1
Introducing Web Services

The underlying software and hardware that provide the connective tissue for the Internet represent some of the most complex technology of the past few decades. Just a few years ago, the Internet was a simple network that relatively few people used for e-mail. Seemingly overnight, HTTP- and HTML-based Web pages emerged. HTTP (Hypertext Transport Protocol) is an application-level protocol that is relatively easy to implement and debug. Web pages based on HTML (Hypertext Markup Language) are easy to author using a human-readable format that Web browsers have been liberal at interpreting.

Several years after the HTTP/HTML revolution, XML (eXtensible Markup Language) came into play. XML is a simple and easy-to-understand format for data markup. From this single, easy manner of data encoding, it wasn't too large a step to create a new application framework that combined HTTP and XML into Web service—and enabled developers to create distributed applications in ways that were impossible before. I say that Web services are an exciting technology because the Web architecture is fundamentally different from many other networking and distributed programming methodologies. Web services inherit many of the better features of the Web, as well as some of the pitfalls.

This chapter provides an overview of the features and drawbacks of Web services. It begins with an introduction to Web services, including a working definition, and places them in the larger context of distributed

application development. By the end of this chapter, you should have a clearer understanding of what Web services are, as well as how this technology fits into the larger landscape of software development.

The Problem: Sharing Data

Simply stated, computers need to share data. Many scenarios bear this out: Businesses need to share information with partners. Divisions need to send data to other divisions. Consumer applications need to work with other consumer applications.

Recently, Microsoft identified several different application types that could use Web services:

- Data providers, for example, those that provide data such as a stock quote
- Business-to-business process integrations, such as those that send a purchase order from one company to another
- Integration with multiple partners, and even with competitors
- Enterprise application integration, for example, integration of a company's e-mail database with its human resources (HR) database

A typical use of Web services would be a help desk application. Help desk applications are often small Windows applications that enable help desk operators to query for internal customer data, and then update this data with whatever information they gather during the course of a help desk call. The developers of this type of application need to pull out information about each employee who calls from the central employee database. They also need to pull out and modify data from their own help desk database.

The Solution: Distributed Application Development

Distributed application development is the art and engineering of getting data from one machine to another. This data can be of almost infinite variation: purchase orders, customer data, digits 100 through 200 of pi, and so on.

There have been many technologies for building applications that can send data back and forth. CORBA (Common Object Request Broker Architecture), RMI (Remote Method Invocation), and DCOM (Distributed Component Object Model) are just a few.

However, all of these have had flaws, and none has ever caught on for ubiquitous and heterogeneous environments. Most haven't even tried to make that a design goal. For example, DCOM is based on COM (Component Object Model), which is a binary standard that has virtually no deployment outside of the Microsoft Windows platform. Although using DCOM across platforms is technically possible, it is wishful thinking to think that DCOM would ever be employed in such an environment. No one has ever really tried and succeeded in a stable and cost-effective manner.

Common Object Request Broker Architecture, or CORBA, is a distributed technology (actually, a specification of a technology) that can and does work in heterogeneous environments. However, for many reasons, including that many developers find it difficult to implement CORBA-based applications and services, CORBA has never really caught on.

Simple Object Access Protocol, or SOAP, is a new step in the world of distributed applications. Web service technologies such as SOAP have made much quicker inroads for distributed application development than did earlier technologies. Among the many reasons for SOAP's success, the most important are that it

- Leverages the Web architecture where appropriate (e.g., XML and other Web standards such as HTTP).
- Uses a modular design.
- Creates a message-passing architecture that doesn't force a particular implementation, programming model, or language.
- Assumes that failure will occur during processing, and allows for processes to detect this easily.

The sections that follow closely examine these reasons for SOAP's success.

The Web Architecture

The Web architecture assumes and deals gracefully with failure, latency, and all kinds of other real-world problems. This is a major strength and one that Web services have for the most part inherited. When you launch a Web browser, you expect that a number of the pages, graphics, and other files that you try to download will challenge you for credentials, or will have moved locations, or simply will no longer exist. Web services are challenged with the realities of this environment.

Assuming failure is a more advanced concept than it seems at first. For example, imagine you are building an application that will download a movie. As the size of your movie increases, the likelihood of some of the packets getting lost increases as well. As the developer of this application, you have a few choices: You can re-download every frame if you lose any one frame, you can ask for just the frames you missed, or you can skip them. Clearly the first choice isn't the best one, but deciding between the next two is difficult.

At its most basic level, the Web makes the last choice: It forgets about what it loses. This is an interesting choice, and it seems like the wrong one on the face of it. But the alternative requires a huge investment in infrastructure to make it work. Clearly, sometimes you need a thick and durably reliable infrastructure, but not always.

Imagine how much more traffic would be generated if your browser acknowledged every image it was downloading, and then the server resent until you received the image. As another example, consider the ubiquitous 404 error code: Built into the HTTP specification itself is the idea that a file may be missing from the server.

The Web is also hugely concurrent. It's the most concurrent system ever seen in computers. Every Web server and every client is operating in parallel. But programming for concurrency is extremely difficult. This is one of the ironies of modern Web development: The system itself is by definition almost unmanageably hard to develop for. The essential challenge is to handle state in an effective manner that takes into consideration the principle of assuming failure.

Web services are a set of technologies that enable you to develop distributed applications which take both failure and concurrency into account. Before we examine in more detail how Web services do this, let's take a moment to define what a Web service is.

Defining Web Services

Web service is one of the more difficult terms to define. Most definitions are inadequate on some level or another, which may be an indication that we are using the term too generally. It may be easier to begin by saying what a Web service is *not.* A Web service is not a Web site that a human reads—at least not in the way this book uses the term. A Web service is not anything with which an end user (even a very sophisticated end user) would directly interact. A Web service is something another process on another machine uses.

The Significance of Internet Standards

Web services are hard to define, but here is a serious attempt: A *Web service* is application logic that is accessible using Internet standards. The idea of getting data from one machine to another—that is, building a distributed application—is decades old. There have been many technologies for exposing application logic—but generally these were based on difficult-to-implement (binary in most cases) and often proprietary protocols. Standards, and in particular Internet standards such as XML, have simplified the building of a distributed application.

What Makes a Standard?

Some would argue that SOAP, for example, isn't a standard because it's not yet (as of this writing) a W3C (World Wide Web Consortium) recommendation. However, I think it is reasonable to say that standards may occur outside of standards bodies. A protocol such as SOAP may become a de facto standard, owing to its wide implementation across many platforms and

(continued)

What Makes a Standard? (*cont.*)

languages. SOAP easily meets this litmus test, as it has more than 50 implementations in every language I can think of, and on every OS I've ever heard of, plus some others I've never heard of! Furthermore, it is based on recommendations from the W3C, such as XML and HTTP.

The Significance of WSDL

Another, semicircular definition of a Web service is to say that it is anything a Web Services Description Language (WSDL) document can describe. Although true, this isn't much better than the first definition, but it is the one that most closely mirrors what I see in my head when I say "Web service." And, when I think of Web service clients, I think of technologies that consume WSDL documents to learn how to communicate with Web services. Of course, one could build a server or client that does nothing with WSDL, but the concepts of WSDL should be present, at the least.

The Significance of Interoperability

So why do Web services exist? There are many other technologies for accessing and exposing application logic, but these technologies usually have serious shortcomings. Web services attempt to solve these problems by promoting heterogeneous, interoperable, loosely coupled, and implementation-independent programs. I would say that the two largest design goals of most Web service technologies are interoperability and loose coupling.

Interoperability is usually the reason people become interested in Web services. The significance of interoperability is fairly easy to comprehend. For example: Web services allow a Perl developer to interoperate with a C++ developer on Microsoft Windows 2000, who in turn can interoperate with an AppleScript programmer on a Macintosh OSX. I've seen this exact scenario, not just in demonstrations, but also in actual deployed applications that allow businesses to integrate more easily than in the past. That is because now they can work with their partners without having to adopt a platform that isn't their usual one. Table 1.1 lists a small sampling of the Web service technologies currently available.

TABLE 1.1: Popular Web Service Technologies

Product	Support OS	Language
Apache SOAP	UNIX, Windows	Java
WASP	UNIX, Windows	C++
GLUE	UNIX, Windows	Java
SOAP BEA WebLogic	UNIX, Windows	Java
MS SOAP Toolkit	Windows	C++, VB, COM
.NET Framework	Windows	C#, VB.NET
SOAP::Lite	UNIX, Windows	Perl
PocketSOAP	WinCE	C++
SOAP for ADA	Linux	Ada
Web Service Behavior	Windows	JavaScript, DHTML
SOAPx4	UNIX	PHP
Delphi 6	Windows	Delphi
Kafka	Windows	XSLT

The Significance of Loose Coupling

Another powerful feature of Web services is how easily they allow you to build loosely coupled systems. *Loosely coupled* is another term that is difficult to define precisely. In general, when I say loosely coupled, I mean that a system exhibits both implementation independence and versioning without breakage.

Implementation independence is an easy enough concept to get across. If I want to implement the same Web service on machine A to machine B, then it shouldn't matter that machine A is a Java/UNIX program and machine B is a C#/Windows program. This is a basic requirement of Web services.

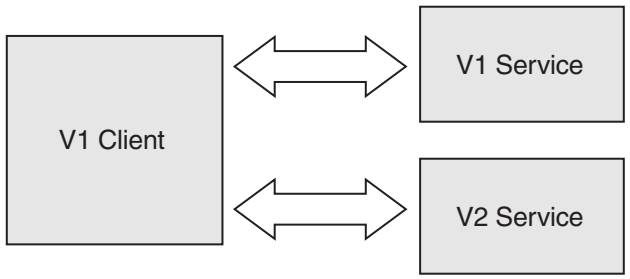

FIGURE 1.1: Versioning Web Services

Versioning ease via loose coupling is a little harder to describe and grasp at first. Clearly, a machine running a version 1 client can expect to interoperate with a version 1 server. But what about mismatched versions? If machine A is running version 1 of a service and sends a message to machine B, which is running version 2, then machine B should be able to receive and understand the message. Of course, Web services don't make this possible in every case because there is always the human element to deal with: business rules and other facts of the real world. Regardless, versioning without breakage is a genuine goal, and a powerful one. Figure 1.1 gives an example of this.

Modular Design

A core concept of Web services (in particular, the wire protocol: SOAP) is that they are very modular. One flaw of many other attempts at distributed application development specifications has been a reliance on getting everything into one specification. For example, if you don't need transactions, then there is no need to implement them. SOAP doesn't try to be a complete distributed application technology. It has many, many holes, and this is *by design*.

By focusing on simplicity and extensibility, it's possible to create distinct specifications for security, transactions, reliability, and so on without drowning in complications. Applications can combine features from specifications they need, including custom modules.

One of the design tenets of this book is that leveraging modularity is critical. Your own designs for Web services should follow a pattern similar to the standards. There are several rules by which to live:

- Allow extensibility points.
- Keep your namespaces easy to version by placing dates in them.
- Don't try to solve every problem with one schema, WSDL, or other file. Break out the problem into pieces.

Message Passing

You may have noticed that I used the term *message passing* earlier. At a very fundamental level, Web services are expressed in messages that are sent from one process to another. Typically (and I would argue that this is a near requirement), those messages are XML based. The message is the core piece of Web services. It is like the function in the C programming language: Just as you can't discuss C without discussing functions, you can't discuss Web services, in particular SOAP, without talking about messages.

Messages, Methods, and Operations

Note that messages do not map to methods. Methods can be modeled by combining related messages into a set called an operation. Operations are analogous to methods in many cases, but I would warn against thinking of the term *operation* as a synonym of *method*. The two are only similar.

As a matter of fact, we can mimic function calls with messages—two to be exact. We can map the first message, the request, to the function call with the parameter values, and express the return value and out parameters as a response message. When we combine a set of messages into a logical unit in this way, we call them an *operation.* The request–response operation is the most common in the Web services world. Listing 1.1 shows a typical SOAP message (in this case a request for a stock quote).

Listing 1.1: A Typical SOAP Request Message

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
   <SOAP-ENV:Header>
       <t:SessionOrder
         xmlns:t="http://example.com"
         xsi:type="xsd:int" mustUnderstand="1">
           5
       </t:SessionOrder>
   </SOAP-ENV:Header>
   <SOAP-ENV:Body>
       <GetStockQuote
         xmlns="http://someexample.com">
           <Price>MSFT</Price>
       </GetStockQuote>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The difficult thing about messages is that they are essentially asynchronous and very concurrent. Nothing in the SOAP specification or the Web architecture in general guarantees that a machine on a network will receive the messages I send in the same order in which I sent them.

Dealing with asynchronous and concurrent behavior is difficult. However, most current Web service implementations already have been fairly successful at mapping the more familiar synchronous programming model on top. This programming model is appropriate for many uses, but never forget that you are really dealing with something unlike a call stack in a computer with a single processor. You are dealing with a much more complex system.

Error Handling

As noted earlier, the Web architecture assumes failure. Web services inherit HTTP's design principle regarding failure, and attempt to improve on it.

The SOAP specification has an entire section devoted to describing exactly how errors can be communicated. Of course, programmatic errors (either in the system or in the application) aren't the only kind of failures you can experience.

For example, when you surf, you will often find that the page you are trying to access no longer exists. In other words, you'll see the Web master's dreaded *404 Not Found* error. There is actually a large set of similar errors that HTTP presupposes may occur, such as pages moving either temporarily or permanently.

Web services inherit this idea of assuming failure and build on it. A significant section of the SOAP specification is about SOAP faults. A SOAP fault is a special message type used specifically for sending errors; Listing 1.2 shows an example. SOAP also specifies faults for an interesting set of failures. For example, SOAP defines the error you are supposed to send when you don't understand a SOAP header.

DEFINITION: SOAP HEADER

A modular and composable piece of information that can be included in any SOAP message, similar to a footnote in a mail message, only more sophisticated. You can learn more about SOAP headers in Chapter 9.

LISTING 1.2: A SOAP Fault Message

```
HTTP/1.1 500 Internal Server Error
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
   <SOAP-ENV:Body>
       <SOAP-ENV:Fault>
           <faultcode>SOAP-ENV:MustUnderstand</faultcode>
           <faultstring>Something bad happened.</faultstring>
       </SOAP-ENV:Fault>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The Web Service Architecture

Microsoft, IBM, and others have created a set of specifications and technologies surrounding their vision for Web service architecture of the future. These specifications fill in the gaps that SOAP and WSDL don't address. You can easily identify these specifications, as they follow the naming scheme WS-*technology,* where the technology is something like security. This architecture is a vision that is shared across the industry and includes many of the ideas presented in this chapter so far, such as modularity and building on Web standards.

This Web service architecture includes a distinct and credible vision for how all of the modular pieces are to be plugged together. This vision is for a set of related specifications that are

- Standards based
- Modular
- Federated
- General purpose

We've looked at why standards and modularity are a good idea. *Federated* designs are useful because they eliminate one of the key problems with most client-server applications: the reliance on a server.

DEFINITION: FEDERATION

Federation is a design methodology for spreading out processes, components, and machines in a way that removes single points of failure. Instead, the pieces of a distributed system are *federated* out over the entire system. Federation enables you to distribute systems across organizational and trust boundaries, and seldom requires a single administrative server. DNS (domain name system) and Usenet are familiar examples of federated services.

Imagine you have a customer relations application that needs to retrieve customer information from a central server. With most distributed application technologies, you would be hard pressed to come up with a more federated design that would allow you to *roll over* in case of failure. Web services enable developers to create a more flexible design with ease. Gnutella, Freenet, and UDDI (Universal Description, Discovery, and Integration) are all popular distributed applications that are also federated ones. In part, their popularity is because of this nonlocalized design.

DEFINITION: ROLLOVER

A process of making resources and services available on a new system when the previous system fails. It is one of the most compelling reasons to use a federated design, but there are more!

Another goal of Web service architecture is to be general purpose. In other words, a mature Web service architecture can be applicable both to high-end server machines and to compact devices such as a cell phone. This general-purpose design means that the standards you use and learn now will be applicable to a wide range of scenarios and targets.

The Baseline Specifications of Web Service Architecture

The baseline specifications are the standards on which the Web service architecture builds to provide many of the other features needed in many distributed applications. The protocols that form the basis of the Web service architecture include SOAP, WSDL, and UDDI.

You can think of XML and HTTP as pre-baselines. But SOAP, UDDI, and WSDL are really where the Web services world begins in earnest. Chapter 2 (XML Web Services Standards) details how these standards are linked together, and how they provide a simple, modular architecture for building, describing, and finding Web services.

SUMMARY

The term *Web services* refers to a set of technologies that are the future of distributed computing. The reasons why they are so apt for distributed computing include the following:

- Web services are designed for interoperability across heterogeneous environments.
- Web services give us a loosely coupled messaging architecture that scales across the Internet.
- The builders of the Web service architecture have the benefit of hindsight and the many successes and failures of earlier attempts to build distributed architectures.