7680CH31.qxd 8/23/06 4:30 PM Page 1157

PART 6

Web Services

CHAPTER 31

Creating Web Services

For years, software developers and architects have struggled to create software components that can be called remotely over local networks and the Internet. In the process, several new technologies and patched-together proprietary solutions were created. Although some of these technologies have been quite successful running back-end systems on internal networks, none has met the challenges of the Internet—a wide, sometimes unreliable, network of computers running on every type of hardware and operating system possible.

This is where XML web services enter the scene. To interact with a web service, you simply need to send an XML message over HTTP. Because every Internet-enabled device supports HTTP, and virtually every programming language has access to an XML parser, there are few limits on what types of applications can use web services. In fact, most programming frameworks include higher-level toolkits that make communicating with a web service as easy as calling a local function.

This chapter provides an overview of web services and the problems they solve. If you are new to web services, you'll learn how to create and consume them in ASP.NET. However, you won't dive into the lower-level details of the underlying protocols just yet. Instead, you'll get started using a web service and then learn to extend it in the next chapter.

WEB SERVICE CHANGES IN .NET 2.0

If you've programmed with web services in .NET 1.*x*, you're probably wondering what has changed in .NET 2.0. From a practical point of view, surprisingly little is new—in fact, the underlying infrastructure is completely the same. However, you will find a number of useful refinements, many of which deal with how web services work with complex types (custom data classes that are used to send or retrieve information to a web method). The changes include the following:

- **Support for property procedures**: If you have a web service that uses a custom type, .NET creates a copy of that class in the client. In .NET 1.*x*, this automatically generated class is built out of public properties. In .NET 2.0, it uses property procedures instead. This minor switch doesn't change how the web service works, but it allows you to use the class in data binding scenarios.
- **Type sharing**: .NET now recognizes when two web services use the same complex type and generates only one client-side class. Best of all, because both proxy classes use the same types, you can easily retrieve an object from one web service and send it directly to another.
- **Custom serialization**: You can now plug into the serialization process for custom classes to take complete control over their XML representation. Although this could be done in ASP.NET 1.*x*, it was undocumented and unsupported.

- **Rich objects**: Need to exchange complex objects, complete with methods and constructors intact? It's possible if you build a new component called a *schema importer*. The schema importer checks the schema of the web service and tells the proxy class what types to use.
- **Contract-first development**: You can now build a .NET web service that conforms to an existing WSDL contract.

Chapter 32 describes all of these improvements.

Many more dramatic changes are just around the corner. Microsoft developers are readying Indigo, a new model for distributed messaging that incorporates the functionality of web services and other .NET technologies, such as remoting. Although there will be a natural upgrade path from ASP.NET web services to Indigo, many of the implementation details will change. Indigo isn't part of .NET 2.0—instead, it's slated to ship with the next version of Windows (and won't arrive any sooner than late 2006). However, Microsoft has hinted it could release an Indigo toolkit for other versions of Windows earlier. For more information, refer to the Microsoft Indigo developer center at `http://msdn.microsoft.com/Longhorn/understanding/pillars/Indigo`.

Web Services Overview

While HTML pages (or the HTML output generated by ASP.NET web forms) are meant to be read by the end user, web services are used by other applications. They are pieces of business logic that can be accessed over the Internet. For example, e-commerce sites can use the web service of a shipping and packaging company to calculate the cost of a shipment. A news site can retrieve the news headlines and articles produced by external news providers and expose them on its own pages in real time. A company can even provide the real-time value of their stock options, reading it from a specialized financial or investment site. All of these scenarios are already taking place on the Web, and major Internet companies such as Amazon, Google, and eBay are providing their own web service offerings to third-party developers.

With web services, you can reuse someone else's business logic instead of replicating it yourself, using just a few lines of code. This technique is similar to what programmers currently do with libraries of APIs, classes, and components. The main difference is that web services can be located remotely on another server and managed by another company.

The History of Web Services

Even though web services are a new technology, you can learn a lot from recent history. Two of the major shifts in software development over the last couple of decades have been the development of object-oriented programming and component-based technology.

Object-oriented programming joined the mainstream in the early 1980s. Many saw object-oriented programming as the solution to the software crisis that resulted from the increasing complexity and size of software applications. Most projects were late and over budget, and the end result was often unreliable. The promise of object-oriented code was that by structuring code into objects, developers could create components that were more reusable, extensible, and maintainable.

The 1990s saw the birth of component technology, which made it possible to build applications by assembling components. Component technology is really an extension of object-oriented principles outside the boundaries of any one particular language so that it becomes a core piece of infrastructure that everyone can use. While object-oriented languages allowed developers to reuse objects in their applications, component-based technologies allowed developers to easily share compiled objects *between* applications. Two dominant component-based technologies emerged—COM (the Component Object Model) and CORBA (Common Object Request Broker Architecture). Since that time, other component technologies have appeared (such as JavaBeans and .NET), but these are designed as proprietary solutions for specific programming frameworks.

Soon after COM and CORBA were created, these standards were applied to distributed components so that an application could interact between objects hosted on different computers in a network. Although both COM and CORBA have a great deal of technical sophistication, they are often difficult to set up and support in network environments, and they can't work together. These headaches became dramatically worse when the Internet appeared and developers began to apply these technologies to create distributed applications that spanned slower, less reliable, WANs (wide area networks). Because of their inherent complexity and proprietary nature, neither COM nor CORBA became truly successful in this environment.

Web services are a new technology that aims to answer these problems by extending component object technology to the Web. Essentially, a web service is a unit of application logic (a component) that can be remotely invoked over the Internet. Many of the promises of web services are the same as those of component technology—the aim is to make it easier to assemble applications from prebuilt application logic, share functionality between organizations and partners, and create more modular applications. Unlike earlier component technologies, web services are designed *exclusively* for this purpose. This means that as a .NET developer, you will still use the .NET component model to share compiled assemblies between .NET applications. However, if you want to share functionality between applications running on different platforms or hosted by different companies, web services fit perfectly.

Web services also place a much greater emphasis on interoperability. All software development platforms that allow programmers to create web services use the same bedrock of open XML-based standards. This ensures that you can create a web service using .NET and call it from a Java client or, alternatively, create a Java web service and call it from a .NET application.

■ **Note** Web services are not limited to the .NET Framework. The standards were defined before .NET was released, and they are exposed, used, and supported by vendors other than Microsoft. The .NET Framework is special because it hides all the plumbing code, and this makes it far easier to expose your own services over the Internet or to access the services provided by other companies. As you'll see, you don't need to know all the details of XML and SOAP to successfully program web services (although, of course, some knowledge helps). ASP.NET abstracts the nitty-gritty stuff and generates wrapper classes that expose a simple object-oriented model to send, receive, and interpret the SOAP messages easily.

Distributed Computing and Web Services

To fully understand the importance of web services, you need to understand the requirements of *distributed computing*. Distributed computing is the partitioning of application logic into units that are executed on two or more computers in a network. The idea of distributed computing has been around a long time, and numerous communication technologies have been developed to allow the distribution and reuse of application logic.

Many reasons exist for distributing application logic. Some of the most important include the following:

High scalability: By distributing the application logic, the load is spread out to different machines. This usually won't improve the performance of the application for individual users (in fact, it may slow it down), but it will almost always improve the scalability, thereby allowing the application to serve a much larger number of users at the same time.

Easy deployment: Pieces of a distributed application may be upgraded without upgrading the whole application. A centrally located component can be updated without needing to update hundreds (or event thousands) of clients.

Improved security: Distributed applications often span company or organization boundaries. For example, you might use distributed components to let a trading partner query your company's product catalog. It wouldn't be secure to let the trading partner connect directly to your company database. Instead, the trading partner needs to use a component running on your servers, which you can control and restrict appropriately.

The Internet has increased the importance and applicability of distributed computing. The simplicity and ubiquity of the Internet makes it a logical choice as the backbone for distributed applications.

Before web services, the dominant protocols were COM (which is called DCOM, or Distributed COM, when used on a network) and CORBA. Although CORBA and DCOM have a lot in common, they differ in the details, making it hard to get the protocols to interoperate. Table 31-1 summarizes some similarities and differences between CORBA, DCOM, and web services. It also introduces a slew of acronyms.

Table 31-1. *Comparing Different Distributed Technologies*

Characteristic	CORBA	DCOM	Web Services
RPC (Remote Procedure Call) mechanism	IIOP (Internet Inter-ORB Protocol)	DCE-RPC (Distributed Computing Environment–Remote Procedure Call)	HTTP (Hypertext Transfer Protocol)
Encoding	CDR (Common Data Representation)	NDR (Network Data Representation)	XML (Extensible Markup Language)
Interface description	IDL (Interface Definition Language)	IDL (Interface Definition Language)	WSDL (Web Service Description Language)
Discovery	Naming service and trading service	Registry	UDDI (Universal Description, Discovery, and Integration)
Firewall friendly?	No	No	Yes
Complexity of protocols	High	High	Low
Cross-platform?	Partly	No	Yes

Both CORBA and DCOM allow for the invocation of remote objects. CORBA uses a standard called IIOP (Internet Inter-ORB Protocol), and DCOM uses a variation of a standard named DCERPC (Distributed Computing Environment Remote Procedure Call). The encoding of data in CORBA is based on a format named CDR (Common Data Representation). In DCOM, the encoding of data is based on a similar but incompatible format named NDR (Network Data Representation). These layers of standards make for significant complexity!

Also, differences exist between the languages that both protocols support. DCOM supports a wide range of languages (C++, Visual Basic, and so on) but was used primarily on Microsoft operating systems. CORBA supported different platforms but mostly gained traction with Java-based applications. As a result, developers had two platforms that had the technical ability to support systems of distributed objects but couldn't work together.

The Problems with Distributed Component Technologies

Interoperability is only part of the problem with CORBA and DCOM. Other technical challenges exist. Both protocols were developed before the Internet, and as such they aren't designed with the needs of a loosely coupled, sometimes unreliable, heavily trafficked network in mind. For example, both protocols are connection-oriented. This means a DCOM client holds onto a connection to the DCOM server to make multiple calls. The server-side DCOM component can also retain information about the client in memory. This provides a rich, flexible programming model, but it's a poor way to design large-scale applications that use the stateless protocols of the Internet. If the client simply disappears without properly cleaning up the connection, unnecessary resources are wasted. Similarly, if thousands of clients try to connect at once, the server can easily become swamped, running out of memory or connections.

Another problem is that both protocols are exceedingly complex. They combine distributed-object technology with features for network security and lifetime management. Web services are so much easier to use in large part because they don't include this level of sophistication. However, that doesn't mean you can't create a secure web service. It just means that if you do, you'll need to rely on web services *and* another standard, such as SSL (as implemented by the web server) or WS-Security and XML Encryption (as implemented by the programming framework).

■**Note** The danger here is that developers could be swamped by a proliferation of standards that aren't required for basic web services but are required for sophisticated web service applications. However, this model still represents the best trade-off between complexity and simplicity. The advantage is that architects can develop innovations for web services (such as transactional support), without compromising the basic level of interoperability provided by the core web service standards.

The Benefits of Web Services

Web services are interesting from several perspectives. From a technological perspective, web services try to solve some problems faced when using tightly coupled technologies such as CORBA and DCOM. These are problems such as getting through firewalls, dealing with the complexities of lower-level transport protocols, and integrating heterogeneous platforms. Web services are also interesting from an organizational and economic perspective, because they open doors for new ways of doing business and integrating systems between organizations.

DCOM and CORBA are fine for building enterprise applications with software running on the same platform and in the same closely administered local network. They are not fine, however, for building applications that span platforms, span the Internet, and need to achieve Internet scalability. They were simply not designed for this purpose.

This is where web services come in. Web services represent the next logical step in the evolution of component-based distributed technologies. Some key advantages include the following:

Web services are simple: That simplicity means they can be easily supported on a wide range of platforms.

Web services are loosely coupled: The web service may extend its interface and add new methods without affecting the clients as long as it still provides the old methods and parameters.

Web services are stateless: A client makes a request to a web service, the web service returns the result, and the connection is closed. There is no permanent connection. This makes it easy to scale up and out to many clients and use a server farm to serve the web services. The underlying HTTP used by web services is also stateless. Of course, it is possible to provide some state by using additional techniques such as the ones you use in ASP.NET web pages, including cookies. However, these techniques aren't standardized.

Web services are firewall-friendly: Firewalls can pose a challenge for distributed object technologies. The only thing that almost always gets through firewalls is HTTP traffic on ports 80 and 443. Because web services use HTTP, they can pass through firewalls without explicit configuration.

■**Note** It is still possible to use a firewall to block SOAP traffic. This is possible since the HTTP header of a web service message identifies it as a SOAP message, and an administrator may configure the firewall to stop SOAP traffic. For business-to-business scenarios, the firewall may allow SOAP traffic only from selected ranges of IP addresses.

Of course, the simplicity of web services does come with a cost. Namely, web services don't have all the features of more complex distributed component technologies. For example, there is no support for bidirectional communication, which means the web server cannot call back to a client after the client disconnects. In this respect, tightly coupled protocols such as DCOM and CORBA are more powerful than web services. The .NET Framework also has a new technology, *remoting*, that is ideal for communicating between distributed .NET applications in an internal network. Remoting is the successor to DCOM on the .NET platform. If you want to know more about remoting, you can read *Advanced .NET Remoting, Second Edition* (Apress, 2005). For information about whether you should use remoting instead of web services, refer to the sidebar "When to Use Web Services."

WHEN TO USE WEB SERVICES

Microsoft suggests two rules of thumb for deciding whether to use web services. If you need to cross platform boundaries (for example, communicate between a .NET and a Java application) or trust boundaries (for example, communicate between two companies), web services make great sense. Web services are also a good choice if you want to use built-in ASP.NET features such as caching or IIS features such as SSL security or Windows authentication. They also make sense if you want to leave yourself open to third-party integration in the future.

However, if you simply want to share functionality between two .NET applications, web services can be overkill—and they may introduce unnecessary overhead. For example, if you simply want web applications to have access to specific business logic, a much better approach is to create a class library assembly (which is compiled to a DLL) and use it in both applications. This avoids the overhead of out-of-process or network communication, which can be significant.

Finally, if you want to distribute functionality so it can be accessed remotely, but both the client and server are built using the .NET Framework, you might want to consider using .NET remoting instead. Remoting doesn't provide the same level of support for open standards such as SOAP, but it does give you the freedom to use different types of communication, proprietary .NET data types, stateful objects, and faster TCP/IP communication. In short, remoting offers more features and the possibility to enhance performance for .NET-only solutions.

Making Money with Web Services

A new technology is doomed if it does not give new opportunities for the people concerned with making money. From a businessperson's perspective, web services open new possibilities for the following reasons:

New payment structures: The user of a web service can pay a subscription fee for using the service. One example may be the news feed from Associated Press. Another possibility is a pay-per-view, or *micro payment*, model. A provider of a credit verification service, for instance, may charge per request.

Real-time interaction and collaboration: Today, data is typically replicated and used locally. Web services enable real-time queries to remote data. An example is an e-commerce site selling computer games. The e-commerce site may hook up to a warehouse to get the number of items in stock in real time. This enables the e-commerce site to provide a better service. Nothing is more frustrating than buying something over the Internet just to learn the next day that the product you wanted is out of stock.

Aggregated services: A web service may aggregate other web services, screen-scraped websites, legacy components exposed using proprietary protocols, and so on. A typical example of an aggregated service is a comparative service giving you the best deal on products. Another type of service is one that groups related services. For example, imagine you're moving to a new home. Someone could provide you with a service that can update your address at the post office, find the transportation company to move all your possessions, and so on.

Web services are by no means the only technology that can provide these solutions. Many similar solutions are available today using existing technology. However web services have the momentum and standards to make these kinds of services generally available.

The Web Service Stack

The key to the success of web services is that they are based on open standards *and* that major vendors such as Microsoft, IBM, and Sun are behind these standards. Still, open standards do not automatically lead to interoperability. First, the vendors must implement all the standards. Furthermore, they must implement the standards in a compatible way.

Several specifications are used when building web services. Figure 31-1 shows the web service stack as it exists today.

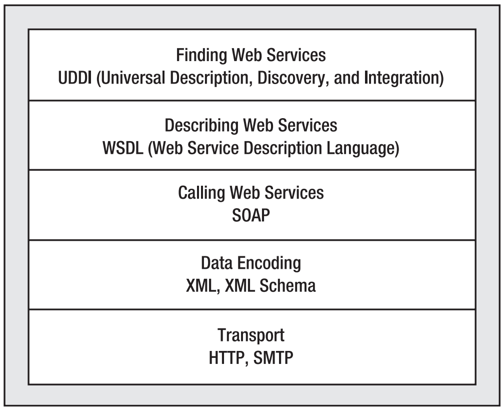


Figure 31-1. *The web service technology stack*

You'll learn much more about the SOAP, WSDL, and UDDI protocols that support web services in the next chapter. However, before you get started creating and using web services, it's important to get a basic understanding of the role these standards play. Table 31-2 summarizes these standards, and the next few sections fill in some of the details. You'll get into much more detail in the next chapter.

Table 31-2. *Web Service Standards*

Standard	Description
WSDL	Used to create an interface definition for a web service. The WSDL document tells a client what methods are present in a web service, what parameters and return values each method uses, and how to communicate with them.
SOAP	The message format used to encode information (such as data values) before sending it to a web service.
HTTP	The protocol over which all web service communication takes place. For example, SOAP messages are sent over HTTP channels.
DISCO	Used to create discovery documents that provide links to multiple web service endpoints. This standard is Microsoft-specific and will eventually be replaced by a similar standard named WS-Inspection.
UDDI	A standard for creating business registries that catalog companies, the web services they provide, and the corresponding URLs for their WSDL contracts.

Finding Web Services

In a simple application, you may already know the URL of the web service you want to use. If so, you can hard-code it or place it in a configuration file. No other steps are required.

In other situations, you might want to search for the web service you need at runtime. For example, you might use a standardized service that's provided by different hosting companies and is not always available at the same URL. Or, you may just want an easy way to find all the web services provided by a trading partner. In both of these situations, you need to use *discovery* to programmatically locate the web services you need.

Two specifications help in the discovery of a web service:

DISCO (an abbreviation of *discovery*): The DISCO standard creates a single file that groups a list of related web services. A company can publish a DISCO file on its server that contains links to all the web services it provides. Then clients simply need to request this file to find all the available web services. This is useful when the client already knows a company that it's offering services and wants to see what web services they expose and find links to the details of its services. It's not very useful to search for new web services over the Internet, but it may be helpful for local networks where a client connects to the server and can see what and where services are available.

UDDI (Universal Description, Discovery, and Integration): UDDI is a centralized directory where web services are published by a group of companies. It's also the place where potential clients can go to search for their specific needs. Different organizations and groups of companies may use different UDDI registries. To retrieve information from a UDDI directory or register your components, you use a web service interface.

Discovery is one of the newest and least mature parts of the web service protocol stack. DISCO is supported only by Microsoft and is slated to be replaced by a similar more general standard named WS-Inspection in future .NET releases. UDDI is designed for web services that are intended to be shared publicly or among a consortium of companies or organizations. It's not incorporated into the .NET Framework, although you can download a separate .NET component to search UDDI directories and register your components (see `http://msdn.microsoft.com/library/en-us/uddi/uddi/portal.asp`). Because there aren't yet any well-established UDDI directories, and because many web services are simply designed for use in a single company or between a small set of known trading partners, it's likely that most web services will not be published in UDDI.

Describing a Web Service

For a client to know how to access a web service, the client must know what methods are available, what parameters each method uses, and what the data type of each parameter is. The WSDL (Web Service Description Language) is an XML-based language that describes all these details. It describes the request message a client needs to submit to the web service and the response message the web service returns. It also defines the transport protocol you need to use (typically HTTP) and the location of the web service.

WSDL is a complex standard. But as you'll see in this chapter, certain tools consume WSDL information and automatically generate helper classes that hide the low-level plumbing required to interact with web services.

The Wire Format

To communicate with a web service, you need a way to create request and response messages that can be parsed and understood on any platform. SOAP (formerly Simple Object Access Protocol but no longer considered an acronym) is the XML-based language you use to create these messages.

It's important to understand that SOAP defines the messages you use to exchange data (the message format), but it doesn't describe how you send the message (the transport protocol). With ASP.NET web services, the transport protocol is HTTP. In other words, to communicate with a web service, a client opens an HTTP connection and sends a SOAP message.

.NET also supports HTTP GET and HTTP POST, two simpler approaches for interacting with web services that aren't as standardized and don't offer the same rich set of features. In both these cases, an HTTP channel is used for communication, and data is sent as a simple collection of name/value pairs, not as a full-blown SOAP message. The only place you're likely to see this simpler approach used in the .NET environment is in the simple browser-based page ASP.NET provides for testing your web services. In fact, by default the ASP.NET 1.1 machine.config file allows only HTTP POST requests from the local computer and disables HTTP GET support entirely.

Figure 31-2 summarizes the web service life cycle. First, the web service consumer finds the web service, either by going directly to the web service URL or by using a UDDI server or DISCO file. Next, the client retrieves the web service WSDL document, which describes how to interact with the web service. Both of these tasks take place at design time. When you run the application and actually interact with the web service, the client sends a SOAP message to trigger to the appropriate web method.

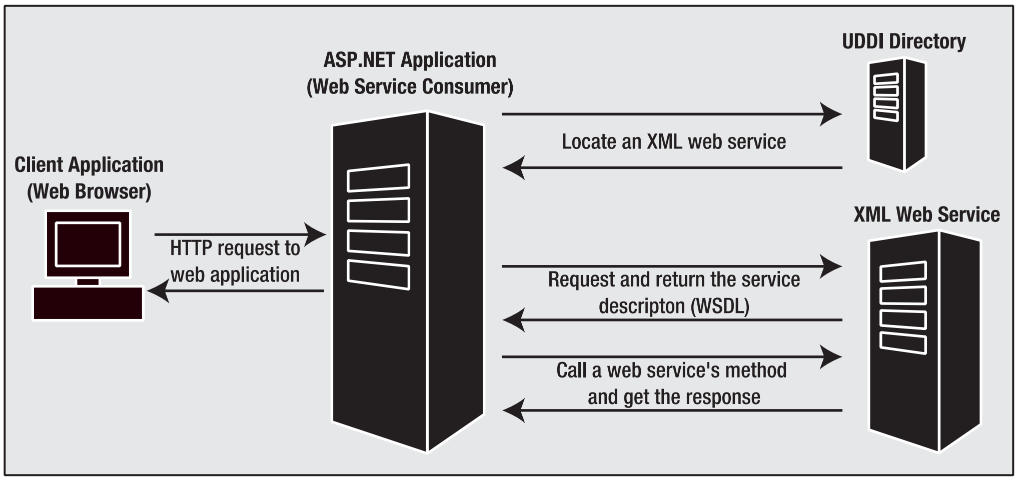


Figure 31-2. *The web service life cycle*

Building a Basic Web Service

In the following sections, you'll see how to build a web service with ASP.NET and how to test it in a browser. In this example, you'll develop a web service called EmployeesService, which will count and return the number of employees for a specific city (or the total of all employees) by querying the Employees table in SQL Server's sample Northwind database.

The Web Service Class

A web service begins as a stateless class with one or more methods. It's these methods that the remote clients call to invoke your code.

EmployeeService is designed to allow remote clients to retrieve information about the employees in a company. It provides a GetEmployees() method that returns a DataSet with the full set of employee information. It also provides a GetEmployeesCount() method, which simply returns the number of employees in the database. To provide these methods, you need nothing more than some basic ADO.NET code. Here's the full class listing:

```
public class EmployeesService
{
    private string connectionString;
    public EmployeesService()
    {
        string connectionString =
          WebConfigurationManager.ConnectionStrings["Northwind"].ConnectionString;
    }

    public int GetEmployeesCount()
    {
        SqlConnection con = new SqlConnection(connectionString);
        string sql = "SELECT COUNT(*) FROM Employees";
        SqlCommand cmd = new SqlCommand(sql, con);
        // Open the connection and get the value.
        con.Open();
        int numEmployees = -1;
        try
        {
            numEmployees = (int)cmd.ExecuteScalar();
        }
        finally
        {
            con.Close();
        }
        return numEmployees;
    }

    public DataSet GetEmployees()
    {
        // Create the command and the connection.
        string sql = "SELECT EmployeeID, LastName, FirstName, Title, " +
         "TitleOfCourtesy, HomePhone FROM Employees";
        SqlConnection con = new SqlConnection(connectionString);
        SqlDataAdapter da = new SqlDataAdapter(sql, con);
        DataSet ds = new DataSet();

        // Fill the DataSet.
        da.Fill(ds, "Employees");
        return ds;
    }
}
```

Both GetEmployees() and GetEmployeesCount() are public methods, which means the class can be used from any web page in the same project. (Or, if you compile this class into a separate DLL, you can use it in any project that references this assembly.) However, you need to take a few extra steps to make this class ready for a web service.

Web Service Requirements

Before you transform your class into a web service, you need to make sure it's compatible with the requirements of the underlying XML-based standards (which you'll learn more about in the next chapter). Because web services are executed by ASP.NET, which hosts the CLR, you can use any valid .NET code. That means you can use .NET classes to access data with ADO.NET or validate data with regular expressions. As with any back-end business component, you can't show any user interface, but other than that your code has no restrictions.

However, there *are* limitations on what information your code can accept (in the form of parameters) and return (in the form of a return value). That's because web services are built on XML-based standards for exchanging data. As a result, the set of data types web services can use is limited to the set of data types recognized by the XML Schema standard. This means you can use simple data types such as strings and numbers, but you have no automatic way to send proprietary .NET objects such as a FileStream, an Image, or an EventLog. This restriction makes a lot of sense. Clearly, other programming languages have no way to interpret these more complex classes, so even if you could devise a way to send them over the wire, the client might not be able to interpret them, which would thwart interoperability. (.NET remoting is an example of a distributed component technology that *does* allow you to use .NET-specific types. However, the cost of this convenience is that it won't support non-.NET clients.)

Table 31-3 lists the supported web service data types.

Table 31-3. *Web Service Data Types for Parameters and Return Values*

Data Type	Description
The Basics	Simple C# data types such as integers (short, int, long), unsigned integers (ushort, uint, ulong), nonintegral numeric types (float, double, decimal), and a few other miscellaneous types (bool, string, char, byte, and DateTime).
Arrays	You can use arrays of any supported type. You can also use an ArrayList (which is simply converted into an array), but you can't use more specialized collections such as the Hashtable. You can also use binary data through byte arrays. Binary data is automatically Base64 encoded so that it can be inserted into an XML web service message.
Custom Objects	You can pass any object you create based on a custom class or structure. The only limitation is that only public data members are transmitted, and all public members and properties must use one of the other supported data types. If you use a class that includes custom methods, these methods will not be transmitted to the client, and they will not be accessible to the client.
Enumerations	Enumerations types (defined in C# with the enum keyword) are supported. However, the web service uses the string name of the enumeration value (not the underlying integer).
XmlNode	Objects based on System.Xml.XmlNode are representations of a portion of an XML document. You can use this to send arbitrary XML.
DataSet and DataTable	You can use the DataSet and DataTable to return information from a relational database. Other ADO.NET data objects, such as Data-Columns and DataRows, aren't supported. When you use a DataSet or DataTable, it's automatically converted to XML in a similar way as if you had used the GetXml() or WriteXml() method.

■**Note** The supported web service data types are based on the types defined by the XML Schema standard. These map fairly well to the basic set of C# data types.

The EmployeesServices class follows these rules. The only data types it uses for parameters and return values are int and DataSet, both of which are supported. Of course, some web service programmers prefer to steer clear of the DataSet (see the sidebar "The DataSet and XML Web Services"), but it's still a reasonable, widely used approach.

One other requirement is that your web services should be stateless. In fact, the web service architecture works in the same way as the web-page architecture—a new web service object is created at the beginning of the request, and the web service object is destroyed as soon as the request has been processed and the response has been returned. The EmployeesServices class fits well with this model, because it doesn't retain any state in class member variables. The only exception is the connectionString variable, which is initialized with the required value every time the class is created.

THE DATASET AND XML WEB SERVICES

You'll notice that the DataSet is one of the few specialized .NET classes that is supported by web services. That's because the DataSet has the ability to automatically serialize itself to XML. However, this support comes with a significant caveat—even though non-.NET clients can use a web service that returns a DataSet, they might not be able to do anything useful with the DataSet XML! That's because other languages won't be able to automatically convert the DataSet into a manageable objects. Instead, they will be forced to use their own XML programming APIs. Although these work in theory, they can be tedious in practice, especially with complex, proprietary XML. For that reason, developers usually avoid the DataSet when creating web services that need to support clients on a wide range of platforms.

It's worth noting that Microsoft could have used the DataSet approach with many other .NET classes in order to make it possible for them to be serialized into XML. However, Microsoft wisely restrained itself from adding these features, realizing this would make it far too easy for programmers to create applications that used web service standards but weren't practical in cross-platform scenarios. (Not so long ago, Microsoft might have pursued exactly this "embrace and extend" philosophy, but fortunately it has recognized the need to foster integration and broad compatibility between applications.)

So that still leaves the question of why Microsoft decided to support the DataSet in its web services toolkit. The reason is because the DataSet enables one of the most common uses of web services—returning a snapshot of information from a relational database. The benefit of adding this feature seemed worth the cost of potential interoperability headaches for developers who don't consider their web service architecture carefully.

Support for Generics

Web services support generics. However, this support might not be exactly what you expect.

It's completely acceptable to create a web service method that accepts or returns a generic type. For example, if you want to return a collection of EmployeeDetails objects, you could use the generic List class, as shown here:

```
public List<EmployeeDetails> GetEmployees()
{ ... }
```

In this case, .NET treats your collection of EmployeeDetails objects in the same way as an array of EmployeeDetails objects:

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

Of course, for this to work, there can't be anything that breaks the serialization rules in the EmployeeDetails class or the List class. For example, if these classes have a nonserializable property, the entire object can't be serialized.

The reason .NET supports generics in this example is because it's quite easy for .NET to determine the real class types at compile time. That allows .NET to determine the structure of the XML messages this method will use and add the information to the WSDL document (as you'll see in the next chapter).

However, .NET doesn't support generic methods. For example, this method isn't allowed:

```
public List<T> GetEmployees<T>()
{ ... }
```

Here the GetEmployees() method is itself generic. It allows the caller to choose a type that will be used by the method. Because this method could in theory be used with absolutely any type of document, there's no way to document it properly and determine the appropriate XML message format in advance.

Exposing a Web Service

Now that you've verified that the EmployeesService class is ready for the Web, it's time to convert it to a web service. The crucial first step is to add the System.Web.Services.WebMethod attribute to each method you want to expose as part of your web service. This web service instructs ASP.NET to make this method available for inspection and remote invocation.

Here's the revised class with two web methods:

```
public class EmployeesService
{
    [WebMethod()]
    public int GetEmployeesCount()
    { ... }

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

These two simple changes complete the transformation from your class into a web service. However, the client still has no entry point into your web service—in other words, there's no way for another application to trigger your web methods. To allow this, you need to create an .asmx file that exposes the web service.

■ **Note** In this example, the web service contains the data access code. However, if you plan to use the same code in a web application, it's worth adding an extra layer using database components. To implement this design, you would first create a separate database component (as described in Part 2) and then use that database component directly in your web pages and your web service.

ASP.NET implements web services as files with the .asmx extension. As with a web page, you can place the code for a web service directly in the .asmx or in a class in a code-behind file that the .asmx file references (which is the Visual Studio approach).

For example, you could create a file named EmployeesService.asmx and link it to your EmployeesService class. Every .asmx file begins with a WebService directive that declares the server-side language used in the file and the class. It can optionally declare other information, such as the code-behind file and whether you want to generate debug symbols during the compilation. In this respect it is similar to the Page directive for .aspx files.

Here's an example .asmx file with the EmployeesService:

```
<%@ WebService Language="C#" Class="EmployeesService" %>
```

In this case, you have two choices. You can insert the class code immediately after the WebService attribute, or you can compile it into one of the assemblies in the Bin directory. If you've added the EmployeesService class to a Visual Studio project, it will automatically be compiled as part of the web application DLL, so you don't need to include anything else in the .asmx file.

At this point, you're finished. Your web service is complete, available, and ready to be used in other applications.

■ **Tip** There's no limit to how many web services you add to a single web application, and you can freely mingle web services and web pages.

Web Services in Visual Studio

If you're using Visual Studio, you probably won't go through the process of creating a class, converting it a web service, and then adding an .asmx file. Instead, you'll create the .asmx file and the code-behind in one step, by selecting Website ➤ Add New Item from the menu. You can choose to put the web service code directly in the .asmx file or in a separate code-behind file, just as you can with a web page.

So, you haven't seen two other web service details. First, the web service class inherits from System.Web.Services.WebService, and second, a WebService attribute is applied to the class declaration. Neither of these details is required, but you'll consider their role in the following sections.

Deriving from the WebService Class

When you create a web service in Visual Studio, your web service class automatically derives from the base WebService class, as shown here:

```
public class EmployeesService : System.Web.Services.WebService
{ ... }
```

Inheriting from the WebService class is a convenience that allows you to access the built-in ASP.NET objects (such as Application, Session, and User) just as easily as you can in a web form. These objects are provided as properties of the WebService class, which your web service acquires through inheritance. If you don't need to use any of these objects (or if you're willing to go through the static HttpContext.Current property to access them), you don't need to inherit.

Here's how you would access Application state in a web service if you derive from the base WebService class:

```
// Store a number in session state.
Session["Counter"] = 10;
```

Here's the equivalent code you would need to use if your web service class doesn't derive from WebService:

```
// Store a number in session state.
HttpContext.Current.Session["Counter"] = 10;
```

This technique won't actually work as intended (in other words, the client won't keep the same session across multiple web method calls) unless you take some extra steps, as described later in the "EnableSession" section.

Table 31-4 lists the properties you receive by inheriting from WebService.

Table 31-4. *WebService Properties*

Property	Description
Application	An instance of the HttpApplicationState class that provides access to the global application state of the web application
Context	An instance of the HttpContext class for the current request
Server	An instance of the HttpServerUtility class
Session	An instance of the HttpSessionState class that provides access to the current session state
User	An IPrincipal object that allow you to examine user credentials and roles, if the user has been authenticated

Since the .NET Framework supports only single inheritance, inheriting from WebService means your web service class cannot inherit from other classes. This is really the only reason not to inherit from WebService.

■ **Note** An interesting point with inheriting from WebService is that WebService is derived from the System.MarshalByRefObject class. This class is the base class used for .NET remoting. As a result, when you create a class that derives from WebService, you gain the ability to use your class in several ways. You can use it as any other local class (and access it directly in your web pages), you can expose it as part of a web service, or you can expose it as a distributed object in a .NET remoting host. To learn more about .NET remoting, refer to *Advanced .NET Remoting, Second Edition* (Apress, 2005).

Documenting a Web Service

Web services are self-describing, which means ASP.NET automatically provides all the information the client needs about what methods are available and what parameters they require. This is provided by the XML-based standard called WSDL, which you'll explore in the next chapter. However, although a WSDL document describes the mechanics of the web service, it doesn't describe its purpose or the meaning of the information supplied to and returned from each method. Most web services will provide this information in separate developer documents. However, you can (and should) include a bare minimum of information with your web service by using the WebMethod and WebService attributes.

You can add descriptions to each method through the Description property of the WebMethod attribute and to the entire web service as a whole using the Description property of the WebService attribute. You can also apply a descriptive name to the web service using the Name property of the WebService attribute. Here's an example of how you might insert this information in the EmployeesService:

```
[WebService(Name="Employees Service",
 Description="Retrieve the Northwind Employees")]
public class EmployeesService : System.Web.Services.WebService
{
    [WebMethod(Description="Returns the total number of employees.")]
    public int GetEmployeesCount()
    { ... }

    [WebMethod(
     Description="Returns the full list of employees.")]
    public DataSet GetEmployees()
    { ... }
}
```

These custom descriptions are added to the WSDL document that describes your service. It's also shown in the automatically generated test page that you'll use in the next section.

You should supply one other detail for your web service—a unique XML namespace. This allows your web service (and the XML messages it generates) to be uniquely identified. XML namespaces were first introduced in Chapter 12. By default, ASP.NET web services use the default XML namespace `http://tempuri.org/`, which is suitable only for testing. If you don't set a custom namespace, you'll see a warning message in the test page advising you to use something more distinctive. Note that the XML namespace has no relationship to the concept of .NET namespaces. It doesn't affect how your code works or how the client uses your web service. Instead, the XML namespace simply identifies your web service. XML namespaces usually look like URLs. However, they don't need to correspond to a valid Internet location.

Ideally, the namespace you use will refer to a URL address that you control. Often, this will incorporate your company's Internet domain name as part of the namespace. For example, if your company uses the website `http://www.mycompany.com`, you might give the Employees web service a namespace such as `http://www.mycompany.com/EmployeesService`.

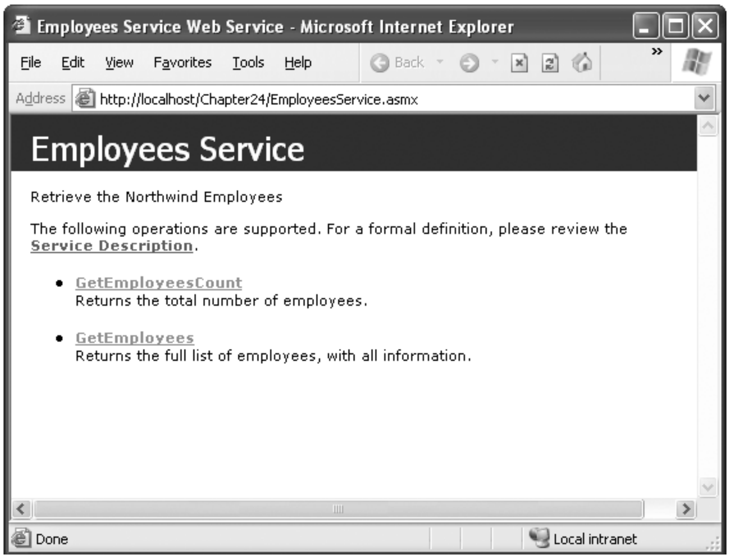
The namespace is specified through the WebService attribute, as shown here:

```
[WebService (Name="Employees Service",
 Description="Retrieve the Northwind Employees",
 Namespace="http://www.apress.com/ProASP.NET/")]
public class EmployeesService : System.Web.Services.WebService
{ ... }
```

Testing a Web Service

Now that you've seen how to create a simple web service, you're ready to test it. Fortunately, you don't need to write a client application to test it because .NET includes a test web page that ASP.NET uses automatically when you request the URL of an .asmx file in a browser. This page uses reflection to read and show information about the web services, such as the names of the methods it provides.

To try the test page, request the EmployeesService.asmx file in your browser. (In Visual Studio, you simply need to set this as the start page for your application and then run it.) Figure 31-3 shows the test page you'll see.


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

Employees Service

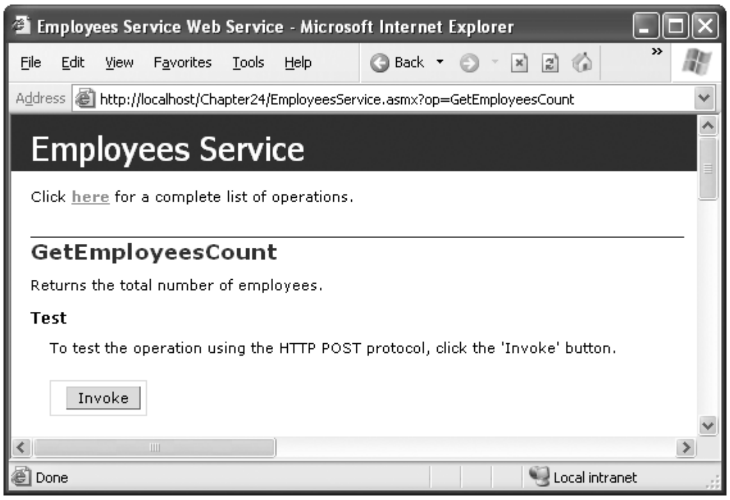
Retrieve the Northwind Employees

The following operations are supported. For a formal definition, please review the Service Description.

- GetEmployeesCount
  Returns the total number of employees.
- GetEmployees
  Returns the full list of employees, with all information.

Figure 31-3. *The web service test page*

Note that the page displays the two web methods with their descriptions, and the page's title is the name of the web service. If you click one of the methods, you'll see a page that allows you to test the method (and supply the data for any method parameters). Figure 31-4 shows the page that allows you to test the GetEmployeesCount() method.


Employees Service Web Service - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back
Address
Done
Local intranet

http://localhost/Chapter24/EmployeesService.asmx?op=GetEmployeesCount

Employees Service

Click here for a complete list of operations.

GetEmployeesCount

Returns the total number of employees.

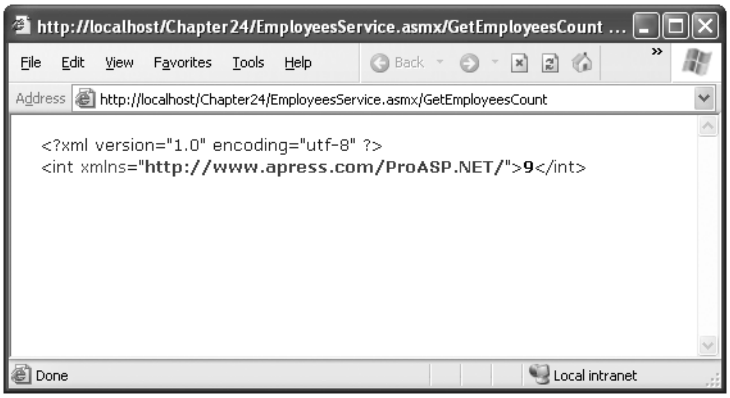
Test

To test the operation using the HTTP POST protocol, click the 'Invoke' button.

Invoke

Figure 31-4. *Testing a web method*

When you click the Invoke button, a new web page appears with an XML document that contains the requested data. Looking at Figure 31-5, you will see nine employee records. If you look at the URL, you'll see that it incorporates the .asmx file, followed by the web service method name.


http://localhost/Chapter24/EmployeesService.asmx/GetEmployeesCount ...
File Edit View Favorites Tools Help
Back
Address
Done
Local intranet

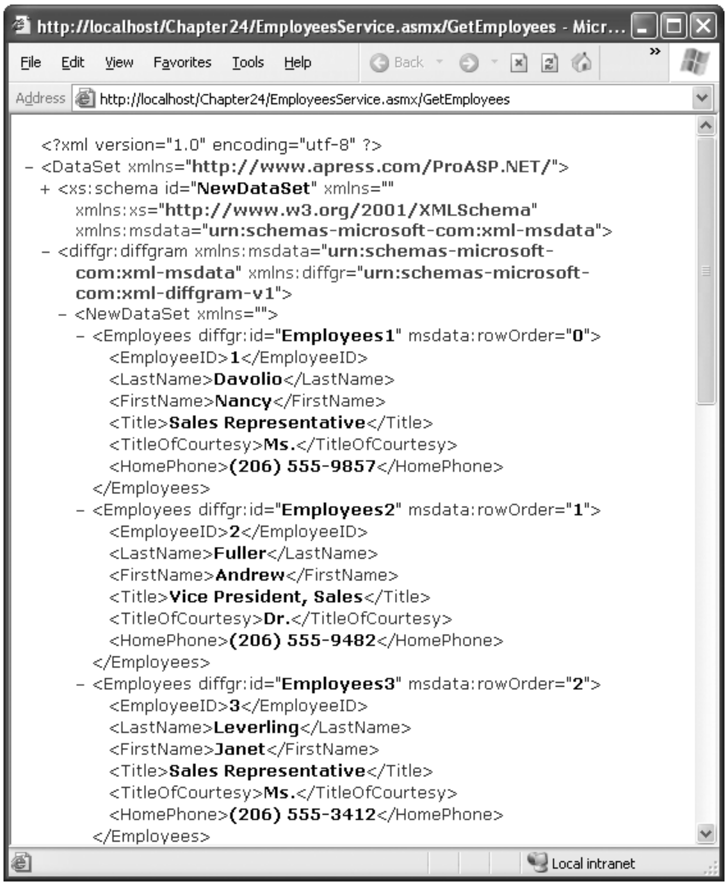
http://localhost/Chapter24/EmployeesService.asmx/GetEmployeesCount

```
<?xml version="1.0" encoding="utf-8" ?>
<int xmlns="http://www.apress.com/ProASP.NET/">9</int>
```

Figure 31-5. *The results for GetEmployeesCount()*

You can repeat this process to invoke GetEmployees(), in which case you'll see the much more detailed XML that represents the entire DataSet contents (as shown in Figure 31-6).

As you can see, thanks to this helper page, testing a basic web service is quite straightforward and doesn't require you to build a client.


http://localhost/Chapter24/EmployeesService.asmx/GetEmployees - Micr...
File Edit View Favorites Tools Help
Back
Address http://localhost/Chapter24/EmployeesService.asmx/GetEmployees
Local intranet

```
  <?xml version="1.0" encoding="utf-8" ?>
- <DataSet xmlns="http://www.apress.com/ProASP.NET/">
  + <xs:schema id="NewDataSet" xmlns=""
      xmlns:xs="http://www.w3.org/2001/XMLSchema"
      xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
  - <diffgr:diffgram xmlns:msdata="urn:schemas-microsoft-
      com:xml-msdata" xmlns:diffgr="urn:schemas-microsoft-
      com:xml-diffgram-v1">
    - <NewDataSet 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>
      - <Employees diffgr:id="Employees3" msdata:rowOrder="2">
          <EmployeeID>3</EmployeeID>
          <LastName>Leverling</LastName>
          <FirstName>Janet</FirstName>
          <Title>Sales Representative</Title>
          <TitleOfCourtesy>Ms.</TitleOfCourtesy>
          <HomePhone>(206) 555-3412</HomePhone>
        </Employees>
```

Figure 31-6. *The results for GetEmployees()*

The test pages aren't part of the web services standards; they're just a frill provided by ASP.NET. In fact, the test page is rendered by ASP.NET on the fly using the web page c:\[WinDir]\Microsoft.NET\Framework\[Version]\Config\DefaultWsdlHelpGenerator.aspx. In some cases, you may want to modify the appearance or behavior of this page. If so, you simply need to copy the DefaultWsdlHelpGenerator.aspx file to your web application directory, modify it, and then change the web.config file for the application to point to the new rendering page by adding the <wsdlHelpGenerator> element, as shown here:

```
<configuration>
  <system.web>
    <webServices>
      <wsdlHelpGenerator href="MyWsdlHelpGenerator.aspx"/>
    </webServices>
    <!-- Other settings omitted. -->
  </system.web>
</configuration>
```

This technique is most commonly used to change the look of the test page. For example, you might use this technique to substitute a version of the page that has a company logo or copyright notice.

Consuming a Web Service

Before a client can use a web service, the client must be able to create, send, receive, and understand XML-based messages. This process is easy in principle but fairly tedious in practice. If you had to implement it yourself, you would need to write the same low-level infrastructure code again and again.

Fortunately, .NET provides a solution with a dedicated component called a *proxy class*, which performs the heavy lifting for your application. The proxy class wraps the calls to the web service's methods. It takes care of generating the correct SOAP message format and managing the transmission of the messages over the network (using HTTP). When it receives the response message, it also converts the results back to the corresponding .NET data types.

■**Note** To access a web service from another computer, the web service needs to be available. This means you can't rely on just the built-in Visual Studio web server (which dynamically chooses a new port each time you run it). Instead, you need to create a virtual directory for your web service (as described in Chapter 18). Once you've taken this step, you should try requesting the web service in your browser using the virtual directory name to make sure it's accessible. You can then add a reference to the web service by following the steps in this section.

Figure 31-7 represents this process graphically. In this example, a browser is running an ASP.NET web page, which is using a web service from another server behind the scenes. The ASP.NET web page uses the proxy class to contact this external web service.

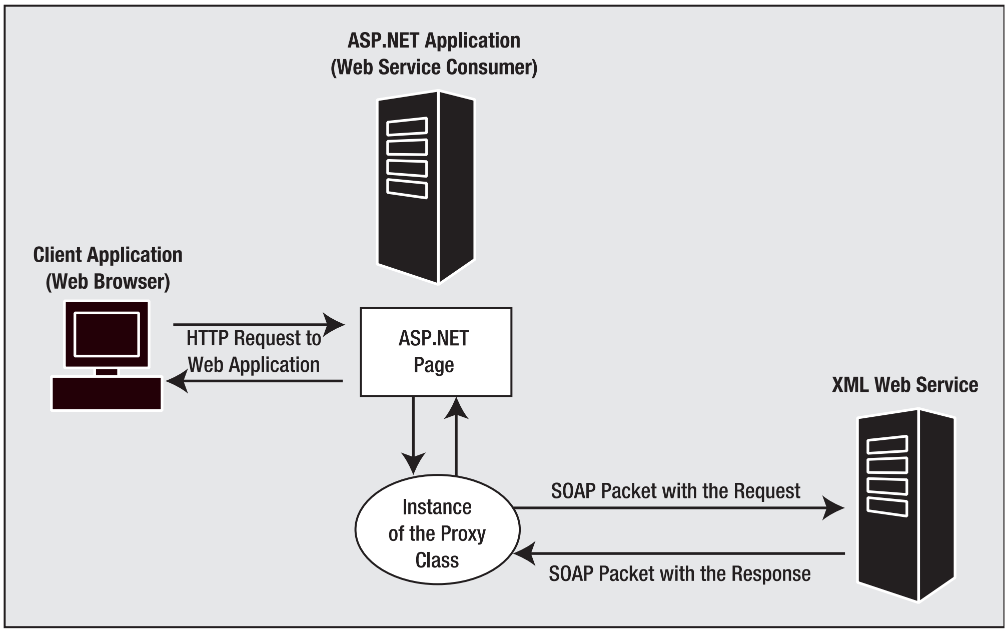


Figure 31-7. *The web service proxy class*

■Note Thanks to the proxy class, you can call a method in a web service as easily as you call a method in a local component. Of course, this behavior isn't always a benefit. Web services have different characteristics than local components. For example, it takes a nontrivial amount of time to call a web method, because every call needs to be converted to XML and sent over the network. The danger is that the more this reality is hidden from developers, the less likely they are to take it into account and design their applications accordingly.

You can create a proxy class in .NET in two ways:

- You can use the wsdl.exe command-line tool.
- You can use the Visual Studio web reference feature.

Both of these approaches produce essentially the same result, because they use the same classes in the .NET Framework to perform the actual work. In fact, you can even harness these classes (which are found in the System.Web.Services namespaces) to generate your own proxy classes programmatically, although this approach isn't terribly practical.

In the following sections, you'll learn how to use wsdl.exe and Visual Studio to create proxy classes. You'll learn how to consume a web service in three types of clients—an ASP.NET web page, a Windows application, and a classic ASP page.

■Tip One difference between the wsdl.exe approach and the web reference feature is that if you use the web reference feature in a web application, you won't be able to actually see the proxy code (because it's generated later in the compilation process). This means if you want to tweak the proxy class code or just peek under the hood, and you're creating a web client, you need to use wsdl.exe. This limitation doesn't apply to other types of clients. They don't use the ASP.NET compilation model, so the proxy class code is added directly to the project.

Generating the Proxy Class with wsdl.exe

The wsdl.exe tool takes a web service and generates the source code of the proxy class in either VB .NET or C#. The name WSDL stems from the web service standard (Web Services Description Language) that's used to describe the functionality provided by a web service. You'll learn more about WSDL in the next chapter.

You can find the wsdl.exe file in the .NET Framework directory, which is typically in a path similar to c:\Program Files\Microsoft Visual Studio 2005\SDK\v2.0\Bin (depending on the version of Visual Studio you have installed). This file is a command-line utility, so it's easiest to use by opening a command prompt window.

In ASP.NET, you can request a WSDL document by specifying the URL of the web service plus the ?WSDL parameter. (Alternatively, you can use a different URL or even a file containing the WSDL content.) The minimum syntax to generate the class is the following:

```
wsdl http://localhost/WebServices1/EmployeesService.asmx
```

By default the generated class is in the C# language, but you can change it by adding the /language parameter, as follows:

```
wsdl /language:VB http://localhost/WebServices1/EmployeesService.asmx
```

By default the generated file also has the same name as the web service (specified in the Name property of the WebService attribute). You can change it by adding a /out parameter to the wsdl.exe command, and you can use a /namespace parameter to change the namespace for the generated class. Here's an example (split over two lines to fit the book's page margins):

```
wsdl /namespace:ApressServices /out:EmployeesProxy.cs
  http://localhost/WebServices1/EmployeesService.asmx
```

Table 31-5 lists the supported parameters.

Table 31-5. *Wsdl.exe Parameters*

Parameter	Description
<url or path>	A URL or path to a WSDL contract, an XSD schema, or a .discomap document.
/nologo	Suppresses the banner.
/language:<language>	The language to use for the generated proxy class. Choose from CS, VB, or JS, or provide a fully qualified name for a class implementing System.CodeDom.Compiler.CodeDomProvider. The default is C#. Short form is /l.
/server	Generate an abstract class for a web service implementation based on the contracts. The default is to generate client proxy classes.
/namespace:<namespace>	The namespace for the generated proxy or template. The default namespace is the global namespace. Short form is /n.
/out:<fileName>	The filename for the generated proxy code. The default name is derived from the service name. Short form is /o.
/protocol:<protocol>	Override the default protocol to implement. Choose from SOAP (for SOAP 1.1), SOAP12 (for SOAP 1.2), HTTP-GET, HTTP-POST, or a custom protocol as specified in the configuration file.
/username:<username> /password:<password> /domain:<domain>	The credentials to use when connecting to a server that requires authentication. Short forms are /u, /p, and /d.
/proxy:<URL>	The URL of the proxy server to use for HTTP requests. The default is to use the system proxy setting.
/proxyusername:<username> /proxypassword:<password> /proxydomain:<domain>	The credentials to use when connecting to a proxy server that requires authentication. Short forms are /pu, /pp, and /pd.
/appsettingurkey:<key>	The configuration key to use in the code generation to read the default value for the URL property. The default is to not read from the config file. Short form is /urlkey.
/appsettingbaseurl:<baseURL>	The base URL to use when calculating the URL fragment. The appsettingurlkey option must also be specified. The URL fragment is the result of calculating the relative URL from the appsettingbaseurl to the URL in the WSDL document. Short form is /baseurl.
/fields	If set, any complex types used by the web service will consist of public fields instead of public properties. Chapter 32 discusses how complex types work with web services in much more detail.

Parameter	Description
/sharetypes	Allows you to add a reference to two or more web services that use the same complex types. This technique is described in the next chapter.
/serverinterface	Generates an interface with just the methods of the WSDL document. You can implement this interface to create your web service. This technique is described in the next chapter.

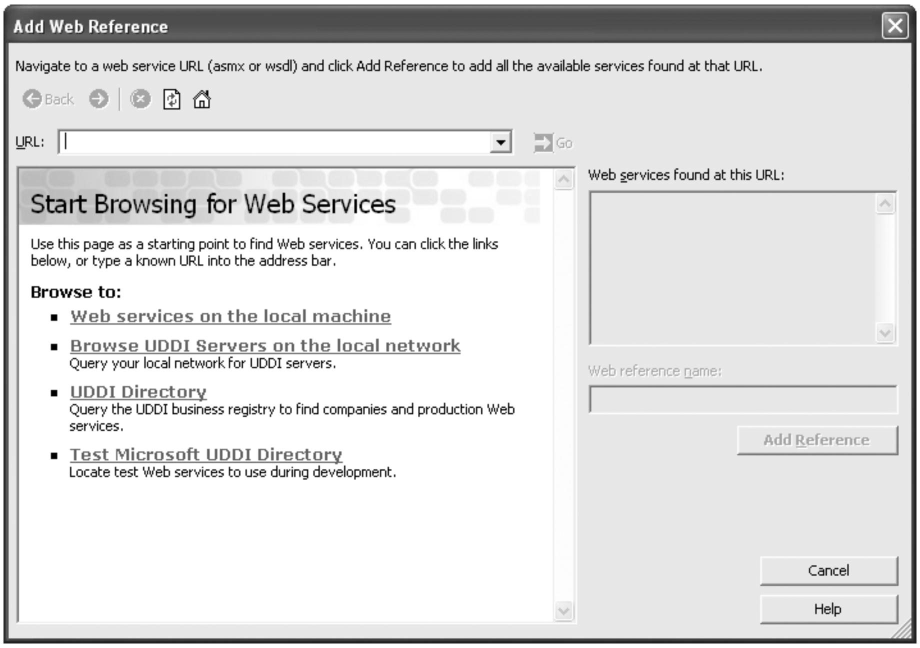
Once you've created this file, you need to copy it to the App_Code directory so that the class is available to the pages in your web application. If you're creating a rich client application (such as a Windows Forms application), you would instead add this file directly to the project so it is compiled into the final EXE.

Generating the Proxy Class with Visual Studio

In Visual Studio, you create the proxy class by adding a web reference in the client project. Web references are similar to ordinary references, but instead of pointing to assemblies with ordinary .NET types, they point to a web service URL with a WSDL contract.

To create a web reference, follow these steps:

1. Right-click the client project in the Solution Explorer, and select Add Web Reference.
2. The Add Web Reference dialog box opens, as shown in Figure 31-8. This window provides options for searching web registries or entering a URL directly. It also has a link that allows you to browse all the web services on the local computer or search a UDDI registry.



Navigate to a web service URL (asmx or wsdl) and click Add Reference to add all the available services found at that URL.

Start Browsing for Web Services

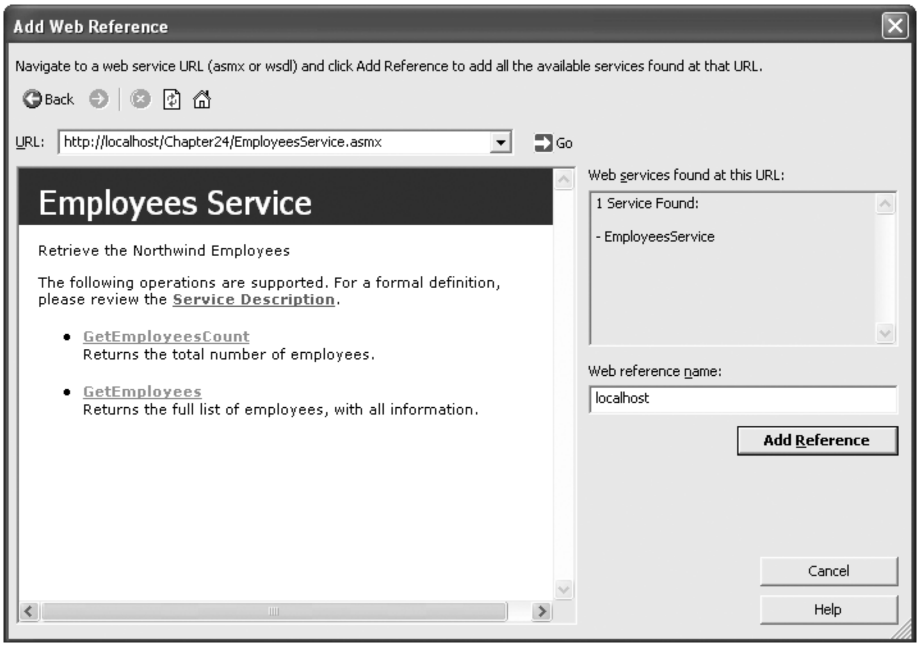
Use this page as a starting point to find Web services. You can click the links below, or type a known URL into the address bar.

Browse to:

- Web services on the local machine
- Browse UDDI Servers on the local network
  Query your local network for UDDI servers.
- UDDI Directory
  Query the UDDI business registry to find companies and production Web services.
- Test Microsoft UDDI Directory
  Locate test Web services to use during development.

Figure 31-8. *The Add Web Reference dialog box*

3. You can browse directly to your web service by entering a URL that points to the .asmx file. The test page will appear in the window (as shown in Figure 31-9), and the Add Reference button will be enabled.



Navigate to a web service URL (asmx or wsdl) and click Add Reference to add all the available services found at that URL.

URL: http://localhost/Chapter24/EmployeesService.asmx

Employees Service

Retrieve the Northwind Employees

The following operations are supported. For a formal definition, please review the Service Description.

- GetEmployeesCount
  Returns the total number of employees.
- GetEmployees
  Returns the full list of employees, with all information.

Web services found at this URL:
1 Service Found:
- EmployeesService

Web reference name: localhost

Figure 31-9. *Adding a web reference*

4. In the Web Reference Name text box you can change the namespace in which the proxy class will be generated.
5. To add the reference to this web service, click Add Reference at the bottom of the window.
6. Now the web reference will appear in the Web References group for your project in the Solution Explorer window.

The web reference you create uses the WSDL contract and information that exists at the time you add the reference. If the web service is changed, you'll need to update your proxy class by right-clicking the web reference and choosing Update Web Reference. Unlike local components, web references aren't updated automatically when you recompile the application.

■ **Tip** When developing and testing a web service in Visual Studio, it's often easiest to add both the web service and the client application to the same solution. This allows you to test and change both pieces at the same time. You can even use the integrated debugger to set breakpoints and step through the code in both the client and the server, as though they were really a single application. To choose which application you want Visual Studio to launch when you click Start, right-click the appropriate project name in the Solution Explorer and select Set As StartUp Project.

When you add a web reference, Visual Studio saves a copy of the WSDL document in your project. Where it stores this information depends on the type of project.

In a web application, Visual Studio creates the App_WebReferences folder (if it doesn't already exist) and then creates a folder inside it with the web reference name (which you chose in the Add Web Reference dialog box). Finally, Visual Studio places the web service files in that folder. However, Visual Studio *doesn't* generate the proxy class. Instead, it's built and cached as part of the ASP.NET compilation process. This is a change in the behavior from ASP.NET 1.*x*.

In any other type of application, Visual Studio creates a WebReferences folder and then creates a folder inside it with the web reference name. Inside that folder it places all the support files you see in a web application (most important is a copy of the WSDL document). It also creates a file named Reference.cs (assuming it's a C# application) with the proxy class source code, as shown in Figure 31-10. By default, this class is hidden from view. To see it, select Project ➤ Show All Files.

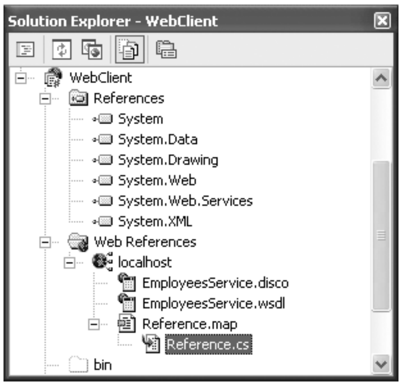


Figure 31-10. *The WSDL contract and proxy class*

Dynamic URLs

In previous versions of .NET, the web service URL was (by default) hard-coded in the class constructor. However, when you create a web reference with Visual Studio 2005, the location is always stored in a configuration file. This is useful, because it allows you to change the location of the web service when you deploy the application, without forcing you to regenerate the proxy class.

The exact location of this setting depends on the type of application. If the client is a web application, this information will be added to the web.config file, as shown here:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <appSettings>
    <add key="localhost.EmployeesService"
     value="http://localhost/WebServices1/EmployeesService.asmx"/>
  </appSettings>
  ...
</configuration>
```

If you're creating a different type of client application, such as a Windows application, the configuration file will have a name in the format [AppName].exe.config. For example, if your application is named SimpleClient.exe, the configuration file will be SimpleClient.exe.config. You must follow this naming convention.

■ **Tip** Visual Studio uses a little sleight of hand with named configuration files. In the design environment, the configuration file will have the name App.config. However, when you build the application, this file will be copied to the build directory and given the appropriate name (to match the executable file). The only exception is if the client application is a web application. All web applications use a configuration file named web.config, no matter what filenames you use. That's what you'll see in the design environment as well.

If you want to control the name of the setting, you need to use the wsdl.exe utility with /appsettingurlkey. For example, you could use this command line:

```
wsdl http://localhost/WebServices1/EmployeesService.asmx /appsettingurlkey:WsUrl
```

In this case, the key is stored with the key WsUrl in the <appSettings> section.

The Proxy Class

Once you create the proxy class, it's worth taking a closer look at the generated code to see how it works.

The proxy class has the same name as the web service class. It inherits from SoapHttpClient-Protocol, which has properties such as Credentials, Url, and Timeout, which you'll learn about in the following sections. Here's the declaration for the proxy class that provides communication with the EmployeesService:

```
public class EmployeesService :
 System.Web.Services.Protocols.SoapHttpClientProtocol
{ ... }
```

The proxy class contains a copy of each method in the web service. However, the version in the proxy class doesn't contain the business code. (In fact, the client has no way to get any information about the internal workings of your web service code—if it could, this would constitute a serious security breach.) Instead, the proxy class contains the code needed to query the remote web service and convert the results. For example, here's the GetEmployeesCount() method in the proxy class:

```
[System.Web.Services.Protocols.SoapDocumentMethodAttribute()]
public int GetEmployeesCount()
{
    object[] results = this.Invoke("GetEmployeesCount", new object[0]);
    return ((int)(results[0]));
}
```

This method calls the base SoapHttpClientProcotol.Invoke() to actually create the SOAP message and start waiting for the response. The second line of code converts the returned object into an integer.

■ **Note** The proxy also has other methods that support asynchronous calls to the web methods. You'll learn more about asynchronous calls and see practical examples of how to use them in Chapter 33.

The proxy class concludes with the proxy code for the GetEmployees() method. You'll notice that this code is nearly identical to the code used for the GetEmployeesCount() method—the only differences are the method name that's passed to the Invoke() method and that the return value is converted to a DataSet rather than an integer.

```
[System.Web.Services.Protocols.SoapDocumentMethodAttribute()]
public System.Data.DataSet GetEmployees()
{
    object[] results = this.Invoke("GetEmployees", new object[0]);
    return ((System.Data.DataSet)(results[0]));
}
```

Creating an ASP.NET Client

Now that you have a web service and proxy class, it's quite easy to develop a simple web-page client. If you're using Visual Studio, the first step is to create a new web project and add a web reference to the web service. If you're using another tool, you'll need to compile a proxy class first using wsdl.exe and then place it in the new web application Bin directory.

The following example uses a simple web page with a button and a GridView control. When the user clicks the button, the web page posts back, creates the proxy class, retrieves the DataSet of employees from the web service, and then displays the result by binding it to the grid.

Before you add this code, it helps to import the proxy class namespace. In Visual Studio, the namespace is automatically the namespace of the current project, plus the namespace you specified in the Add Web Reference dialog box (which is localhost by default). Assuming your project is named WebClient, the web service is on the local computer, and you didn't make any changes in the Add Web Reference dialog box, you'll use this namespace:

```
using WebClient.localhost;
```

Now you can add the code that uses the proxy class to retrieve the data:

```
private void cmdGetData_Click(object sender, System.EventArgs e)
{
    // Create the proxy.
    EmployeesService proxy = new EmployeesService();

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

    // Bind the results.
    GridView1.DataSource = ds.Tables[0];
    GridView1.DataBind();
}
```

Because the proxy class has the same name as the web service class, when the client instantiates the proxy class it seems as though the client is actually instantiating the web service. To help emphasize the difference, this code names the object variable proxy.

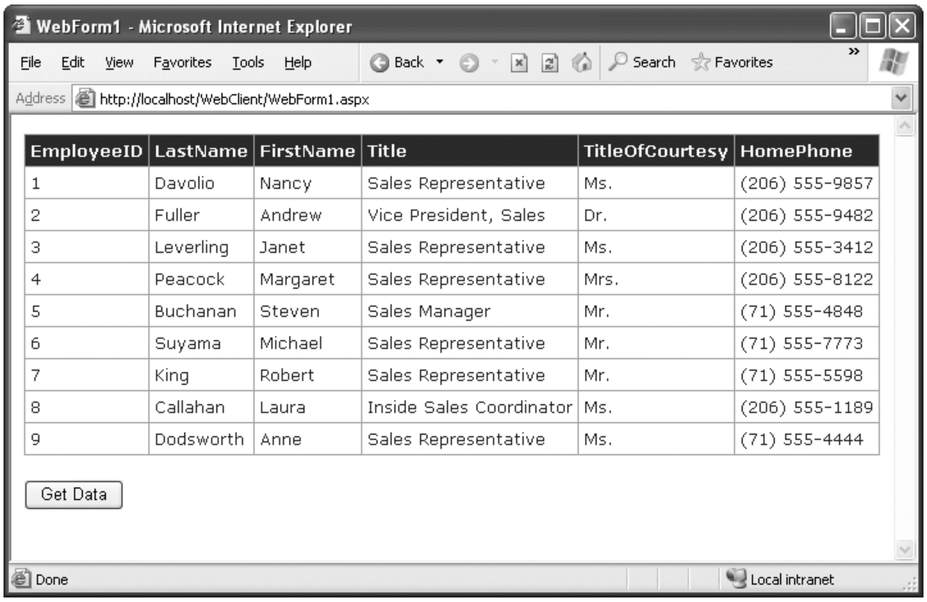
If you run the page, you'll see the page shown in Figure 31-11.

Interestingly, you don't need to perform the data binding manually. You can use the ObjectDataSource (described in Chapter 9) to bind directly to the corresponding proxy class, no code required:

```
<asp:ObjectDataSource ID="ObjectDataSource1" runat="server"
 SelectMethod="GetEmployees" TypeName="localhost.EmployeesService" />
<asp:GridView ID="GridView1" runat="server" AutoGenerateColumns="True"
 DataSourceID="ObjectDataSource1"/>
```

From the point of view of your web-page code, there's no difference between calling a web service and using an ordinary stateless class. However, you must remember that the web service that actually implements the business logic could be on a web server on the other side of the world. As a result, you need to reduce the number of times you call it and be prepared to handle exceptions resulting from network problems and connectivity errors.

7680CH31.qxd 8/23/06 4:30 PM Page 1186


WebForm1 - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://localhost/WebClient/WebForm1.aspx
Get Data
Done
Local intranet

EmployeeID	LastName	FirstName	Title	TitleOfCourtesy	HomePhone
1	Davolio	Nancy	Sales Representative	Ms.	(206) 555-9857
2	Fuller	Andrew	Vice President, Sales	Dr.	(206) 555-9482
3	Leverling	Janet	Sales Representative	Ms.	(206) 555-3412
4	Peacock	Margaret	Sales Representative	Mrs.	(206) 555-8122
5	Buchanan	Steven	Sales Manager	Mr.	(71) 555-4848
6	Suyama	Michael	Sales Representative	Mr.	(71) 555-7773
7	King	Robert	Sales Representative	Mr.	(71) 555-5598
8	Callahan	Laura	Inside Sales Coordinator	Ms.	(206) 555-1189
9	Dodsworth	Anne	Sales Representative	Ms.	(71) 555-4444

Figure 31-11. *Displaying data from a web service in a web page*

Timeouts

The proxy class includes a Timeout property that allows you to specify the maximum amount of time you're willing to wait, in milliseconds. By default, the timeout is 100,000 milliseconds (10 seconds).

When using the Timeout property, you need to include error handling. If the Timeout period expires without a response, an exception will be thrown, giving you the chance to notify the user about the problem.

Here's how you could rewrite the ASP.NET web-page client to use a timeout of three seconds:

```
private void cmdGetData_Click(object sender, System.EventArgs e)
{
    // Create the proxy.
    EmployeesService proxy = new EmployeesService();

    // This timeout will apply to all web service method calls.
    proxy.Timeout = 3000;  // 3,000 milliseconds is 3 seconds.

    DataSet ds = null;
    try
    {
        // Call the web service and get the results.
        ds = proxy.GetEmployees();
    }
    catch (System.Net.WebException err)
    {
        if (err.Status == WebExceptionStatus.Timeout)
        {
            lblResult.Text = "Web service timed out after 3 seconds.";
        }
        else
        {
            lblResult.Text = "Another type of problem occurred.";
        }
    }

    // Bind the results.
    if (ds != null)
    {
        GridView1.DataSource = ds.Tables[0];
        GridView1.DataBind();
    }
}
```

You can also set the timeout to -1 to indicate that you'll wait as long as it takes. However, this will make your web application unacceptably slow if you attempt to perform a number of operations with an unresponsive web service.

Connecting Through a Proxy

The proxy class also has some built-in intelligence that allows you to reroute its HTTP communication with special Internet settings. By default the proxy class uses the Internet settings on the current computer. In some networks, this may not be the best approach. You can override these settings by using the Proxy property of the web service proxy class.

■ **Tip** In this case, the word *proxy* is being used in two ways: as a proxy that manages communication between a client and a web service and as a proxy server in your organization that manages communication between a computer and the Internet.

For example, if you need to connect through a computer called ProxyServer using port 80, you could use the following code before you called any web service methods:

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

// Specify a proxy server for network communication.
WebProxy connectionProxy = new WebProxy("ProxyServer", 80);
proxy.Proxy = connectionProxy;
```

The WebProxy class has many other options that allow you to configure connections and set authentication information in more complicated scenarios.

Creating a Windows Forms Client

One of the main advantages of web services is the way they allow you to web enable local applications, such as rich client applications. Using a web service, you can create a desktop application that gets up-to-the-minute data from a web server. The process is almost entirely transparent. In fact, as high-speed access becomes more common, you may not even be aware of which portions of functionality depend on the Internet and which ones don't.

You can use web service functionality in a Windows application in the same way you would use it in an ASP.NET application. First, you create the proxy class using Visual Studio or the wsdl.exe utility. Next, add code to create an instance of the proxy class and call a web method. The only difference is the user interface the application uses.

If you haven't explored desktop programming with .NET yet, you'll be happy to know that you can reuse much of what you've learned in ASP.NET development. Many web controls (such as labels, buttons, text boxes, and lists) closely parallel their .NET desktop equivalents, and the code you write to interact with them can often be transferred from one environment to the other with few changes. In fact, the most significant difference between desktop programming and web programming in .NET is the extra steps you need to take in web applications to preserve information between postbacks and when transferring the user from one page to another.

To begin creating your Windows client in Visual Studio, create a new Windows application project, and then add the web reference. Web projects start with a single startup form, which you can design in much the same way as you design a web page. For this example, you simply need to drag and drop a Button and a DataGridView control from the Toolbox.

Once again, begin by importing the namespace you need at the top of the form class file, as you did in the ASP.NET page:

```
using WindowsClient.localhost;
```

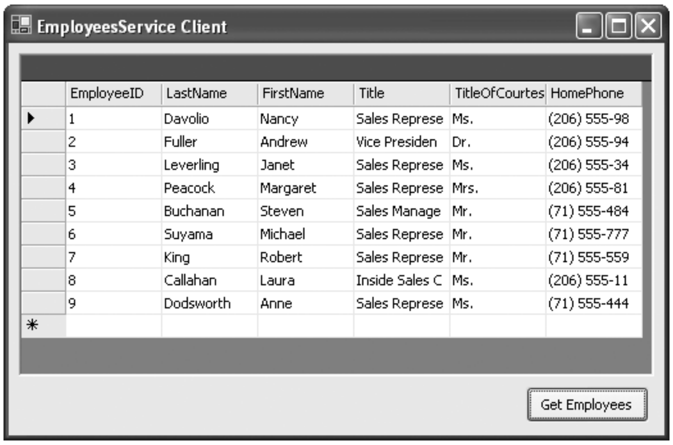
Next, add the event-handling code for the button. This code retrieves the DataSet and displays it in the form. Data binding works slightly differently in a Windows application; for example, you don't have to call an explicit DataBind() method after you set the data source. This code also introduces one refinement—it explicitly sets the application to use an hourglass cursor while the web service call is underway so the user knows that the operation is in progress. Other than that, the code is identical:

```
private void cmdGetData_Click(object sender, System.EventArgs e)
{
    this.Cursor = Cursors.WaitCursor;

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

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

    // Bind the results.
    dataGridView1.DataSource = ds.Tables[0];

    this.Cursor = Cursors.Default;
}
```

Figure 31-12 shows what you'll see when you run the Windows client and click the button to retrieve the web service data.

Of course, Windows development contains many other possibilities, which are covered in many other excellent books. The interesting part from your vantage point is the way that a Windows client can interact with a web service just like an ASP.NET application does. This raises a world of new possibilities for integrated Windows and web applications. For example, you could extend this Windows application so that it allows the user to modify the employee data. You could then add methods to the EmployeesService that allow the client to submit the changed data and commit the changes to the back-end database.

It's important to understand that what you do to consume your sample web service is exactly what you would do to consume any other third-party web service. Web service providers don't need to distribute their proxy classes, because programming platforms such as .NET include the tools to generate them automatically.



	EmployeeID	LastName	FirstName	Title	TitleOfCourtes	HomePhone
▶	1	Davolio	Nancy	Sales Represe	Ms.	(206) 555-98
	2	Fuller	Andrew	Vice Presiden	Dr.	(206) 555-94
	3	Leverling	Janet	Sales Represe	Ms.	(206) 555-34
	4	Peacock	Margaret	Sales Represe	Mrs.	(206) 555-81
	5	Buchanan	Steven	Sales Manage	Mr.	(71) 555-484
	6	Suyama	Michael	Sales Represe	Mr.	(71) 555-777
	7	King	Robert	Sales Represe	Mr.	(71) 555-559
	8	Callahan	Laura	Inside Sales C	Ms.	(206) 555-11
	9	Dodsworth	Anne	Sales Represe	Ms.	(71) 555-444
*						

Figure 31-12. *Displaying data from a web service in a Windows form*

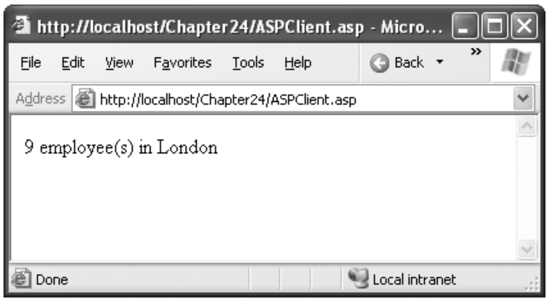
■ **Tip** If you'd like to try consuming some non-.NET web services, you can search the web service catalog at XMethods (`http://www.xmethods.com`). Or, for more practice with genuinely useful web services, Microsoft's MapPoint (`http://msdn.microsoft.com/library/en-us/dnanchor/html/anch_mappointmain.asp`) is an interesting example that enables you to access high-quality maps and geographical information. There's also Microsoft's TerraService (`http://terraservice.net/webservices.aspx`), which is based on the hugely popular TerraServer site where web surfers can view topographic maps and satellite photographs of the globe. Using TerraService you can query information about different locations on the globe and even download tiles with satellite photography of specific regions.

Creating an ASP Client with MSXML

It's also interesting to demonstrate how a web service can be called by a legacy application of any type and platform. The following example shows a bare-bones approach to displaying data in a legacy ASP page:

```
<script language="VBScript" runat="Server">
Option Explicit

Dim URL
URL = "http://localhost/WebServices1/EmployeesService.asmx/GetEmployeesCount"
Dim objHTTP
Set objHTTP = CreateObject("Microsoft.XMLHTTP")

' Send an HTTP_POST command to the URL.
objHTTP.Open "POST", URL, False
objHTTP.Send

' Read and display the value of the root node.
Dim numEmp
numEmp = objHTTP.responseXML.documentElement.Text
Response.Write(numEmp & " employee(s) in London")
</script>
```

This code simply sets the URL to point to the web method in the web service. It then uses the Microsoft.XMLHTTP class (from the Microsoft XML Parser, a COM component that provides classes to manipulate XML data, send HTTP commands and receive the respective responses) to open an HTTP connection and to send the command in a synchronous manner. In this case, the code is accessing the service through an HTTP POST command, which ASP.NET web services support only on the local computer. When the send method returns, the response text is saved in the respon-seXML property. It's provided as an MSXML2.DOMDocument object with a documentElement property that points to the root node of the returned XML data. Using this object, you can navigate the XML of the response. In this case, because the data simply contains an integer result, you can use the text property to read the value of that element. Figure 31-13 shows the result.


http://localhost/Chapter24/ASPClient.asp - Micro...
File Edit View Favorites Tools Help
Back
Address http://localhost/Chapter24/ASPClient.asp
Done
Local intranet

9 employee(s) in London

Figure 31-13. *Displaying data from a web service in an ASP page*

The interesting part of this example is that it uses the Microsoft XML library, which has classes to send commands via HTTP, receive the response text, and parse XML. That's all you need. If you don't already have this common component, you can download it from `http://msdn.microsoft.com/library/en-us/xmlsdk/html/xmmscxmlinstallregister.asp`. Note that you can use the previous code, with minor modifications, in any VBScript or VBA application, including a Microsoft Office macro or a WSH (Windows Scripting Host) client.

Now consider a more complex example—the GetEmployees() web method that returns a complete DataSet. To interact with this data, you need to dig through the XML response. Here's the code that loops through the <Employees> tags and extracts several pieces of information about each employee to create a list:

```
<script language="VBScript" runat="Server">
Option Explicit

Dim URL
URL = "http://localhost/WebServices1/EmployeesService.asmx/GetEmployeesCount"
Dim objHTTP
Set objHTTP = CreateObject("Microsoft.XMLHTTP")

' Send an HTTP_POST command to the URL.
objHTTP.Open "POST", URL, False
objHTTP.Send

' Retrieve the XML response.
Dim Doc
Set Doc = objHTTP.responseXML
```

```
' Dig into the XML DataSet structure.
' Skip down one node to get past the schema. Then go one level deeper
' to the root DataSet element.
' Finally, loop through the contained tags, each of which
' represents an employee.
Dim Child
For Each Child In Doc.documentElement.childNodes(1).childNodes(0).childNodes
    ' The first node is the ID.
    Response.Write(Child.childNodes(0).Text + "<br>")
    ' The second node is the first name.
    Response.Write(Child.childNodes(1).Text)
    ' The third node is the last name.
    Response.Write(Child.childNodes(2).Text + "<br><br>")
Next
</script>
```

Creating an ASP Client with the SOAP Toolkit

The previous example showed the lowest common denominator for web service invocation—posting a message over HTTP and parsing the returned XML by hand. Most clients have access to more robust toolkits that directly support SOAP. One example is the Microsoft SOAP Toolkit, which is a COM component that you can use to call any type of web service that provides a valid WSDL document. Thus, the SOAP Toolkit supports .NET web services and web services created on other platforms. You can use the SOAP Toolkit to use web services in COM-based applications like those created in Visual Basic 6, Visual C++ 6, and ASP. To download the latest version of the Microsoft SOAP Toolkit, refer to `http://msdn.microsoft.com/webservices/_building/soaptk`.

To use the Microsoft SOAP Toolkit, you need to know the location of the WSDL for the service you want to use. Using this WSDL document, the SOAP Toolkit will dynamically generate a proxy. You can't see the proxy (because it's created at runtime), but you can use it to have access the same higher-level web services model.

The following example rewrites the ASP page to use the SOAP Toolkit. In this case, the WSDL document is retrieved directly from the web server. To improve performance, it's recommended that you save a local copy of the WSDL file and use that to configure the SoapClient object.

```
<script language="VBScript" runat="Server">
Option Explicit

Dim SoapClient
Set SoapClient = CreateObject("MSSOAP.SoapClient"

' Generate a proxy.
Dim WSDLPath
WSDLPath = "http://localhost/WebServices1/EmployeesService.asmx?WSDL"
SoapClient.MSSoapInit WSDLPath

' Read the number of employees.
Dim numEmp
numEmp = SoapClient.GetEmployeesCount()
Response.Write(numEmp & " employee(s) in London")
</script>
```

Notice that in this example, you have no need to read any XML. Instead, the client calls the GetEmployeesCount() method directly using the SoapClient object. However, this approach still won't help you manipulate a DataSet, because a COM equivalent for this class doesn't exist. Instead, you'll need to fall back on parsing XML, as shown in the previous example.

Of course, Java, C++, Delphi, and so on, have their own components, libraries, and APIs to establish HTTP connections, send commands, and parse XML text, but the essential approach is the same.

Refining a Web Service

So far you've seen how to create a basic web service with a couple of methods and create a web page and Windows application that use it. However, we haven't touched on a number of web service features yet. For example, a web method can cache data, use session state, and perform transactions. In the remainder of this chapter, you'll look at how you can use these techniques.

The secret to applying these features is the WebMethod attribute. So far the examples you've seen have used the WebMethod attribute to mark the methods you want to expose as part of a web service and to attach a description (using the Description property). However, several additional WebMethod properties exist, as described in Table 31-6.

Table 31-6. *Properties of the WebMethod Attribute*

Argument	Description
Description	The method's description.
MessageName	The alias name for the method, which is used if you have overloaded versions of the method or if you want to expose the method with a different name. This technique is deprecated.
CacheDuration	This is the number of seconds that the method's response will be maintained in cache. The default is zero, meaning it's not cached.
EnableSession	Gets or sets whether the method can access information in the Session collection.
BufferResponse	Gets or sets whether the method's response is buffered. It is true by default, and it should be set to false only if you know that the request will take long to complete and you want to send sections of data earlier.
TransactionOption	Gets or sets whether the method supports transaction and of what type. Allowed values are disabled, NotSupported, Supported, Required, and RequiresNew. Because of the stateless nature of web services, they can participate only as the root object of a transaction.

CacheDuration

As you learned in Chapter 11, ASP.NET has built-in support for two types of caching: output caching and data caching. Web services can use both these forms of caching, as you'll see in the following sections.

Output Caching

The simplest kind of web service caching is output caching. Output caching works with web services in the same way it does with web pages: identical requests (in this case, requests for the same method and with the same parameters) will receive identical responses from the cache, until the cached information expires. This can greatly increase performance in heavily trafficked sites, even if you store a response only for a few seconds.

You should use output caching only for straightforward information retrieval or data-processing functions. You should not use it in a method that needs to perform other work, such as changing session items, logging usage, or modifying a database. This is because subsequent calls to a cached method will receive the cached result, and the web method code will not be executed.

To enable caching for a function, you use the CacheDuration property of the WebMethod attribute. Here's an example with the GetEmployees() method:

```
[WebMethod(CacheDuration=30)]
public DataSet GetEmployees()
{ ... }
```

This example caches the employee DataSet for 30 seconds. Any user who calls the GetProducts() method in this time span will receive the same DataSet, directly from the ASP.NET output cache.

Output caching becomes more interesting when you consider how it works with methods that require parameters. Here's an example that caches a GetEmployeesByCity() web method for ten minutes:

```
[WebMethod(CacheDuration=600)]
public DataSet GetEmployeesByCity(string city)
{ ... }
```

In this case, ASP.NET is a little more intelligent. It reuses only those requests that supply the same city value. For example, here's how three web service requests might unfold:

1. A client calls GetEmployeesByCity() with the city parameter of London. The web method calls, contacts the database, and stores the result in the web service cache.
2. A client calls GetEmployeesByCity() with the city parameter of Kirkland. The web method calls, contacts the database, and stores the result in the web service cache. The previously cached DataSet is not reused, because the city parameter differs.
3. A client calls GetEmployeesByCity() with the city parameter of London. Assuming ten minutes haven't elapsed since the request in step 1, ASP.NET automatically reuses the first cached result. No code is executed.

Whether it makes sense to cache this version of GetEmployeesByCity() really depends on how much traffic your web service receives and how many different cities exist. If there are only a few different city values, this approach may make sense. If there are dozens and your web server memory is limited, it's guaranteed to be inefficient.

Data Caching

ASP.NET also supports data caching, which allows you to store full-fledged objects in the cache. As with all ASP.NET code, you can use data caching through the Cache object (which is available through the Context.Cache property in your web service code). This object can temporarily store information that is expensive to create so that the web method can reuse it for other calls by other clients. In fact, the data can even be reused in other web services or web pages in the same application.

Data caching makes a lot of sense in the version of the EmployeesService that provides two GetEmployees() methods, one of which accepts a city parameter. To ensure optimum performance but cut down on the amount of data in the cache, you can store a single object in the cache: the full employee DataSet. Then, when a client calls the version of GetEmployees() that requires a city parameter, you simply need to filter out the rows for the city the client requested.

The following code shows this pattern at work. The first step is to create a private method that uses the cache, called GetEmployeesDataSet(). If the DataSet is available in the cache, GetEmployeesDataSet() uses that version and bypasses the database. Otherwise, it creates a new DataSet and fills it with the full set of employee records. Here's the complete code:

```
private DataSet GetEmployeesDataSet()
{
    DataSet ds;

    if (Context.Cache["EmployeesDataSet"] != null)
    {
        // Retrieve it from the cache
        ds = (DataSet)Context.Cache["EmployeesDataSet"];
    }
    else
    {
        // Retrieve it from the database.
        string sql = "SELECT EmployeeID, LastName, FirstName, Title, " +
         "TitleOfCourtesy, HomePhone, City FROM Employees";
        SqlConnection con = new SqlConnection(connectionString);
        SqlDataAdapter da = new SqlDataAdapter(sql, con);
        ds = new DataSet();
        da.Fill(ds, "Employees");

        // Track when the DataSet was created. You can
        // retrieve this information in your client to test
        // that caching is working.
        ds.ExtendedProperties.Add("CreatedDate", DateTime.Now);

        // Store it in the cache for ten minutes.
        Context.Cache.Insert("EmployeesDataSet", ds, null,
         DateTime.Now.AddMinutes(10), TimeSpan.Zero);
    }
    return ds;
}
```

Both the GetEmployees() and GetEmployeesByCity() methods can use the private GetEmployees-DataSet() method. The difference is that GetEmployeesByCity() loops through the records and manually removes each record that doesn't match the supplied city name. Here are both versions:

```
[WebMethod(Description="Returns the full list of employees.")]
public DataSet GetEmployees()
{
    return GetEmployeesDataSet();
}

[WebMethod(Description="Returns the full list of employees by city.")]
public DataSet GetEmployeesByCity(string city)
{
    // Copy the DataSet.
    DataSet dsFiltered = GetEmployeesDataSet().Copy();

    // Remove the rows manually.
    // This is a good approach (rather than using the
    // DataTable.Select() method) because it is impervious
    // to SQL injection attacks.
    foreach (DataRow row in dsFiltered.Tables[0].Rows)
    {
        // Perform a case-insensitive compare.
        if (String.Compare(row["City"].ToString(), city.ToUpper(), true) != 0)
        {
            row.Delete();
        }
    }

    // Remove these rows permanently.
    dsFiltered.AcceptChanges();

    return dsFiltered;
}
```

Generally, you should determine the amount of time to cache information depending on how long the underlying data will remain valid. For example, if a stock quote were being retrieved, you would use a much smaller number of seconds than you might for a weather forecast. If you were storing a piece of information that seldom changes, such as the results of a yearly census poll, your considerations would be entirely different. In this case, the information is almost permanent, but the amount of returned information will be larger than the capacity of ASP.NET's output cache. Your goal in this situation would be to limit the cache duration enough to ensure that only the most popular requests are stored.

Of course, you should also base caching decisions on how long it will take to re-create the information and how many clients will be using the web service. You may need to perform substantial real-world testing and tuning to achieve perfection. For more information on data caching, refer to Chapter 11.

■**Tip** The data cache is global to an entire application (on a single web server). That means you can store information in the cache in a web service and retrieve it in a web page in the same web application, and vice versa.

EnableSession

The best practice for ASP.NET web services is to disable session state. In fact, by default, web services do not support session state. Most web services should be designed to be stateless in order to achieve high scalability. Sometimes, however, you might decide to use state management to retain user-specific information or optimize performance in a specialized scenario. In this case, you need to use the EnableSession property, as shown here:

```
[WebMethod(EnableSession=true)]
public DataSet StatefulMethod()
{ ... }
```

What happens when you have a web service that enables session state management for some methods but disables it for others? Essentially, disabling session management just tells ASP.NET to ignore any in-memory session information and withhold the Session collection from the current procedure. It doesn't cause existing information to be cleared out of the collection (that will happen only when the session times out). The only performance benefit you're receiving is from not having to look up session information when it isn't required. You don't need to take the same steps to allow your code to use Application state—this global state collection is always available.

Session state handling is not a part of the SOAP specification. As a result, you must rely on the support of the underlying infrastructure. ASP.NET relies on HTTP cookies to support session state. The session cookie stores a session ID, and ASP.NET uses the session ID to associate the client with the session state on the server. However, when you use a stateful web service, there's no guarantee that the client will support cookies. In fact, many will not. If the client doesn't support cookies, ASP.NET state management won't work, and a new session will be created with each new request. Unfortunately, your code has no way to identify this error condition.

To try session state (and observe the potential problems), you can create the simple web service shown here. It stores a single piece of personalized information (the user name) and allows you to retrieve it later.

```
public class StatefulService : System.Web.Services.WebService
{
    [WebMethod(EnableSession=true)]
    public void StoreName(string name)
    {
        Session["Name"] = name;
    }

    [WebMethod(EnableSession=true)]
    public string GetName()
    {
        if (Session["Name"] == null)
        {
            return "";
        }
        else
        {
            return (string)Session["Name"];
        }
    }
}
```

When you test the StoreName() and GetName() web methods using the ASP.NET test page, you get the expected behavior. When you call GetName(), you receive whatever string you supplied the last time you called StoreName(). That's because web browsers support cookies without a hitch.

By default, the proxy class doesn't share this ability. To see this problem in action, add a reference to the StatefulService in the Windows client. Then add a new button with the following event-handling code:

```
private void cmdTestState_Click(object sender, System.EventArgs e)
{
    // Create the proxy.
    StatefulService proxy = new StatefulService();


    // Set a name.
    proxy.StoreName("John Smith");

    // Try to retrieve the name.
    MessageBox.Show("You set: " + proxy.GetName());
}
```

Unfortunately, this code doesn't work as you might expect. When you run it, you'll see the empty string shown in Figure 31-14.

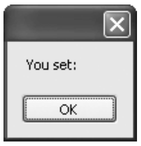

You set:

Figure 31-14. *A failed stateful service*

To resolve this problem, you need to explicitly prepare the web service proxy to accept the session cookie by creating a cookie container (an instance of the System.Net.CookieContainer class).

To correct this code, you can create the cookie container as a form-level variable, as shown here. This ensures that it lives as long as the enclosing class (the form) and can be reused in multiple methods in the same form, without losing the current web service session.

```
public partial class Form1 : System.Windows.Forms.Form
{
    private System.Net.CookieContainer cookieContainer =
      new System.Net.CookieContainer();
    ...
}
```

Now you simply need to attach this cookie container to the proxy class before you call any web method:

```
private void cmdTestState_Click(object sender, System.EventArgs e)
{
    StatefulService proxy = new StatefulService();
    proxy.CookieContainer = cookieContainer;

    proxy.StoreName("John Smith");
    MessageBox.Show("You set: " + proxy.GetName());
}
```

Now both web method calls use the same session, and the user name appears in the message box, as shown in Figure 31-15.

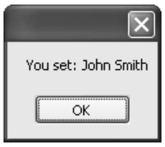

You set: John Smith

Figure 31-15. *A successful stateful service*

You need to keep the cookie container around as long as you need to keep the session cookie. For example, if your client is a web application and you want to be able to perform web service operations after every postback without losing the session, you'll need to store the session cookie in the session state of the current page. Note that the session state of the web application is different from the session state of the web service. Not only are they separate applications, but also they are also probably running on completely separate web servers.

By now, you're probably seeing a hint of the complexity of trying to use sessions with web services. Because web services are destroyed after every method call, they don't provide a natural mechanism for storing state information. You can use the Session collection to compensate for this limitation, but this approach raises the following complications:

- Session state will disappear when the session times out. The client will have no way of knowing when the session times out, which means the web service may behave unpredictably.
- Session state is tied to a specific user, not to a specific class or object. This can cause problems if the same client wants to use the same web service in two different ways or creates two instances of the proxy class at once.
- Session state is maintained only if the client preserves the session cookie. The state management you use in a web service won't work if the client fails to take these steps.

For these reasons, web services and state management don't offer a natural fit.

BufferResponse

The BufferResponse property allows you to control when the data returned from the web service is sent to the client. By default the BufferResponse property is set to true. This means that the entire result is serialized before it is sent to the client. By setting this property to false (as follows), ASP.NET will start returning output as it is serialized.

```
[WebMethod(BufferResponse=false)]
public byte[] GetLargeStreamOfData()
{ ... }
```

The web service method will always finish executing before anything is returned. The Buffer-Response setting applies to the serialization that takes place *after* the method has executed. With buffering turned off, the first part of the result is serialized and sent. Then the next part of the result is serialized and sent, and so on.

Setting BufferResponse to false makes sense only when the web service returns a large amount of data. Even then, it rarely makes any difference, because the automatically generated .NET proxy class doesn't have the ability to start processing the returned data piece by piece. This means that the proxy class will still wait for all the information to be received before it passes it back to your application. However, you can change this behavior by taking direct control over the XML message processing with the IXmlSerializable interface described in the next chapter.

TransactionOption

Web services, like any other piece of .NET code, can initiate ADO.NET transactions. Additionally, web services can easily participate in COM+ transactions. COM+ transactions are interesting because they allow you to perform a transaction that spans multiple different data sources (for example, a SQL Server database and an Oracle database). COM+ transactions also commit or rollback automatically. However, you must pay a price for these added features and convenience—because COM+ transactions use a two-stage commit protocol, they are always slower than using ADO.NET client-initiated transactions or stored procedure transactions.

The support for COM+ transactions in a web service is also somewhat limited. Because of the stateless nature of HTTP, web service methods can act only as the root object in a transaction. This means that a web service method can start a transaction and use it to perform a series of related tasks, but multiple web services cannot be grouped into one transaction. As a result, you may have to put in some extra thought when you're creating a transactional web service. For example, it won't make sense to create a financial web service with separate DebitAccount() and CreditAccount() methods, because they won't be able to be grouped into a transaction. Instead, you can make sure both tasks are executed as a single unit using a transactional TransferFunds() method.

To use a transaction in a web service, you first have to add a reference to the System.Enterprise-Services assembly. To do this in Visual Studio, right-click References in the Solution Explorer, select Add Reference, and choose System.EnterpriseServices. You should then import the corresponding namespace so that the types you need (TransactionOption and ContextUtil) are at your fingertips:

```
using System.EnterpriseServices;
```

To start a transaction in a web service method, set the TransactionOption property of the WebMethod attribute. TransactionOption is an enumeration that provides several values that allow you to specify whether a code component uses or requires transactions. Because web services must be the root of a transaction, most of these options don't apply. To create a web service method that starts a transaction automatically, use the following attribute:

```
[WebMethod(TransactionOption=TransactionOption.RequiresNew)]
public DataSet TransactionMethod()
{ ... }
```

The transaction is automatically committed when the web method completes. The transaction is rolled back if any unhandled exception occurs or if you explicitly instruct the transaction to fail using the following code:

```
ContextUtil.SetAbort();
```

Most databases support COM+ transactions. The moment you use these databases in a transactional web method, they will automatically be enlisted in the current transaction. If the transaction is rolled back, the operations you perform with these databases (such as adding, modifying, or removing records) will be automatically reversed. However, some operations (such as writing a file to disk) aren't inherently transactional. This means that these operations will not be rolled back if the transaction fails.

Now consider the following web method, which takes two actions: it deletes records in a database and then tries to read from a file. However, if the file operation fails and the exception isn't handled, the entire transaction will be rolled back, and the deleted records will be restored. Here's the transactional code:

```
[WebMethod(TransactionOption=TransactionOption.RequiresNew)]
public void UpdateDatabase()
{
    // Create ADO.NET objects.
    SqlConnection con = new SqlConnection(connectionString);
    SqlCommand cmd = new SqlCommand("DELETE * FROM Employees", con);

    // Apply the update. This will be registered as part of the transaction.
    using (con)
    {
        con.Open();
        cmd.ExecuteNonQuery();
    }

    // Try to access a file. This generates an exception that isn't handled.
    // The web method will be aborted and the changes will be rolled back.
    FileStream fs = new FileStream("does_not_exist.bin", IO.FileMode.Open);

    // (If no errors have occurred, the database changes
    // are committed here when the method ends).
}
```

Another way to handle this code is to catch the error, perform any cleanup that's required, and then explicitly roll back the transaction, if necessary:

```
[WebMethod(TransactionOption=TransactionOption.RequiresNew)]
public void UpdateDatabase()
{
    // Create ADO.NET objects.
    SqlConnection con = new SqlConnection(connectionString);
    SqlCommand cmd = new SqlCommand("DELETE * FROM Employees", con);

    // Apply the update.
    try
    {
        con.Open();
        cmd.ExecuteNonQuery();

        FileStream fs = new FileStream("does_not_exist.bin",
          IO.FileMode.Open);
    }
    catch
    {
        if (con.State != ConnectionState.Closed) con.Close();
        ContextUtil.SetAbort();
    }
    finally
    {
        con.Close();
    }
}
```

Does a web service need to use COM+ transactions? It all depends on the situation. If multiple updates are required in separate data stores, you may need to use transactions to ensure your data's integrity. If, on the other hand, you're modifying values only in a single database (such as SQL Server 2000), you can probably use the data provider's built-in transaction features instead, as described in Chapter 7.

■**Note** In the future, other emerging standards, such as XLANG and WS-Transactions, may fill in the gaps by defining a cross-platform standard that will let different web services participate in a single transaction. However, this goal is still a long way from being realized.

Summary

In this chapter, you learned what web services are and why they are important for businesses. You also took your first look at how to create and consume web services in .NET and test web services with nothing but a browser. In the next two chapters, you'll dig into the underlying standards and learn how to extend the web service infrastructure.