Kenn Scribner
Mark C. Stiver

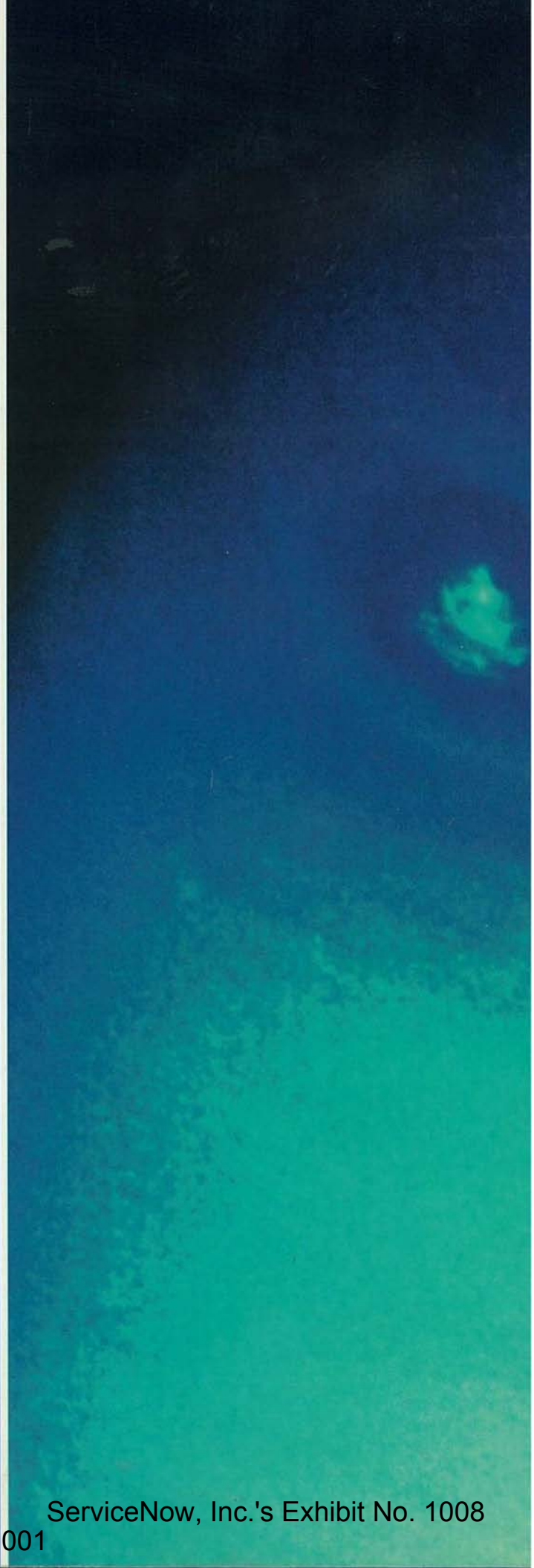
Applied SOAP:

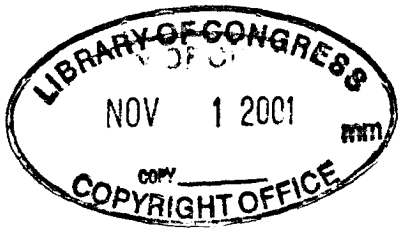
Implementing .NET XML Web Services

"The experience that Mark and Kenn have with XML Web Services really shines through in their writing, their explanations, and their examples. This book provides a great foundation upon which to build your understanding of how SOAP works within .NET."

—**Scott Seely**,
MSDN Architectural Samples Team

SAMS





Applied SOAP: Implementing .NET XML Web Services

Kenn Scribner and Mark C. Stiver

SAMS

201 West 103rd St., Indianapolis, Indiana, 46290

Applied SOAP: Implementing .NET XML Web Services

International Standard Book Number: 0-672-32111-4

Library of Congress Catalog Card Number: 00-110536

Printed in the United States of America

First Printing: October 2001

04 03 02 01 4 3 2 1

Associate Publisher
Linda Engelman

Acquisitions Editor
Linda Scharp

Development Editor
Laurie McGuire

Managing Editor
Charlotte Clapp

Project Editor
Leah Kirkpatrick

Copy Editor
Krista Hansing

Indexer
Larry Sweazy

Proofreader
Wendy Ott

Technical Editor
Scott Seely

Team Coordinator
Lynne Williams

Media Developer
Dan Scherf

Interior Designer
Gary Adair

Cover Designer
Gary Adair

Page Layout
Gloria Schurick

Overview

Contents

PART II Implementing Web Services

6 Web Services in ASP.NET 193

7 Consuming .NET Web Services 245

PART III More Advanced Web Services

Web Service Fundamentals

CHAPTER 1

IN THIS CHAPTER

It's pretty hard to pick up a trade magazine these days without seeing a headline about Web Services. With phrases such as "a new paradigm" being proliferated, are we really witnessing the genesis of a new technology?

Unfortunately, the answer isn't black and white—a lot depends on your perspective. Web Services can be used in a wide variety of ways, including these:

- Participating in business-to-business (B2B) transactions
- Exposing software functionality to customers
- Integrating heterogeneous platforms and programming languages
- Providing a simplified platform for product development

What Are Web Services?

Web Services can be described as any functionality that is accessible over the Internet, generally (but not necessarily) using one or more eXtensible Markup Language (XML) messages in the communications protocol. Web Services use the concept of an operation to represent the association of a request message to zero or more response messages. When these operations are combined to satisfy some particular purpose, they form an interface.

The Poor Man's Web Service

The Internet is already flooded with conventional types of Web Services, better known as Web pages. Users are expected to interact with the functionality behind the Web page through typical user-interface widgets such as forms, buttons, and so on.

We already know how to reuse Web functionality by embedding other Web pages into our own pages through frames and links. But this presentation-based approach severely limits the things that you can accomplish. If you embed another Web site within a frame of your own, you generally have no control over the colors, graphics, or other aspects of the presentation. Another problem is that any information entered by a user in the embedded page never gets back to your controlling application. In other words, you're out of the loop!

One way around this is for your application to act as a proxy for the user. Many developers have already written simple applications that navigate to a particular URL, screen-scrape the Web site's HTML for information, and use that information to build new Web content. Consider the following HTML that describes the current weather temperature:

```
<HTML>
   <HEAD>
      <TITLE>Today's Weather</TITLE>
   </HEAD>
   <BODY>
```

```
      <P>City: <B>Los Angeles</B></P>
      <P>State: <B>California</B></P>
      <P>Temperature: <B>83</B></P>
   </BODY>
</HTML>
```

In this case, it would be fairly simple to programmatically locate the temperature value within the markup. However, over an extended period of time using this service, you can be sure that the underlying Web page will change and ultimately break your application. It would be nearly impossible to develop software that could automatically adjust to fluctuations in the type of markup, as shown:

```
<HTML>
<HEAD>
   <TITLE>Today's Weather</TITLE>
</HEAD>
<BODY>
   <TABLE WIDTH="500" CELLPADDING="10" CELLSPACING="15">
      <TR>
         <TD ALIGN="LEFT" VALIGN="MIDDLE" WIDTH="100"><B>City</B></TD>
         <TD ALIGN="LEFT" VALIGN="MIDDLE" WIDTH="200">Los Angeles</TD>
      </TR>
      <TR>
         <TD ALIGN="LEFT" VALIGN="MIDDLE" WIDTH="100"><B>State</B></TD>
         <TD ALIGN="LEFT" VALIGN="MIDDLE" WIDTH="200">California</TD>
      </TR>
      <TR>
         <TD ALIGN="LEFT" VALIGN="MIDDLE" WIDTH="100">
            <B>Temperature</B>
         </TD>
         <TD ALIGN="LEFT" VALIGN="MIDDLE" WIDTH="200">83</TD>
         <TD ALIGN="LEFT" VALIGN="TOP">
            <IMG SRC="sunny.gif" WIDTH="25" HEIGHT="33">
         </TD>
      </TR>
   </TABLE>
</BODY>
</HTML>
```

Now consider integrating multiple systems using this approach. Recall that in the Web Services paradigm, many systems likely could participate in some business process. Because HTML content changes at such a fast pace, you likely will not ever be able to construct a reliable integrated solution.

For example, consider one application that monitors the temperature Web site and another Web site that posts the average speed of traffic on a nearby highway:

```
<HTML>
   <HEAD>
      <TITLE>Interstate Traffic Report</TITLE>
   </HEAD>
   <BODY>
      <CENTER>Average Speed</CENTER>
      <CENTER>67</CENTER>
   </BODY>
</HTML>
```

By relating these two axes of data, the application might be capable of determining whether there is a correlation between sunny days and fast driving. Although it might be an interesting problem to solve, the likelihood of the application working with 24×7 reliability is extremely low. The plain-and-simple fact is that relying on presentation-oriented data leads to a tightly coupled and brittle system.

The question isn't whether the concept of integrating Web content is valid; the problem lies within the information that can be obtained from a source. Without rich content markup, programs don't have much of a chance of locating pertinent information. Of course, this is where XML markup makes an important difference. Given standardized markup that describes information in a particular domain space, an application should always be capable of finding the right data.

Taking the concept of standardized markup into account, Web Services can be better defined as functionality that is accessed over the Web and that provides information in a reliable and predictable manner. In many cases, this predictability will be realized through the use of XML markup for describing information.

Although Web Services are not limited to the following technologies, you will find that a large percentage of Web Service implementations are built upon the Hypertext Transfer Protocol (HTTP), SOAP/XML as a messaging protocol, and Web Services Description Language (WSDL) as a way to describe service interfaces.

The basic idea behind Web Services isn't really new. In many ways, we are just reusing technologies that most of us have used for years. Surprisingly, many developers have already built systems using Web Service techniques, but in a very ad hoc and proprietary way. The main difference is that the industry is now supporting Web Services with standards, tools, and implementations.

First reactions about Web Services usually revolve around performance. Most people recognize that transmitting XML is not the most expeditious way for systems to communicate. So why use XML? We use XML because it provides us with a predictable way to package information that is structured, extensible, and yet still very easy to use—not something that can be said for other packaging protocols. Let's take a closer look.

XML Messages

By nature, interface-based programming enables us to build loosely coupled systems, meaning that the client and Web Service are independent of one another. This has been true in object-oriented programming for many years, within the confines of a particular programming language. Web Services reinforce loosely coupled systems by removing dependence upon a common programming language or even a common platform. This is realized through the use of XML messages, which define the operations inside a Web Service interface.

The importance of this feature is well understood by distributed application developers who have been using systems such as CORBA and DCOM. Historically, building applications on top of binary protocols and their associated runtimes results in a very tight dependency between the client and the server. This forces developers to repeatedly build and distribute new interface components (such as proxies and stubs), which is a very tedious and error-prone process. More importantly, though, XML lets you focus on the interface semantics rather than having to worry so much about synchronizing parameter lists of remote methods.

Syntax Versus Semantics

Recall that *syntax* is the detailed representation of information. It's the way you organize instructions in a programming language or arrange tags in an XML document.

Semantics, on the other hand, refers to the meanings or concepts behind a syntactical representation. Because semantics represents information from a logical standpoint, there might be several ways to syntactically represent that information, all of which should convey the same meaning to the information consumer.

To better contrast syntax and semantics, consider the following sample XML:

```
<ChargeCreditCard>
    <amount>150.00</amount>
    <creditCardNumber>123456789</creditCardNumber>
    <expirationDate>2003-01-31</expirationDate>
</ChargeCreditCard>
```

In this case, the syntax is fairly simple—an XML message consisting of start and end tags, structured with a single root element and its descendants. We could have just as easily used the following text:

```
Please charge $150.00 to credit card number 123456789, which expires January
31, 2003.
```

Semantically, the information represented by both syntaxes allows you to bill someone's credit card, which is really what we're interested in. Obviously, the latter syntax is more pleasing to humans, and the former XML message is much more acceptable for application consumption.

SOAP uses XML to define a syntax, which makes it very easy to represent information in a structured form. However, SOAP also carries some important protocol semantics that allow SOAP processors to serialize/deserialize data, handle faults, and mandate that certain information be present in a message.

As the creator of a Web Service, you have the task of defining your own set of semantics for your Web Service. Some simple semantics might be to get a stock quote or to retrieve the time and temperature. A more advanced semantic might be to schedule a vacation, which includes reserving a hotel room, airfare, and ground transportation. Arriving at a reasonable set of semantics requires you to use standard software engineering practices such as working with domain experts.

The long-term vision of Web Services is for developers to be able to construct applications by integrating one or more units of functionality into a single service (as in the vacation example). The most significant aspect of this model is that you can incorporate distinct units of functionality from a wide variety of sources and successfully complete some larger task or business process. It is like code reuse, without the programming language compatibility problems.

To describe the interaction between Web Services and their clients, it's important to define some terminology that will be used in this context.

Web Service Terminology

The Web Services model uses several terms that help to identify the various roles in a typical Web Service scenario.

A *service provider* is an entity that hosts a Web Service that exposes some functionality. The service provider is responsible for defining the semantics of the service interface as well as constructing the appropriate physical representation as depicted in a Web Service description document.

The service provider can then publish the interface description to a *service registry*. Here, information about the provider and the service are persisted for service discovery. The service registry exposes its own set of interface semantics that allows others to create new entries, update registry information, or query for specific registry parameters. For more detailed information about service registries, refer to Chapter 5, "Web Service Description and Discovery."

At some point after service publication, a *service requestor* (sometimes referred to as the client) can discover the Web Service and its interface description, and bind to this service to fulfill the service requestor's needs.

As you can see from Figure 1.1, three major processes take place. First, the service provider *publishes* service information. Next, the service requestor *finds* the service in the service

registry. Finally, the service requestor *binds* to the service to execute some functionality. This *publish, find, and bind* model is consistent with other networking protocols such as DNS.

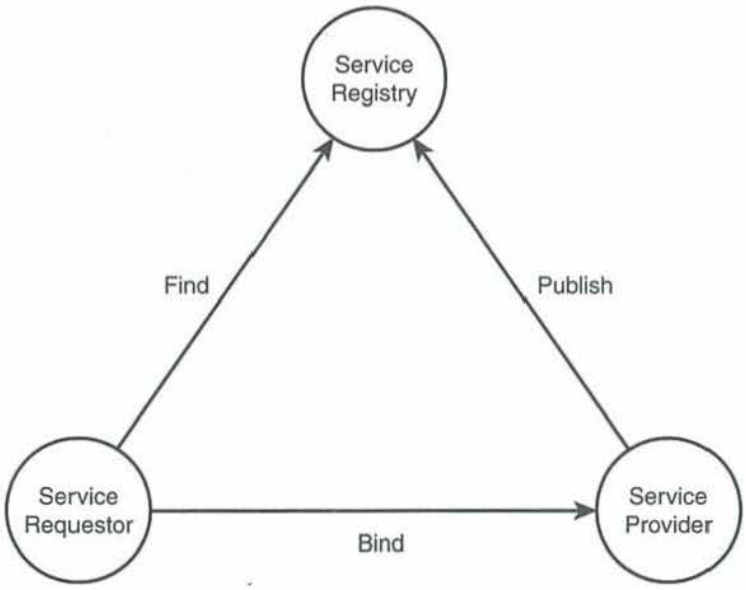


FIGURE 1.1
The Publish, Find, and Bind Model.

Now that you are familiar with the concept of Web Services, let's take a quick tour of how Web Services came about and why.

The Road to Web Services

Most things in software engineering happen for a very good reason—or, at least, we hope so. This is how concepts such as *abstraction* and *encapsulation* have become mainstream. As with any technology, we learn from past mistakes and capitalize upon our successes.

Web Services are no different. They have materialized from a variety of Web technologies that have been proven to work in the widely distributed environment of the Internet. The Internet itself has evolved over the years, spawning many new ideas and concepts that have contributed to the Web Service approach.

Waves of the Internet

Since the beginning of the Internet, many changes have come about in networking technology, security, system scalability, and many other areas of distributed computing. Overall, we believe that the Internet has succumbed to four major waves of development.

The first wave of the Internet started around the 1970s with some very important government research, specifically the Defense Advanced Research Projects Agency (DARPA). This is

where Transmission Control Protocol/Internet Protocol (TCP/IP) was born. Its goal was to interconnect computer systems through a complex architecture of networks and subnetworks. Over the years, a variety of physical networks (such as Ethernet) and routing technologies evolved to the point that, in 1990, more than 200,000 computers were interconnected on the Internet.

Although Internet connectivity was one of the most significant achievements in computing, it meant very little without applications to drive it. This is where the second wave of the Internet began (roughly in the 1980s). Tools such as FTP and Telnet gained in popularity by allowing system users to remotely access other computers. Although the tools were crude, compared to today's standards, the underlying protocols that they used where quite elegant.

In the early 1990s, the Internet began to seep into more sophisticated applications and led to the dawn of the Web, which marks the third wave. Browsers—and eventually Java applets—allowed the general consumer to experience interconnected communities of users. Of course, where there are consumers, there are vendors. This spawned more electronic business opportunities, as evidenced by the plethora of electronic storefronts and shopping carts.

All this, of course, has brought about the fourth wave of the Internet, which is the focus of this book—Web Services. Here, the goal is for multiple *diverse* applications to communicate so that they can execute some task. Not only does this improve the user's experience, but it also offers the ability for you to integrate functionality at a much lower cost than developing it all yourself. From the user's standpoint, all of this is orchestrated from a single application. But behind the scenes, one or more additional applications will likely participate. The key is that the applications work while remaining oblivious to vendor-specific technologies being used by the participating services.

Looking back, each wave introduced new Internet standards that facilitated the next wave of development.

Internet Standards

In April 1969, the first Request for Comments (RFC) was published at UCLA (RFC 1), and thus began the process of sharing ideas in computing for a much greater cause. Then, in 1986, the Internet Engineering Task Force (IETF) was officially created. Its charter was (and still is) to evolve the architecture of the Internet using open contributions from the research and development community.

After inventing the Web, Tim Berners-Lee decided to create the World Wide Web Consortium (W3C) in October 1994. The W3C's purpose is to promote interoperability and open forum discussions about the Web and its protocols.

These organizations have led the way to standardization, a process that has resulted in a strong foundation for the Web Service infrastructure.

Several standards are prevalent in current Internet development. Some have existed for years, and others are relatively new, and not necessarily standards. This section summarizes these technologies and shows how they apply to Web Services.

HTTP and SMTP

As stated before, TCP/IP is the foundation of Internet communication protocols. However, TCP/IP without an application is a little like a car without a driver. How an application uses TCP/IP also determines the semantics of that application's protocol.

Application protocols such as HTTP and Simple Mail Transfer Protocol (SMTP) already have predefined semantics and behavior that determine how they should be used. For HTTP, the semantic implies a request/response model designed to serve Web resources such as HTML or JPG files. SMTP, on the other hand, implies a one-way request/acknowledge semantic designed to transmit text-based email messages in a *fire-and-forget* manner.

NOTE

Fire-and-forget is a military warfare term that has been overloaded for networking purposes. We use it to describe the process of sending a message: The sender does not require any acknowledgement that the recipient actually received the message.

In the context of Web Services, these *application protocols* are used to carry additional semantics, such as those specified by SOAP. SOAP, in turn, provides a way for you to define your application semantics that are also carried over these application protocols. This layering of semantics just reemphasizes the flexibility of Web Service protocols.

Because HTTP is the dominant protocol being used for Web Services, let's take a quick look at the two most common aspects of HTTP being exploited, the `GET` and `POST` verbs.

The following is a sample HTTP `GET` request:

```
GET /default.htm HTTP/1.1
Accept: text/*
Host: www.mcp.com
{CR}{LF}
```

In this case, the client is requesting that the www.mcp.com server return the default.htm resource. The client would like to use version 1.1 of the HTTP protocol in this transaction and is willing to accept the resource as some form of text. Notice that the message is terminated by the carriage return/line feed pair following the message.

Many times you need to provide application-specific information to the server that is not represented in the semantics of the HTTP protocol. For instance, you can pass parameters on the URL, as shown:

```
GET /GetStockQuote.asp?symbol=MSFT HTTP/1.1
Accept: text/*
Host: www.mcp.com
{CR}{LF}
```

Given this sample request for obtaining stock quote information, the `symbol` parameter and its value are passed on the query string, and you can expect the server to respond with some form of HTML, such as the following:

```
HTTP/1.1 200 OK
Content-type: text/html

<html><head><title>Microsoft Stock Price</title></head>
<body>
    <b>Microsoft: 80.75</b>
</body>
</html>
```

The server's response contains the HTTP version, a status code and message, and the content type that is associated with the related payload following the carriage return/line feed pair.

However, using the `GET` request is less than optimal when dealing with large and complex parameters. Certain HTTP implementations and older firewalls have been known to truncate URLs based on poorly chosen size limits. URLs also undergo encoding for a large number of characters, which complicates processing procedures.

Instead, we can use the HTTP `POST` verb, which places information within the Body of the request message:

```
POST /GetStockQuote.asp HTTP/1.1
Accept: text/*
Host: www.mcp.com
Content-type: text/xml
Content-length: nnnn


<Symbol>MSFT</Symbol>
```

Similar to the `GET` verb, the `POST` verb also identifies a resource, the version of HTTP, and the content that it expects to receive. However, two additional HTTP fields are provided—`Content-type` and `Content-length`. The two new fields refer to the remainder of the message, which, in this case, happens to carry an XML message.

Here you can see that we are no longer bound by limitations of the URL. There is less character encoding taking place, and we are able to transmit more complicated payloads—this is by far the most helpful aspect of HTTP when used in the Web Service model.

NOTE

Although we generally recommend that you use the `POST` method, in some situations using `GET` might make sense. `GET` is best used for simple semantics that require only minimal message structure and in situations when client applications don't have control over the `POST` payload.

An example of this is with the Visual Studio .NET test pages that are generated with your Web Services. Because the browser's `POST` feature does not allow you to place customized information in the message Body, the `GET` verb comes in very handy.

Also note that your test pages won't support complicated parameter types (same goes for the WSDL for HTTP `GET` or `POST`), so you'll be required to manually build a client that can exercise your service in this case.

eXtensible Markup Language (XML)

Although XML has such a wide variety of uses, it makes a great foundation for Web Services for many reasons:

- In a world where global business arrangements are becoming the norm, XML natively supports different character sets through Unicode (UTF-8, UTF-16, and so on).
- XML promotes interoperability in a platform-agnostic way by promoting the convergence of information to a common, vendor-neutral state.
- Probably most importantly, XML is simple.

As we briefly mentioned before, XML is really just syntax for application semantics that you must define. To define semantics and the associated XML syntax for your semantics, you need to establish the appropriate structure and restrictions of your markup—you do this through a schema.

XML Schemas

When you want to describe (and possibly validate) an XML document, you might use a Document Type Definition (DTD), such as this one:

```
<!-- House DTD -->
<!ELEMENT house (address)>
<!ATTLIST house bedrooms CDATA #REQUIRED
                bathrooms CDATA #REQUIRED>
<!ELEMENT address (street, city, state, zip)>
```

```
<!ELEMENT street (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ELEMENT state (#PCDATA)>
<!ELEMENT zip (#PCDATA)>
```

As you can see, though, DTDs are limited by their somewhat cryptic syntax and lack of type checking.

In February 2001, the XML Schema specification was promoted to Recommendation status by the W3C. XML schemas not only encapsulate the same feature-function as DTDs, but they also offer complex type checking, all wrapped up in an XML language. This makes XML schemas the preferred method of describing all forms of XML documents, including XML messages. The same example is shown in XML schema format, as follows:

```
<!-- House XML Schema -->
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:complexType name="AddressType">
    <xsd:sequence>
      <xsd:element name="street" type="xsd:string" />
      <xsd:element name="city" type="xsd:string" />
      <xsd:element name="state" type="xsd:string" />
      <xsd:element name="zip" type="xsd:decimal" />
    </xsd:sequence>
    <xsd:attribute name="bedrooms" type="xsd:positiveInteger" use="required" />
    <xsd:attribute name="bathrooms" type="xsd:positiveInteger" use="required"/>
  </xsd:complexType>

  <xsd:element name="house" type="AddressType" />
</xsd:schema>
```

SOAP

Although SOAP isn't officially an Internet standard, it has been widely adopted by the Internet community, including the Electronic Business XML (ebXML) organization, for its transport and routing layer.

To summarize SOAP in a single word, *packaging* is the most appropriate description. Many developers have created ad-hoc approaches for sending XML messages between their applications. The creators of XML-RPC took the concept to the next level by exposing a publicly available specification for XML messaging. Taking XML-RPC a step further, SOAP basically defines a standard yet extensible way to wrap information in XML so that both ends of the connection (and potentially everything in between) can understand how to open this package. Let's take a quick look at a SOAP request message:

```
<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:transId xmlns:t="http://www.mcp.com/trans">
          87654
       </t:transId>
    </SOAP-ENV:Header>
    <SOAP-ENV:Body>
       <m:GetStockQuote xmlns:m="http://www.mcp.com/stock">
          <Symbol>MSFT</Symbol>
       </m:GetStockQuote>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

This simple message shows the general packaging of a SOAP message. The Envelope contains an optional Header and a mandatory Body. The Header is used for out-of-band information that doesn't necessarily apply to the semantics of the message Body. The Body is used to carry the application-specific message content. This is definitely not all that SOAP represents, but it captures the spirit of what SOAP set out to accomplish. We'll leave it to Chapter 4, ".NET Web Services and SOAP," to provide the gory details about the remainder of the SOAP protocol.

WSDL and UDDI

WSDL is the description language that is used to describe how software must interact with a particular Web Service. Clients use WSDL documents to understand the logical structure and the syntax of a Web Service. WSDL also provides message-exchange patterns, service bindings, and references to the location of a service.

Growing in popularity, Universal Description, Discovery, and Integration (UDDI) is one way for the publish, find, and bind process to be accomplished. UDDI servers allow WSDL to be published and propagated across the Internet so that clients can ultimately consume a given service.

Although neither WSDL nor UDDI has been standardized, the industry is giving both the most attention of any similar mechanisms. A great deal more about WSDL and UDDI will be explained in Chapter 5.

Uses for Web Services

At the beginning of this chapter, we briefly mentioned several areas where Web Services can have an impact on the solutions that you build. Let's look at a few of these areas a little closer.

Business-to-Business

One area that clearly deserves attention is in the business-to-business (B2B) paradigm. Most companies today still operate on paper systems. This is due in part to the simplicity and low

cost of entry into paper-based systems. For more than two decades, developers have been trying to integrate business processes using a combination of software and communication protocols.

Electronic Data Interchange (EDI) has been the technology of choice for many years. Unfortunately, EDI has been a technology that only large corporations could afford to leverage. Not only was EDI expensive, but it also could require years of effort to become fully integrated into a business process. Although the intentions were good, the complexity of the underlying technologies was not cost-effective for many companies.

When XML took center stage, it became apparent that XML could facilitate the low-cost and simplistic approach that was necessary for the B2B task.

Right around the advent of SOAP, the Organization for the Advancement of Structured Information Standards (OASIS) formed a group to design an XML-based technology that could become the de facto standard for business communication. This effort is known as Electronic Business XML (ebXML) and is sponsored by OASIS and the United Nation's Center for Trade Facilitation and Electronic Business (UN/CEFACT).

Because most of the ebXML group has been working with EDI for years, it had a clear idea of how to build a global electronic marketplace. Initially, the ebXML team set a course to deliver a framework within 18 months of the group's forming. In May 2001, the group did just that.

What was so significant about ebXML was that the ebXML Transport, Routing, and Packaging team began developing an XML-based protocol that did not involve SOAP. Fortunately, before the release in May 2001, the TRP team reevaluated the SOAP specifications and eventually incorporated SOAP and its use with Multipart MIME.

Exposing Functionality to Customers

Although this is a similar topic, don't confuse this type of service with standard B2B operations as addressed by ebXML. Here we're talking about something like a billable service or a deliverable that you expose to a customer through a Web Service.

An example might be the history report of a car—but not just any report, especially not one that is displayed in a Web page. Rather, this information must be provided in XML so that a car dealer can build a used-car evaluation system that automatically approves or rejects trade-ins.

Consumer-related services such as the Hailstorm services being developed by Microsoft provide the capability to store documents, calendar information, and even your favorite Web sites in a common repository.

Integrating Heterogeneous Systems

It's normal for companies to have computing systems dispersed across multiple platforms. This can be the result of system evolution, cost reduction, or varying developer experience. Commonly, it's the result of architectural disagreements between developers that prefer one platform to another.

The typical RPC-like protocols have never proved to be a satisfactory solution to this problem. While it's possible to get disparate systems to communicate, the time involved in doing so usually outweighs the benefits. Why is it so difficult to get disparate systems to communicate? One reason is that vendors choose to deviate from standards to squeeze more performance out of their platforms. Interpretation of standards is another cause of protocol incompatibility, mostly because specifications are complex and have areas of ambiguity that aren't easily resolved without vendor cooperation.

If you can reduce the development time to a manageable amount and remove the vendor-specific protocols, you can capitalize on Web Services in the enterprise in many ways.

We won't try to convince you that Web Service performance is on par with binary protocols—the additional overhead in text processing alone suggests that they're not. But we can tell you that it takes less than a week to get two dissimilar systems communicating using XML and HTTP. And with network and processor speeds rapidly improving, ease of development may be more beneficial than raw speed. Ultimately, you'll have to decide which type of performance means the most to you.

Rapid Development Environment

Web Services can make your project development environment operate faster, for several reasons.

As mentioned before, heterogeneous systems can be hooked together to build a solution. This enables developers to work in their preferred environment, while still allowing them to produce an integrated solution. If at some point you decide to move a service from one platform to another, clients will be unaware of the change, as long as you maintain the same interface. A platform-independent programming environment such as .NET adds to this capability by reducing the amount of code that actually needs to be rewritten.

Web Services also force you to think in terms of interfaces. You don't have to worry about compiling another developer's source code or linking in libraries. Stack traces and memory dumps are partially replaced by traces of XML messages.

This usually has the positive side effect of breaking down services into logical subcomponents. You can then isolate functionality and test it accordingly. Because of this, Web Services can be

easier to test. You can fairly easily trace Request messages and save them in text files so that later they can be played back to regression test your interfaces.

Web Services can offer such an improved development environment because of the fundamental Web Service properties. We'll take a look at these next.

Web Service Properties

Many times we become so enamored with new technologies that we don't always realize the benefits and costs of using them. It's important to explore these so that you can make informed decisions about the systems you are building.

Performance

Although there are many types of performance, we are specifically referring to raw communications performance at this point. This type of performance always seems to be one of the first topics that come up in discussions of Web Services. It's a perfectly reasonable concern because we're talking about using a text-based protocol (such as SOAP and XML).

Losses Versus Gains

The fact is, when using XML and HTTP, you definitely sacrifice network performance when compared to binary protocols such as CORBA. Various unofficial tests have shown performance degradation at five times slower (or worse) than a binary protocol. Although most of the performance cost can be attributed to XML parsing, character encoding and socket setup and teardown costs also have an impact. However, as we've already mentioned, network and processor speed improvements will eventually make these concerns nonissues.

But consider what you gain from this loss in performance. First, you gain native Unicode support. By default, XML is designed to accommodate most forms of languages, making it a truly global protocol. This might not be important to you now, but in an Internet economy that has no physical borders, globalization will play an important part in future B2B efforts.

You also gain loose coupling between systems. This is by far one of the most positive aspects about Web Services—and SOAP, in particular. Developers are constantly challenged to work with different operating systems, runtime platforms, programming languages, and so on. Loose coupling enables you to hide the implementation details of a service from the service requestor, giving both parties a great deal of flexibility and choice. This leads to the topic of another type of performance: developer performance. Because Web Services are agnostic to implementation details, programmers have the opportunity to choose the programming environment that makes them the most productive. In a multi–Web Service environment, loose coupling also allows developers to work independently on these services. This helps to improve parallel development efforts.

Finally, you gain an extensible protocol that can grow over time. With so many changes occurring in the Internet, you cannot afford to stagnate. Systems must be capable of adapting faster and faster. Built-in mechanisms for extensibility (as demonstrated in SOAP) are necessary to protect systems from becoming brittle.

Improving the Performance of Web Services

But don't go away feeling that all performance is lost. You can do a few things to improve the overall performance of your application.

One of the most significant ways is to be aware of how you define your interfaces. We tend to think in terms of standard distributed RPC architectures. With these systems, we usually have a distributed runtime that is managing state and maintaining an open connection at all times. RPC architectures do this because setup and teardown of connections takes a significant amount of time. When using protocols such as HTTP for Web Services, you have to plan on connections coming and going to achieve high scalability.

To account for this, you must maximize the amount of information that you transmit on each call. However, this is not always an easy task because you can't afford to change the semantics of an interface just to improve performance. You will need to spend a great deal of design time determining the best way to convey your semantics in the fewest number of messages possible. Just remember, you're better off sending a little too much information than not sending enough and requiring additional round trips.

It's also important to understand the needs of your service's typical service requestor and adjust the interface appropriately. For instance, if you know that a client will typically ask for A and then almost immediately ask for B, maybe your interface should always return B with A, saving that additional round trip. This is a simple but common example in distributed systems development.

You should also be aware that how a service requestor is implemented could make a significant difference in the service requestor's performance. Although you shouldn't care about the service requestor's implementation—and, in most situations, it's out of your control—the fact remains that poor implementations can be directly attributed to lousy performance. A common improvement can be found by using SAX parsers over DOM parsers. There are certainly advantages to each, but, in many cases, SAX will help your applications get faster. Don't be overly discouraged by parsing speed, though; vendors are developing new ways to improve parse times, as you've already seen in newer generations of XML parsers.

User Perceptions

Another client-related area that you might have control over deals with *perceived performance*. This type of performance is directly related to the end user's experience. You can identify these

areas for improvement when your XML messages have a direct effect on a user interface. In this case, it might be possible to provide partial user feedback before all transactions in a business process complete. This is similar to the way Web pages are rendered in a browser—as a user, it's helpful to see the first screen of a document while the remainder is downloaded in the background. Be careful when architecting your systems around presentation-oriented behavior—you want to avoid sacrificing semantics for performance whenever possible.

Simplicity

Compared to many of the distributed systems currently in the industry, Web Service dwarfs the others when it comes to simplicity.

Granted, some of the technologies surrounding Web Services have their share of complexities, but XML and HTTP generally are not very difficult to use. Plenty of software is available to use XML and HTTP, not to mention the growing number of SOAP implementations that are freely available (see Appendix D, ".NET Web Service Resources").

Building software to use a Web Service is also very easy. When you understand the semantics of the Web Service, constructing the code to interact with the service is usually straightforward. This is because the programming model for Web Service is fairly simple. It forces you to separate interface from implementation, a concept that is well understood by the object-oriented programming community.

Security

The Internet has brought security to the forefront of everyone's minds. Viruses, hackers, and the like all lead to concerns from both a consumer and a developer standpoint. The question of whether Web Services make you more susceptible to security risks doesn't have a clear answer. On one hand, Web Services don't introduce any new technologies, but they do introduce yet another application of existing technologies that can be attacked.

The Secure Socket Layer (SSL) is one level of security that encrypts SOAP messages at the HTTP transport layer, usually referred to as HTTPS. Without the proper SSL certificate information, an interceptor would have no way of knowing the actual contents of the underlying HTTPS message.

In addition to SSL, a lot of interest has arisen in extending SOAP's security prospects. This has led to a W3C submission called *SOAP Security Extensions: Digital Signature*. In a nutshell, this promotes a standard way to use digital signatures in XML to sign SOAP messages. Although this does not prevent others from peering into your messages or replaying messages to your applications, it does allow you to verify the integrity of the message as well as its origin. To accomplish this, the specification proposes using the SOAP Headers to carry digital signatures that relate to some other portion of the SOAP Envelope.

At this point, the major aspects of security that you should concentrate on are authentication, authorization, and encryption. You can perform authentication in several ways, including using Microsoft's Passport, using client certificates, or implementing your own authentication scheme. In the latter case, SSL remains the definitive mechanism for securing a connection so that you can perform your authentication work.

Reliability and Availability

How many times have you attempted to point your browser to a Web page, only to find nothing but a sleepy status bar followed by that ominous and yet oh-so-familiar HTTP 404 error?

When you consider the amount of network traffic that the Internet carries today and the number of complex systems that must work together to keep the Internet running, it's amazing that this type of error doesn't occur more often. However, for many business processes (such as a financial transaction), even a minor hiccup in the network can cause great damage.

The best way to solve this problem is to use a transport protocol that guarantees delivery. For instance, a message queue that supports ACID transactions can ensure that messages arrive at their destination.

NOTE

Recall that ACID stands for Atomic (interrupted work can be undone), Consistent (resource integrity is preserved), Isolated (it works independent of other transactions), and Durable (the results are permanent).

For many people, this level of reliability is not worth the incompatibilities of message-queuing systems. This is why we're using Web Services in the first place—so that we don't force both ends of a connection to share some proprietary protocol.

Until more sophisticated and standardized systems are put in place, the best recommendation that we can make is to build your systems with the most atomic interfaces possible. This minimizes the sequencing of requests, thus reducing the risk of taking a particular message out of context. And as any distributed application programmer knows, you program defensively for communication problems and deal with exceptions in the most robust way possible, trying to eliminate the possibility of leaving your system in an unstable state. This suggests that, at a bare minimum, you build in mechanisms so that changes can be *backed out*.

Consistency

Although this can be related to reliability and availability, we refer to consistency as the capability of a service requestor to faithfully expect the interface to perform in a manner similar to

past requests. In other words, you can't arbitrarily change the semantics or syntax of a published interface without potentially causing problems for clients. And if you are charging for your service, you can be sure that customers will not find change very pleasing.

We mentioned that you shouldn't change the syntax of an interface, and to some extent that's true. However, as discussed earlier, syntax is normally the part of a message that computers can understand. Thus, it's not overly unreasonable to expect SOAP infrastructures to eventually be capable of automatically adjusting for small syntactic changes, assuming that the semantics remain the same.

You've read a lot about Web Services and many of the issues surrounding them. Now it's time to dive into some code and see a Web Service in action.

Creating a Web Service in Visual Studio .NET

In this section, you will see how Visual Studio .NET helps you generate a Web Service, as well as build a consumer application of that service. You will see how WSDL plays a critical part in bringing together the service requestor and the service provider. Finally, you will have an opportunity to use a network-tracing application to analyze request and response SOAP messages.

Creating the Service

In this example, you will create a Persistence application that allows a client to store name/value pairs on the server. Although this implementation of a *property bag* is far from full-featured, you can see how a simple Web Service can bridge the gab between heterogeneous systems wanting to share state information. The following steps walk you through creating a Persistence application:

1. Start a new ASP.NET Web Service project called Ch1WebService—in this case, using C#.
2. Select http://localhost for the location of your service. (This step assumes that you are running a local copy of IIS with the appropriate server extensions.)
3. Look under your system's wwwroot folder (by default located in C:\inetpub\wwwroot), for a folder with the Ch1WebService project name. Within that folder there are several hidden folders and a bin folder that, at this point, should be empty. After you build the project, the bin folder will be populated with the necessary files for your service to execute.

NOTE

Hidden files can be seen by enabling this option under the Tools, Folder Options menu, under the View tab in Windows Explorer.

4. Go back to the Ch1WebService project folder. Among the many files available, there is a Service1.asmx file that is the entry point into your Web Service. You might want to take a look at the contents of this file to see how it is organized.
5. Looking at the Solution Explorer in Visual Studio .NET, note that Service1.asmx is also listed under the Ch1WebService project. To change the name of your service, you must alter the properties of the Service1.asmx file and change it to Persistence.asmx. You will also need to change the contents of this file so that the `Class=` value reflects the new Persistence service name in this assembly. Finally, you must change the name of the `Service1` class in the source code.
6. Press F5 (Debug mode) to see Visual Studio .NET build your project and launch a browser window showing a sample Web page. At the top of the page, you will see a hyperlink to the service description of your service. Notice that there isn't much here because you haven't created any Web-enabled methods yet. You will also see a warning on the page about the default namespace being used; we will assign a real namespace to the service shortly.
7. Close the browser window. Visual Studio .NET should exit Debug mode.
8. Enter the service code. Listing 1.1 shows the final code that will be used for this example.

LISTING 1.1 Form1.cs

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Web;
using System.Web.Services;

namespace Ch1WebService
{
   /// <summary>
   /// Summary description for Persistence.
   /// </summary>
   [WebService(Namespace="http://www.mcp.org/WebServices/Persistence")]
   public class Persistence : System.Web.Services.WebService
   {
      public Persistence()
      {
         //CODEGEN: This call is required by the ASP.NET Web Services Designer
         InitializeComponent();
      }
```

LISTING 1.1 Continued

```
#region Component Designer generated code
/// <summary>
/// Required method for Designer support - do not modify
/// the contents of this method with the code editor.
/// </summary>
private void InitializeComponent()
{
}
#endregion

/// <summary>
/// Clean up any resources being used.
/// </summary>
protected override void Dispose( bool disposing )
{
}

[WebMethod]
public string Get(string myName)
{
   string s;

   try
   {
      // Retrieve the named value from the Application property
      s = Application[myName].ToString();
   }
   catch (Exception e)
   {
      s = e.Message;
   }

   return s;
}

[WebMethod]
public void Put(string myName, string myValue)
{
   try
   {
      // Assign the named value to the Application property
      Application[myName] = myValue;
   }
   catch (Exception)
```

LISTING 1.1 Continued

```
        {
            // do nothing
        }
        return;
    }
  }
}
```

Listing 1.1 takes advantage of the `Application` state-management facilities provided through .NET. This allows you to store name/value pairs across multiple sessions of the service. Of course, this is really only a simulation of persistence and would be better implemented by actually storing the information in a database. With the current code base, the state is simply lost when the IIS service stops executing.

NOTE

As you develop a service, you might find yourself wanting to use the `Session` feature of .NET to maintain state for a specific client.

This is easy enough to do: Just use the `[WebMethod(true)]` attribute on each Web method to enable session state, and use `Session` in place of `Application`.

But be warned—`Session` relies upon HTTP cookies in the transport protocol. When testing with the browser (using HTTP `GET`), your application will perform exactly as you expected because IE will automatically pass cookies back and forth for you. However, client applications like the one you are about to build will not automatically support cookies. Thus, the application will not be capable of maintaining state across Web Service methods because each request will be considered a new session.

Overall, you should avoid using transport-level facilities such as cookies that bleed into application behavior. In other words, cookies don't appear anywhere in the SOAP message, but they can have a significant impact on the behavior of your application. Instead, you should create your own state-management values that are contained within the interface (ideally the SOAP Header) so that, regardless of the transport you choose, clients can always be sure of proper system behavior. Chapter 4 provides more information about modifying the SOAP Header.

Now that you've built a fully functional Web Service, you need to create a client that will access the service through the SOAP protocol.

Creating the Client

You've already seen the standard test Web client that Visual Studio .NET automatically creates for your project. This example shows you how to build a simple Windows application that can interact with your service.

1. Start by creating a new project. This time, though, you want to create a Windows application in C#. Call the project Ch1Client.
2. Place two buttons (named Get and Put) and two text boxes (named myName and myValue) on your form.
3. Next, right-click on References in the Solution Explorer window, and select Add Web Reference from the menu.
4. Click on the Web References on Local Web Server hyperlink, which displays all available services on your machine.
5. Click the Ch1WebService link in the right pane, which displays two additional hyperlinks, View Contract and View Documentation. If you click on the View Contract hyperlink, the WSDL for this service will be displayed. We'll cover this shortly.
6. Click on the Add Reference button to inject the Web Service description into your client project.
7. In addition to the code that is generated by Visual Studio .NET, add the Web Service-specific code fragment as shown in Listing 1.2.

LISTING 1.2 Form1.cs

```
private void Get_Click(object sender, System.EventArgs e)
{
   localhost.Persistence p = new localhost.Persistence();

   System.Windows.Forms.MessageBox.Show(p.Get(this.myName.Text));
}

private void Put_Click(object sender, System.EventArgs e)
{
   localhost.Persistence p = new localhost.Persistence();

   p.Put(this.myName.Text, this.myValue.Text);
   this.myName.Clear();
   this.myValue.Clear();
}
```

Adding the Web reference to your project (see Step 7) enables you to instantiate a `localhost.Persistence` object within the application code. When adding a Web Reference, you are physically including the WSDL file into your application. The compiler also creates a folder in the project (under Web References) called *localhost*, which includes a .CS file that contains a proxy for the service. If the Web Service changes at some point, you can simply right-click on the `localhost` folder and select Update Web Reference option. Be aware that you also might need to update your application code to conform to any new interface changes. However, the compiler will most likely catch these as well.

While we're on the topic of WSDL, take a quick look at the WSDL file shown in Listing 1.3.

LISTING 1.3 Persistence.wsdl

```
<?xml version="1.0" encoding="utf-8"?>
<definitions xmlns:s="http://www.w3.org/2001/XMLSchema"
             xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
             xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
             xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
             xmlns:s0="www.mcp.org/WebServices/Persistence"
             targetNamespace="www.mcp.org/WebServices/Persistence"
             xmlns="http://schemas.xmlsoap.org/wsdl/">
  <types>
    <s:schema attributeFormDefault="qualified" elementFormDefault="qualified"
              targetNamespace="www.mcp.org/WebServices/Persistence">
      <s:element name="Get">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="1" maxOccurs="1" name="myName"
nillable="true" type="s:string" />
          </s:sequence>
        </s:complexType>
      </s:element>
      <s:element name="GetResponse">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="1" maxOccurs="1" name="GetResult"
nillable="true" type="s:string" />
          </s:sequence>
        </s:complexType>
      </s:element>
      <s:element name="Put">
        <s:complexType>
          <s:sequence>
```

LISTING 1.3 Continued

```
            <s:element minOccurs="1" maxOccurs="1" name="myName"
nillable="true" type="s:string" />
            <s:element minOccurs="1" maxOccurs="1" name="myValue"
nillable="true" type="s:string" />
          </s:sequence>
        </s:complexType>
      </s:element>
      <s:element name="PutResponse">
        <s:complexType />
      </s:element>
      <s:element name="string" nillable="true" type="s:string" />
    </s:schema>
  </types>
  <message name="GetSoapIn">
    <part name="parameters" element="s0:Get" />
  </message>
  <message name="GetSoapOut">
    <part name="parameters" element="s0:GetResponse" />
  </message>
  <message name="PutSoapIn">
    <part name="parameters" element="s0:Put" />
  </message>
  <message name="PutSoapOut">
    <part name="parameters" element="s0:PutResponse" />
  </message>
  <message name="GetHttpGetIn">
    <part name="myName" type="s:string" />
  </message>
  <message name="GetHttpGetOut">
    <part name="Body" element="s0:string" />
  </message>
  <message name="PutHttpGetIn">
    <part name="myName" type="s:string" />
    <part name="myValue" type="s:string" />
  </message>
  <message name="PutHttpGetOut" />
  <message name="GetHttpPostIn">
    <part name="myName" type="s:string" />
  </message>
  <message name="GetHttpPostOut">
    <part name="Body" element="s0:string" />
  </message>
  <message name="PutHttpPostIn">
    <part name="myName" type="s:string" />
```

LISTING 1.3 Continued

```
    <part name="myValue" type="s:string" />
  </message>
  <message name="PutHttpPostOut" />
  <portType name="PersistenceSoap">
    <operation name="Get">
      <input message="s0:GetSoapIn" />
      <output message="s0:GetSoapOut" />
    </operation>
    <operation name="Put">
      <input message="s0:PutSoapIn" />
      <output message="s0:PutSoapOut" />
    </operation>
  </portType>
  <portType name="PersistenceHttpGet">
    <operation name="Get">
      <input message="s0:GetHttpGetIn" />
      <output message="s0:GetHttpGetOut" />
    </operation>
    <operation name="Put">
      <input message="s0:PutHttpGetIn" />
      <output message="s0:PutHttpGetOut" />
    </operation>
  </portType>
  <portType name="PersistenceHttpPost">
    <operation name="Get">
      <input message="s0:GetHttpPostIn" />
      <output message="s0:GetHttpPostOut" />
    </operation>
    <operation name="Put">
      <input message="s0:PutHttpPostIn" />
      <output message="s0:PutHttpPostOut" />
    </operation>
  </portType>
  <binding name="PersistenceSoap" type="s0:PersistenceSoap">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
style="document" />
    <operation name="Get">
      <soap:operation soapAction="www.mcp.org/WebServices/Persistence/Get"
style="document" />
      <input>
        <soap:body use="literal" />
      </input>
      <output>
        <soap:body use="literal" />
```

LISTING 1.3 Continued

```
      </output>
    </operation>
    <operation name="Put">
      <soap:operation soapAction="www.mcp.org/WebServices/Persistence/Put"
style="document" />
      <input>
        <soap:body use="literal" />
      </input>
      <output>
        <soap:body use="literal" />
      </output>
    </operation>
  </binding>
  <binding name="PersistenceHttpGet" type="s0:PersistenceHttpGet">
    <http:binding verb="GET" />
    <operation name="Get">
      <http:operation location="/Get" />
      <input>
        <http:urlEncoded />
      </input>
      <output>
        <mime:mimeXml part="Body" />
      </output>
    </operation>
    <operation name="Put">
      <http:operation location="/Put" />
      <input>
        <http:urlEncoded />
      </input>
      <output />
    </operation>
  </binding>
  <binding name="PersistenceHttpPost" type="s0:PersistenceHttpPost">
    <http:binding verb="POST" />
    <operation name="Get">
      <http:operation location="/Get" />
      <input>
        <mime:content type="application/x-www-form-urlencoded" />
      </input>
      <output>
        <mime:mimeXml part="Body" />
      </output>
    </operation>
    <operation name="Put">
```

LISTING 1.3 Continued

```
      <http:operation location="/Put" />
      <input>
        <mime:content type="application/x-www-form-urlencoded" />
      </input>
      <output />
    </operation>
  </binding>
  <service name="Persistence">
    <port name="PersistenceSoap" binding="s0:PersistenceSoap">
      <soap:address location="http://localhost/Ch1WebService/Persistence.asmx" />
    </port>
    <port name="PersistenceHttpGet" binding="s0:PersistenceHttpGet">
      <http:address location="http://localhost/Ch1WebService/Persistence.asmx" />
    </port>
    <port name="PersistenceHttpPost" binding="s0:PersistenceHttpPost">
      <http:address location="http://localhost/Ch1WebService/Persistence.asmx" />
    </port>
  </service>
</definitions>
```

Starting at the bottom of the file, the service is exposed through three different WSDL ports listed under the `<service>` element. One port operates on HTTP `GET` requests; another operates on HTTP `POST` requests. Most importantly, the third supports the SOAP protocol.

When running the Visual Studio .NET debugger, you've probably already figured out that the test pages generated for your service use the HTTP `GET` port.

NOTE

Be aware that Visual Studio .NET does not generate test pages for complex interface types. Therefore, you are responsible for manually constructing a test client.

You will also find at the top of the WSDL file an XML schema that describes the logical structure of your Web Service messages. Note that there is only one logical structure for each message, but each port represents a potentially different syntactical representation of each message.

Chapter 5 provides extensive coverage of WSDL, but it's helpful for you to see the types of information that a client application needs to communicate with the service.

At this point, you have learned how the service requestor and service provider are constructed and how they ultimately interoperate. Now let's take a closer look at the actual SOAP transactions that are transmitted over the wire.

Tracing Messages on the Network

When debugging your Web Service in Visual Studio .NET, you might have noticed that the generated test page displays sample messages for SOAP and HTTP `GET/POST`. At times, you will find it very helpful to see the actual messages that are being sent back and forth between endpoints.

The trace utility (MSSOAPT.EXE), provided with the SOAP Toolkit 2.0 binaries, is a favorite tool. For the trace to work, you need to start a new trace that will listen to a port (usually 8080) and forward requests to port 80, where your local IIS copy should be listening.

You then need to configure your client application to point to the newly created 8080 port. To do this in the sample client that you just created, open the WSDL file that is listed under the Solution Explorer window.

NOTE

Rather than changing the WSDL file, you may prefer to change your code by modifying the `p.Url` (from Listing 1.2) to point to the new location.

Edit the WSDL file so that the three ports' address value references `localhost:8080`, as follows:

```
<service name="Persistence">
    <port name="PersistenceSoap" binding="s0:PersistenceSoap">
      <soap:address
➥location="http://localhost:8080/Ch1WebService/Persistence.asmx" />
    </port>
    <port name="PersistenceHttpGet" binding="s0:PersistenceHttpGet">
      <http:address
➥location="http://localhost:8080/Ch1WebService/Persistence.asmx" />
    </port>
    <port name="PersistenceHttpPost" binding="s0:PersistenceHttpPost">
      <http:address
➥location="http://localhost:8080/Ch1WebService/Persistence.asmx" />
    </port>
  </service>
```

NOTE

You don't necessarily have to modify the location of all three bindings. Generally, you're probably interested only in the SOAP binding.

Rebuild and execute your project, and you will see the trace utility capture both the request and the response messages. For example, using the operations that you just built will result in the traces shown in Listings 1.4–1.7.

LISTING 1.4 PUT Request

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <soap:Body>
    <Put xmlns="www.mcp.org/WebServices/Persistence">
      <myName>phone</myName>
      <myValue>555-1234</myValue>
    </Put>
  </soap:Body>
</soap:Envelope>
```

LISTING 1.5 PUT Response

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <soap:Body>
    <PutResponse xmlns="www.mcp.org/WebServices/Persistence" />
  </soap:Body>
</soap:Envelope>
```

LISTING 1.6 GET Request

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <soap:Body>
    <Get xmlns="www.mcp.org/WebServices/Persistence">
```

LISTING 1.6 Continued

```
      <myName>phone</myName>
    </Get>
  </soap:Body>
</soap:Envelope>
```

LISTING 1.7 GET Response

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <soap:Body>
    <GetResponse xmlns="www.mcp.org/WebServices/Persistence">
      <GetResult>555-1234</GetResult>
    </GetResponse>
  </soap:Body>
</soap:Envelope>
```

This is clearly the only way that you can debug interoperability problems. Sometimes the *unformatted* trace also will be more appropriate because it shows the entire HTTP message, including the Header and the Body.

If you already have some experience with SOAP, you might have noticed that these messages are encoded using the literal form of an XML schema rather than SOAP's special encoding format (see Section 5 of the SOAP specification). This is the default behavior of a Visual Studio .NET–generated Web Service, but it can easily be switched using a Web Service attribute. More about this will be covered in Chapter 6, "Web Services in ASP.NET."

Interface Design Tips

You've already been introduced to interface semantics and the underlying concepts. Hopefully you came away with a clear understanding about the differences between syntax and semantics, and how semantics play such a critical part in building valuable Web Services.

This section presents a few practices that will help you build better Web Service interfaces.

Learning from the Past

It didn't take the software development community very long to figure out that interfaces are one of the best ways to decouple complex systems. This technique has been used for many years, especially in the electronics industry.

For example, consider the RCA jacks on a television. Most people know that there are three separate connectors—one for right channel audio, one for left channel audio, and one for the video signal. Each has a standard color-coding scheme, which makes it very easy to connect components.

You can take away several ideas from this example:

- **Ease of use**—Like the colors on connectors, interfaces should be easily recognizable and understood.
- **Distribution of functionality**—You need to strike a balance between having too many interfaces and having one overly complex and monolithic interface. One wire is easier for a consumer to connect, but if technical challenges of combining signals into a single connector force you to charge $10,000 for your DVD player, nobody wins.
- **Compatibility and overloading**—New components need to maintain compatibility with older interfaces, and new interfaces should be created when old ones no longer meet your needs. This is analogous to the creation of component-video output for DVD players. We needed higher-quality video, but the existing RCA jacks couldn't support the technology. Conversely, just as you wouldn't try to run power to your television through the cable jack, you shouldn't force an interface to accommodate something that it wasn't designed to handle. In the world of electronics, you know when you've done something wrong when smoke billows from your equipment. But with software, the consequences might not be so obvious. You might not realize the problems that you've created until much later, when it is more costly to fix.

Before getting too far into the nuts and bolts of interfaces, we need to come to terms on what an interface really is.

What Is an Interface?

Webster defines an interface as follows:

Interface: (n): A point at which independent systems interact.

However, an interface should be a multitude of things—some are obvious, while others are somewhat intangible.

Naturally, an interface should be interesting to users. This can be done with the content that your interface provides, the speed at which the interface performs, or simply because you offer a reliable service.

The other side of the coin is that a poorly designed interface can interfere with its potential for use. Many good things have fallen to the wayside just because users found it difficult to learn how to make them work.

The bottom line is, proper design of interfaces is a huge responsibility—the interface becomes your *storefront* and establishes how users view your Web Service.

Let's take a moment to slightly modify Webster's definition of an interface. You might suggest that an interface is a *logical* point at which independent systems interact. Why *logical*? It's logical because, as you've already seen, there could be more than one physical representation of an interface. There's no reason why an interface should change just because a developer chooses to use SMTP rather than HTTP.

Another property of your interfaces that you can control is the way that SOAP packages your messages. Deciding whether to use SOAP encoding could have an impact on the capabilities of your interface.

Using SOAP to Encode Information

SOAP defines an encoding style (see Section 5 of the SOAP v1.1 specification) for serializing an information graph as XML. This is an important feature of SOAP that is extremely useful for RPC serialization; it is denoted by using the `encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"` attribute within the Envelope contents. By using SOAP's encoding style, you get some of the following benefits:

- Serialization of objects by value and by reference
- Array serialization, including multidimensional arrays
- Partially transmitted and sparse array serialization

Although you can describe type information using XML schemas, in some cases, you might want to use an XML schema to validate messages entering or leaving your system. Section 5 allows for so many different variations of encoding information, so you will find that it's difficult to generate a schema that covers the exhaustive list of possibilities.

Rather than use the encoding mechanisms as defined by SOAP, you can define message structures that are completely describable in an XML schema.

NOTE

The UDDI framework (as described in Chapter 5) uses the message-based encoding style rather than Section 5 encoding as defined by SOAP.

Interface Versioning

Versioning interfaces requires a syntactical approach to encoding messages. In the case of SOAP, consider the following service example, which retrieves the temperature for a location based on its ZIP code:

```
POST /GetTemperature.asmx HTTP/1.1
Host: www.mcp.com
Accept: text/*
Content-type: text/xml; charset=utf-8
Content-length: nnnn
SOAPAction: "http://www.mcp.com/Temperature/GetTemperature"
{CR}{LF}
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
   <SOAP-ENV:Body>
      <GetTemperature xmlns="http://www.mcp.com/Temperature">
         <ZipCode>12345</ZipCode>
      </GetTemperature>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

In this case, the URI `http://www.mcp.com/Temperature` is used as the mechanism for identifying an interface from other interfaces. By using XML namespaces, the URI scopes the operation `GetTemperature` to that interface. In the case of HTTP, the `SOAPAction` field may also reflect this URI to verify the intent of the message. If `SOAPAction` is left empty, the HTTP message declares the intent of the request.

Now consider changing the temperature service to accept a city/state pair rather than a ZIP code. You have several options for exposing this new operation.

Creating a New Interface

The first approach that you can take is to create a new URI that represents a completely new interface, as follows:

```
POST /GetTemperature.asmx HTTP/1.1
Host: www.mcp.com
Accept: text/*
Content-type: text/xml; charset=utf-8
Content-length: nnnn
SOAPAction: "http://www.mcp.com/Temperature2/GetTemperature"
{CR}{LF}
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
   <SOAP-ENV:Body>
      <GetTemperature xmlns="http://www.mcp.com/Temperature2">
         <City>San Jose</City>
         <State>CA</State>
      </GetTemperature>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Here, the URI `http://www.mcp.com/Temperature2` has been applied to the XML namespace and `SOAPAction` field appropriately. This requires you to build a completely new WSDL description and XML schema to describe this new URI.

Adding an Operation to an Existing Interface

The next approach that you can take is to create a new operation name under an existing URI:

```
POST /GetTemperature.asmx HTTP/1.1
Host: www.mcp.com
Accept: text/*
Content-type: text/xml; charset=utf-8
Content-length: nnnn
SOAPAction: "http://www.mcp.com/Temperature/GetTemperatureByCity"
{CR}{LF}
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
   <SOAP-ENV:Body>
      <GetTemperatureByCity xmlns="http://www.mcp.com/Temperature">
         <City>San Jose</City>
         <State>CA</State>
      </GetTemperatureByCity>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Rather than change the URI `http://www.mcp.com/Temperature`, you only need to add a new operation for the given WSDL description and XML schema. Because you won't change any of the existing operations in your WSDL file, existing clients will be unaware of the new interface.

Modifying an Existing Operation

Finally, you have the option of modifying an existing operation to facilitate the new functionality.

```
POST /Router.pl HTTP/1.1
Host: www.mcp.com
Accept: text/*
Content-type: text/xml
Content-length: nnnn
SOAPAction: "http://www.mcp.com/Temperature/GetTemperature"
{CR}{LF}
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENV:Body>
      <m:GetTemperature xmlns:m="http://www.mcp.com/Temperature">
         <City>San Jose</City>
         <State>CA</State>
```

```
      </m:GetTemperature>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

This can be a more difficult approach than the previous options because your service code must be capable of interpreting the intention of incoming parameters. At times it can be difficult resolving ambiguity in requests.

Using the temperature example, what happens if your service receives a message that has a city, state, and ZIP code. Which parameters take precedence over others? Should you fail the request altogether? What happens when you start mixing mandatory and optional fields? These are all questions that you must answer when you start to change an existing operation.

All these questions lead to the topic of interface complexity.

Interface Complexity

Measuring complexity is a difficult task because it's one of those concepts that is purely based on your perspective. Unfortunately, this means that there's no silver bullet, no recipes that you can apply to guarantee a simple interface.

The following section provides an example to get you thinking about the issues at hand and the options that you have.

Additional Operations Versus Additional Parameters

It is often difficult to determine the best way to expose your system's functionality. Should you create a collection of small, succinct operations? Or possibly create a single, all-inclusive interface that offers a wide variety of parameters? In either case, you can probably fulfill the system requirements. So does it really matter which direction you choose?

Consider the example XML fragment showing a simple request to place an order:

```
<PlaceOrder>
   <partNumber>EVH5150</partNumber>
   <accountNumber>317</accountNumber>
   <quantity>8</quantity>
</PlaceOrder>
```

This is followed by a typical response containing an order number:

```
<PlaceOrderResponse>
   <orderNumber>670221</orderNumber>
</PlaceOrderResponse>
```

When a customer has placed an order, it's reasonable to assume that the customer will want to periodically check the status of the order, as shown in the following request:

```
<CheckStatus>
   <orderNumber>670221</orderNumber>
</CheckStatus>
```

The client can expect a standard status response message:

```
<CheckStatusResponse>
   <status>submitted</status>
</CheckStatusResponse>
```

However, one alternative to having separate operations (and requiring two round trips) is to combine the two operations into a single request:

```
<PlaceOrderAndCheckStatus>
   <partNumber>EVH5150</partNumber>
   <accountNumber>8675309</accountNumber>
   <quantity>7</quantity>
</PlaceOrderAndCheckStatus>
```

The semantics seem simple. The customer is interested in knowing whether the order was automatically shipped at the time it was placed. The response message seems innocent:

```
<PlaceOrderAndCheckStatusResponse>
   <orderNumber>670221</orderNumber>
   <status>submitted</status>
</PlaceOrderAndCheckStatusResponse>
```

Here, we have made a 100% improvement in performance by reducing the process to a single round trip. Therefore, this must be the correct way to design this interface. But wait—what happens when the user needs to check the order status again? Well, you could change the `PlaceOrderAndCheckStatus` operation so that it can accept optional parameters. This allows two types of messages to be valid:

```
<PlaceOrderAndCheckStatus>
   <partNumber>EVH5150</partNumber>
   <accountNumber>8675309</accountNumber>
   <quantity>7</quantity>
</PlaceOrderAndCheckStatus>
```

and

```
<PlaceOrderAndCheckStatus>
   <orderNumber>670221</orderNumber>
</PlaceOrderAndCheckStatus>
```

But because the interface semantics are becoming confusing, what happens if someone tries to place a new order and check the status of an existing order in the same request?

```
<PlaceOrderAndCheckStatus>
   <partNumber>EVH5150</partNumber>
   <accountNumber>8675309</accountNumber>
   <quantity>7</quantity>
   <orderNumber>670221</orderNumber>
</PlaceOrderAndCheckStatus>
```

Should you process the new order and then check status for the second order? Which status do you return? Possibly both? If so, how does the client know which status goes with a particular operation?

```
<PlaceOrderAndCheckStatusResponse>
   <orderNumber>670221</orderNumber>
   <status>submitted</status>
   <status>back order</status>
</PlaceOrderAndCheckStatusResponse>
```

Better yet, instead of allowing this ambiguity to creep into the interface, what if you just deny requests of this nature and inform the client of the poorly formed request? But should you really force clients to discover the interface semantics through trial and error? Generally, if you want a wide audience to use your service, your best bet is to keep the interface simple and spend less time worrying about performance. In cases where performance needs do exist, you should provide good documentation that explains exactly how your interface works and why.

The problem with making the blanket statement that you should be minimizing round trips is that you could end up requiring clients to receive information that they never intended to use. Under normal circumstances, a little bit of extra information is better than the overhead of requiring extra round trips. In some pathological situations in which the payload is extremely large, however, this might not be a fair trade.

Criteria for Managing Complexity

To summarize from the preceding example, here are some general questions that you should ask yourself before you make your final interface decisions:

- How difficult will it be for me to validate an operation's syntax?
- How complex will the server logic need to be for this request to be processed and a intelligible response to be sent?
- Will clients be able to quickly understand the semantics behind this operation?
- How much unrelated information is a client receiving that doesn't pertain to the request?
- How easy will it be for clients to upgrade to newer versions of my interface?
- How many round trips will the client need to make before getting the desired information?

Summary

We've covered a lot of ground in this chapter, touching on many aspects of Web Services to set the stage for the remainder of the book.

By now you should have a better understanding of why semantics are so important to Web Services and what properties contribute to a valuable service. You've also learned about many of the Internet protocols that play a big part in the Web Service paradigm.

You've been introduced to Visual Studio .NET and have seen a working demonstration of its Web Service tools for both clients and servers. It's pretty clear that .NET provides a flexible and robust environment for building and consuming services.

Finally, you explored Web Service interfaces and some of the issues surrounding their development, maintenance, and use.

In the next chapter, you will learn about the Microsoft .NET Framework and how it applies to Web Services. This will provide you with a better understanding of the facilities that .NET provides and will reinforce the topics that were discussed in this chapter.