CHAPTER 32

Web Service Standards and Extensions

In the previous chapter, you learned how .NET hides the low-level plumbing of web services, allowing you to create and consume sophisticated web services without needing to know anything about the low-level details of the protocols you're using. This higher-level abstraction is a general theme of modern programming—for example, few Windows developers worry about the individual pixels in their business applications, and ASP.NET developers rarely need to write raw markup to the output stream. Of course, sometimes high-level frameworks aren't quite enough. For example, Windows developers creating real-time games just might need to work with low-level video hardware, and web developers creating custom controls will probably need to immerse themselves in a thorny tangle of JavaScript and HTML.

The same principle is true of web services. In most cases, the high-level web services model in .NET is all you need. It ensures fast, productive, error-proof coding. Sometimes, however, you need to dig a little deeper. This is particularly true if you need to send complex objects to non-.NET clients or build extensions that plug into the .NET web services model. In this chapter, you'll take a look at this lower level, and learn more about the underlying SOAP and WSDL protocols.

Tip If you're a web service expert looking for new .NET 2.0 features, pay special attention to the "Customizing SOAP Messages" section toward the end of this chapter. It describes how to control the serialization process with IXmlSerializable and how to create schema importer extensions to allow types that web services wouldn't ordinarily support. Also, look for the "Implementing an Existing Contract" section that describes how to perform contract-first development with WSDL.

WS-Interoperability

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

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

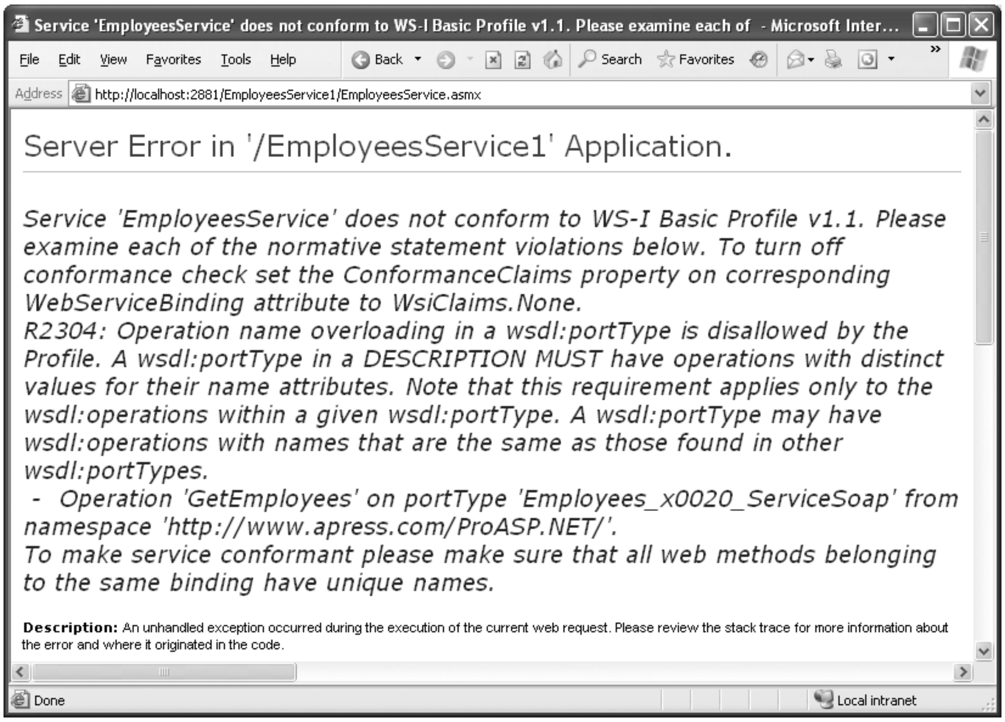
specifications, and it lays out a few ground rules. WS-Interoperability is strongly backed by all web service vendors (including Microsoft, IBM, Sun, and Oracle).

Ideally, as a developer you shouldn't need to worry about the specifics of WS-Interoperability. Instead, .NET should respect its rules implicitly. The way .NET 2.0 handles this is with a new WebServiceBinding attribute that is automatically added to your web service class when you create it in Visual Studio:

```
[WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
public class MyService : WebService
{ ... }
```

The WebServiceBinding attribute indicates the level of compatibility you're targeting. Currently, the only option is WsiProfiles.BasicProfile1_1, which represents the WS-Interoperability Basic Profile 1.1. However, as standards evolve there will be newer versions of SOAP and WSDL, as well as newer versions of the WS-Interoperability profile that go along with them.

Once you have the WebServiceBinding attribute in place, .NET will warn you with a compile error if your web service strays outside the bounds of allowed behavior. By default, all .NET web services are compliant, but you can inadvertently create a noncompliant service by adding certain attributes. For example, it's possible to create two web methods with the same name, as long as their signatures differ and you give them different message names using the MessageName property of the WebMethod attribute. However, this behavior isn't allowed according to the WS-Interoperability profile; it generates the error page shown in Figure 32-1 when you try to run the web service.


Service 'EmployeesService' does not conform to WS-I Basic Profile v1.1. Please examine each of - Microsoft Inter...
File Edit View Favorites Tools Help
Back
Search
Favorites
Address
http://localhost:2881/EmployeesService1/EmployeesService.asmx
Done
Local intranet

Server Error in '/EmployeesService1' Application.

Service 'EmployeesService' does not conform to WS-I Basic Profile v1.1. Please examine each of the normative statement violations below. To turn off conformance check set the ConformanceClaims property on corresponding WebServiceBinding attribute to WsiClaims.None.
R2304: Operation name overloading in a wsdl:portType is disallowed by the Profile. A wsdl:portType in a DESCRIPTION MUST have operations with distinct values for their name attributes. Note that this requirement applies only to the wsdl:operations within a given wsdl:portType. A wsdl:portType may have wsdl:operations with names that are the same as those found in other wsdl:portTypes.
- Operation 'GetEmployees' on portType 'Employees_x0020_ServiceSoap' from namespace 'http://www.apress.com/ProASP.NET/'.
To make service conformant please make sure that all web methods belonging to the same binding have unique names.

Description: An unhandled exception occurred during the execution of the current web request. Please review the stack trace for more information about the error and where it originated in the code.

Figure 32-1. *The SOAP message*

You can also choose to advertise your conformance with the EmitConformanceClaims property, as shown here:

```
[WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1,
 EmitConformanceClaims=true)]
public class MyService : WebService
{ ... }
```

In this case, additional information is inserted into the WSDL document to indicate that your web service is conformant. It's important to understand that this is for informational purposes only—your web service can be conformant without explicitly stating that it is.

In rare cases you might choose to violate one of the WS-Interoperability rules in order to create a web service that can be consumed by an older, noncompliant application. In this situation, your first step is to turn off compliance by removing the WebServiceBinding attribute. Alternatively, you can disable compliance checking and document that fact by using the WebServiceBinding attribute without a profile:

```
[WebServiceBinding(ConformsTo = WsiProfiles.None)]
public class MyService : WebService
{ ... }
```

SOAP

SOAP is a cross-platform standard used to format the messages sent between web services and client applications. The beauty of SOAP is its flexibility. Not only can you use SOAP to send any type of XML data (including your own proprietary XML documents), you can also use SOAP with transport protocols other than HTTP. For example, you could send SOAP messages over a direct TCP/IP connection.

■**Note** The .NET Framework includes support only for the most common use of SOAP, which is SOAP over HTTP. If you want more flexibility, you might want to consider the WSE (Web Services Enhancements) toolkit available from Microsoft, which is introduced in the next chapter.

SOAP is a fairly straightforward standard. The first principle is that every SOAP message is an XML document. This XML document has a single root element that is the SOAP envelope. The rest of the data for the message is stored inside the envelope, which includes a header section and a body.

■**Note** SOAP was originally considered an acronym (Simple Object Access Protocol). However, in the current versions of the SOAP standard, this is no longer the case. For the detailed SOAP specifications, surf to `http://www.w3.org/TR/soap`.

Essentially, .NET web services use two types of SOAP messages. The client sends a *request message* to a web service to trigger a web method. When processing is complete, the web service sends back a *response message*. Both of these messages have the same format, which is shown in Figure 32-2.

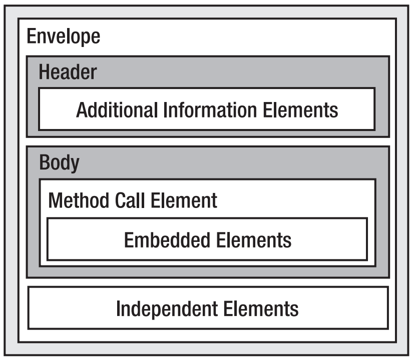


Figure 32-2. *The SOAP message*

SOAP Encoding

Two different (but closely related) styles of SOAP exist. *Document-style* SOAP views the data exchanged as documents. In other words, each SOAP message you send or receive contains an XML document in its body. *RPC-style* SOAP views the data exchange as method calls on remote objects. The remote object may be a Java object, a COM component, a .NET object, or something else entirely. In RPC-style SOAP, the outermost element in the request is always named after the method, and there is an element for each parameter on that method. In the response, the outermost element has the same name as the method with the text *Response* appended.

Seeing as .NET web services embrace the object-oriented RPC model, you might assume that .NET web services use RPC-style SOAP. However, this isn't the case. The simple reason is that document-style SOAP is more flexible—it gives you the ability to exchange arbitrary XML documents between the web service and the web service consumer. However, even though .NET uses document-style SOAP, it formats messages in a similar way to RPC-style SOAP, using many of the same conventions.

To make life even more interesting, data in a SOAP message can be encoded in two ways—literal and SOAP section 5. *Literal encoding* means that the data is encoded according to a specific XML schema. *SOAP section 5 encoding* means the data is encoded according to the similar, but more restricted, rules set out in section 5 of the SOAP specification. The section 5 rules are a bit of a throwback. The underlying reason they exist is because SOAP was developed before the XML Schema standard was finalized.

■ **Note** By default, all .NET web services use document-style SOAP with literal encoding. You should consider changing this behavior only if you need to be compatible with a legacy application.

At this point, you might wonder why you need to know any of these lower-level SOAP details. In most cases, you don't. However, sometimes you might want to change the overall encoding of your web service. One reason might be that you need to expose a web method that needs to be called by a client that supports only RPC-style SOAP. Although this scenario is becoming less and less common (and it violates WS-Interoperabiltiy), it can still occur.

ASP.NET has two attributes (both of which are found in the System.Web.Services.Protocols namespace) that you can use to control the overall encoding of all methods in a web service:

- **SoapDocumentService**: Use this to make every web service use document-style SOAP (which is already the default). However, you can use the SoapBindingUse parameter to specify SOAP section 5 encoding instead of document encoding.
- **SoapRpcService**: Use this to make every web service use RCP style SOAP with SOAP section 5 encoding.

You can also use the following two attributes on individual web methods:

- **SoapDocumentMethod**: Add this attribute to use document-style SOAP for a single web method. You can specify the encoding to use.
- **SoapRpcMethod**: Add this attribute to use RPC-style SOAP for a single web method.

This is useful if you want to expose two methods in a web service that perform a similar function but support a different type of SOAP encoding. However, in this chapter you'll focus on understanding the SOAP message style that .NET uses by default (document/literal).

SOAP Versions

The most common version of SOAP in use today is SOAP 1.1. The only other variant is the more recent SOAP 1.2, which clarifies many aspects of the SOAP standard, introduces some minor refinements, and formalizes the extensibility model.

■**Note** No matter which version of SOAP you use, the capabilities of a .NET web service are the same. In fact, the only reason you would pay particular attention to which version is being used is when you need to ensure compatibility with non-.NET clients. The examples in this chapter use SOAP 1.1.

.NET 1.*x* supported SOAP 1.1 only. However, a web service created in .NET 2.0 automatically supports both SOAP 1.1 and SOAP 1.2. If you want to change this, you can disable either one through the web.config file:

```
<configuration>
  <system.web>
    <webServices>
      <protocols>
        <!-- Use this to disable SOAP 1.2 -->
        <remove name="HttpSoap12"/>

        <!-- Use this to disable SOAP 1.1 -->
        <remove name="HttpSoap"/>
      </protocols>
   </webServices>
  ...
  </system.web>
</configuration>
```

When you create a proxy class with .NET, it uses SOAP 1.1 by default, unless only SOAP 1.2 is available. You can override this behavior programmatically by setting the SoapVersion property of the proxy class before you call any web methods:

```
proxy.SoapVersion = System.Web.Services.Protocols.SoapProtocolVersion.Soap12;
```

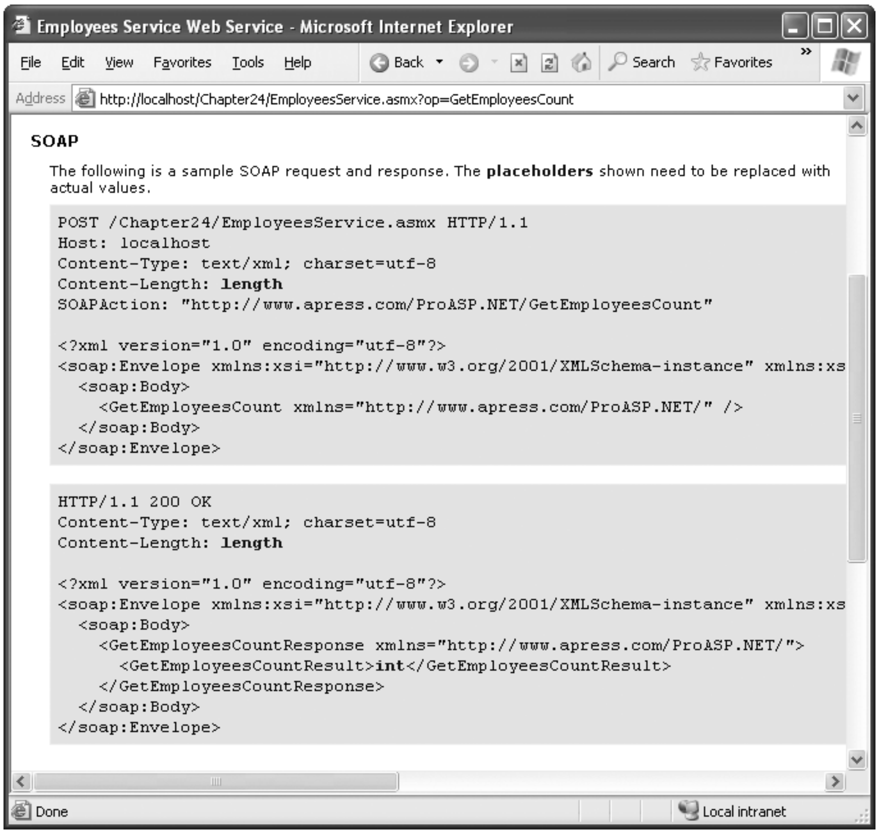
Alternatively, you can build a proxy class that always uses SOAP 1.2 by default by using the /protocol:SOAP12 command-line switch with wsdl.exe.

Tracing SOAP Messages

Before looking at the SOAP standard in detail, it's worth exploring how you can look at the SOAP messages sent to and from a .NET web service. Unfortunately, .NET doesn't include any tools for tracing or debugging SOAP messages. However, it's fairly easy to look at the underlying SOAP using other tools.

The first approach is to use the browser test page. As you know, the browser test page doesn't use SOAP—instead, it uses the scaled-down HTTP POST protocol that encodes data as name/value pairs. However, the test page does include an example of what a SOAP message should look like for a particular web method.

For example, consider the EmployeesService developed in the previous chapter. If you load the test page, click the link for the GetEmployeesCount() method, and scroll down the page, you'll see a sample request and response message with placeholders where the data values should go. Figure 32-3 shows part of this data example. You can also scroll farther down the page to see the format of the simpler HTTP POST messages.


Employees Service Web Service - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back
Search
Favorites
Address
http://localhost/Chapter24/EmployeesService.asmx?op=GetEmployeesCount
Done
Local intranet

SOAP

The following is a sample SOAP request and response. The **placeholders** shown need to be replaced with actual values.

```
POST /Chapter24/EmployeesService.asmx HTTP/1.1
Host: localhost
Content-Type: text/xml; charset=utf-8
Content-Length: length
SOAPAction: "http://www.apress.com/ProASP.NET/GetEmployeesCount"

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xs
  <soap:Body>
    <GetEmployeesCount xmlns="http://www.apress.com/ProASP.NET/" />
  </soap:Body>
</soap:Envelope>
```

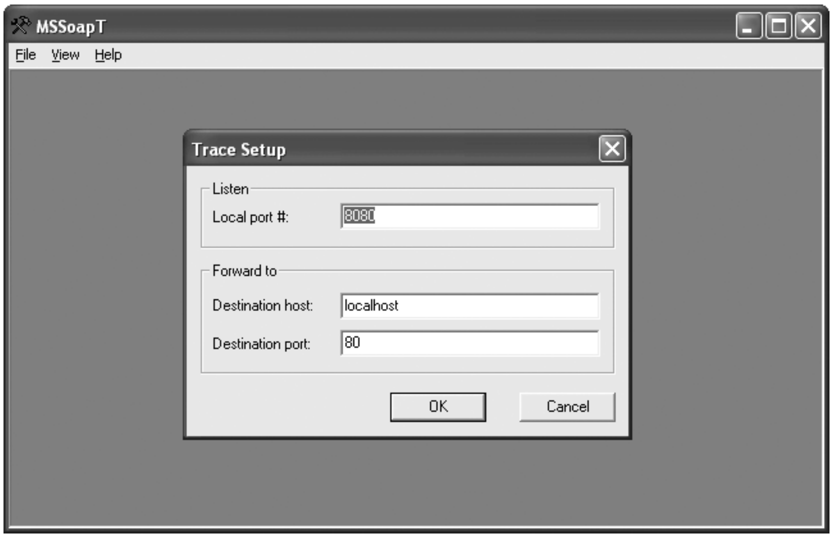
```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xs
  <soap:Body>
    <GetEmployeesCountResponse xmlns="http://www.apress.com/ProASP.NET/">
      <GetEmployeesCountResult>int</GetEmployeesCountResult>
    </GetEmployeesCountResponse>
  </soap:Body>
</soap:Envelope>
```

Figure 32-3. *Sample SOAP messages for GetEmployeesCount()*

These examples can help you understand the SOAP standard, but they aren't as useful if you want to see live SOAP messages, perhaps to troubleshoot an unexpected compatibility problem between a .NET web service and another client. Fortunately, there is an easy way to capture real SOAP messages as they flow across the wire, but you need to use another tool. This is the Microsoft

SOAP Toolkit, a COM library that includes objects that allow you to consume web services in COM-based languages such as Visual Basic 6 and Visual C++ 6. Along with these tools, the SOAP Toolkit also includes an indispensable tracing tool for peeking under the covers at SOAP communication.

To download the SOAP Toolkit, surf to `http://msdn.microsoft.com/webservices/_building/soaptk`. Once you've installed the SOAP Toolkit, you can run the trace utility by selecting Microsoft SOAP Toolkit ➤ Trace Utility from the Start menu. Once the trace utility loads, select File ➤ New ➤ Formatted Trace. You'll see the window shown in Figure 32-4.



Local port #: 8080
Destination host: localhost
Destination port: 80

Figure 32-4. *Starting a new SOAP trace*

The default settings indicate that the trace utility will listen for communication on port 8080 and forward all messages to port 80 (which is where the IIS web server is listening for unencrypted HTTP traffic, including GET and POST requests and SOAP messages). Click OK to accept these settings.

You need one additional detail. By default, your web service clients will bypass the tracing tool by sending their SOAP messages directly to port 80, not 8080. You need to tweak your client code so that it sends the SOAP messages to port 8080 instead. To do this, you simply need to use a URL that specifies the port, as shown here:

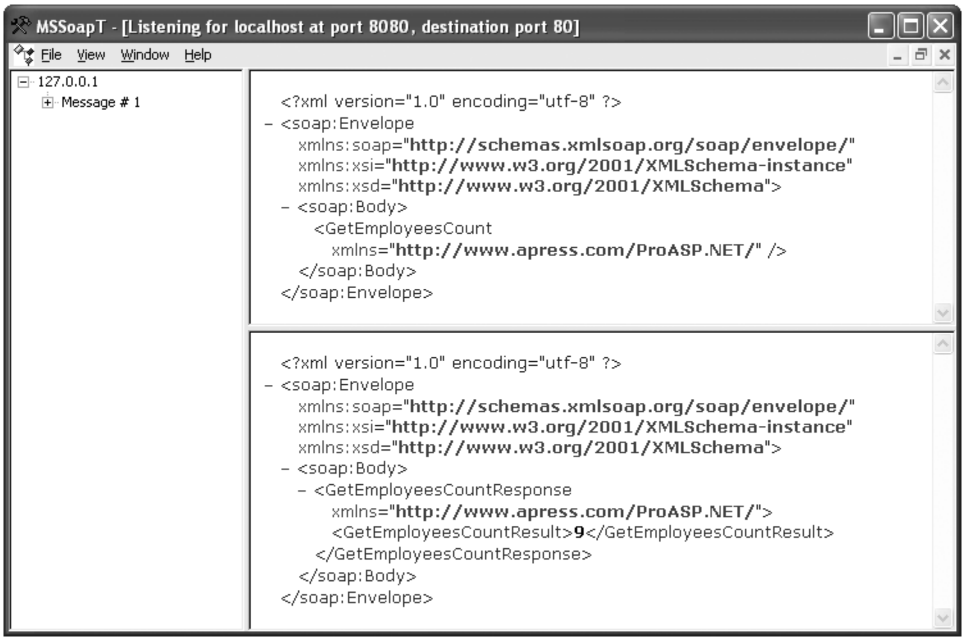
```
http://localhost:8080/MyWebSite/MyWebService.asmx
```

To change the URL, you need to modify the Url property of the proxy class before you invoke any of its methods. Rather than hard-coding a new URL, you can use the code shown here, which uses the System.Uri class to generically redirect any URL to port 8080:

```
// Create the proxy.
EmployeesService proxy = new EmployeesService();

Uri newUrl = new Uri(proxy.Url);
proxy.Url = newUrl.Scheme + "://" + newUrl.Host + ":8080" + newUrl.AbsolutePath;

// Call the web service and get the results.
DataSet ds = proxy.GetEmployeesCount();
```

You don't need to make a similar change to the web service, because it automatically sends its response message back to the port where the request message originated—in this case 8080. The trace utility will then log the response message and forward it back to the client application.

Once you've finished calling the web service, you can expand the tree in the trace utility to look at the request and response messages. Figure 32-5 shows the result of running the previous code snippet. In the top window is the request message for the GetEmployeesCount() method. In the bottom window is the response with the current number of employees in the table (nine). As you invoke more web methods, additional nodes will be added to the tree.


MSSoapT - [Listening for localhost at port 8080, destination port 80]
File View Window Help
127.0.0.1
Message # 1

```
  <?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>
      <GetEmployeesCount
        xmlns="http://www.apress.com/ProASP.NET/" />
    </soap:Body>
  </soap:Envelope>
```

```
  <?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>
    - <GetEmployeesCountResponse
        xmlns="http://www.apress.com/ProASP.NET/">
        <GetEmployeesCountResult>9</GetEmployeesCountResult>
      </GetEmployeesCountResponse>
    </soap:Body>
  </soap:Envelope>
```

Figure 32-5. *Capturing SOAP messages*

The SOAP trace tool is a powerful tool for looking at SOAP messages, particularly if you want to see how an unusual data type is serialized, test a custom extension, or troubleshoot an interoperability problem. Best of all, you don't need to install any special software on the web server. Instead, you simply need to forward the client's messages to the local trace utility.

In the following sections, you'll take a closer look at the SOAP format. You may want to use the SOAP trace utility to test these examples and take a look at the underlying SOAP messages for yourself.

The SOAP Envelope

Every SOAP message is enclosed in a root <Envelope> element. Inside the envelope, there is an optional <Header> element and a required <Body> element. Here's the basic skeleton:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/" >
  <soap:Header>
  </soap:Header>
  <soap:Body>
  </soap:Body>
</soap:Envelope>
```

Notice that the <Envelope>, <Body>, and <Header> elements all exist in the SOAP envelope namespace. This is a requirement.

The <Body> element contains the message payload. It's here that you place the actual data, such as the parameters in a request message or the return value in the response message. You can also specify fault information to indicate an error condition and *independent elements*, which define the serialization of complex types.

Request Messages

With automatically generated .NET web services, the first element in the <Body> element identifies the name of the method you are invoking. For example, here's a complete SOAP message for calling the GetEmployeesCount() method:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/" >
  <soap:Body>
    <GetEmployeesCount xmlns="http://www.apress.com/ProASP.NET/" />
  </soap:Body>
</soap:Envelope>
```

In this example, all you need is an empty <GetEmployeesCount> element. However, if the method requires any information (in the form of parameters), these are encoded in the <GetEmployeesCount> element with the appropriate parameter name. For example, the following SOAP represents a call to GetEmployeesByCity() that specifies a city name of London:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/" >
  <soap:Body>
     <GetEmployeesByCity xmlns="http://www.apress.com/ProASP.NET/">
      <city>London</city>
    </GetEmployeesByCity>
  </soap:Body>
</soap:Envelope>
```

You'll notice that in both these examples, the data inside the <Body> element is given the namespace of the web service (in this example, `http://www.apress.com/ProASP.NET`). The SOAP protocol is actually flexible enough to allow any XML markup in the <Body> element. This means that rather than treating SOAP as a protocol for remote method invocation, you can also use it as a way to exchange complex XML documents. In a business-to-business scenario, different parts of this document might be created by different companies in an automated workflow, and they might even include digital signatures. Unfortunately, the .NET programming model makes it hard to work in this way, because it wraps the SOAP and WSDL details with an object-oriented abstraction. However, leading web service developers are challenging this approach, and it's likely that future versions of ASP.NET will include increased flexibility.

■ **Note** At this point, you may be wondering if there's the possibility to create incompatible SOAP messages. For example, .NET web services use the name of a method for the first element in the <Body>, but other web service implementations may not follow this convention. The solution to these challenges is WSDL, a standard you'll consider in the "WSDL" section later in this chapter. It allows .NET to define the message format for your web services in great detail. In other words, it doesn't matter if the naming and organization of a .NET SOAP message is slightly different from that of a competitor's platform, because the non-.NET platform will read the rules in the WSDL document to determine how it should interact with the web service.

Response Messages

After sending a request message, the client waits to receive the response from the web server (as with any HTTP request). This response message uses a similar format to the request message. By .NET convention (again, not a requirement of the SOAP specification), the first child element in the <Body> element has the name of the method with the suffix *Response* appended. For example, after calling the GetEmployeesCount() method, you might receive a response like this:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/" >
  <soap:Body>
    <GetEmployeesCountResponse xmlns="http://www.apress.com/ProASP.NET/">
      <GetEmployeesCountResult>9</GetEmployeesCountResult>
    </GetEmployeesCountResponse>
  </soap:Body>
</soap:Envelope>
```

As with the request message, the response message is namespace-qualified using the namespace of the web service. Inside the <GetEmployeesCountResponse> element is an element with the return value named <GetEmployeesCountResult>. By .NET convention, this has the name of the web method, followed by the suffix *Result*. Interestingly, this isn't the only piece of information you can find in the <XxxResponse> element. If the method uses ref or out parameters, the data for those parameters will also be included. This allows the client to update its parameter values after the method call completes, which gives the same behavior as when you use out or ref parameters with a local method call.

Fault Messages

The SOAP standard also defines a way to represent error conditions. If an error occurs on the server, a message is sent with a <Fault> element as the first element inside the <Body> element. Fortunately, .NET follows this standard and applies it automatically. If an unhandled exception occurs while a web method is running, .NET sends a SOAP fault message back to the client. When the proxy class receives the fault message, it throws a client-side exception to notify the client application. However, as you'll see, this process of converting a web service exception to a client application exception isn't entirely seamless.

Consider, for example, what happens if you call the GetEmployeesCount() method when the database server isn't available. A SqlException is thrown on the web server side and caught by ASP.NET, which returns the following (somewhat abbreviated) fault message:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/" >
  <soap:Body>
    <soap:Fault>
      <faultcode>soap:Server</faultcode>
      <faultstring>System.Web.Services.Protocols.SoapException: Server was unable
to process request. ---> System.Data.SqlClient.SqlException: SQL Server does not
exist or access denied. at ... </faultstring>
      <detail />
    </soap:Fault>
  </soap:Body>
</soap:Envelope>
```

In general, the Fault element contains a <faultcode>, <faultstring>, and <detail> elements. The <faultcode> takes one of several predefined values, including ClientFaultCode (there was a problem with the client's SOAP request), MustUnderstandFaultCode (a required part of the SOAP message was not recognized), ServerFaultCode (an error occurred on the server), and VersionMismatchFaultCode (an invalid namespace was found). The <faultstring> element contains a full description of the problem. You can use the optional <detail> element to store additional information about the error that occurred (although it's empty in this example).

The problem is that the <Fault> element doesn't map directly to the .NET exception class. When the proxy receives this message, it can't identify the original exception object (and it has no way of knowing if that exception class is even available on the client). As a result, the proxy class simply throws a generic SoapException with the full <faultstring> details.

To understand how this works, consider what happens if you write the following code in your client:

```
EmployeesService proxy = new EmployeesService();

int count = -1;
try
{
    count = proxy.GetEmployeesCount();
}
catch (SqlException err)
{ ... }
```

In this case, the exception will never be caught, because it's a SoapException, not a SqlException (even though the root cause of the problem and the original exception object *is* a SqlException). Even if you catch the SqlException in the web method and manually throw a different exception object, it will still be converted into a SoapException on the client. That makes it difficult for the client to distinguish between different types of error conditions. The client can catch only a System.Net.WebException (which represents a timeout or a general network problem) or a System.Web.Services.Protocols.SoapException (which represents any .NET exception that occurred in the web service).

You have one other option. You can catch the exception in the web method on the server side and throw the supported SoapException yourself. The advantage of this approach is that before your web service throws the SoapException object, you can configure it by inserting additional XML in the <detail> element. The client can then read the content and use it to programmatically determine what really happened.

For example, here's a faulty version of the GetEmployeesCount() method that uses this approach to add the original exception type name to the SoapException using a custom <ExceptionType> element. You could extend this approach to add any combination of elements, attributes, and data.

```
[WebMethod()]
public int GetEmployeesCountError()
{
    SqlConnection con = null;
    try
    {
        con = new SqlConnection(connectionString);

        // Make a deliberately faulty SQL string
        string sql = "INVALID_SQL COUNT(*) FROM Employees";
        SqlCommand cmd = new SqlCommand(sql, con);

        con.Open();
        return (int)cmd.ExecuteScalar();
    }
    catch (Exception err)
    {
        // Create the detail information
        // an <ExceptionType> element with the type name.
```

```
            XmlDocument doc = new XmlDocument();
            XmlNode node = doc.CreateNode(XmlNodeType.Element,
             SoapException.DetailElementName.Name,
             SoapException.DetailElementName.Namespace);
            XmlNode child = doc.CreateNode(XmlNodeType.Element,
             "ExceptionType", SoapException.DetailElementName.Namespace);
            child.InnerText = err.GetType().ToString();
            node.AppendChild(child);

            // Create the custom SoapException.
            // Use the message from the original exception,
            // and add the detail information.
            SoapException soapErr = new SoapException(err.Message,
             SoapException.ServerFaultCode, Context.Request.Url.AbsoluteUri, node);

            // Throw the revised SoapException.
            throw soapErr;
        }
        finally
        {
            con.Close();
        }
}
```

The client application can read the <ExceptionType> element to get the additional information you've added. Here's an example that displays the exception name in a Windows message box (see Figure 32-6):

```
EmployeesService proxy = new EmployeesService();
try
{
    int count = proxy.GetEmployeesCountError();
}
catch (SoapException err)
{
    MessageBox.Show("Original error was: " + err.Detail.InnerText);
}
```

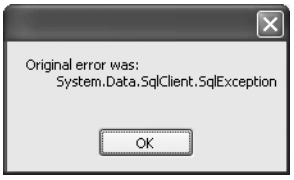

```
Original error was:
    System.Data.SqlClient.SqlException
```

Figure 32-6. *Retrieving additional SOAP fault information*

The SOAP Header

SOAP also defines a <Header> section where you can place out-of-band information. This is typically information that doesn't belong in the message payload. For example, a SOAP header might contain user authentication credentials or a session ID. These details might be required for processing the request, but they aren't directly related to the method you're calling. By separating these two portions, you achieve two improvements:

- **The method interface is simpler**: For example, you don't need to create a version of the GetEmployees() method that accepts a user name and password as parameters. Instead, that information is passed in the header, keeping the method less cluttered.
- **The service is more flexible**: For example, if you add an authentication service using a SOAP header, you have the freedom to change how that service works and what information it requires without changing the interface of your web methods. As in all types of programming, loosely coupled solutions are almost always preferable.

The <Header> element is optional, and it allows for an unlimited number of child elements to be placed within the header. To define new headers for use with a .NET web service, you create classes that derive from System.Web.Services.Protocols.SoapHeader.

For example, imagine you want to design a better way to support state in a web service. Instead of trying to use the session cookie (which requires an HTTP cookie and can't be defined in the WSDL document), you could pass the session ID as a header with every SOAP message. The following sections implement this design.

The Custom Header

The first step to implement this design is to create a custom class that derives from SoapHeader and includes the information you want to transmit as public properties.

Here's an example:

```
public class SessionHeader: SoapHeader
{
    public string SessionID;

    public SessionHeader(string sessionID)
    {
        SessionID = sessionID;
    }

    // A default constructor is required for automatic deserialization.
    public SessionHeader()
    {}
}
```

The SoapHeader class is really nothing more than a data container that can be serialized in and out of the <Header> element in a SOAP message. The custom SessionHeader adds a string SessionID variable with the session key.

Linking the Header to a Web Service

To use the SessionHeader in the web service, you need to create a public member variable in the web service for the header, as shown here:

```
public class SessionHeaderService : System.Web.Services.WebService
{
    public SessionHeader CurrentSessionHeader;
    ...
}
```

When you build a proxy class for this web service, it automatically includes a CurrentSession-Header property. You can read or set the session header using this property. The definition for the custom SessionHeader class is also added to the proxy class file.

Headers are linked to individual methods using the SoapHeader attribute. For example, if you want to use the SessionHeader service in a web method named DoSomething(), you would apply the WebMethod and SoapHeader attributes like this:

```
[WebMethod()]
[SoapHeader("CurrentSessionHeader")]
public void DoSomething()
{}
```

Note that the SoapHeader attribute takes the name of the public member variable where you want .NET to store the SOAP header. In the DoSomething() method, the SoapHeader attributes tells ASP.NET to create a new SessionHeader object using the header information that's been received from the client and store it in the public CurrentSessionHeader member variable of the web service. ASP.NET uses reflection to find this member variable at runtime. If it's not present, an error will occur. The SoapHeader attribute can also accept a named Direction property. Direction specifies whether the SOAP header will be sent from the client to the web service, from the web service to the web client, or both.

The following example shows how you can use the session header to create a simple system for storing state. First, a CreateSession() web method allows the client to initiate a new session. At this point, a new session ID is generated for a new SessionHeader object. Next, a new Hashtable collection is created in the Application collection, indexed under the session ID. Because the session ID uses a GUID, it's statistically guaranteed to be unique among all users.

```
[WebMethod()]
[SoapHeader("CurrentSessionHeader", Direction=SoapHeaderDirection.Out)]
public void CreateSession()
{
    // Create the header.
    CurrentSessionHeader = new SessionHeader(Guid.NewGuid().ToString());

    // From now on, all session data will be indexed under that key.
    Application[CurrentSessionHeader.SessionID] = new Hashtable();
}
```

This Hashtable will be used to store the additional session information. This isn't the best approach (for example, the Application collection isn't shared between computers in a web farm, doesn't persist if the web application restarts, and isn't scalable with large numbers of users). However, you could easily extend this approach to use a combination of a back-end database and caching. That solution would be much more scalable, and it would use the same system of session headers that you see in this example.

You'll also notice that the CreateSession() method uses the direction SoapHeaderDirection.Out, because it creates the header and sends it back to the client. Here's the interesting part: when the client receives the custom header, it's stored in the CurrentSessionHeader property of the proxy class. The best part is that from that point on, whenever the client application calls a method in the web service that requires the header, it's submitted with the request. In fact, as long as the client uses the same proxy class, the headers are automatically transmitted and the session management system is completely transparent.

To test this, you need to add two more methods to the web service. The first method, SetSessionData(), accepts a DataSet and stores it in the Application slot for the current user's session.

```
[WebMethod()]
[SoapHeader("CurrentSessionHeader", Direction=SoapHeaderDirection.In)]
public void SetSessionData(DataSet ds)
{
    Hashtable session = (Hashtable)Application[CurrentSessionHeader.SessionID];
    session.Add("DataSet", ds);
}
```

■**Note** You don't need to lock the Application collection in this example. That's because no two clients use the same session ID, so there's no possibility for two users to attempt to change that slot of the Application collection at the same time.

Next, you can use a GetSessionData() method to retrieve the DataSet for the current user's session and return it:

```
[WebMethod()]
[SoapHeader("CurrentSessionHeader", Direction=SoapHeaderDirection.In)]
public DataSet GetSessionData()
{
    Hashtable session = (Hashtable)Application[CurrentSessionHeader.SessionID];
    return (DataSet)session["DataSet"];
}
```

Of course, if you were creating a real-world implementation of this model, you wouldn't store the session information in application state, because it isn't robust or scalable enough. (For a rehash of the problems with application state, refer to Chapter 6.) Instead, you'd probably choose to store the information in a back-end database and cache it in the data cache (see Chapter 11) for quick retrieval.

Consuming a Web Service That Uses a Custom Header

When a web method requires a SOAP header, there's no way to test it using the simpler HTTP GET or HTTP POST protocols. As a result, you can't test the code in the browser test page. (In fact, the Invoke button won't even appear on this page.) Instead, you need to create a simple client.

The following code shows an example test. It creates the session (at which point it receives the SOAP header), stores a new, empty DataSet on the server, and then retrieves it.

```
SessionHeaderService proxy = new SessionHeaderService();
proxy.CreateSession();
proxy.SetSessionData(new DataSet("TestDataSet"));
DataSet ds = proxy.GetSessionData();
```

The SOAP message used for the call to CreateSession() is similar to the previous examples:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/" >
  <soap:Body>
    <CreateSession xmlns="http://www.apress.com/ProASP.NET/" />
  </soap:Body>
</soap:Envelope>
```

The response message includes the SOAP header:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/" >
  <soap:Header>
    <SessionHeader xmlns="http://tempuri.org/">
      <SessionID>bbc0bfed-c3c2-4552-b70e-dfa5564447fd</SessionID>
    </SessionHeader>
  </soap:Header>
  <soap:Body>
    <CreateSessionResponse xmlns="http://www.apress.com/ProASP.NET/" />
  </soap:Body>
</soap:Envelope>
```

Now, subsequent method invocations also have the SOAP header automatically included, as shown here:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/" >
  <soap:Header>
    <SessionHeader xmlns="http://tempuri.org/">
      <SessionID>bbc0bfed-c3c2-4552-b70e-dfa5564447fd</SessionID>
    </SessionHeader>
  </soap:Header>
  <soap:Body>
    <GetSessionData xmlns="http://www.apress.com/ProASP.NET/" />
  </soap:Body>
</soap:Envelope>
```

The result, in this example, is a web service that provides an alternate session state mechanism that uses SOAP headers instead of less reliable HTTP cookies. However, you can also use SOAP headers for many more web service extensions. In fact, in the next chapter you'll see how they allow you to leverage new and emerging web service standards with Microsoft's Web Services Enhancements component.

WSDL

WSDL (Web Service Description Language) is an XML-based language used to describe the public interface of a web service and the communication protocols it supports. A WSDL document is essentially a contract that tells the client what it needs to know in order to interact with a web service. Essentially, a WSDL document plays the same role as a type library for a COM component. (There's no direct analog to a type library in the .NET world, because all the descriptive type information you need is embedded into the compiled assembly as metadata.)

SOAP provides the ability to communicate with a web service. However, it doesn't tell you how to format your messages. Without WSDL, it would be up to you to document and explain the XML format your web services expect in the SOAP envelope. After locating a web service, the client developers would need to understand this information and handcraft the SOAP request and response messages accordingly. If this sort of human intervention was required in order to access every new service, the move toward web services would certainly be inhibited.

WSDL fills in the gaps by describing the supported protocols and expected message formats used by a web service. The power of WSDL is that it is not tied to any particular platform or object model. It is an XML language that provides an interface to web services across all platforms.

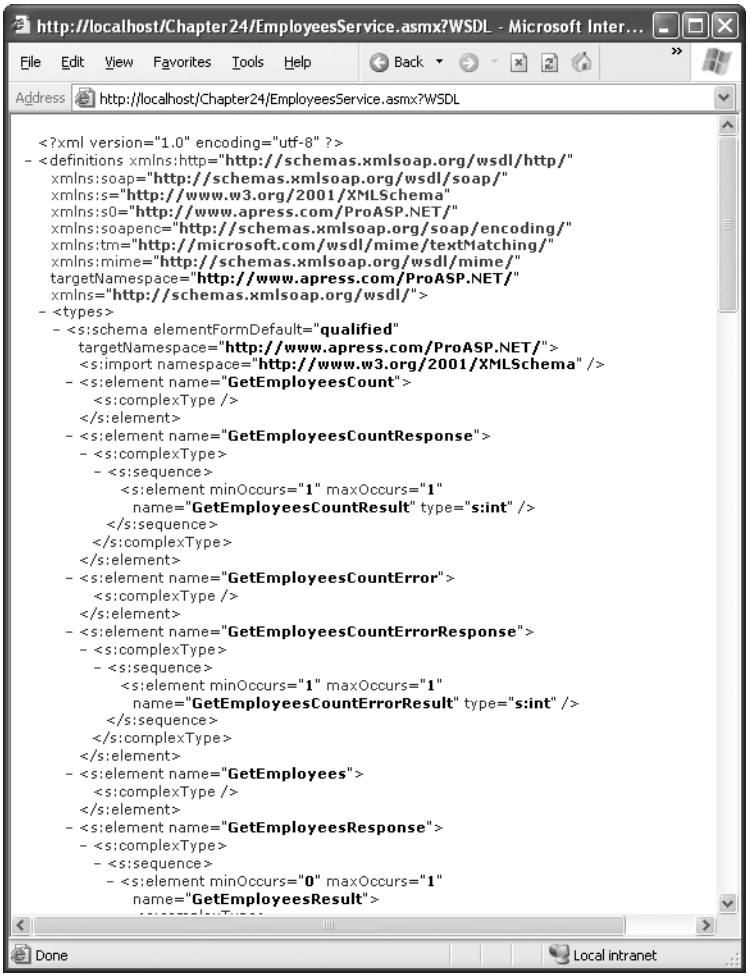
You can find the full WSDL standard at `http://www.w3.org/TR/wsdl`. The standard is fairly complex, but its underlying logic is hidden from the developer in ASP.NET programming, just as ASP.NET web controls abstract away the messy details of HTML tags and attributes.

■Tip Depending on the type of web services you create, you may not need to view the WSDL information—instead, you may be content to let .NET generate it for you automatically. However, if you need to support third-party clients or you plan to use contract-first development techniques (described in the section "Implementing an Existing Contract"), you'll need a solid understanding.

Viewing the WSDL for a Web Service

Once you've created a web service, you can easily get ASP.NET to generate the corresponding WSDL document. All you need to do is request the web service .asmx file, and add ?WSDL to the end of the

URL. (Another option is to click the Service Description link on the browser test page, which requests this URL.) Figure 32-7 shows part of the WSDL document you'll see for the Employees-Service developed in the previous chapter.


http://localhost/Chapter24/EmployeesService.asmx?WSDL - Microsoft Inter...
File Edit View Favorites Tools Help
Back
Address http://localhost/Chapter24/EmployeesService.asmx?WSDL
Done
Local intranet

```
  <?xml version="1.0" encoding="utf-8" ?>
- <definitions xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:s="http://www.w3.org/2001/XMLSchema"
    xmlns:s0="http://www.apress.com/ProASP.NET/"
    xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"
    xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
    targetNamespace="http://www.apress.com/ProASP.NET/"
    xmlns="http://schemas.xmlsoap.org/wsdl/">
  - <types>
    - <s:schema elementFormDefault="qualified"
        targetNamespace="http://www.apress.com/ProASP.NET/">
        <s:import namespace="http://www.w3.org/2001/XMLSchema" />
      - <s:element name="GetEmployeesCount">
          <s:complexType />
        </s:element>
      - <s:element name="GetEmployeesCountResponse">
        - <s:complexType>
          - <s:sequence>
              <s:element minOccurs="1" maxOccurs="1"
                name="GetEmployeesCountResult" type="s:int" />
            </s:sequence>
          </s:complexType>
        </s:element>
      - <s:element name="GetEmployeesCountError">
          <s:complexType />
        </s:element>
      - <s:element name="GetEmployeesCountErrorResponse">
        - <s:complexType>
          - <s:sequence>
              <s:element minOccurs="1" maxOccurs="1"
                name="GetEmployeesCountErrorResult" type="s:int" />
            </s:sequence>
          </s:complexType>
        </s:element>
      - <s:element name="GetEmployees">
          <s:complexType />
        </s:element>
      - <s:element name="GetEmployeesResponse">
        - <s:complexType>
          - <s:sequence>
            - <s:element minOccurs="0" maxOccurs="1"
                name="GetEmployeesResult">
```

Figure 32-7. *A WSDL document for the EmployeesService*

WSDL is keenly important because it allows the designers of programming frameworks such as .NET to create tools that can create proxy classes programmatically. When you add a web reference in Visual Studio (or use wsdl.exe), you point it to the WSDL document for the web service. (If it's a .NET web service, you can save a step by pointing it to the .asmx web service file, because both tools are smart enough to add the ?WSDL to the end of the query string to get a WSDL document for a .NET web service.) The tool then scans the WSDL document and creates a proxy class that uses the same methods, parameters, and data types. Other languages and programming platforms provide tools that work similar magic.

■**Note** The WSDL document contains information for communication between a web service and client. It doesn't contain any information that has anything to do with the code or implementation of your web service methods—that is unnecessary and would compromise security. Remember, when you add a web reference, all you need is the WSDL document. Neither Visual Studio nor the wsdl.exe tool has the ability to examine web service code directly.

WSDL documents tend to be extremely long and take more work to navigate than a simple SOAP message. In the next few sections, you'll examine a sample WSDL document for the EmployeesService.

The Basic Structure

WSDL documents consist of five main elements that combine to describe a web service. The first three of these are abstract and define the messaging. The last two are concrete and define the protocol and address information.

The three abstract elements, the <types>, <message>, and <portType> elements, combine to define the interface of the web service. They define the methods, parameters, and the properties of a web service. The two concrete elements, the <binding> element and the <port> element, combine to provide the protocol (SOAP over HTTP) and address information (the URI) of a web service. Separating the message definition elements from the location and protocol information provides the flexibility to reuse a common set of messages and data types within different protocols. It makes WSDL documents quite a bit more complicated than they would be otherwise, but it also gives unlimited flexibility for the future. For example, in the future WSDL documents could be used to describe how to interact with web services of SMTP, FTP, or some entirely different network-based protocol.

The WSDL 1.1 specification comes with SOAP over HTTP, HTTP GET, HTTP POST, and MIME extensions layered on top of the base specification. ASP.NET supports all of these except the MIME extension. Since WSDL is most commonly implemented utilizing SOAP over HTTP and since that is the default in ASP.NET, this chapter will focus on WSDL in relation to SOAP and HTTP. SOAP is really dominant, and there seems to be no real competitor to it at this point. Microsoft seems to be more interested in supporting raw SOAP over TCP, a transfer protocol known as Direct Internet Message Encapsulation (DIME), and building workarounds to the HTTP request-response structure, than they are in supporting MIME.

The <definitions> element is the root element of a WSDL document. It is where most of the namespaces are defined. Inside the <definitions> element are five main elements—one of which, <message>, commonly occurs multiple times.

Here's the basic structure of a WSDL document:

```
<?xml version="1.0" encoding="utf-8" ?>
<definitions>
  <types></types>
  <message></message>
  <message></message>
  <portType></portType>
  <binding></binding>
  <service></service>
</definitions>
```

Here is an overview of the main elements of a WSDL document:

- **Types**: This section is where all the web service data types are defined. This includes custom data types and the message formats.
- **Messages**: This section provides details about the request and response messages used to communicate with the web service.
- **PortType**: This section groups messages in input/output pairs. Each pair represents a method.
- **Binding**: This section provides information about the transport protocols supported by the web service.
- **Service**: This section provides the endpoint (URI address) for the web service.

The following sections examine these elements.

Types Section

The <types> section is where the web service data types are defined. The <types> element is actually an embedded XML schema, and all data types defined in the XML Schema standard are valid. You can add other type systems through extensibility if required.

In a .NET web service, every message is defined as a complex type. The complex type definition describes the method name, its parameters, the minimum and maximum times the element can occur, and the data types. For example, consider the GetEmployeesCount() method. The request message needs no data and is defined using XML schema syntax, like this:

```
<s:element name="GetEmployeesCount">
  <s:complexType />
</s:element>
```

The response message returns an integer (the int type from the XML Schema standard). It's defined like this:

```
<s:element name="GetEmployeesCountResponse">
  <s:complexType>
    <s:sequence>
      <s:element minOccurs="1" maxOccurs="1" name="GetEmployeesCountResult"
       type="s:int" />
    </s:sequence>
  </s:complexType>
</s:element>
```

The <types> section will typically be quite lengthy, because it defines two complex types for each web method.

A more interesting example is the web service that returns a custom object. For example, consider this version of the GetEmployees() method, which returns an array of EmployeeDetails objects:

```
[WebMethod()]
public EmployeeDetails[] GetEmployees()
{ ... }
```

If you look in the <types> section for this web service, you'll find that the request message is unchanged:

```
<s:element name="GetEmployees">
  <s:complexType />
</s:element>
```

However, the response refers to another complex type, named ArrayOfEmployeeDetails:

```
<s:element name="GetEmployeesResponse">
  <s:complexType>
    <s:sequence>
      <s:element minOccurs="0" maxOccurs="1" name="GetEmployeesResult"
       type="s0:ArrayOfEmployeeDetails" />
    </s:sequence>
  </s:complexType>
</s:element>
```

This ArrayOfEmployeeDetails is a complex type that's generated automatically. It represents a list of zero or more EmployeeDetails objects, as shown here:

```
<s:complexType name="ArrayOfEmployeeDetails">
  <s:sequence>
    <s:element minOccurs="0" maxOccurs="unbounded" name="EmployeeDetails"
     nillable="true" type="s0:EmployeeDetails" />
  </s:sequence>
</s:complexType>
```

The EmployeeDetails data class is also defined as a complex type in the <types> section. It's made up of an EmployeeID, FirstName, LastName, and TitleOfCourtesy, as shown here:

```
<s:complexType name="EmployeeDetails">
  <s:sequence>
    <s:element minOccurs="1" maxOccurs="1" name="EmployeeID" type="s:int" />
    <s:element minOccurs="0" maxOccurs="1" name="FirstName" type="s:string" />
    <s:element minOccurs="0" maxOccurs="1" name="LastName" type="s:string" />
    <s:element minOccurs="0" maxOccurs="1" name="TitleOfCourtesy"
     type="s:string" />
  </s:sequence>
</s:complexType>
```

You'll learn more about how complex types work in web services later in this chapter (in the "Customizing SOAP Messages" section).

Another ingredient that will turn up in the <types> section is the definition for any SOAP headers you use. For example, the stateful web service test developed earlier in this chapter defines the following type to represent the data in the session header:

```
<s:element name="SessionHeader" type="s0:SessionHeader" />
<s:complexType name="SessionHeader">
  <s:sequence>
    <s:element minOccurs="0" maxOccurs="1" name="SessionID" type="s:string" />
  </s:sequence>
</s:complexType>
```

Message Section

Messages represent the information exchanged between a web service method and a client. When you request a stock quote from the simple web service, ASP.NET sends a message, and the web service returns a different message. You can find the definition for these messages in the <message> section of the WSDL document. Here's an example:

```
<message name="GetEmployeesCountSoapIn">
  <part name="parameters" element="s0:GetEmployeesCount" />
</message>
<message name="GetEmployeesCountSoapOut">
  <part name="parameters" element="s0:GetEmployeesCountResponse" />
</message>
```

In this example, you'll notice that ASP.NET creates both a GetEmployeesCountSoapIn and a GetEmployeesCountSoapOut message. The naming is a matter of convention, but it underscores that a separate message is required for input (sending parameters and invoking a web service method) and output (retrieving a return value from a web service method).

The data used in these messages is defined in terms of the information in the <types> section. For example, the GetEmployeesCountSoapIn request message uses the GetEmployeesCount message, which is defined as an empty complex type in the <types> section.

PortType Section

The information in the <portType> section of the WSDL document provides a catalog of the functionality available in a web service. Unlike the <message> element just discussed, which contained independent elements for the input and output messages, these operations are tied together in a request-response grouping. The operation name is the name of the method. The <portType> is a collection of operations, as shown here:

```
<portType name="EmployeesServiceSoap">
  <operation name="GetEmployeesCount">
    <documentation>Returns the total number of employees.</documentation>
    <input message="s0:GetEmployeesCountSoapIn" />
    <output message="s0:GetEmployeesCountSoapOut" />
  </operation>
  <operation name="GetEmployees">
    <documentation>Returns the full list of employees.</documentation>
    <input message="s0:GetEmployeesSoapIn" />
    <output message="s0:GetEmployeesSoapOut" />
  </operation>
</portType>
```

Additionally, you'll see a <documentation> tag with the information added through the Description property of the WebMethod attribute.

■**Note** Four types of operations exist: one-way, request-response, solicit-response, and notification. The current WSDL specification defines bindings only for the one-way and request-response operation types. The other two can have bindings defined via binding extensions. The latter two are simply the inverse of the first two; the only difference is whether the endpoint in question is on the receiving or sending end of the initial message. HTTP is a two-way protocol, so the one-way operations will work only with MIME (which is not supported by ASP.NET) or with another custom extension.

Binding Section

The <binding> elements link the abstract data format to the concrete protocol used for transmission over an Internet connection. So far, the WSDL document has specified the data type used for various pieces of information, the required messages used for an operation, and the structure of each message. With the <binding> element, the WSDL document specifies the low-level communication protocol that you can use to communicate with a web service. It links this to an <operation> from the <portType> section.

Although we won't go into all the details of SOAP encoding, here's an example that defines how SOAP communication should work with the GetEmployeesCount() method of the EmployeesService:

```
<binding name="EmployeesServiceSoap" type="s0:EmployeesServiceSoap">
  <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
   style="document" />
  <operation name="GetEmployeesCount">
    <soap:operation
     soapAction="http://www.apress.com/ProASP.NET/GetEmployeesCount"
     style="document" />
     <input>
      <soap:body use="literal" />
    </input>
    <output>
      <soap:body use="literal" />
    </output>
  </operation>
```

If your method uses a SOAP header, that information is added as a <header> element. Here's an example with the CreateSession() method from the custom state web service developed earlier in this chapter. The CreateSession() method doesn't require the client to submit a header, but it does return one. As a result, only the output message references the header:

```
<operation name="CreateSession">
  <soap:operation soapAction="http://tempuri.org/CreateSession"
   style="document" />
  <input>
    <soap:body use="literal" />
  </input>
  <output>
    <soap:body use="literal" />
    <soap:header message="s0:CreateSessionSessionHeader"
     part="SessionHeader" use="literal" />
  </output>
</operation>
```

In addition, if your web service supports SOAP 1.2, you'll find a duplicate <binding> section with similar information:

```
<binding name="EmployeesServiceSoap12" type="s0:EmployeesServiceSoap">
  ...
</binding>
```

Remember, .NET web services support both SOAP 1.1 and SOAP 1.2 by default, but you can change this using configuration files, as shown earlier in this chapter.

Service Section

The <service> section defines the entry points into your web service, as one or more <port> elements. Each <port> provides address information or a URI. Here's an example from the WSDL for the EmployeesService:

```
<service name="EmployeesService">
  <documentation>Retrieve the Northwind Employees</documentation>
  <port name="EmployeesServiceSoap" binding="s0:EmployeesServiceSoap">
    <soap:address location="http://localhost/WebService1/EmployeesService.asmx" />
  </port>
</service>
```

The <service> section also includes a <documentation> element with the Description property of the WebService attribute, if it's set.

Implementing an Existing Contract

Since web services first appeared, a fair bit of controversy has existed about the right way to develop them. Some developers argue that the best approach is to use platforms such as .NET that abstract away the underlying details. They want to work with a higher-level framework of *remote procedure calls*. But XML gurus argue that you should look at the whole system in terms of *XML message passing*. They believe the first step in any web service application should be to develop a WSDL contract by hand.

As with many controversies, the ultimate solution probably lies somewhere in between. Application developers will probably never write WSDL contracts by hand—it's just too tedious and error-prone. On the other hand, developers who need to use web services in broader cross-platform scenarios will need to pay attention to the underlying XML representation of their messages and use techniques such as XML serialization attributes (described in the next section) to make sure they adhere to the right schema.

.NET 1.*x* was unashamedly oriented toward remote procedure calls. It significantly restricted the ability of developers to get underneath the web service facade and tinker with the low-level details. .NET 2.0 addresses this limitation with increased support for XML-centric approaches. One example is *contract-first* development.

In the web service scenarios you've seen so far, the web service code is created first. ASP.NET generates a matching WSDL document on demand. However, using .NET 2.0, you can approach the problem from the other end. That means you can take an existing WSDL document and feed it into the wsdl.exe command-line utility to create a basic web service skeleton. All you need is the new /serverInterface command-line switch

For example, you could use the following command line to create a class definition for the EmployeesService:

```
wsdl /serverInterface http://localhost/EmployeesService/EmployeesService.asmx?WSDL
```

You'll end up with an interface like this:

```
public interface IEmployeesServiceSoap
{
    [WebMethod()]
    DataSet GetEmployees();
}
```

You can then implement the interface in another class to add the web service code for the GetEmployees() method. By starting with the WSDL control first, you ensure that your web service implementation matches it exactly.

■ **Note** The interface isn't quite this simple. To make sure your interface *exactly* matches the WSDL, .NET adds a number of attributes that specifically set details such as namespaces, SOAP encoding, and XML element names. This clutters the interface, but the basic structure is as shown here.

You could also use this trick with a third-party web service. For example, you might want to create your own version of the stock-picking web service on XMethods. You want to ensure that clients can call your web method without needing to get a new WSDL document or be recompiled. To ensure this, you can generate and implement an exact interface match:

```
wsdl /serverInterface
 http://services.xmethods.net/soap/urn:xmethods-delayed-quotes.wsdl
```

Of course, for your web service to really be compatible, your code needs to follow certain assumptions that aren't set out in the WSDL document. These details might include how you parse strings, deal with invalid data, handle exceptions, and so on.

Contract-first development is unlikely to replace the simpler class-first model. However, it's a useful feature for developers who need to adhere to existing WSDL contracts, particularly in a cross-platform scenario.

Customizing SOAP Messages

In many cases, you don't need to worry about the SOAP serialization details. You'll be happy enough to create and consume web services using the infrastructure that .NET provides. However, in other cases you may need to extend your web services to use custom types or serialize your types to a specific XML format (for cross-platform compatibility). In the following sections, you'll see how you can control these details.

Serializing Complex Data Types

As you learned in the previous chapter, the SOAP specification supports all the data types defined by the XML Schema standard. These are considered *simple types*. Additionally, SOAP supports *complex types*, which are structures built out of an arrangement of simple types. You can use complex types for a web method return value or as a parameter. However, if a web method requires complex type parameters, you can interact with it only using SOAP. The simpler HTTP GET and HTTP POST mechanisms won't work, and the browser test page won't allow you to invoke the web method.

You've already used one example of a complex type: the DataSet. When you call the GetEmployees() method in the EmployeesService, .NET returns an XML document that describes the schema of the DataSet and its contents. Here's a partial listening of the SOAP response message:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/" >
  <soap:Body>
    <GetEmployeesResponse xmlns="http://www.apress.com/ProASP.NET/">
      <GetEmployeesResult>
        <xs:schema id="NewDataSet" xmlns=""
         xmlns:xs="http://www.w3.org/2001/XMLSchema"
         xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
        <!-- Schema omitted. -->
        </xs:schema>
        <diffgr:diffgram xmlns:msdata="urn:schemas-microsoft-com:xml-msdata"
         xmlns:diffgr="urn:schemas-microsoft-com:xml-diffgram-v1">
        <EmployeesDataSet xmlns="">
          <Employees diffgr:id="Employees1" msdata:rowOrder="0">
            <EmployeeID>1</EmployeeID>
            <LastName>Davolio</LastName>
            <FirstName>Nancy</FirstName>
            <Title>Sales Representative</Title>
            <TitleOfCourtesy>Ms.</TitleOfCourtesy>
            <HomePhone>(206) 555-9857</HomePhone>
          </Employees>
          <Employees diffgr:id="Employees2" msdata:rowOrder="1">
            <EmployeeID>2</EmployeeID>
            <LastName>Fuller</LastName>
            <FirstName>Andrew</FirstName>
            <Title>Vice President, Sales</Title>
            <TitleOfCourtesy>Dr.</TitleOfCourtesy>
            <HomePhone>(206) 555-9482</HomePhone>
          </Employees>
          ...
        </EmployeesDataSet></diffgr:diffgram>
      </GetEmployeesResult>
    </GetEmployeesResponse>
  </soap:Body>
</soap:Envelope>
```

You can also use your own custom classes with .NET web services. In this case, when you build the proxy, a copy of the custom class will automatically be added to the client (in the appropriate language of the client).

The process of converting objects to XML is known as *serialization*, and the process of reconstructing the objects from XML is know as *deserialization*. The component that performs the serialization is the System.Xml.Serialization.XmlSerializer class. You shouldn't confuse this class with the serialization classes you learned about in Chapter 13, such as the BinaryFormatter and SoapFormatter. These classes perform .NET-specific serialization that works with proprietary .NET objects, as long as they are marked with the Serializable attribute. Unlike the BinaryFormatter and SoapFormatter, the XmlSerializer works with any class, but it's much more limited than the BinaryFormatter and SoapFormatter and can extract only public data.

To use the XmlSerializer and send your custom objects to and from a web service, you need to be aware of a few restrictions:

- **Any code you include is ignored in the client**: This means the client's copy of the custom class won't include methods, constructor logic, or property procedure logic. Instead, these details will be stripped out automatically.
- **Your class must have a default zero-argument constructor**: This allows .NET to create a new instance of this object when it deserializes a SOAP message that contains the corresponding data.
- **Read-only properties are not serialized**: In other words, if a property has only a get accessor and not a set accessor, it cannot be serialized. Similarly, private properties and private member variables are ignored.

Clearly, the need to serialize a class to a piece of cross-platform XML imposes some strict limitations. If you use custom classes in a web service, it's best to think of them as simple data containers, rather than true participants in object-oriented design.

Creating a Custom Class

To see the XmlSerializer in action, you need to create a custom class and a web method that uses it. In the next example, we'll use the database component first developed in Chapter 8. This database component doesn't use the disconnected DataSet objects. Instead, it returns the results of a query using the custom EmployeeDetails class.

Here's what the EmployeeDetails class looks like currently, without any web service–related enhancements:

```
public class EmployeeDetails
{
    private int employeeID;
    public int EmployeeID
    {
        get {return employeeID;}
        set {employeeID = value;}
    }
    private string firstName;
    public string FirstName
    {
        get {return firstName;}
        set {firstName = value;}
    }

    private string lastName;
    public string LastName
    {
        get {return lastName;}
        set {lastName = value;}
    }

    private string titleOfCourtesy;
    public string TitleOfCourtesy
    {
        get {return titleOfCourtesy;}
        set {titleOfCourtesy = value;}
    }

    public EmployeeDetails(int employeeID, string firstName, string lastName,
      string titleOfCourtesy)
    {
        this.employeeID = employeeID;
        this.firstName = firstName;
        this.lastName = lastName;
        this.titleOfCourtesy = titleOfCourtesy;
    }
}
```

The EmployeeDetails class uses property procedures instead of public member variables. However, you can still use it because the XmlSerializer will perform the conversion automatically. The EmployeeDetails class doesn't have a default zero-parameter constructor, so before you can use it in a web method you need to add one, as shown here:

```
public EmployeeDetails(){}
```

Now the EmployeeDetails class is ready for a web service scenario. To try it, you can create a web method that returns an array of EmployeeDetail objects. The next example shows one such method—a GetEmployees() web method that calls the EmployeeDB.GetEmployees() method in the database component. (For the full code for this method, you can refer to Chapter 8 or consult the downloadable code.)

Here's the web method you need:

```
[WebMethod()]
public EmployeeDetails[] GetEmployees()
{
    EmployeeDB db = new EmployeeDB();
    return db.GetEmployees();
}
```

Generating the Proxy

When you generate the proxy (either using wsdl.exe or adding a web reference), you'll end up with two classes. The first class is the proxy class used to communicate with the web service. The second class is the definition for EmployeeDetails.

It's important to understand that the client's version of EmployeeDetails doesn't match the server-side version. In fact, the client doesn't even have the ability to see the full code of the server-side EmployeeDetails class. Instead, the client reads the WSDL document, which contains the XML schema for the EmployeeDetails class. The schema simply lists all the public properties and fields (without distinguishing between the two) and their data types.

When the client builds a proxy class, .NET uses this WSDL information to generate a client-side EmployeeDetails class. For every public property or field in the server-side definition of Employee-Details, .NET adds a matching public property to the client-side EmployeeDetails class.

Here's the code that's generated for the client-side EmployeeDetails class:

```
public partial class EmployeeDetails
{
    private int employeeIDField;
    private string firstNameField;
    private string lastNameField;
    private string titleOfCourtesyField;

    public int EmployeeID
    {
        get { return this.employeeIDField; }
        set { this.employeeIDField = value; }
    }

    public string FirstName
    {
        get { return this.firstNameField; }
        set { this.firstNameField = value; }
    }

    public string LastName
    {
        get { return this.lastNameField; }
        set { this.lastNameField = value; }
    }

    public string TitleOfCourtesy
    {
        get { return this.titleOfCourtesyField; }
        set { this.titleOfCourtesyField = value; }
    }
}
```

In this example, the client-side version is quite similar to the server-side version, because the server-side version didn't include much code. The only real difference (other than the renaming of private fields) is the missing nondefault constructor. As a general rule, the client-side version doesn't preserve nondefault constructors, any code in property procedures or constructors, any methods, or any private members.

■ **Note** The client-side version of a data class always uses property procedures, even if the original server-side version used public member variables. That gives you the ability to bind collections of client-side EmployeeDetails objects to a grid control. This is a change from .NET 1.*x*.

Testing the Custom Class Web Service

The next step is to write the code that calls the GetEmployees() method. Because the client now has a definition of the EmployeeDetails class, this step is easy:

```
EmployeesServiceCustomDataClass proxy = new EmployeesServiceCustomDataClass();
EmployeeDetails[] employees = proxy.GetEmployees();
```

The response message includes the employee data in the <GetEmployeesResult> element. By default, the XmlSerializer creates a structure of child elements based on the class name (EmployeeDetails) and the public property or variable names (EmployeeID, FirstName, LastName, TitleOfCourtesy, and so on). Interestingly, this default structure looks quite a bit like the XML used to model the DataSet, without the schema information.

Here's a somewhat abbreviated example of the response message:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/" >
  <soap:Body>
    <GetEmployeesResponse xmlns="http://www.apress.com/ProASP.NET/">
      <GetEmployeesResult>
        <EmployeeDetails>
          <EmployeeID>1</EmployeeID>
          <FirstName>Nancy</FirstName>
          <LastName>Davolio</LastName>
          <TitleOfCourtesy>Ms.</TitleOfCourtesy>
        </EmployeeDetails>
        <EmployeeDetails>
          <EmployeeID>2</EmployeeID>
          <FirstName>Andrew</FirstName>
          <LastName>Fuller</LastName>
          <TitleOfCourtesy>Dr.</TitleOfCourtesy>
        </EmployeeDetails>
      </GetEmployeesResult>
      ...
    </GetEmployeesResponse>
  </soap:Body>
</soap:Envelope>
```

When the client receives this message, the XML response is converted into an array of EmployeeDetails objects, using the client-side definition of the EmployeeDetails class.

Customizing XML Serialization with Attributes

Sometimes, you may want to customize the XML representation of a custom class. This approach is most useful in cross-platform programming scenarios when a client expects XML in a certain form. For example, you might have an existing schema that expects EmployeeDetails to use an EmployeeID attribute instead of a nested <EmployeeID> tag. .NET provides an easy way to apply these rules, using attributes. The basic idea is that you apply attributes to your data classes (such as EmployeeDetails). When the XmlSerializer creates a SOAP message, it reads these attributes and uses them to tailor the XML payload it generates.

The System.Xml.Serialization namespace contains a number of attributes that can be used to control the shape of the XML. Two sets of attributes exist: one where the attributes are named Xml*Xxx* and another where the attributes are named Soap*Xxx*. Which attributes you use depends on how the parameters are encoded.

As discussed earlier in this chapter, two types of SOAP serialization exist: literal and SOAP section 5 encoding. The XmlXxx attributes apply when you use literal style parameters. As a result, they apply in the following cases:

- When you use a web service with the default encoding. (In other words, you haven't changed the encoding by adding any attributes.)
- When you use the HTTP GET or HTTP POST protocols to communicate with a web service.
- When you use the SoapDocumentService or SoapDocumentMethod attribute with the Use property set to SoapBindingUse.Literal.
- When you use the XmlSerializer on its own (outside a web service).

The SoapXxx attributes apply when you use encoded-style parameters. That occurs in the following cases:

- When you use the SoapRpcService or SoapRpcMethod attributes
- When you use the SoapDocumentService or SoapDocumentMethod attribute with the Use property set to SoapBindingUse.Encoded

A class member may have both the SoapXxx and the XmlXxx attributes applied at the same time. Which one is used depends on the type of serialization being performed.

Table 32-1 lists most of the available attributes. Most of the attributes contain a number of properties. Some properties are common to most attributes, such as the Namespace property (used to indicate the namespace of the serialized XML) and the DataType property (used to indicate a specific XML Schema data type that might not be the one the XmlSerializer would choose by default). For a complete reference that describes all the attributes and their properties, refer to the MSDN Help.

Table 32-1. *Attributes to Control XML Serialization*

Xml Attribute	SOAP Attribute	Description
XmlAttribute	SoapAttribute	Used to make fields or properties into XML attributes instead of elements.
XmlElement	SoapElement	Used to name the XML elements.
XmlArray		Used to name arrays.
XmlIgnore	SoapIgnore	Used to prevent fields or properties from being serialized.
XmlInclude	SoapInclude	Used in inheritance scenarios. For example, you may have a property or field that's typed as some base class but may actually reference some derived class. In this case, you can use XmlInclude to specify all the derived class types that you may use.
XmlRoot		Used to name the top-level element.
XmlText		Used to serialize fields directly in the XML text without elements.
XmlEnum	SoapEnum	Used to give the members of an enumeration a name different from the name used in the enumeration.
XmlType	SoapType	Used to control the name of types in the WSDL file.

To see how SOAP serialization works, you can apply these attributes to the EmployeeDetails. For example, consider the following modified class declaration that uses several serialization attributes:

```
public class EmployeeDetails
{
    [XmlAttribute("id")]
    public int EmployeeID
    {
        get {return employeeID;}
        set {employeeID = value;}
    }

    [XmlElement("First")]
    public string FirstName
    {
        get {return firstName;}
        set {firstName = value;}
    }

    [XmlElement("Last")]
    public string LastName
    {
        get {return lastName;}
        set {lastName = value;}
    }

    [XmlIgnore()]
    public string TitleOfCourtesy
    {
        get {return titleOfCourtesy;}
        set {titleOfCourtesy = value;}
    }

    // (Constructors and private data omitted.)
}
```

Here's what a serialized EmployeeDetails will look like in the SOAP message:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/" >
  <soap:Body>
    <GetEmployeesResponse xmlns="http://www.apress.com/ProASP.NET/">
      <GetEmployeesResult>
        <EmployeeDetails id="1">
          <First>Nancy</First>
          <Last>Davolio</Last>
          </EmployeeDetails>
        <EmployeeDetails id="2">
          <First>Andrew</First>
          <Last>Fuller</Last>
        </EmployeeDetails>
      ...
      </GetEmployeesResult>
    </GetEmployeesResponse>
  </soap:Body>
</soap:Envelope>
```

■**Tip** If you want to experiment with different serialization attributes, you can also use the XmlSerializer class directly. Just create an instance of the XmlSerializer and pass the type of the object you want to serialize as a constructor parameter. You can then use the Serialize() method to convert the object to XML and write the data to a stream or a TextWriter object. You can use Deserialize() to read the XML data from a stream or TextReader and re-create the original object. You can also use a command-line tool called xsd.exe that's included with the .NET Framework to generate C# class definitions based on XML schema documents. The class declaration will automatically include the appropriate serialization attributes.

This example has only one limitation. Although you can control how the EmployeeDetails object is serialized, you can't use the same attributes to shape the element that wraps the list of employees. To take this step, you have two options. You could create a custom collection class and apply the XML serialization attributes to that class. Or, if you want to continue using an ordinary array, you must add an XML attribute that applies directly to the return value of the web method, like this:

```
[return: XmlArray("EmployeeList")]
public EmployeeDetails[] GetEmployees()
{ ... }
```

Now when you call the web method, you'll get this XML:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/" >
  <soap:Body>
    <GetEmployeesResponse xmlns="http://www.apress.com/ProASP.NET/">
      <EmployeeList>
        <EmployeeDetails id="1">
          <First>Nancy</First>
          <Last>Davolio</Last>
          </EmployeeDetails>
        <EmployeeDetails id="2">
          <First>Andrew</First>
          <Last>Fuller</Last>
        </EmployeeDetails>
      ...
      </EmployeeList>
    </GetEmployeesResponse>
  </soap:Body>
</soap:Envelope>
```

You can do a fair bit more to configure the details. For example, you can insert XML serialization attributes immediately before your parameters to change the required XML of the incoming request message. You can also use the SoapDocument attribute (discussed earlier) to change the name and namespace of the XML element that wraps the return value of your function (in this example, it's named <GetEmployeesReponse>).

Type Sharing

In .NET 1.*x*, you could run into headaches if more than one web service used the same custom class. For example, you might call the Store.GetOrder() method from one web service to get an Order object and then send that Order object to the Shipping.TrackOrder() method from another web service. The problem is that when you add a reference to both the Store and Shipping web services, you end up with two copies of the Order object data class, in two different namespaces. And even though these class definitions are equivalent, you still can't use them interchangeably. That means if you receive a version of an object from one web service, you can't pass it along to another web service.

.NET 2.0 handles this problem with a new *type sharing* feature. With type sharing, you can generate one client-side copy of a data class and use it with all compatible web services. To be considered compatible, the custom data class must meet these requirements:

- It must have the same XML representation. In other words, the XML schema of the type must be identical.
- It must have the same XML namespace. Different namespaces indicate different documents.

Many other details aren't important. For example, these factors have no influence on your ability to perform type sharing:

- The location of the web service
- The language of the web service
- The name of the class or properties (as long as you apply the XML serialization attributes to make sure the serialized form matches)

For example, the following server-side Employee class looks a fair bit different from the EmployeeDetails class shown in the previous section, but the serialized form is identical:

```
[XmlElement("EmployeeDetails")]
public class Employee
{
    [XmlAttribute("id")]
    public int ID;

    [XmlElement("First")]
    public string FirstName;

    [XmlElement("Last")]
    public string LastName;
}
```

This class matches the first requirement (identical XML schema), but it may not match the second (same namespace). You have two ways to make sure the class is in the same namespace. Your first option is to use the same namespace in the WebService attribute for both web services:

```
[WebService(Namespace="http://www.apress.com/ProASP.NET/")
public class EmployeesServiceCompatible : System.Web.Services.WebService
{ ... }
```

Usually, this isn't the approach you want. It implies that both web services are the same. A better choice is to use a unique namespace to identify shared XML data structures. You can do this by applying the XmlRoot or XmlElement attribute to the EmployeeDetails and Employee classes, as shown here:

```
[XmlElement("EmployeeDetails",
 Namespace="http://www.apress.com/ProASP.NET/EmployeeDetails")]
public class Employee
{ ... }
```

Now, the serialized XML in the response message looks like this:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/" >
  <soap:Body>
    <GetEmployeesResponse xmlns="http://www.apress.com/ProASP.NET/">
      <GetEmployeesResult>
        <EmployeeDetails id="1"
         xmlns="http://www.apress.com/ProASP.NET/EmployeeDetails">
          <First>Nancy</First>
          <Last>Davolio</Last>
          </EmployeeDetails>
        <EmployeeDetails id="2"
         xmlns="http://www.apress.com/ProASP.NET/EmployeeDetails">
          <First>Andrew</First>
          <Last>Fuller</Last>
        </EmployeeDetails>
      ...
    </GetEmployeesResponse>
  </soap:Body>
</soap:Envelope>
```

Assuming you've satisfied these two requirements, you're ready to generate the shared type. Currently, this is possible only by using the wsdl.exe command-line utility with the /sharetypes switch. You must also supply the address of all the web services that use the same type. Here's an example:

```
wsdl /sharetypes
 http://localhost/EmployeesService2/EmployeesServiceCompatible.asmx?WSDL
 http://localhost/EmployeesService2/EmployeesService.asmx?WSDL
```

If the types aren't identical for some reason, you won't be warned about the problem. Instead, your proxy class will contain multiple versions of the data class, such as EmployeeDetails, EmployeeDetails1, and so on.

Customizing XML Serialization with IXmlSerializable

The XML serialization attributes work well when you can take advantage of a one-to-one mapping between properties and XML elements or attributes. However, in some scenarios developers need more flexibility to create an XML representation of a type that fits a specific schema. For example, you might need to change the representation of your data types, control the order of elements, or add out-of-band information (such as comments or the date the document was serialized). In other cases, it may be technically possible to use the XML serialization attributes, but it may involve creating an unreasonably awkward class model.

Fortunately, .NET 2.0 provides an IXmlSerializable interface that you can implement to get complete control over your XML. The IXmlSerializable attribute has existed in .NET since version 1.0. However, it was used as a proprietary way to customize the serialization of the .NET DataSet, and it wasn't made available for general use. Now it's fully supported. IXmlSerializable mandates the three methods listed in Table 32-2.

Table 32-2. *IXmlSerializable Methods*

Method	Description
WriteXml()	In this method you write the XML representation of an instance of your object using an XmlWriter. You need this method in your web service in order for your web service to serialize an object and send it as a return value.
ReadXml()	In this method you read the XML from an XmlReader and generate the corresponding object. It's quite possible you won't need this method (in which case it's safe to throw a NotImplementedException. However, you will need it if you have to deserialize an object that your web service is accepting as an input parameter or if you decide to deploy this custom class to the client.
GetSchema()	This method is deprecated, and you should return null. If you want the ability to generate the XML schema for your class (which will be incorporated in the WSDL document), you must use the XmlSchemaProvider attribute instead. The XmlSchemaProvider names the method in your class that returns the XML schema document (XSD).

Chapter 12 discussed the XmlReader and XmlWriter classes in detail. Using them is quite straightforward. Here's an example of a custom class that handles its own XML generation:

```
public class EmployeeDetailsCustom : IXmlSerializable
{
    public int ID;
    public string FirstName;
    public string LastName;

    const string ns = "http://www.apress.com/ProASP.NET/CustomEmployeeDetails";

    void IXmlSerializable.WriteXml(XmlWriter w)
    {
        w.WriteStartElement("Employee", ns);

        w.WriteStartElement("Name", ns);
        w.WriteElementString("First", ns, FirstName);
        w.WriteElementString("Last", ns, LastName);
        w.WriteEndElement();

        w.WriteElementString("ID", ns, ID.ToString());
        w.WriteEndElement();
    }

    void IXmlSerializable.ReadXml(XmlReader r)
    {
        r.MoveToContent();
        r.ReadStartElement("Employee");

        r.ReadStartElement("Name");
        FirstName = r.ReadElementString("First", ns);
        LastName = r.ReadElementString("Last", ns);
        r.ReadEndElement();
        r.MoveToContent();
        ID = Int32.Parse(r.ReadElementString("ID", ns));
        reader.ReadEndElement();
    }
    System.Xml.Schema.XmlSchema IXmlSerializable.GetSchema()
    {
        return null;
    }

    // (Constructors omitted.)
}
```

■ **Tip** Make sure you read the full XML document, including the closing element tags in the ReadXml() method. Otherwise, .NET may throw an exception when you attempt to deserialize the XML.

Now, if you create a web method like this:

```
[WebMethod()]
public EmployeeDetailsCustom GetCustomEmployee()
{
    return new EmployeeDetailsCustom(101, "Joe", "Dabiak");
}
```

here's the XML you'll see:

```
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <GetCustomEmployeeResponse xmlns="http://www.apress.com/ProASP.NET/">
      <GetCustomEmployeeResult>
        <Employee xmlns="http://www.apress.com/ProASP.NET/CustomEmployeeDetails">
          <Name>
            <First>Joe</First>
            <Last>Tester</Last>
          </Name>
          <ID>1</ID>
        </Employee>
      </GetCustomEmployeeResult>
    </GetCustomEmployeeResponse>
  </soap:Body>
</soap:Envelope>
```

■ **Note** When using IXmlSerializable, the only serialization attributes that have any effect are the ones you apply to the method and the class declaration. Attributes on individual properties and fields have no effect. However, you could use .NET reflection to check for your own attributes and then use them to tailor the XML markup you generate.

Schemas for Custom Data Types

The only limitation in this example is that the client has no way to determine what XML to expect. If you look at the <types> section of the WSDL document for this example, you'll see that the schema is left wide open with the <any> element. This allows any valid XML content.

```
<s:element name="GetCustomEmployeeResponse">
  <s:complexType>
    <s:sequence>
      <s:element minOccurs="0" maxOccurs="1" name="GetCustomEmployeeResult">
        <s:complexType>
          <s:sequence>
            <s:element ref="s:schema" />
            <s:any />
          </s:sequence>
        </s:complexType>
      </s:element>
    </s:sequence>
  </s:complexType>
</s:element>
```

On the client side, you could deal with the data as an XML fragment, in which case you need to write the XML parsing code. However, a better idea is to supply an XML schema for your custom XML representation.

To do this, you need to add a static method to your class that returns the XML schema document as an XmlQualifiedName object, as shown here:

```
public static XmlQualifiedName GetSchemaDocument(XmlSchemaSet xs)
{
    // Get the path to the schema file.
    string schemaPath = HttpContext.Current.Server.MapPath("EmployeeDetails.xsd");

    // Retrieve the schema from the file.
    XmlSerializer schemaSerializer = new XmlSerializer(typeof(XmlSchema));
    XmlSchema s = (XmlSchema)schemaSerializer.Deserialize(
      new XmlTextReader(schemaPath), null);
    xs.XmlResolver = new XmlUrlResolver();
    xs.Add(s);

    return new XmlQualifiedName("EmployeeDetails", ns);
}
```

■ **Tip** This example retrieves the schema document from a file. For best performance, you would cache this document or construct it programmatically.

Now you need to point .NET to the right method using the XmlSchemaProvider attribute:

```
[XmlSchemaProvider("GetSchemaDocument")]
public class EmployeeDetailsCustom : IXmlSerializable
{ ... }
```

Now, ASP.NET will call this static method when it generates the WSDL document. It will then add the schema information to the WSDL document. However, remember that when you build a client .NET will generate the data class to match the schema, which means the client-side EmployeeDetails will differ quite a bit from the server-side version. (In this example, the client-side EmployeeDetails class will have a nested Name class because of the organization of XML elements, which probably isn't what you want.)

So, what can you do if you want the same version of EmployeeDetails on both the client and server ends? You could manually change the generated proxy code class, although this change will be discarded each time you rebuild the proxy. A more permanent option is to use schema importer extensions, which you'll tackle in the "Schema Importer Extensions" section.

Custom Serialization for Large Data Types

One reason you might use IXmlSerializable is to build web services that send large amounts of data. For example, imagine you want to send a large block of binary data that contains the content from a file. You could use a web service like this:

```
[WebMethod()]
public byte[] DownloadFile(string fileName)
{ ... }
```

The problem is that this approach assumes you'll read the entire data of the file into memory at once, as a byte array. If the file is several gigabytes in size, this will cripple the computer. A better solution is to use IXmlSerializable to implement chunking. That way, you can send an arbitrarily large amount of data over the wire by writing it one chunk at a time.

In the following sections, you'll see an example the uses IXmlSerializable to dramatically reduce the overhead of sending a large file.

The Server Side

The first step is to create the signature for your web method. For this strategy to work, the web method needs to return a class that implements IXmlSerializable. This example uses a class named FileData. Additionally, you need to turn off ASP.NET buffering to allow the response to be streamed across the network.

```
[WebMethod(BufferResponse = false)]
[SoapDocumentMethod(ParameterStyle = SoapParameterStyle.Bare)]
public FileData DownloadFile(string serverFileName)
{ ... }
```

The most laborious part is implementing the custom serialization in the FileData class. The basic idea is that when you create a FileData object on the server, you'll simply specify the corresponding filename. When the FileData object is serialized and IXmlSerializable.WriteXml() is called, the FileData object will create a FileStream and start sending binary data one block at a time.

Here's the bare skeleton of the FileData class:

```
[XmlRoot(Namespace="http://www.apress.com/ProASP.NET/FileData")]
[XmlSchemaProvider("GetSchemaDocument")]
public class FileData : IXmlSerializable
{
    // Namespace for serialization.
    const string ns = "http://www.apress.com/ProASP.NET/FileData";

    // The server-side path.
    private string serverFilePath;

    // When the FileData is created, make sure the file exists.
    // This won't defend against other problems reading the file (like
    // insufficient rights, the file is currently locked by another process,
    // and so on).
    public FileData(string serverFilePath)
    {
        if (!File.Exists(serverFilePath))
        {
            throw new FileNotFoundException("Source file not found.");
        }
        this.serverFilePath = serverFilePath;
    }
    void IXmlSerializable.WriteXml(System.Xml.XmlWriter writer)
    { ... }

    System.Xml.Schema.XmlSchema IXmlSerializable.GetSchema()
    {
        return null;
    }

    void IXmlSerializable.ReadXml(System.Xml.XmlReader reader)
    {
        throw new NotImplementedException();
    }

    public static XmlQualifiedName GetSchemaDocument(XmlSchemaSet xs)
    {
        // Get the path to the schema file.
        string schemaPath = HttpContext.Current.Server.MapPath("FileData.xsd");

        // Retrieve the schema from the file.
        XmlSerializer schemaSerializer = new XmlSerializer(typeof(XmlSchema));
        XmlSchema s = (XmlSchema)schemaSerializer.Deserialize(
          new XmlTextReader(schemaPath), null);
        xs.XmlResolver = new XmlUrlResolver();
        xs.Add(s);
        return new XmlQualifiedName("FileData", ns);
    }
}
```

You'll notice that this class supports writing the file data to XML but not reading it. That's because you're looking at the server's version of the code. It sends FileData objects, but doesn't receive them.

In this example, you want to create an XML representation that splits data into separate Base64-encoded chunks. It will look like this:

```
<FileData xmlns="http://www.apress.com/ProASP.NET/FileData">
  <fileName>sampleFile.xls</fileName>
  <size>66048</size>
  <content>
    <chunk>...</chunk>
    <chunk>...</chunk>
    ...
  </content>
</FileData>
```

Here's the WriteXml() implementation that does the job:

```
void IXmlSerializable.WriteXml(System.Xml.XmlWriter writer)
{
    // Open the file (taking care to allow it to be opened by other threads
    // at the same time).
    FileStream fs = new FileStream(serverFilePath, FileMode.Open,
      FileAccess.Read, FileShare.Read);

    // Write filename.
    writer.WriteElementString("fileName", ns, Path.GetFileName(serverFilePath));

    // Write file size (useful for determining progress.)
    long length = fs.Length;
    writer.WriteElementString("size", ns, length.ToString());
    // Start the file content.
    writer.WriteStartElement("content", ns);

    // Read a 4 KB buffer and write that (in slightly larger Base64-encoded chunks).
    int bufferSize = 4096;
    byte[] fileBytes = new byte[bufferSize];
    int readBytes = bufferSize;
    while (readBytes > 0)
    {
        readBytes = fs.Read(fileBytes, 0, bufferSize);
        writer.WriteStartElement("chunk", ns);

        // This method explicitly encodes the data. If you use another method,
        // it's possible to add invalid characters to the XML stream.
        writer.WriteBase64(fileBytes, 0, readBytes);
        writer.WriteEndElement();
        writer.Flush();
    }
    fs.Close();

    // End the XML.
    writer.WriteEndElement();
}
```

Now you can complete the web service. The DownloadFile() method shown here looks for a user-specified file in a hard-coded directory. It creates a new FileData object with the full path name and returns it. At this point, the FileData serialization code springs into action to read the file and begin writing it to the response stream.

```
public class FileService : System.Web.Services.WebService
{
    // Only allow downloads in this directory.
    string folder = @"c:\Downloads";

    [WebMethod(BufferResponse = false)]
    [SoapDocumentMethod(ParameterStyle = SoapParameterStyle.Bare)]
    public FileData DownloadFile(string serverFileName)
    {
        // Make sure the user only specified a filename (not a full path).
        serverFileName = Path.GetFileName(serverFileName);

        // Get the full path using the download directory.
        string serverFilePath = Path.Combine(folder, serverFileName);

        // Return the file data.
        return new FileData(serverFilePath);
    }
}
```

You can try this method using the browser test page and verify that the data is split into chunks by looking at the XML.

The Client Side

On the client side, you need a way to retrieve the data one chunk at a time and write it to a file. To provide this functionality, you need to change the proxy class so that it returns a custom IXmlSerializable type. You'll place the deserialization code in this class.

Tip You can implement both the serialization code and the deserialization code in the same class and distribute that class as a component to the client and the server. However, it's usually better to observe a strict separation between both ends of a web service application. This makes it easier to update the client with new versions.

When you create the proxy class, .NET will try to create a suitable copy of the FileData class. However, it won't succeed. Without the schema information, it will simply try to convert the returned value to a DataSet. Even if you add the schema information, all .NET can do is create a class representation that exposes all the details (the name, size, and content) through separate properties. This class won't have the chunking behavior—instead, it will attempt to load everything into memory at once.

To fix this problem, you need to customize the proxy class by hand. If you're creating a web client, you need to first generate the proxy class with wsdl.exe so you have the code available. Here's the change you need to make:

```
public FileDataClient DownloadFile(string serverFileName)
{
    object[] results = this.Invoke("DownloadFile", new object[] {
      serverFileName});
    return ((FileDataClient)(results[0]));
}
```

Obviously, modifying the proxy class is a brittle solution, because every time you refresh the proxy class your change will be wiped out. A better choice is to implement a schema importer extension, as described in the next section.

Here's the basic outline of the FileDataClient class:

```
[XmlRoot(Namespace="http://www.apress.com/ProASP.NET/FileData")]
public class FileDataClient : IXmlSerializable
{
    private string ns = "http://www.apress.com/ProASP.NET/FileData";

    // The location to place the downloaded file.
    private static string clientFolder;
    public static string ClientFolder
    {
        get { return clientFolder; }
        set { clientFolder = value; }
    }

    void IXmlSerializable.ReadXml(System.Xml.XmlReader reader)
    { ... }

    System.Xml.Schema.XmlSchema IXmlSerializable.GetSchema()
    {
        return null;
    }

    void IXmlSerializable.WriteXml(System.Xml.XmlWriter writer)
    {
        throw new NotImplementedException();
    }
}
```

One important detail is the static property ClientFolder, which keeps track of the location where you want to save all downloaded files. You must set this property before the download begins, because the ReadXml() method uses that information to determine where to create the file. The ClientFolder property must be a static property, because the client doesn't get the chance to create and configure the FileDataClient object it wants to use. Instead, .NET creates a FileDataClient instance automatically and uses it to deserialize the data. By using a static property, the client can set this piece of information before starting the download, as shown here:

```
FileDataClient.ClientFolder = @"c:\MyFiles";
```

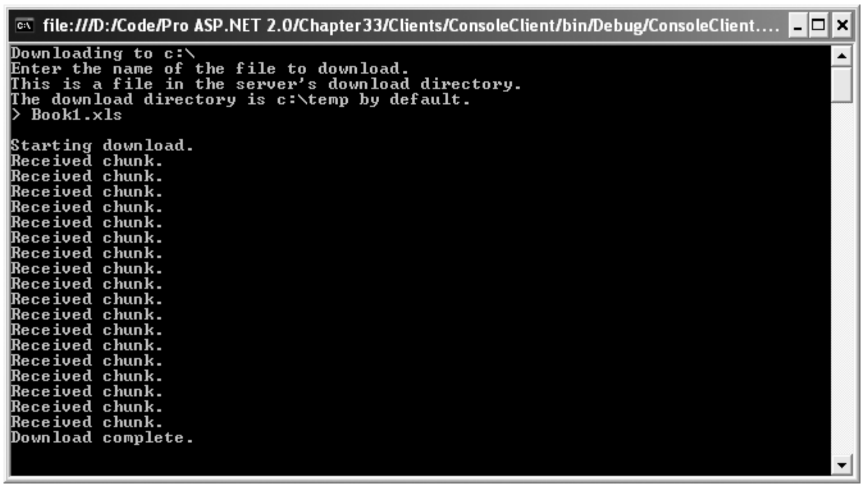
The deserialization code performs the reverse task of the serialization code—it steps through the chunks and writes them to the new file. Here's the complete code:

```
void IXmlSerializable.ReadXml(System.Xml.XmlReader reader)
{
    if (FileDataClient.ClientFolder == "")
    {
        throw new InvalidOperationException("No target folder specified.");
    }
    reader.ReadStartElement();

    // Get the original filename.
    string fileName = reader.ReadElementString("fileName", ns);

    // Get the size (not currently used).
    double size = Convert.ToDouble(reader.ReadElementString("size", ns));

    // Create the file.
    FileStream fs = new FileStream(Path.Combine(ClientFolder, fileName),
      FileMode.Create, FileAccess.Write);

    // Read the XML and write the file one block at a time.
    byte[] fileBytes;
    reader.ReadStartElement("content", ns);
    double totalRead = 0;

    while (true)
    {
        if (reader.IsStartElement("chunk", ns))
        {
            string bytesBase64 = reader.ReadElementString();
            totalRead += bytesBase64.Length;
            fileBytes = Convert.FromBase64String(bytesBase64);
            fs.Write(fileBytes, 0, fileBytes.Length);
            fs.Flush();

            // You could report progress by raising an event here.
            Console.WriteLine("Received chunk.");
        }
        else
        {
            break;
        }
    }
    fs.Close();
    reader.ReadEndElement();
    reader.ReadEndElement();
}
```

Here's a complete console application that uses the FileService:

```
static void Main()
{
    Console.WriteLine("Downloading to c:\\");
    FileDataClient.ClientFolder = @"c:\";

    Console.WriteLine("Enter the name of the file to download.");
    Console.WriteLine("This is a file in the server's download directory.");
    Console.WriteLine("The download directory is c:\\temp by default.");
    Console.Write("> ");
    string file = Console.ReadLine();

    FileService proxy = new FileService();
    Console.WriteLine();
    Console.WriteLine("Starting download.");
    proxy.DownloadFile(file);
    Console.WriteLine("Download complete.");
}
```

Figure 32-8 shows the result.


file:///D:/Code/Pro ASP.NET 2.0/Chapter33/Clients/ConsoleClient/bin/Debug/ConsoleClient....

```
Downloading to c:\
Enter the name of the file to download.
This is a file in the server's download directory.
The download directory is c:\temp by default.
> Book1.xls

Starting download.
Received chunk.
Received chunk.
Received chunk.
Received chunk.
Received chunk.
Received chunk.
Received chunk.
Received chunk.
Received chunk.
Received chunk.
Received chunk.
Received chunk.
Received chunk.
Received chunk.
Received chunk.
Received chunk.
Received chunk.
Received chunk.
Download complete.
```

Figure 32-8. *Downloading a large file with chunking*

You can find this example online, with a few minor changes (for example, the client and server methods for working with the file are combined into one FileData class).

Schema Importer Extensions

One of the key principles of service-oriented design is that the client and the server share contracts, not classes. That level of abstraction allows clients on widely different platforms to interact with the same web service. They send the same serialized XML, but they have the freedom to use different programmatic structures (such as classes) to prepare their messages.

In some cases, you might want to bend these rules to allow your clients to work with rich data types. For example, you might want to take a custom data class, distribute it to both the client and server, and allow them to send or receive instances of this class with a web service. .NET 2.0 makes this possible with a new feature called *schema importer extensions*.

■Tip Think twice about using custom data types. The danger of this approach is that it can easily lead you to develop a proprietary web service. Although your web service will still use XML (which can always be read on any platform), once you start tailoring your XML to fit platform-specific types, it might be prohibitively difficult for other clients to parse that XML or do anything practical with it. For example, most non-.NET clients don't have an easy way to consume the XML generated for the DataSet.

Before developing a schema importer extension, make sure your service is using the XmlSchemaProvider attribute to designate a method that returns schema information. Without the schema information, the proxy generation tool won't have the information it needs to identify your custom data types, so any schema importers you create will be useless.

For the FileData class, the schema is drawn from this schema file:

```
<xs:schema id="FileData" targetNamespace=http://www.apress.com/ProASP.NET/FileData
 elementFormDefault="qualified" xmlns:xs="http://www.w3.org/2001/XMLSchema">

    <xs:complexType name="FileData" >
      <xs:sequence>
        <xs:element name="fileName" type="xs:string" />
        <xs:element name="size" type="xs:int" />
        <xs:element name="content" >
          <xs:complexType >
            <xs:sequence>
              <xs:element name="chunk" type="xs:base64Binary"
               maxOccurs="unbounded"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>

</xs:schema>
```

Now you're ready to develop the schema importer that allows the client to recognize this data type.

Using a schema importer involves two steps: creating the extension and then registering it. To create the extension, you need to create a new class library component (DLL assembly). In this assembly, add a class that derives from SchemaImporterExtension.

When the proxy generator comes across a complex type (as it generates a proxy class), it calls the ImportSchemaType() method of every schema importer extension defined in the machine.config file. Each schema importer can check the namespace and schema of the type and then decide to handle it by mapping the XML type to a known .NET type.

Here's an example with a FileDataSchemaImporter that configures the proxy to use the FileDataClient class:

```
public class FileDataSchemaImporter : SchemaImporterExtension
{
    public override string ImportSchemaType(string name, string ns,
      XmlSchemaObject context, XmlSchemas schemas, XmlSchemaImporter importer,
      CodeCompileUnit compileUnit, CodeNamespace mainNamespace,
      CodeGenerationOptions options, CodeDomProvider codeProvider)
    {
        if (name.Equals("FileData") &&
         ns.Equals("http://www.apress.com/ProASP.NET/FileData"))
        {
            mainNamespace.Imports.Add(new CodeNamespaceImport("FileDataComponent"));
            return "FileDataClient";
        }
        else
        {
            // Chose not to handle the type.
            return null;
        }
    }
}
```

This is an extremely simple schema importer. It does two things:

- It instructs the proxy class to use the class named FileDataClient for this type. That means the proxy class will use your existing class and refrain from generating a client-side FileData class automatically (the standard behavior).
- It instructs the proxy class generator to add a namespace import for the FileDataComponent namespace. It's still up to you to make sure the assembly with the FileDataComponent.File-Data class is available in your project.

Once you've created the schema importer, you need to install it in the global assembly cache. Give it a strong name (use the Signing tab in the project properties) and then drag and drop it into the c:\[WinDir]\Assembly directory, or use the gacutil.exe command-line utility.

Once your schema importer is safely installed in the cache, use Windows Explorer to find out its public key token. Armed with that information, you can register your schema importer in the machine.config file using settings like these:

```
<configuration>
...
  <system.xml.serialization>
    <schemaImporterExtensions>
      <add name="FileDataSchemaImporter" type=
"SchemaImporter.FileDataSchemaImporter, SchemaImporter, Version=1.0.0.0,
Culture=neutral, PublicKeyToken=6c8e0bfd71c11c40" />
    </schemaImporterExtensions>
  </system.xml.serialization>
</configuration>
```

The type attribute is the important part. Make sure you use this format on a single line:

```
<Namespace-qualified class name>, <Assembly name without the extension>,
<Version>, <Culture>, <Public key token>
```

Now you're ready to use your schema importer. Try running wsdl.exe on the FileService:

```
http://localhost/WebServices2/FileService.asmx
```

The generated code proxy code will use the type name you specified and include the new namespace import. However, it won't create the FileData class—instead, you'll use the custom version you created in the FileData component.

Finally, add this generated proxy class to your client project. You can now download files with the chunk-by-chunk streaming support that's provided by the FileData class.

■**Note** Currently, only the wsdl.exe uses schema importers. Schema importers don't come into play when you generate a web reference with Visual Studio. Expect this to change in later releases.

Summary

In this chapter, you took an in-depth look at the two most important web service protocols: SOAP and WSDL. SOAP is an incredibly lightweight protocol for messaging. WSDL is a flexible, extensible protocol for describing web services. Together, they ensure that web services can be created and consumed on virtually any programming platform for years to come. This chapter also discussed in detail how you can tailor the XML returned by your web service.