PART 5

Web Services

CHAPTER 21

Web Services Architecture

Microsoft has promoted ASP.NET web services more than almost any other part of the .NET Framework. But despite Microsoft's efforts, confusion is still widespread about what web services are and, more important, what they're meant to accomplish. This chapter introduces web services and explains their role in Microsoft's vision of the programmable Web. Along the way, you'll learn about the open-standards plumbing that allows web services to work, including technical standards such as WSDL (Web Services Description Language) and SOAP.

Internet Programming Then and Now

To understand the place of web services, you have to understand the shortcomings of the current architecture of distributed applications—applications that work over large networks or the Internet. Most of the applications you use over the Internet today can be considered "monolithic" because they combine a variety of services behind a single proprietary user interface. For example, you may already use your bank's website to do everything from checking exchange rates and reviewing stock quotes to paying bills and transferring funds. This is a successful model of development, but it has the following unavoidable shortcomings:

- Monolithic applications take a great deal of time and resources to create. They are often tied to a specific platform or to specific technologies, and they can't be easily extended and enhanced.

- Getting more than one application to work together is a full project of its own. Usually, an integration project involves a lot of custom work that's highly specific to a given scenario. And what's worse, every time you need to interact with another business, you need to start the integration process all over again. Currently, most websites limit themselves to extremely simple methods of integration. For example, you might be provided with a link that opens another website in an existing frame on the current web page.

- Most important, units of application logic can't easily be reused between one application and another. With ASP.NET, source code can be shared using .NET classes, but this isn't possible for applications created by different companies or written using different programming languages. And if you want to perform more sophisticated or subtle integration, such as between a website and a Windows application, or between applications hosted on different platforms, no easy solutions are available.

- Sometimes, you might want to get extremely simple information from a web application, such as an individual stock quote. To get this information, you usually need to access and navigate through the entire web application, locate the correct page, and then perform the required task. You have no way to access information or perform a task without working through the graphical user interface, which can be cumbersome over a slow connection or unworkable on a portable device such as a cell phone.

Components and the COM Revolution

This state of affairs may sound familiar if you know the history of the Windows platform. A similar situation existed in the world of desktop applications many years ago. Developers found they were spending the majority of their programming time solving problems they had already solved. Programmers needed to structure in-house applications carefully with DLLs or source code components in order to have any chance of reusing their work. Third-party applications usually could be integrated only through specific pipelines. For example, one program might need to export to a set file format, which would then be manually imported into a different application. Companies focused on features and performance but had no easy way to share data or work together.

The story improved dramatically when Microsoft introduced its COM technology (parts of which also go by the more market-friendly name ActiveX). With COM, developers found they could develop components in their language of choice and reuse them in a variety of programming environments, without needing to share source code. Similarly, less need existed to export and import information using proprietary file formats. Instead, COM components allowed developers to package functionality into succinct and reusable chunks with well-defined interfaces.

COM helped end the era of monolithic applications. Even though we all still use database and office productivity applications, such as Microsoft Word, that seem to be monolithic, integrated applications, much of the functionality in these applications is delegated to separate components behind the scenes. As with all object-oriented programming, this makes code easier to debug, alter, and extend.

Web Services and the Programmable Web

Web services enable the same evolution that COM did, with a twist. Web services are individual units of programming logic that exist on a web server. They can be easily integrated into all sorts of applications, including everything from other ASP.NET applications to simple command-line applications. The twist is that, unlike COM, which is a platform-specific standard, web services are built on a foundation of open standards. These standards allow web services to be created with .NET but consumed on other platforms—or vice versa. In fact, the idea of web services didn't originate at Microsoft. Other major computer forces such as IBM helped to develop the core standards that Microsoft uses natively in ASP.NET.

The root standard for all the individual web service standards is XML. Because XML is text-based, web service invocations can pass over normal HTTP channels. Other distributed object technologies, such as DCOM, are much more complex, and as a result, they are exceedingly difficult to configure correctly, especially if you need to use them over the Internet. So not only are web services governed by cross-platform standards, but they're also easier to use.

You can look at web services in two ways. Application programmers (and the .NET Framework) tend to treat a web service as a set of methods that you can call over the Internet. Of course, these methods have all the capabilities that ASP.NET programmers are used to, such as the automatic security and session state facilities discussed in other parts of this book. XML gurus take a different perspective. They prefer to treat web services as a way to exchange XML messages.

Which perspective you take depends to some extent on the type of web service you are creating. For example, if you need to pass messages through several intermediaries as part of a long-running business-to-business transaction, you'll have an easier time looking at your web service as a message-passing system. On the other hand, if you're calling a web service just to get some information—such as a product catalog or stock quote—you'll probably treat it like any other useful function.

When Web Services Make Sense

With the overbearing web services hype, developers sometimes forget to ask tough questions about when web services should and should *not* be used. Although web services are an impressive piece of technology, they aren't the best choice for all applications.

Microsoft recommends you use web services when your application needs to cross *platform boundaries* or *trust boundaries*. You cross a platform boundary when your system incorporates a non-.NET application. In other words, web services are a perfect choice if you need to provide data to a Java client running on a Unix computer. Because web services are based on open standards, Java developers simply need to use a web service toolkit that's designed for the Java platform. They can then call your .NET web services seamlessly, without worrying about any conversion issues.

You cross a trust boundary when your system incorporates applications from more than one company or organization. In other words, web services work well if you need to provide some information from a database (such as a product catalog or customer list) to an application written by other developers. If you use web services, you won't need to supply the third-party developers with any special information—instead, they can get all the information using an automated tool. You also won't need to give them access to privileged resources. For example, instead of connecting directly to your database or to a proprietary component, they can interact with the web service, which will retrieve the data for them. In fact, you can even use some of the same security settings that you use with web pages to protect your web services.

If you aren't crossing platform or trust boundaries, web services might not be a great choice. For example, web services are generally a poor way to share functionality between two web applications on your web server or to share functionality between different types of applications in your company. Instead, it's a much better idea to develop and share a dedicated .NET component. This technique ensures optimum performance, because you do not need to translate data into XML or send messages over the network. You'll find more details of this technique in Chapter 24, which tackles component-based development.

The Open-Standards Plumbing

Before using web services, it helps to understand a little about the architecture that makes it all possible. Strictly speaking, this knowledge isn't required to work with web services. In fact, you can skip to the next chapter and start creating your first web service right now. However, understanding a little bit about the way web services work can help you determine how to use them best.

Remember, web services are designed from the ground up with open-standard compatibility in mind. To ensure the greatest possible compatibility and extensibility, the web service architecture has been made as generic as possible. This means few assumptions are made about the format and encoding used to send information to and from a web service. Instead, all these details are explicitly defined, in a flexible way, using standards such as SOAP and WSDL. And as you'll see, in order for a client to be able to connect to a web service, a lot of mundane work has to go on behind the scenes to process and interpret this SOAP and WSDL information. This mundane work does exert a bit of a performance overhead, but it won't hamper most well-designed web services.

Table 21-1 summarizes the standards this chapter examines.

Table 21-1. *Web Service Standards*

Standard	Description
WSDL	Tells a client what methods are present in a web service, what parameters and return values each method uses, and how to communicate with them.
SOAP	The preferred way to encode information (such as data values) before sending it to a web service.
HTTP	The protocol over which all web service communication takes place. For example, SOAP messages are sent over HTTP channels.
DISCO	The discovery standard that contains links to web services or that can be used to provide a dynamic list of web services in a specified path.
UDDI	A standard for creating business registries that list information about companies, the web services they provide, and the corresponding URLs for DISCO file or WSDL contracts. Unfortunately, UDDI is still too new to be widely accepted and useful.

Web Services Description Language

Web Services Description Language (WSDL) is an XML-based standard that specifies how a client can interact with a web service, including details such as how parameters and return values should be encoded in a message and what protocol should be used for transmission over the Internet. Currently, three standards are supported for the actual transmission of web service information: HTTP GET, HTTP POST, and SOAP (over HTTP).

You can find the full WSDL standard at `http://www.w3.org/TR/wsdl`. The standard is fairly complex, but its underlying logic is hidden from the developer in ASP.NET programming, just as ASP.NET web controls abstract away the messy details of HTML tags and attributes. As you'll see in the next chapter, ASP.NET creates WSDL documents for your web services automatically. ASP.NET can also create a proxy class based on a WSDL document. This proxy class allows a client to call a web service without worrying about networking or formatting issues. Many non-.NET platforms provide similar tools to make these chores relatively painless. For example, Visual Basic 6 or C++ developers can use Microsoft's SOAP Toolkit. (To find the download, surf to `http://msdn.microsoft.com`, and search for *SOAP Toolkit*.)

■ **Note** The WSDL document contains information for communication between a web service and client. It doesn't contain information that has anything to do with the code or implementation of your web service methods—that is unnecessary and would compromise security.

SOAP

A client can use three protocols to communicate with a web service in .NET:

- HTTP GET, which communicates with a web service by encoding information in the query string and retrieves information as a basic XML document.
- HTTP POST, which places parameters in the request body (as form values) and retrieves information as a basic XML document.
- SOAP, which uses XML for both request and response messages. Like HTTP GET and HTTP POST, SOAP works over HTTP, but it uses a more detailed XML-based language for bundling information. SOAP messages are widely supported by many platforms.

Although .NET has the ability to support all three of these protocols, it restricts the first two for better security. By default, it disables HTTP GET, and it restricts HTTP POST to the local computer. This means you can use it to test a web service (as you'll see in the next chapter), but you can't use it to call a web service from a remote computer. You can change this setup by modifying the web.config file, but that's not recommended.

A Sample SOAP Message

Essentially, when you use SOAP, you're simply using the SOAP standard to encode the information in your messages. SOAP messages follow an XML-based standard and look something like this:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<soap:Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <GetStockQuote xmlns ="http://www.prosetech.com/Stocks/">
      <Ticker>MSFT</Ticker>
    </GetStockQuote>
  </soap:Body>
</soap:Envelope>
```

Looking at the preceding SOAP message, you can see that the root element is a <soap:Envelope>, which contains the <soap:Body> of the request. Inside the body is information that indicates what's taking place—essentially, a client is calling a web service method named GetStockQuote(). The client is supplying a parameter (named Ticker) with the value MSFT. Although this is a fairly straightforward method call, SOAP messages can easily contain entire structures representing custom objects or DataSets.

In response to this request, a SOAP response message will be returned. Here's an example of what you can expect:

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <GetStockQuoteResponse xmlns="http://www.prosetech.com/Stocks">
      <GetStockQuoteResult>29.23</GetStockQuoteResult>
```

```
    </GetStockQuoteResponse>
  </soap:Body>
</soap:Envelope>
```

This message returns the result of the stock quote, which is the number 29.23, wrapped in the appropriate XML tags.

This example demonstrates why the SOAP message format is superior to HTTP GET and HTTP POST. When using SOAP, both request and response messages are formatted using XML. When using HTTP GET and HTTP POST, the response messages use XML, but the request messages use simple name/value pairs to supply the parameter information. This means HTTP GET and HTTP POST won't allow you to use complex objects as parameters and won't be natively supported on most non-.NET platforms. HTTP GET and HTTP POST are primarily included for testing purposes.

Remember, your applications won't directly handle SOAP messages. Instead, .NET will translate the information in a SOAP message into the corresponding .NET data types before the data reaches your code. This allows you to interact with web services in the same way you interact with any other object.

For information about the SOAP standard, you can read the full specification at `http://www.w3.org/TR/SOAP`. (Once again, these technical details explain a lot about how SOAP works but are rarely implemented in day-to-day programming.)

Communicating with a Web Service

The WSDL and SOAP standards enable the communication between web services and clients, but they don't show how it happens. The following three components play a role:

- A custom web service class that provides some piece of functionality.
- A client application that wants to use this functionality.
- A proxy class that acts as the interface between the two. The proxy class contains a representation of all the web service methods and takes care of the details involved in communicating with the web service by the chosen protocol.

The actual process works like this (see Figure 21-1):

1. The client creates an instance of a proxy class.
2. The client invokes the method on the proxy object, exactly as though it were using a normal, local class.
3. Behind the scenes, the proxy object sends the information to the web service in the appropriate format (usually SOAP) and receives the corresponding response, which is converted to the corresponding data or object.
4. The proxy object returns the result to the calling code.

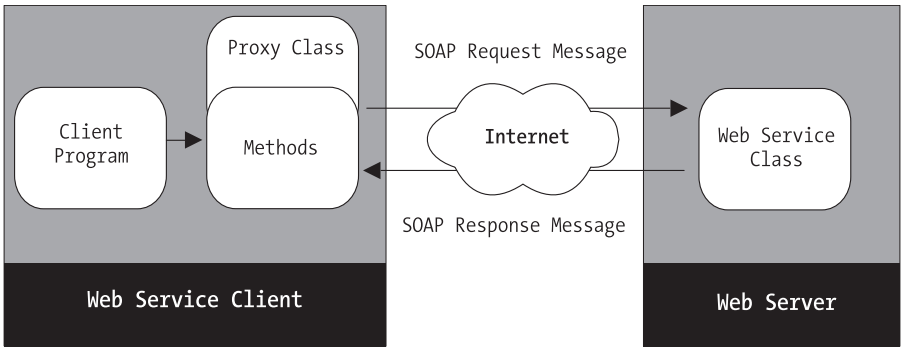

Figure 21-1. *Web service communication*

Perhaps the most significant detail is that the client doesn't need to be aware that a remote function call to a web service is taking place. The process is completely transparent and works as though you were calling a function in your own local code!

Of course, the following additional limitations and considerations apply:

- Not all data types are supported for method parameters and return values. For example, you can't pass many .NET class library objects (the DataSet is one important exception).
- The network call takes a short but measurable amount of time. If you need to use several web service methods in a row, this delay can start to add up.
- Unless the web service takes specific steps to remember its state information, this data will be lost. Basically, this means you should treat a web service like a stateless utility class, composed of as many independent methods as you need. You shouldn't think of a web service as an ordinary object with properties and member variables.
- A variety of new errors can occur and interrupt your web method (for example, network problems). You may need to take this into account when building a robust application.

You'll examine all these issues in detail throughout the next two chapters.

Web Service Discovery

Imagine you've created a web service that uses WSDL to describe how it works and transmits information in SOAP packets. Inevitably, you'll start to ask how clients can find your web services. At first thought, it seems to be a trivial question. Clearly, a web service is available at a specific URL address. Once clients have this address, they can retrieve the WSDL document by adding ?WSDL to the URL, and all the necessary information is available. So why is a discovery standard required at all?

The straightforward process described earlier works great if you need to share a web service only with specific clients or inside a single organization. However, as web services become more and more numerous and eventually evolve into a common language for performing business transactions over the Web, this manual process seems less practical. For instance, if a company provides 12 web services, how does it communicate each of these 12 URLs to prospective clients? E-mailing them to individual developers is sure to take time and create inefficiency. Trying to recite them over the telephone is worse. Providing an HTML page that consolidates all the appropriate links is a start, but it will still force client application developers to manually enter information into their programs. If this information changes later, it will result in painstaking minor changes that could cause countless headaches. Sometimes, trivial details aren't so trivial.

DISCO

The DISCO standard picks up where the "HTML page with links" concept ends. When following the DISCO standard, you provide a .disco file that specifies where a web service is located. Tools such as Visual Studio can read the .disco file and automatically provide you with the list of corresponding web services.

Here is a sample .disco file:

```
<disco:discovery  xmlns:disco="http://schemas.xmlsoap.org/disco"
  xmlns:wsdl="http://schemas.xmlsoap.org/disco/wsdl">
    <wsdl:contractRef
          ref="http://localhost/WebServices/StockQuote.asmx?WSDL"/>
</disco:discovery>
```

The benefit of a .disco file is that it is clearly used for web services (while .html and .aspx files can contain any kind of content). The other advantage is that you can insert <disco> elements for as many web services as you want, including ones that reside on other web servers. In other words, a .disco file provides a straightforward way to create a repository of web service links that can be used automatically by .NET. However, you don't need to create a .disco file to use a web service.

■ **Note** The DISCO standard is Microsoft-specific, and it's a bit of a dead end. It's slated for eventual replacement by WS-Inspection, a similar standard that's backed by all web service vendors.

Universal Description, Discovery, and Integration

Universal Description, Discovery, and Integration (UDDI) is one of the youngest and most rapidly developing standards in the web service family. UDDI is an initiative designed to make it easier for you to locate web services on any server.

With discovery files, the client still needs to know the specific URL location of the discovery file. Discovery files may make life easier by consolidating multiple web services into one document, but they don't provide any obvious way to examine the web services offered by a company without navigating to its website and looking for a .disco hyperlink. The goal of UDDI, on the other hand, is to provide repositories where businesses can advertise all the web services they have. For example, a company might list the services it has for exchanging business documents. To submit this information, a business must be registered with the service.

In some ways, UDDI is the equivalent of Google for web services, with one significant difference. Most web search engines attempt to catalog the entire Internet. Setting up a UDDI registry with all the web services of the world doesn't have much point, because different industries have different needs, and a single disorganized collection won't please anyone. Instead, it's much more likely that groups of companies and consortiums will band together to set up their own UDDI registries organized into specific industries. In all likelihood, many of these registries will be restricted so that they aren't publicly available.

Interestingly enough, the UDDI registry defines a complete programming interface that specifies how SOAP messages can be used to retrieve information about a business or register the web services for a business. In other words, the UDDI registry is itself a web service! This standard is still not in widespread use, but you can find detailed specifications at `http://uddi.microsoft.com`.

WS-Interoperability

Web services have developed rapidly, and standards such as SOAP and WSDL are still evolving. In early web service toolkits, different vendors interpreted parts of these standards in different ways, leading to interoperability headaches. Adding to the confusion is that some features from the original standards are now considered obsolete.

Negotiating these subtle differences is a small minefield, especially if you need to create web services that will be accessed by clients using other programming platforms and web service toolkits. Fortunately, another standard has appeared recently that sets out a broad range of rules and recommendations designed to guarantee interoperability across the web service implementations of different vendors. This document is the WS-Interoperability Basic Profile (see `http://www.ws-i.org`). It specifies a recommended subset of the full SOAP 1.1 and WSDL 1.1 specifications and lays out a few ground rules. WS-Interoperability is strongly backed by all web service vendors (including Microsoft, IBM, Sun, and Oracle).

Ideally, as a developer, you shouldn't need to worry about the specifics of WS-Interoperability. However, you'll be happy to learn that ASP.NET 2.0 web services follow its guidelines automatically.

Note With a fair bit of work, you can configure your web service to break these rules, but by default .NET won't let you. Instead, your web service will throw an exception when it's compiled. In the next chapter, you'll learn how to turn WS-Interoperability checking on and off.

WHERE ARE WEB SERVICES TODAY?

Currently, first-generation web services are being used to bridge the gap between modern applications and older technologies. For example, an organization might use a web service to provide access to a legacy database. Internal applications can then contact the web service instead of needing to interact directly with the database, which could be much more difficult. Similar techniques are being used to allow different applications to interact. For example, web services can act as a kind of "glue" that allows a payroll system to interact with another type of financial application in the same company.

Second-generation web services are those that allow partnering companies to work together. For example, an e-commerce company might need to submit orders or track parcels through the web service provided by a shipping company. Second-generation web services require two companies to work closely together to devise a strategy for exposing the functionality they each need. Second-generation web services are in their infancy but are gaining ground quickly.

The third generation of web services will allow developers to create much more modular applications by aggregating many different services into one application. For example, you might add a virtual hard drive to your web applications using a third-party web service. You would pay a subscription fee to the web service provider, but the end user wouldn't be aware of what application functionality is provided by you and what functionality relies on third-party web services. This third generation of web services will require new standards and enhancements that will allow web services to better deal with issues such as reliability, discovery, and performance. These standards are constantly evolving, and it's anyone's guess how long it will be before third-generation web services begin to flourish, but it's probably just a matter of time.

Already, you can use third-party web services from companies such as eBay, Amazon, and Google. These web services act as part of a value-added proposition and may eventually evolve into separate cost-based services. But if you're curious, you can seek these web services out today and use them in your own .NET applications.

The Last Word

This chapter introduced web services, explained the role they play in distributed applications, and dissected the standards and technologies they rely on to provide their magic. In the next chapter, you'll see just how easy .NET makes it to create your own web services.