
   <serviceHostingEnvironment aspNetCompatibilityEnabled="true" />
   <services>
      <service name="Core35.Services.Wcf.TimeService">
         <endpoint address=""
                   behaviorConfiguration="TimeServiceAspNetAjaxBehavior"
                   binding="webHttpBinding"
                   contract="Core35.Services.Wcf.ITimeService" />
      </service>
   </services>
</system.serviceModel>
```

First, you register the list of behaviors for endpoints. In doing so, you define a behavior for your service—named *TimeServiceAspNetAjaxBehavior*—and state that it accepts requests from the Web via script. The *enableWebScript* element is logically equivalent to the *ScriptService* attribute you use to decorate a Web service class for the same purpose.

Next, you list the services hosted by the current ASP.NET application. This preceding *web.config* file has just one service coded in the class *Core35.Services.Wcf.TimeService* with one

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23

endpoint using the *ITimeService* contract and the *webHttpBinding* binding model. The *name* attribute must be set to the type of the class implementing the service.

Testing the Service

The service is pretty much all set. How would you use it from the <*script*> section of a client ASP.NET page? The steps required from a developer aren't much different from those required to invoke a Web service. You start by registering the service with the script manager using the SVC endpoint:

```
<asp:ScriptManager ID="ScriptManager1" runat="server">
    <Services>
        <asp:ServiceReference Path="~/TimeService.svc" />
    </Services>
</asp:ScriptManager>
```

When processing the markup, the *ScriptManager* control triggers additional requests to generate and download the JavaScript proxy class for the specified WCF service. The client page uses the proxy class to place calls.

The proxy class is named after the *Namespace* and *Name* properties of the *ServiceContract* attribute of the contract. If you leave the parameters to their default values, the JavaScript proxy class is named *Tempuri.org.ITimeService*, where *Tempuri.org* is the default namespace and the interface name is the default name of the contract.

There's no relationship between the name of the service class and JavaScript proxy class, not even when you use the same class to provide both the contract and implementation of the service. The name of the JavaScript proxy class depends on namespace and name of the service contract. It is common to assign the namespace a unique URI, such as *http://www.Core35-Book.com*. In this case, the name of the proxy class comes out a bit mangled, like *http.www.Core35Book.com*. It is recommended, therefore, that you use plain strings to name the namespace of a contract being used in an AJAX-enabled WCF service.

Let's assume the following, instead:

```
[ServiceContract(Namespace = "Core35.Services", Name="WcfTimeService")]
```

In this case, the following JavaScript can be used to invoke the method *GetTimeFormat*:

```
<script language="javascript" type="text/javascript">
    function getTime()
    {
        Core35.Services.WcfTimeService.GetTimeFormat(
             "dd-mm-yyyy [hh:mm:ss]", onMethodCompleted);
    }

    function onMethodCompleted(results)
    {
        $get("lblCurrentTime").innerText = results;
    }
</script>
```


Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23

```
<form id="Form1" runat="server">
    <asp:ScriptManager ID="ScriptManager1" runat="server">
        <Services>
            <asp:ServiceReference Path="~/Services/TimeService.svc" />
        </Services>
    </asp:ScriptManager>

    <h1>What time is on the server? Set your clock...</h1>
    <input type="button" value="Get time" onclick="getTime()" />
    <hr />
    <asp:Label runat="server" ID="lblCurrentTime" />
</form>
```

The JavaScript proxy class is made of static methods whose name and signature match the prototype of the WCF service endpoints. In addition, and like the ASP.NET AJAX Web services, each JavaScript proxy method supports a bunch of additional parameters—callback functions to handle the success or failure of the operation.

ASP.NET Compatibility Modes

When you create a new WCF service for ASP.NET AJAX, the service class is also decorated by default by the *AspNetCompatibilityRequirements* attribute, which deserves a few words of its own.

```
[AspNetCompatibilityRequirements(
        RequirementsMode=AspNetCompatibilityRequirementsMode.Allowed)]
public class TimeService : ITimeService
{
    :
}
```

Although they are designed to be transport independent, when WCF services are employed in the context of an ASP.NET AJAX application, they might actually work in a manner very similar to ASMX services. By using the *AspNetCompatibilityRequirements* attribute, you state your preference of having WCF and ASMX services work according to the same model. One practical repercussion of this setting is that when a WCF service is activated, the runtime checks declared endpoints and ensures that all of them use the Web HTTP binding model.

Their compatibility with ASMX services enables WCF services to access, for example, the *HttpContext* object and subsequently other ASP.NET intrinsic objects. The compatibility is required at two levels. The first level is in the *web.config* file, where you use the following:

```
<system.serviceModel>
  :
  <serviceHostingEnvironment aspNetCompatibilityEnabled="true" />
</system.serviceModel>
```

Second, developers need to explicitly choose the compatibility mode for a given WCF service by using the service *AspNetCompatibilityRequirements* attribute.

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23

Building a Less Simple Service

In Chapter 19, we discussed the *AutoComplete* extender to provide suggestions to users typing into a text box. The extender calls into a service to receive an array of suggestions. The service can be either a scriptable Web service or an AJAX-enabled WCF service. Let's see what it takes to create a helper WCF service for the *AutoComplete* extender.

The *Suggestions* Service

A service for the auto-complete extender can have any number of operations, but all have the same prototype. Here's a possible contract:

```
[ServiceContract(Namespace = "Core35.Services", Name = "SuggestionService")]
public interface ISuggestionService
{
    [OperationContract]
    string[] GetCustomerNames(string prefixText, int count);

    [OperationContract]
    string[] GetCustomerIDs(string prefixText, int count);
}
```

The implementation contains the code to query for customer names and ID and, optionally, for some server-side caching. Here's a fragment of the service class:

```
[AspNetCompatibilityRequirements(
     RequirementsMode = AspNetCompatibilityRequirementsMode.Allowed)]
public class SuggestionService : ISuggestionService
{
    public string[] GetCustomerIDs(string prefixText, int count)
    {
        int i=0;
        DataView data = GetData();
        data = FilterDataByID(data, prefixText);
        string [] suggestions = new string[data.Count];

        foreach (DataRowView row in data) {
           suggestions[i++] = row["customerID"].ToString();
        }

        return suggestions;
    }

    // Other methods here
    ...
}
```

The next step is creating the endpoint for the service. Let's call it *suggestions.svc*:

```
<%@ ServiceHost
    Service="Core35.Services.Wcf.SuggestionService"
    CodeBehind="~/App_Code/WcfSuggestionService.cs" %>
```


Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23

At this point, you link the auto-complete extender to the WCF service, as shown next:

```
<act:AutoCompleteExtender runat="server" ID="AutoCompleteExtender1"
     ...
     ServicePath="~/Services/Suggestions.svc"
     ServiceMethod="GetCustomerIDs" />
```

One more step is left—publishing the contract in a service host. For an AJAX-enabled WCF service, the host is Microsoft Internet Information Services (IIS). However, you still need to publish the endpoint.

Service Without Configuration

Publishing a given contract means binding the contract to a public endpoint. This usually requires adding a new <*service*> block in the *web.config* file under the <*services*> section, as shown here:

```
<services>
      <service name="Core35.Services.Wcf.SuggestionService">
         <endpoint behaviorConfiguration="StandardServiceAspNetAjaxBehavior"
                   binding="webHttpBinding"
                   contract="Core35.Services.Wcf.ISuggestionService" />
      </service>
</services>
```

The key thing to notice is that AJAX-enabled WCF services can also be deployed without configuration. All that you have to do is add a new *Factory* attribute to the *@ServiceHost* directive in the SVC file:

```
<%@ ServiceHost
    Factory="System.ServiceModel.Activation.WebScriptServiceHostFactory"
    Service="Core35.Services.Wcf.SuggestionService"
    CodeBehind="~/App_Code/WcfSuggestionService.cs" %>
```

By taking this approach, you do not need to make changes in the *web.config* file, and creating a WCF service for AJAX pages is as easy as creating the class and defining the endpoint.

Data Contracts

Any nonprimitive data to be sent or received over WCF methods must be marked with the *DataContract* attribute. Imagine you have the following service:

```
[ServiceContract(Namespace = "Core35.Services.Wcf")]
[AspNetCompatibilityRequirements(
        RequirementsMode = AspNetCompatibilityRequirementsMode.Allowed)]
public class CustomerService
{
    [OperationContract]
    public CustomerDTO LookupCustomer(string id)
    {
```


Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23

```
        NorthwindDataContext context = new NorthwindDataContext();
        var data = (from c in context.Customers
                    where c.CustomerID == id
                    select c).SingleOrDefault();
       if (data != null)
       {
            CustomerDTO dto = new CustomerDTO((Customer)data);
            return dto;
       }
       else
            return null;
    }
}
```

The method *LookupCustomer* is expected to return a custom object. This object must be decorated with ad hoc *DataContract* attributes:

```
namespace Core35.Services.Wcf
{
    [DataContract]
    public class CustomerDTO
    {
        private Customer _c;
        public CustomerDTO(Customer c)
        {
            this._c = c;
        }

        [DataMember]
        public string CustomerID
        {
            get { return _c.CustomerID; }
            set { _c.CustomerID = value; }
        }

        ...
    }
}
```

In this particular case, the class being used over WCF is a data-transfer object (DTO)—that is, a helper class that moves the content of domain model objects across tiers.

Note Could you directly use the *Customer* class obtained from Linq-to-SQL? Yes, as long as the class and its members are flagged with the *DataContract* and *DataMember* attributes. Linq-to-SQL classes as generated by the Visual Studio 2008 O/R designer are kind of anemic objects (only data, no behavior) and, as such, they are just perfect as DTOs. However, you need to make sure that they contain the right *DataContract* and *DataMember* attributes. You can add these attributes automatically by setting the *SerializationMode* property of the data context class to *Unidirectional*. (See Chapter 10, "The Linq-to-SQL Programming Model.")

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23

Remote Calls via Page Methods

As we've seen, Web and WCF services are simple and effective ways of implementing a server API. When the ASP.NET AJAX runtime engine has generated the proxy class, you're pretty much done and can start calling methods as if they were local to the client. Web and WCF services, though, are not free of issues. They require an extra layer of code and additional files or assembly references to be added to the project. Is this a big source of concern for you? If so, consider that you have an alternative—page methods.

Introducing Page Methods

Page methods are simply public, static methods exposed by the code-behind class of a given page and decorated with the *WebMethod* attribute. The runtime engine for page methods and AJAX-enabled Web services is nearly the same. Using page methods saves you from the burden of creating and publishing a service; at the same time, though, it binds you to having page-scoped methods that can't be called from within a page different from the one where they are defined. We'll return later to the pros and cons of page methods. For now, let's just learn more about them.

Defining a Page Method

Public and static methods defined on a page's code-behind class and flagged with the *WebMethod* attribute transform an ASP.NET AJAX page into a Web service. Here's a sample page method:

```
public class TimeServicePage : System.Web.UI.Page
{
    [WebMethod]
    public static DateTime GetTime()
    {
        return DateTime.Now;
    }
}
```

You can use any data type in the definition of page methods, including .NET Framework types as well as user-defined types. All types will be transparently JSON-serialized during each call.

Note The page class where you define methods might be the direct code-behind class or, better yet, a parent class. In this way, in the parent class you can implement the contract of the public server API and keep it somewhat separated from the rest of event handlers and methods that are specific to the page life cycle and behavior. Because page methods are required to be *static* (*shared* in Microsoft Visual Basic .NET), you can't use the syntax of interfaces to define the contract. You have to resort to abstract base classes.

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23

Alternatively, you can define Web methods as inline code in the *.aspx* source file as follows (and if you use Visual Basic, just change the type attribute to *text/VB*):

```
<script type="text/C#" runat="server">
    [WebMethod]
    public static DateTime GetTime()
    {
        return DateTime.Now;
    }
</script>
```

Note that page methods are specific to a given ASP.NET page. Only the host page can call its methods. Cross-page method calls are not supported. If they are critical for your scenario, I suggest that you move to using Web or WCF services.

Enabling Page Methods

When the code-behind class of an ASP.NET AJAX page contains *WebMethod*-decorated static methods, the runtime engine emits a JavaScript proxy class nearly identical to the class generated for a Web service. You use a global instance of this class to call server methods. The name of the class is hard-coded to *PageMethods*. We'll return to the characteristics of the proxy class in a moment.

Note, however, that page methods are not enabled by default. In other words, the *PageMethods* proxy class that you use to place remote calls is not generated unless you set the *EnablePageMethods* property to *true* in the page's script manager:

```
<asp:ScriptManager runat="server" ID="ScriptManager1" EnablePageMethods="true" />
```

For the successful execution of a page method, the ASP.NET AJAX application must have the *ScriptModule* HTTP module enabled in the *web.config* file:

```
<httpModules>
  <add name="ScriptModule"
     type="System.Web.Handlers.ScriptModule, System.Web.Extensions" />
</httpModules>
```

Among other things, the module intercepts the application event that follows the loading of the session state, executes the method, and then serves the response to the caller. Acquiring session state is the step that precedes the start of the page life cycle. For page method calls, therefore, there's no page life cycle and child controls are not initialized and processed.

Why No Page Life Cycle?

In the early days of ASP.NET AJAX (when it was code-named Atlas), page methods were instance methods and required view state and form fields to be sent with every call. The sent view state was the last known good view state for the page—that is, the view state downloaded to the client. It was common for developers to expect that during the page

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23

method execution, say, a *TextBox* was set to the same text just typed before triggering the remote call. Because the sent view state was the last known good view state, that expectation was just impossible to meet. At the same time, a large share of developers was also complaining that the view state was being sent at all during page method calls. View state is rarely small, which serves to increase the bandwidth and processing requirements for handling page methods.

In the end, ASP.NET AJAX extensions require static methods and execute them just before starting the page life cycle. The page request is processed as usual until the session state is retrieved. After that, instead of the page method call going through the page life cycle, the HTTP module kicks in, executes the method via reflection, and returns.

Coded in this way, the execution of a remote page method is quite effective and nearly identical to having a local Web service up and running. The fact that static methods are used and no page life cycle is ever started means one thing to you—you can't programmatically access page controls and their properties.

Consuming Page Methods

The collection of page methods is exposed to the JavaScript code as a class with a fixed name—*PageMethods*. The schema of this class is similar to the schema of proxy classes for AJAX-enabled Web services. The class lists static methods and doesn't require any instantiation on your own. Let's take a look at the *PageMethods* class.

The Proxy Class

Unlike the proxy class for Web services, the *PageMethods* proxy class is always generated as inline script in the body of the page it refers to. That's a fairly obvious choice given the fixed naming convention in use; otherwise, the name of the class should be different for each page. Here's the source code of the *PageMethods* class for a page with just one Web method, named *GetTime*:

```
<script type="text/javascript">
var PageMethods = function()
{
    PageMethods.initializeBase(this);
    this._timeout = 0;
    this._userContext = null;
    this._succeeded = null;
    this._failed = null;
}
PageMethods.prototype =
{
    GetTime:function(succeededCallback, failedCallback, userContext)
            {
               return this._invoke(PageMethods.get_path(),
                           'GetTime', false, {}, succeededCallback,
                           failedCallback, userContext);
            }
```


Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23

```
}
PageMethods.registerClass('PageMethods', Sys.Net.WebServiceProxy);
PageMethods._staticInstance = new PageMethods();

PageMethods.set_path = function(value) {
     var e = Function._validateParams(arguments,
                                      [{name: 'path', type: String}]);
     if (e) throw e;
     PageMethods._staticInstance._path = value;
}
PageMethods.get_path = function() {
    return PageMethods._staticInstance._path;
}
PageMethods.set_timeout = function(value) {
    var e = Function._validateParams(arguments,
                                     [{name: 'timeout', type: Number}]);
    if (e) throw e;
    if (value < 0)
       throw Error.argumentOutOfRange('value', value,
                                      Sys.Res.invalidTimeout);
    PageMethods._staticInstance._timeout = value;
}
PageMethods.get_timeout = function() {
    return PageMethods._staticInstance._timeout;
}
PageMethods.set_defaultUserContext = function(value) {
    PageMethods._staticInstance._userContext = value;
}
PageMethods.get_defaultUserContext = function() {
    return PageMethods._staticInstance._userContext;
}
PageMethods.set_defaultSucceededCallback = function(value) {
    var e = Function._validateParams(arguments,
                    [{name: 'defaultSucceededCallback', type: Function}]);
    if (e) throw e;
    PageMethods._staticInstance._succeeded = value;
}
PageMethods.get_defaultSucceededCallback = function() {
    return PageMethods._staticInstance._succeeded;
}
PageMethods.set_defaultFailedCallback = function(value) {
    var e = Function._validateParams(arguments,
        [{name: 'defaultFailedCallback', type: Function}]);
    if (e) throw e;
    PageMethods._staticInstance._failed = value;
}
PageMethods.get_defaultFailedCallback = function() {
    return PageMethods._staticInstance._failed;
}

PageMethods.set_path("/Core35/Ch20/CallPageMethod.aspx");
PageMethods.GetTime = function(onSuccess,onFailed,userContext) {
   PageMethods._staticInstance.GetTime(onSuccess,onFailed,userContext);
}
</script>
```


Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23

As you can see, the structure of the class is nearly identical to the proxy class of an AJAX Web service. You can define default callbacks for success and failure, user context data, path, and timeout. A singleton instance of the *PageMethods* class is created, and all callable methods are invoked through this static instance. No instantiation whatsoever is required.

Executing Page Methods

The *PageMethods* proxy class has as many methods as there are Web methods in the code-behind class of the page. In the proxy class, each mapping method takes the same additional parameters you would find with a Web service method: completed callback, failed callback, and user context data. The completed callback is necessary to update the page with the results of the call. The other parameters are optional. The following code snippet shows a locally-defined *getTime* function bound to a client event handler. The function calls a page method and leaves the *methodCompleted* callback the burden of updating the user interface as appropriate.

```
function getTime()
{
    PageMethods.GetTime(methodCompleted);
}
function methodCompleted(results, context, methodName)
{
    // Format the date-time object to a more readable string
    var displayString = results.format("ddd, dd MMMM yyyy");
    $get("Label1").innerHTML = displayString;
}
```

The signature of a page method callback is exactly the same as the signature of an AJAX Web service proxy. The role of the *results*, *context*, and *methodName* parameters is the same as described in Table 20-3.

Timeout, error handling, and user feedback are all aspects of page methods that require the same programming techniques discussed earlier for Web service calls.

Note From page methods, you can access session state, the ASP.NET *Cache*, and *User* objects, as well as any other intrinsic objects. You can do that using the *Current* property on *HttpContext*. The HTTP context is not specific to the page life cycle and is, instead, a piece of information that accompanies the request from the start.

Page Methods vs. AJAX-Enabled Services

From a programming standpoint, no difference exists between service methods and page methods. Performance is nearly identical. A minor difference is the fact that page methods are always emitted as inline JavaScript, whereas this aspect is configurable for services.

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23

Web services are publicly exposed over the Web and, as such, they're publicly callable by SOAP-based clients (unless the protocol is disabled). A method exposed through a Web or WCF service is visible from multiple pages; a page method, conversely, is scoped to the page that defines it. On the other hand, a set of page methods saves you from the additional work of developing a service.

Whatever choice you make, it is extremely important that you don't call any critical business logic from page and service methods. Both calls can be easily replayed by attackers and have no additional barrier against one-click and replay attacks. Normally, the view state, when spiced up with user key values, limits the range of replay attacks. As mentioned, though, there's no view state involved with page and Web service method calls, so even this small amount of protection isn't available for these specific cases. However, if you limit your code to calling UI-level business logic from the client, you should be fine.

Note I repeatedly mentioned that AJAX-enabled services, including Web services, are to be considered local to the application. They are in fact application services implemented as ASP.NET Web services because of the lack of alternatives. With ASP.NET 3.5, though, you have the possibility of using WCF services. What if you want to incorporate data coming from a classic WS-* Web service? You can't invoke the Web service directly from the client, but nothing prevents you from making a server-to-server call using the networking API of the .NET Framework.

Conclusion

ASP.NET offers two approaches to AJAX: partial rendering and scriptable services. Of the two, partial rendering is the one with some hidden costs. Although it can still achieve better performance than classic postbacks, partial rendering moves a lot of data around. In the worst cases, the savings in terms of markup are negligible compared to the quantity of bytes moved. On the other hand, AJAX was developed around the idea of making stateless server-side calls from the client and updating the page via the DOM.

Here's where scriptable services fit in. No hidden costs are buried in this model. As in a classic SOAP-powered Web service call, you send in only input data required by the method being invoked and receive only the return value. Traffic is minimal, and no view state or other hidden fields (for example, event validation) are roundtripped. On the down side, remote method calls require JavaScript skills. You control the execution of the method via JavaScript and use a JavaScript callback to incorporate the results in the page.

You need a server API to plan and execute client-to-server direct calls. How do you expose this API? Using a contract that is often implemented through a plain interface. How do you implement such a server API? There are various options: as a local, application-specific Web service, as an AJAX-enabled WCF service, or through page methods. In all cases, the ASP.NET

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23

AJAX client page is enriched with a system-generated proxy class to make calling the server easy and effective.

Once you have a back end based on services, you orchestrate calls to endpoints from the client browser using whatever programming language the browser provides. JavaScript might not always be optimal; Silverlight 2.0 is just around the corner.

In the next chapter, I'll examine the characteristics of a rich browser plug-in—that is, Silverlight 2.0—and also have a look at the less powerful, but still quite useful, Silverlight 1.0.

Just the Facts

- To definitely move away from postbacks, you need the ability to execute server calls from the client. JavaScript and *XMLHttpRequest* provide this ability.
- You need a standard and reliable way of defining the server API—contract-based services are the right way to go.
- ASP.NET AJAX allows you to define the server API using AJAX-enabled versions of ASP.NET Web services and WCF services.
- In the context of a single page, you can also use page methods—namely, public and static methods defined on the code-behind class of a page.
- WCF services are not supported in versions of ASP.NET AJAX older than version 3.5.
- In the latest version of ASP.NET, WCF services can be auto-configurable and don't require you to edit the *web.config* file for IIS to host them successfully.

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23