|  |  |
| --- | --- |
|  |  |

**Understanding XML Web Services**

Rarely have I seen a new technology hyped as much as Web services. Microsoft even describes .NET as a "platform for building Web services," though it is so much more than that. And it seems every new technical book I pick up nowadays has a chapter on Web services. Are Web services really that revolutionary? Read on to find out.

Interestingly, what Microsoft now calls XML Web services were once called ASP.NET Web services in the early Beta documentation. I suppose many people thought it was a Microsoft proprietary technology, so Microsoft changed the name to XML Web services to stress the fact it uses open standards (like XML). Regardless, in this chapter I refer to the technology as simply Web services, without any additional qualification.

**Web Services Overview**

When I was fresh out of college, I worked for a company that had a successful integrated database/4GL product. I was experimenting with a new technology called the World Wide Web and had successfully "Web-enabled" the product, which meant only that its data could be queried from and displayed in a browser. Being the only Web "expert" at that company, I was called into a high-level strategy meeting where the original architects of the product were brainstorming more significant ways to leverage its strengths for the Web. The idea they presented to me was this: since the Web contained large amounts of data, and the product's strength was its data processing engine, could this engine be used to dynamically query, correlate, and report on data retrieved from Web sites?

I almost laughed but had to stifle it when I realized they were serious. What they did not realize was that the Web is not structured into neat little tables with well-defined relationships for consumption by computerized data engines. Instead, the Web is a chaotic jumble of sites that deliver information designed for human consumption. Each site is organized very differently, and each has its own unique way of presenting the data. Furthermore, Web sites are constantly changing. We humans are incredibly flexible parsers, so we can make sense out of the chaos. We easily adjust when Amazon.com changes its page layout, but it would take an AI application of immense complexity for a computer to do the same. I had to break this bad news to the architects with a straight face.

It turns out that the joke is on me. This vision of a queriable Web is exactly where we are heading. Though we have a long ways to go, Web services are a giant step forward. I think of this vision as a queriable Web, but the working technical term is the *semantic* Web, which refers to a Web containing structured information with well-defined meaning. In other words, it refers to a Web that you can query.

**Why Web Services?**

Technologies such as COM popularized the notion of component-based programming. In this style of programming, developers write small, discrete programs and connect them to compose an application. Other technologies, including DCOM and now .NET Remoting, allow you to build an application by connecting components executing on separate networked machines. These technologies assume a homogeneous environment. For example, .NET Remoting assumes that a .NET application exists on both ends of the communication channel.

Web services make it possible for Web sites and applications to interact like components. Like any component, a Web service exposes its functionality through a set of methods. Unlike DCOM or .NET Remoting, however, Web services are designed for a heterogeneous environment. That is, a client running on any platform can invoke a method on a Web service running on any platform. The Web service responds with data stripped of unnecessary details, allowing the client to use the data in whatever manner it wishes. Therefore, the typical Web service client is *not* a Web browser. Instead, Web service clients are Windows applications, application servers, other Web sites, PDA software, and so on. In other words, the end user of a Web service is a computer program, not a human.

The application of Web services is limitless. In one scenario, data can be gathered from totally unrelated Web sites and correlated in imaginative ways without human intervention. For example, let's say I start a Web site called The VikingsReport.com, which follows the ups and downs of the Minnesota Vikings football team. My theory is that the Vikings have good seasons when the economy is strong, but struggle when the economy is bad. To support my hypothesis, I could write a program that contacts Web services hosted by sites such as NYSE.com, Nasdaq.com, ESPN.com, and FederalReserve.gov, download statistical data ranging from economic reports to passing yards, correlate the data into a series of graphs, charts, and reports, and then post it all on my site for the benefit of Vikings fans. This scenario is summarized in Figure 6-1.

Figure 6-1: Web services allow data to be correlated from various sites.

Other uses of Web services include B2B scenarios in which companies can conduct business using nothing more than the Internet and some agreed-upon protocols. Because Web services are platform agnostic, you can also use them to integrate a diverse back office environment consisting of a variety of modern platforms and legacy mainframe applications.

**Web Service Composition**

Web services achieve platform interoperability because they are built using industry standards and widely supported protocols such as these:

* **Extensible Markup Language (XML).** The foundation of Web services is XML. It provides a platform-agnostic way to mark up data and is the grammar upon which SOAP and WSDL are built.
* **HyperText Transfer Protocol (HTTP).** This is the networking protocol used by the Web. The popularity of the Web ensures all but the most ancient platforms have rich support for the HTTP networking stack.
* **Simple Object Access Protocol (SOAP).** The preferred, but not required, Web service messaging protocol is SOAP. It is quickly being adopted throughout the industry.
* **Web Service Description Language (WSDL).** This is an XML-based grammar that describes the layout of the SOAP messages accepted by a particular Web service.
* **Universal Description, Discovery, and Integration (UDDI).** This is an XML-based grammar that allows Web service providers to advertise the existence of their Web services.

Using these standards, Web services provide three critical infrastructure services:

* **Discovery.** With UDDI, Web services provide a mechanism by which client programs or client programmers can locate a Web service that provides the required functionality.
* **Description.** Once the Web service is located, the client downloads a WSDL document that details the Web service functionality and how to use it. This includes method names, parameter types, and return types.
* **Wire Format.** When the client invokes a Web service method, the method call must be serialized into some format that can be transported to and understood by the Web service. The most common serialized format is SOAP, although you can also use the query-string mechanism supported by HTTP.

**The Role of XML and XML Schema**

The popularity of the Web has driven vendors to create more and more powerful Web server products, until the Web server itself has become the application server. However, Web server-based applications generally return HTML. Although HTML is adept at defining page layout, colors, text emphasis, and so on, it surrounds the raw data with these presentation details. In other words, HTML mixes the data and the presentation. For example, consider the following excerpt from the class schedule page of the Intertech, Inc. Web site:

<TR name ='only row for data'>

 <TD WIDTH='40%' VALIGN='top'>

 <A href='/courses/CourseDetails.asp?LOC=details&**ID=99115**'>

 <P ALIGN='left'>**Extreme .NET**</p></a></TD>

 <TD align='Center' WIDTH='10%' VALIGN='TOP'>

 <Font class='whitefont'> \_\_ </font> </TD>

 <TD align='Center' WIDTH='10%' VALIGN='TOP'>

 <Font class='whitefont'>

 <A href='https://www.intertech-inc.com/enroll.asp?**CourseDateID=889**'>

 <font class='Bold75'>**6-10**</font></A>

 </font> </TD>

 <TD align='Center' WIDTH='10%' VALIGN='TOP'>

 <Font class='whitefont'> \_\_ </font></TD>

 <TD align='Center' WIDTH='10%' VALIGN='TOP'>

 <Font class='whitefont'> \_\_ </font></TD>

</TR>

In all this text, only the items in bold constitute data. The rest is presentation detail. If you had a program that periodically went to this Web page, fetched the course schedule, and updated your Outlook calendar, then it would have to parse through all this presentation detail to retrieve the data. Furthermore, if the page layout changed, you would have to update your program's parsing logic.

In contrast, Web services use XML to transmit pure data mixed only with metadata. For example, the data in the previous excerpt can be represented in XML as follows:

<courseSchedule>

 <course id="99115">

 <name>Extreme .NET</name>

 <date id="889">2002-6-10</date>

 </course>

</courseSchedule>

Obviously, this is much more concise than the HTML example. Furthermore, this layout only changes if the data schema itself changes, which is far less frequent than updates to the look and feel of a Web site. As a result, your parsing code remains valid for much longer. Luckily, if you use .NET to consume Web services, you rarely need to write any parsing code because .NET can convert XML into data that your .NET language of choice can understand. In other words, it can take the following line:

<date id="889">2002-6-10</date>

and turn it into an instance of the CLR's DateTime type. Web services themselves, however, know nothing about the Common Type System (CTS) used by the CLR and all .NET languages. Instead, Web services use a type system that is native to XML and is part of the XML Schema specification. .NET simply translates the XML Schema types to the corresponding CTS types. On the server side, if you use .NET to build the Web service, then the runtime automatically converts CLR types to the corresponding XML Schema types.

**The Role of UDDI and Disco**

If you open a new business, you need to make potential customers aware of it through advertising. The first place businesses advertise is in the Yellow Pages, because it provides a resource that all potential customers are familiar with and can use to look up phone numbers and addresses of businesses offering the required services.

Likewise, if I built a Web service for the Intertech class schedule, I would want people to use it and would therefore advertise its existence and general functionality. UDDI provides a standard XML-based grammar that describes the general services and location of a Web service. Gather these UDDI descriptions into one repository, and you essentially have the Yellow Pages for Web services. Such UDDI sites are already beginning to arrive on the scene. For example, Microsoft provides a UDDI Web site at http://uddi.microsoft.com.

Ironically, UDDI is itself a Web service. In theory, a program that needs the current weather report, for example, can query a UDDI repository, retrieve the locations of several such Web services, and use whichever one it wants. This brings us much closer to the vision of the semantic Web I mentioned earlier. In the short term, however, most Web service clients will just "know" the location of the Web service, or a human will find it by the usual means—surfing the Web. For example, the most effective way for me to advertise the Intertech class schedule Web service would be to provide a link on the Intertech home page. Customers surfing to the Web site could then see that the service is available and choose to build a client program to consume it.

UDDI is in its infancy stages. Because of this, when Microsoft was designing its Web service support, it included its own discovery mechanism called *disco*. Like UDDI, disco is XML-based, but it is much simpler. Unlike UDDI, the intent of disco is just to allow the discovery of Web services hosted by a given Web site. To muddle the issue even more, Visual Studio .NET Web service projects contain files with the .vsdisco extension that are called *dynamic discovery files*, not to be confused with files with the .disco extension that are called *static discovery files*. Dynamic discovery allows ASP.NET to investigate each virtual directory of a Web site in order to find and report on any available Web services. For security reasons, the release version of .NET cripples this feature. Though it is not recommended, you can follow the instructions in Microsoft Knowledge Base article Q307303 (search for it at <http://support.microsoft.com>) to enable dynamic discovery on a Web site.

**The Role of WSDL**

Once you find a Web service, how do you use it? This question is addressed by WSDL. To understand its role, consider the information required to use any CLR-defined class. First, you need the name of each of its methods. Then, for each method, you need the list of input parameters it expects, and the type of data it returns in response. This is exactly the same information required to consume a Web service. A WSDL document contains this information for a given Web service and, because it is formatted using a standard XML grammar, it can be interpreted by any program on any platform.

For example, consider the following Web service:

public class ClassSchedule

{

 [WebMethod] // <-- This attribute described in the next section!

 public DateTime GetNextClassDate(string className)

 { ... }

}

A *small* portion of the WSDL for this Web service is shown here:

<s:element name="**GetNextClassDate**">

 <s:complexType>

 <s:sequence>

 <s:element minOccurs="0" maxOccurs="1" name="className" type="**s:string**" />

 </s:sequence>

 </s:complexType>

</s:element>

<s:element name="**GetNextClassDateResponse**">

 <s:complexType>

 <s:sequence>

 <s:element minOccurs="1" maxOccurs="1" name="GetNextClassDateResult"

 type="**s:dateTime**" />

 </s:sequence>

 </s:complexType>

</s:element>

This portion of the WSDL document describes the parameter types required to call the GetNextClassDate method and the type of data it returns. The actual WSDL document is much longer and quite complex. Luckily .NET provides a tool called Wsdl.exe that builds a .NET proxy class based on this WSDL document. Other development platforms provide similar tools. Thus you can think of a WSDL document as the type metadata for a given Web service.

When writing the client code, you simply call methods on the generated proxy. The proxy class uses the WSDL to determine how to serialize each method call into a message format the Web service understands (typically SOAP) and transmits the message to the Web service.

**The Role of SOAP**

SOAP provides a standard XML-based message format to represent a method call and its response. Because SOAP is an industry standard, a client built on any platform can compose a SOAP message to invoke a method on a Web service running on any platform. By design, SOAP is a very simple specification, defining only three parts: the envelope, the header, and the body. The relationship between these SOAP sections is described here and illustrated in Figure 6-2.

* The SOAP body contains the information required to execute the method call such as its name and the parameters.
* The SOAP header contains other out-of-band data such as security or transactional information. This section is optional.
* The SOAP envelope encapsulates the body and the header.

Figure 6-2: The layout of a SOAP message

For example, the following code invokes the GetNextClassDate method on a client-side proxy to the ClassSchedule Web service:

// Create the Web service proxy

ClassSchedule proxy = new ClassSchedule();

// Call a method on the Web service

DateTime next = proxy.GetNextClassDate("Extreme .NET");

As a result, the proxy class serializes the GetNextClassDate call into the following SOAP message:

<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

 xmlns:xsd="http://www.w3.org/2001/XMLSchema"

 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">

 <soap:Body>

 <**GetNextClassDate** xmlns="http://www.intertech-inc.com">

 <className>**Extreme .NET**</className>

 </GetNextClassDate>

 </soap:Body>

</soap:Envelope>

Because this SOAP message is formatted as indicated by the WSDL document, the Web service can interpret it, execute the requested method, and return the result. The result, too, must be serialized into an understood message format. Here is the SOAP result message:

<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

 xmlns:xsd="http://www.w3.org/2001/XMLSchema"

 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">

 <soap:Body>

 <GetNextClassDateResponse xmlns="http://www.intertech-inc.com">

 <GetNextClassDateResult>**2002-6-10**</GetNextClassDateResult>

 </GetNextClassDateResponse>

 </soap:Body>

</soap:Envelope>

The client-side proxy reads this SOAP message and converts the date data (in bold) to a CLR DataTime object.

The SOAP specification defines the following formatting styles for both the overall contents of the body section and for each individual method parameter:

* **RPC style.** This term refers to the format of the SOAP body. A message that uses RPC style follows the format defined in section 7 of the SOAP specification.
* **Encoded style.** This term refers to the format of the method parameters within the SOAP body. A message falls in the encoded style category if it formats parameters according to section 5 of the SOAP specification.

However, the SOAP specification does allow for some variation. Of course, both the client and the Web service must understand and expect any variation from the specification.

WSDL defines two such variations. These variations use XML Schema as the underlying specification rather than SOAP sections 5 and 7.

* **Document style.** This refers to an alternate format for the SOAP body. In other words, it is an alternate to RPC style.
* **Literal style.** This represents an alternate format for the method parameters within the SOAP body, that is, an alternate to encoded style.

Before this discussion of RPC style, document style, section 5, and section 7 drives you Section 8, keep in mind that you rarely need to know anything other than what the Web service supports. The WSDL document for a Web service defines which styles it uses. So .NET reads the WSDL to determine the appropriate message format. All .NET Web services default to document style with literal parameters, but can also support document style with encoded parameters and RPC style with encoded parameters. RPC style only supports encoded parameters. See Table 6-1 for a summary of these supported combinations.

| Table 6-1: The Body and Parameter Formatting Combinations  |
| --- |
| **Supported Parameter Formats**  |
| ***BODY FORMAT***  | ***ENCODED***  | ***LITERAL***  |
| RPC | Yes | No |
| Document | Yes | Yes |

|  |  |  |
| --- | --- | --- |
|  | Note  | Although SOAP was originally an acronym for Simple Object Access Protocol, once it was handed to the W3C standards committee, the committee renamed it XML Protocol. Due to the popularity of the name, however, it is still referred to as SOAP. So like OLE before it, SOAP has become a name unto itself rather than an acronym. |

SOAP was specifically designed as a transport-agnostic message format. That is, it does not care what underlying network protocol is used to transport the SOAP message. Web services, however, rely exclusively on HTTP for SOAP transport. In fact, if a Web service method only consists of parameters with simple types, you can invoke it using nothing more than the standard HTTP GET and POST verbs instead of SOAP. SOAP, however, allows for the serialization of complex types and thus is the preferred messaging format.

**The World Wide Web Consortium**

As noted, Web services derive their usefulness from the fact that they are based on widely supported standards. The World Wide Web Consortium (W3C) is the organization that reviews, establishes, and documents many of these standards. You can browse their site (<http://www.w3c.org>) for information regarding XML, XML Schema, WSDL, and SOAP.

With all the hype surrounding Web services, a key fact has been overlooked; at the time of this writing, WSDL and SOAP have *not* reached the recommendation stage—that is, these are not official standards endorsed by the W3C. In particular, the WSDL specification is currently only a "note," which means it has not even been reviewed yet. The bottom line is that these specifications can change, so your projects should not rely too heavily on their current incarnations. As you will see in the next section, .NET provides several tools that abstract most of the details of SOAP and WSDL and thus isolate your application code from specification changes. Even so, you must consider any Web service development inherently risky due to these fluid standards.

|  |  |  |
| --- | --- | --- |
|  | Warning  | SOAP, WSDL, and UDDI have not yet been finalized by standards committees. |

|  |  |
| --- | --- |
|  |  |