CHAPTER 22

Creating Web Services

Web services have the potential to dramatically simplify the way distributed applications are built. They might even lead to a new generation of applications that seamlessly integrate multiple remote services into a single web page or desktop interface. However, the greatest programming concept in the world is doomed to fail if it isn't supported by powerful, flexible tools that make its use not only possible but also convenient. Fortunately, ASP.NET doesn't disappoint. It provides classes that allow you to create a web service quickly and easily.

In the previous chapter, you looked at the philosophy that led to the creation of web services and the XML infrastructure that makes it all possible. This chapter delves into the practical side of web services, leading you through the process of creating and using a basic service.

Web Service Basics

As you've already seen in this book, ASP.NET is based on object-oriented principles. Instead of dealing with a slew of miscellaneous functions, you work with discrete classes that wrap code into neat, reusable objects. Web services follow that pattern—in fact, every web service is really an ordinary class. Even better, a client can create an instance of your web service and use its methods just as though it were any other locally defined class. The underlying HTTP transmission and data type conversion that has to happen takes place behind the scenes.

How does this magic work? It's made possible by the .NET Framework and the types in the System.Web.Services namespace.

A typical web service consists of a few basic ingredients:

An .asmx file: This is the web service end point—the URL where the client sends its request messages. The .asmx file plays the same role with web services as the .aspx file plays with web pages.

The web service class: This contains the functionality (the code) for the web service. As with web pages, you can place the web service class directly in the .asmx file or in a separate code-behind file. Typically, your web service class will inherit from System.Web.Services.WebService, but it doesn't need to do so.

One or more web service methods: These are ordinary methods in the web service class that are marked with the WebMethod attribute. This attribute indicates that the corresponding method should be made available though ASP.NET.

As long as you have these basic ingredients in place, ASP.NET will manage the lower-level details for you. For example, you never need to create a WSDL document or a SOAP message by hand. ASP.NET automatically generates the WSDL document for your web service when it's requested. Similarly, .NET converts ordinary method calls to SOAP messages transparently.

The only layer you need to worry about is the business-specific code that actually performs the task (such as inserting data into a database, creating a file, performing a calculation, and so on). You write this code like any other VB method.

Configuring a Web Service Project

You can add web services to any web application. In fact, the only difference between the ASP.NET Web Site and ASP.NET Web Service project types in Visual Studio is that the Web Site project type starts off with one web page, while the Web Service project type has one default web service. Other than that, the project types (and their configurations) are identical.

However, if you want to use web services in a client application (which you usually do), you need to take some extra steps to set up your project. The problem is that you can't rely on the built-in Visual Studio web server to host your web services. The Visual Studio web server dynamically chooses a new port each time you run it, which means your client application would have serious difficulties tracking down the web service. Instead, you need to create a virtual directory for your web application (as described in Chapter 12). Once you've taken this step, you can create the web application in this location. That way, your web application is hosted by IIS at a fixed location, which allows clients to connect to it.

Chapter 12 contains a great deal of information about IIS and web services. However, here's a quick series of steps that will get you started. (These steps assume you're using Windows 2000 or Windows XP—consult Chapter 12 for information about Windows 2003.)

1. First, create the virtual directory for your web service. For example, you could create the directory c:\ASP.NET\WebServices using Windows Explorer.

2. The next step is to turn this folder into a virtual directory and a web application. Start IIS Manager (select Start ➤ Programs ➤ Administrative Tools ➤ Internet Information Services), right-click the Default Website item, and select New Virtual Directory.

3. Follow the Virtual Directory Creation Wizard, choosing an alias for the virtual directory (the examples for this chapter use the name WebServices) and supplying the physical path you created in step 1. Use the default security settings.

4. Now fire up Visual Studio. Select File ➤ New Web Site, and choose the ASP.NET Web Site or ASP.NET Web Service project type (either one works for web services).

5. In the Location box, choose HTTP. Then, supply the virtual path to your web service, which is `http://localhost/` followed by the virtual directory alias, as in `http://localhost/WebServices`.

6. Click OK to create your web service project in the virtual directory.

7. To add a web service to your project, right-click the project in the Solution Explorer, and choose Add ➤ Add New Item. In the Add New Item dialog box, pick Web Service (see Figure 22-1). You can choose to place the code in a separate code-behind file (which is the usual approach) or directly in the web service file. Click Add to add the file.

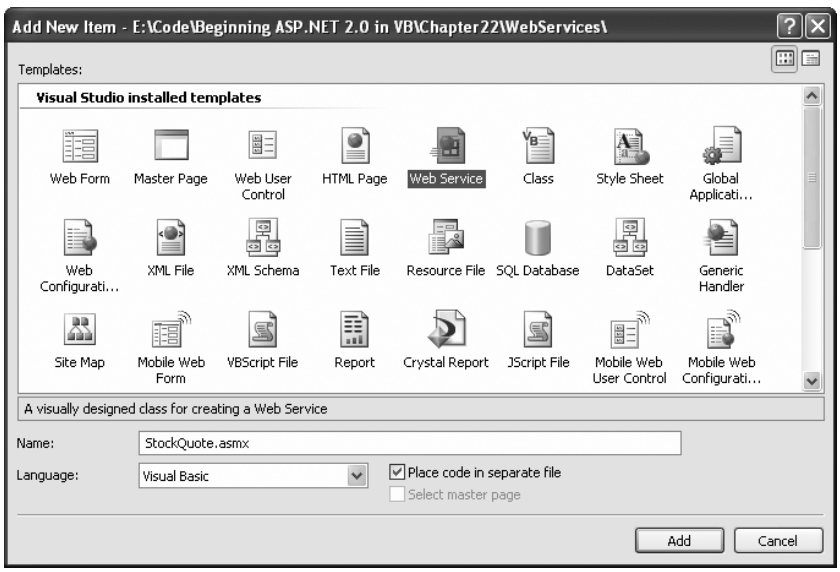

Figure 22-1. *Adding a web service to any project*

Tip You need to use a similar approach to use the web service examples included with the book. You can place the folder with the web services wherever you want, as long as you create the correct virtual directory (`http://localhost/WebServices`). The readme file has complete instructions.

In the following sections, you'll look at the code for a simple web service and learn how it works.

The StockQuote Web Service

The following listing shows the code for the StockQuote web service—a simple web service that contains only one method. In this example, all the code is placed in a single .asmx file (so no code-behind file is used).

```
<%@ WebService Language="VB" Class="StockQuote" %>

Imports System.Web
Imports System.Web.Services
Imports System.Web.Services.Protocols

Public Class StockQuote
    Inherits WebService

    <WebMethod()> _
    Public Function GetStockQuote(ByVal ticker As String) As Decimal
        ' (Perform database lookup here.)
    End Function

End Class
```

As you can see, the StockQuote class looks more or less the same as any .NET class. The two differences—inheriting from WebService and using the WebMethod attribute—are highlighted in bold in the example.

An .asmx file is similar to the standard .aspx file used for graphical ASP.NET pages. As with .aspx files, .asmx files start with a directive that specifies the language and are compiled the first time they are requested in order to increase performance. You can create them with Notepad or any other text editor, but professional developers use Visual Studio. The letter *m* indicates "method" because all web services are built from one or more web methods that provide the actual functionality.

Understanding the StockQuote Service

The code for the StockQuote web service is quite straightforward. The first line specifies the file is used for a web service and is written in VB. The next line imports the web service namespace so all the classes you need are at your fingertips. As with other core pieces of ASP.NET technology, web service functionality is provided through prebuilt types in the .NET class library.

The remainder of the code is the actual web service, which is built from a single class that derives from WebService. In the previous example, the web service class contains a method called GetStockQuote(). This is a normal VB function with a <WebMethod()> attribute before its definition.

.NET attributes are used to *describe* code. The WebMethod attribute doesn't change how your function works, but it does tell ASP.NET that this is one of the procedures it must make available over the Internet. Methods that don't have this attribute won't be accessible (or even visible) to remote clients, regardless of whether they are defined with the Private or Public keyword.

The best way to understand a web service is to think of it as a business object. Business objects support your programs but don't take part in creating any user interface. For example, business objects might have helper functions that retrieve information from a database, perform calculations, or process custom data. Your code creates business objects whenever it needs to retrieve or store specific information, such as report information from a database. Sometimes business objects are called *service providers* because they perform a task (a service), but they rarely retain any information in memory.

The most remarkable part of this example is that the StockQuote web service is already complete. All you need to do is place this .asmx file on your web server in a virtual directory that other clients can access, as you would with an .aspx page. Other clients can then start creating instances of your class and can call the GetStockQuote() method as though it were a locally defined procedure. If you want to call the web method from a .NET client, all you need to do is use .NET to create a proxy class.

Web Services with Code-Behind

As you've seen with web pages, Visual Studio uses code-behind files so you can separate design and code. In the case of .asmx files, this feature isn't really necessary because web service files consist entirely of code and don't contain any user interface elements. However, it's still usually done as a matter of convention.

Visual Studio presents web service code files a little differently than web page code files. Web service code files are automatically placed in the App_Code subfolder (see Figure 22-2). This is because of a minor difference in the way ASP.NET compiles web services. If you don't place the web service code file here, it won't be compiled.

Figure 22-2. *A web service with a code-behind file*

The code-behind version of the StockQuote web service is almost identical. Listing 22-1 shows StockQuote.asmx, and Listing 22-2 shows StockQuote.asmx.vb.

Listing 22-1. *StockQuote.asmx*

```
<%@ WebService Language="VB"
  CodeBehind="~/App_Code/StockService.vb" Class="StockService" %>
```

Listing 22-2. *StockQuote.asmx.vb*

```
Imports System.Web
Imports System.Web.Services
Imports System.Web.Services.Protocols

Public Class StockQuote
    Inherits WebService

    <WebMethod()> _
    Public Function GetStockQuote(ByVal ticker As String) As Decimal
        ' (Perform database lookup here.)
    End Function

End Class
```

Note that the client always connects to a web service using a URL that points to the .asmx file, much as a client requests a web page using a URL that points to an .aspx file. That means clients find the StockQuote service at `http://localhost/WebServices/StockQuote.asmx`.

The ASP.NET Intrinsic Objects

When you inherit from System.Web.Services.WebService, you gain access to several of the standard built-in ASP.NET objects that you can use in an ordinary Web Forms page. These include the following:

Application: Used to store data globally so that it's available to all clients (as described in Chapter 9).

Server: Used for utility functions, such as encoding strings, so they can be safely displayed on a web page (as described in Chapter 5).

Session: Used for client-specific state information. However, you'll need to take some extra steps to make it work with web services. Chapter 23 has the full details.

User: Used to retrieve information about the current client, if the client has been authenticated. Chapter 23 discusses web services and security.

Context: Provides access to Request and Response and, more usefully, the Cache object (described in Chapter 26).

On the whole, these built-in objects won't often be used in web service programming, with the exception of the Cache object. Generally, a web service should look and work like a business object and not rely on retrieving or setting additional information through a built-in ASP.NET object. However, if you need to use per-user security or state management, these objects will be useful. For example, you could create a web service that requires the client to log on and subsequently stores important information in the Session collection. Or you could create a web service that retrieves a large object (such as a DataSet), stores it in server memory using the Session collection, and returns whatever information you need through other web methods that can be invoked as required.

YOU DON'T NEED TO INHERIT FROM WEBSERVICE

Remember, inheriting from WebService is just a convenience for accessing a few common ASP.NET objects. If you don't need to use any of these objects (or if you're willing to go through the shared 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 date in session state.
Application("LastUpdate") = DateTime.Now
```

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.Application("LastUpdate") = DateTime.Now
```

Documenting Your 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 all accomplished through the WSDL document. However, although the 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 attributes.

■ **Tip** Attributes are a special language construct that's built into the .NET Framework. Essentially, attributes describe your code to the .NET runtime by adding extra metadata. The attributes you use for the web service description are recognized by the .NET Framework and provided to clients in automatically generated description documents and through the browser test page. Attributes are used in many other situations, and even if you haven't encountered them before, you're sure to encounter them again in .NET programming.

Descriptions

You can add descriptions to each function through the WebMethod attribute and to the entire web service as a whole using a WebService attribute. For example, you could describe the StockQuote service like this:

```
<WebService(Description:="Retrieve information about a stock.")> _
Public Class StockQuote
    Inherits WebService

    <WebMethod(Description:="Gets a quote for a NASDAQ stock.")> _
    Public Function GetStockQuote(ByVal ticker As String) As Decimal
        ' (Perform database lookup here.)
    End Function

End Class
```

In this example, the Description property is added as a named argument using the = operator.

These custom descriptions will appear in two important places. First, they will be added to the WSDL document that ASP.NET generates automatically. The descriptive information is added as <documentation> tags:

```
<service name="StockQuote">
  <documentation>Retrieve information about a stock.
  </documentation>
  ...
</service>
```

Second, the descriptive information will also appear in the automatically generated browser test page, which may be viewed by the programmer who is designing the client application. The test page is described later in this chapter (in the "Testing Your Web Service" section).

The XML Namespace

You should specify a unique namespace for your web service that can't be confused with the namespaces used for other web services on the Internet. Note that this is an XML namespace, not a .NET namespace. It doesn't affect how your code works or how the client uses your web service. Instead, this namespace uniquely identifies your web service in the WSDL document. XML namespaces typically look like URLs, but they don't need to correspond to a valid Internet location. For more information, refer to the XML overview in Chapter 17.

■ **Tip** Think of XML namespaces as one of the ways that an application or a web service catalog can distinguish between different web services.

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 stock quote web service a namespace like `http://www.mycompany.com/StockQuote`. If you don't specify a namespace, the default (`http://tempuri.org/`) will be used. This is fine for development, but you'll see a warning message in the test page advising you to use something more distinctive.

You specify the namespace through the WebService attribute, as shown here:

```
<WebService(Description:="Gets a quote for a NASDAQ stock.", _
 Namespace:="http://www.prosetech.com/Stocks")> _
Public Class StockQuote
    Inherits WebService
    ...
End Class
```

Conformance Claims

As you learned in Chapter 21, web services have developed rapidly, and the standards web services use (such as SOAP and WSDL) are still evolving. What's more, some of these standards (such as SOAP) provide more than one way to accomplish the same thing. Different platforms may use different approaches, which leads to a lack of consistency and makes it difficult to ensure everyone can really interact.

To deal with the confusion, web gurus created yet another standard: WS-Interoperability. WS-Interoperability sets out a series of guidelines and recommendations web services must follow, thereby strengthening the rules. If the web services and web clients follow the same WS-Interoperability rules, they shouldn't have any trouble communicating, no matter what platforms are used to build web services and client applications.

When you create a new web service, Visual Studio adds a [WebServiceBinding] attribute to your web service declaration. This attribute indicates the level of compatibility you're targeting. Currently, the only option is WsiProfiles.BasicProfile1_1, which represents the WS-Interoperability Basic Profile 1.1. However, as standards evolve, you'll see newer versions of SOAP and WSDL, as well as newer versions of the WS-Interoperability profile to go along with them.

```
<WebServiceBinding(ConformsTo:=WsiProfiles.BasicProfile1_1)> _
<WebService(Description:="Retrieve information about a stock.")> _
Public Class StockQuote
    Inherits WebService
    ...
End Class
```

Once you have the WebServiceBinding attribute in place, .NET will warn you with a compiler error if your web service strays outside the bounds of allowed behavior. By default, all .NET web services are compliant, but you can inadvertently create a noncompliant service by adding certain attributes. For example, it's possible to create two web methods with the same name, as long as their signatures differ and you give them different message names using the MessageName property of the WebMethod attribute. This strange feature isn't recommended, and this behavior isn't allowed according to the WS-Interoperability profile. If you use this feature and try to compile your web service, you'll get a compilation error explaining that you've violated the standard.

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

```
<WebServiceBinding(ConformsTo:=WsiProfiles.BasicProfile1_1, _
 EmitConformanceClaims:=True)> _
<WebService(Description:="Retrieve information about a stock.")> _
Public Class StockQuote
    Inherits WebService
    ...
End Class
```

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

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

```
<WebServiceBinding(ConformsTo:=WsiProfiles.None)> _
<WebService(Description:="Retrieve information about a stock.")> _
Public Class StockQuote
    Inherits WebService
    ...
End Class
```

Testing Your Web Service

Even without creating a client for your web service, you can use the built-in features of the .NET Framework to view information about your web service and perform a rudimentary test.

The most useful testing feature is the ASP.NET test page: an automatically generated HTML page that lets you execute the methods in a web service and review its WSDL document. You don't have to perform any special steps to see this page—ASP.NET generates it automatically when you request an .asmx web service file.

Before continuing, you should modify the GetStockQuote() method in the same web service so it returns a hard-coded value. This will allow you to test that it's working with the test page. For example, you could use the statement `Return ticker.Length`. That way, the return value will be the number of characters you supplied in the Ticker parameter.

```
<WebMethod(Description:="Gets a quote for a NASDAQ stock.")> _
Public Function GetStockQuote(ByVal ticker As String) As Decimal
    Return ticker.Length
End Function
```

The Web Service Test Page

To view the web service test page, you can have one of two options:

- Seeing as you're using IIS to host your web service project, you don't need to run Visual Studio to get the test page. Just fire up a browser, and request the web service URL (such as `http://localhost/WebServices/StockQuote.asmx`).
- Inside Visual Studio you can quickly launch the current web service in the same way you launch a web page—just click the Start button.

Either way, you'll see a simple web page (shown in Figure 22-3) that lists all the available web service methods. (In this case, only one, GetStockQuote(), is available.) The test page also lists whatever description information you may have added through the WebMethod and WebService attributes.

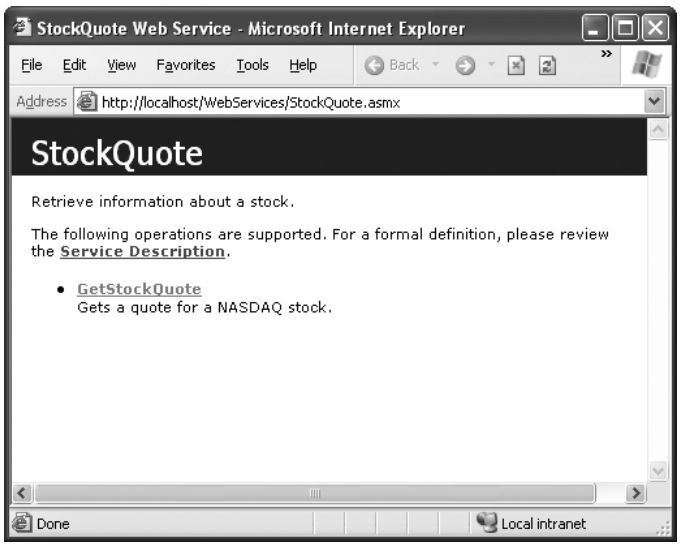

Figure 22-3. *The web service test page*

Remember, you don't need to write any special code to make this page appear, and you can make the request using any browser. ASP.NET generates the page for you automatically, and the page is intended purely as a testing convenience. Clients using your web service won't browse to this page to interact with your web service, but they might use it to find out some basic information about how to use it.

Service Description

You can also click the Service Descriptions link to display the WSDL description of your web service (see Figure 22-4), which you examined in detail in Chapter 21.

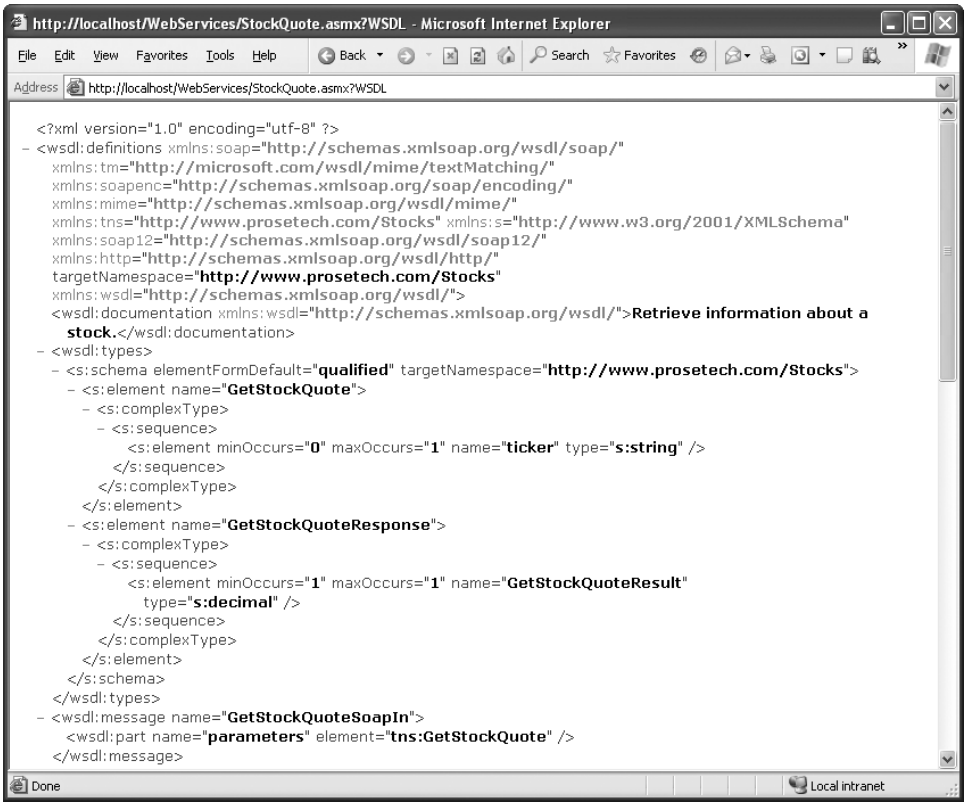

Figure 22-4. *A portion of the StockQuote WSDL document*

When you click this link, the browser makes a request to the .asmx file with a WSDL parameter in the query string:

```
http://localhost/WebServices/StockQuote.asmx?WSDL
```

If you know the location of an ASP.NET web service file, you can always retrieve its WSDL document by adding ?WSDL to the URL. This provides a standard way for a client to find all the information it needs to make a successful connection. Whenever ASP.NET receives a URL in this format, it generates and returns the WSDL document for the web service.

Method Description

You can find out information about the methods in your web service by clicking the corresponding link. For example, in the StockQuote web service, you can click the GetStockQuote link (see Figure 22-5).

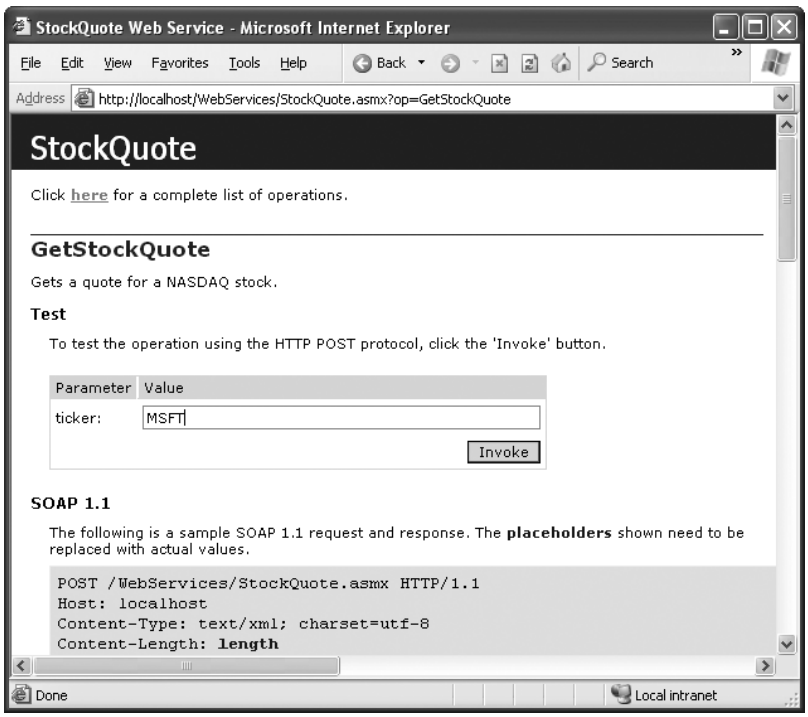

Figure 22-5. *The GetStockQuote() method description*

This window provides two sections. The first part consists of a text box and Invoke button that allows you to run the function without needing to create a client. The second part is a list of the different protocols you can use to connect with the web service (HTTP POST, HTTP GET, and SOAP) and a technical description of the message format for each one.

Testing a Method

To test your method, enter a ticker, and click the Invoke button. The result will be returned to you as an HTML page (see Figure 22-6).

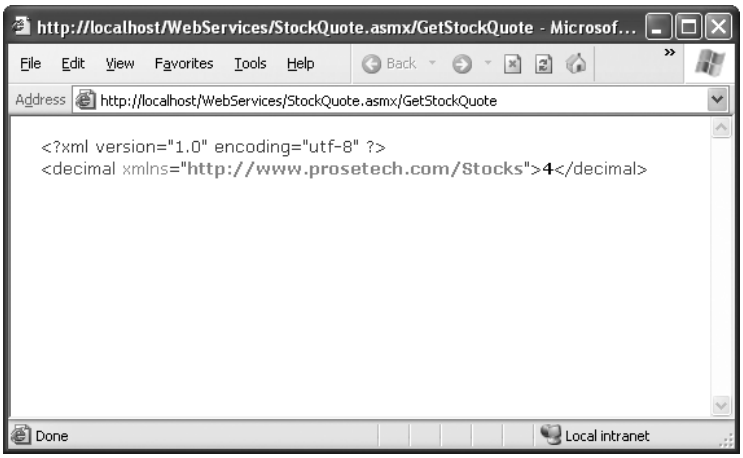

Figure 22-6. *The result from GetStockQuote()*

This may not be the result you expected, but the information is there—wrapped inside the tags of an XML document. This format allows a web service to return complex information, such as a class with several members or even an entire DataSet. If you followed the suggestions for modifying the StockQuote service, you should see a number representing the length of the stock ticker you supplied.

The web service test page is a relatively crude way of accessing a web service. It's never used for any purpose other than testing. In addition, it supports a smaller range of data types. For example, the test page can't call a web method that requires a custom structure or DataSet as a parameter, because the browser won't be able to create and supply these objects.

One question that many users have after seeing this page is, where does this functionality come from? For example, you might wonder whether the test page has additional script code for running your method or whether it relies on features built into the latest version of Internet Explorer. In fact, all that happens when you click the Invoke button is a normal HTTP operation. In version 1.1 of the .NET Framework, this operation is an HTTP POST that submits the parameters you supply. If you look at the URL for the result page, you'll see something like this:

```
http://localhost/WebServices/StockQuote.asmx/GetStockQuote
```

The request URL follows this format:

```
http://localhost/[VirtualDirectory]/[AsmxFile]/[Method]
```

The actual parameter values are posted to this URL in the body of the request (similar to a web page postback), so you won't see them in the URL. The results are sent back to your browser as an XML document.

The clients that use your web service will almost never use this HTTP POST approach Instead, they will send full SOAP messages over HTTP. The actual physical connection (also known as the *transport protocol*) remains the same. The only difference is in the data you send (otherwise known as the *message protocol*). With HTTP POST, you send the values in a simple name/value collection, according to the HTML standard. With SOAP over HTTP, you post the data in an XML package called a SOAP message.

SOAP allows for more flexibility. Because it was designed with web services in mind, it lets you send custom objects or DataSets as parameters. But the fact that you can access your web method through a simple HTTP request demonstrates the simplicity of web services. Surely, if you can run your code this easily in a basic browser, true cross-platform integration can't be that much harder.

Web Service Data Types

Although the client can interact with a web service method as though it were a local function or subroutine, some differences exist. The most significant of these are the restrictions on the data types you can use. Table 22-1 lists the data types that are supported for web service parameters and return values.

■ **Tip** These limitations are designed to ensure cross-platform compatibility. There's no reason .NET couldn't create a way to convert objects to XML and back and then use that to allow you to send complex objects to a web service. However, this "extension" would limit the ability of non-.NET clients to use the web services.

Table 22-1. *Web Service Data Types*

Data Type	Description
The basics	Standard types such as integers and floating-point numbers, Boolean variables, dates and timespans, and strings are fully supported.
Enumerations	Enumeration types (defined in VB with the Enum keyword) are fully supported.
DataSet and DataTable	This gives you an easy package to send information drawn from a relational database. For this to work, the web service actually converts the DataSet or DataTable into an XML document. The client converts it back into an object (if it's a .NET client) or just works with the XML (if it's a client on another platform). The DataRow class isn't supported.
XmlNode	Objects based on System.Xml.XmlNode are representations of a portion of an XML document. Under the hood, all web service data is passed as XML. This class allows you to directly support a portion of XML information whose structure may change.

Continued

Table 22-1. *Continued*

Data Type	Description
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. If you use a class with defined methods, these methods will not be transmitted to the client, and they will not be accessible to the client. You won't be able to successfully use most other .NET classes.
Arrays and collections	You can use arrays of any supported type, including DataSets, XmlNodes, and custom objects. You can also use many collection types, such as the ArrayList and generic lists.

The full set of objects is supported for return values and for parameters when you're communicating through SOAP. If you're communicating through HTTP GET or HTTP POST, you'll be able to use only basic types, enumerations, and arrays of basic types or enumerations. This is because complex classes and other types cannot be represented in the query string (for HTTP GET) or form body (for an HTTP POST operation).

The StockQuote Service with a Data Object

If you've ever used a stock quote service over the Internet, you've probably noticed that the example so far is somewhat simplified. The GetStockQuote() function returns one piece of information—a price quote—whereas popular financial sites usually produce a full quote with a 52-week high and 52-week low and other information about the volume of shares traded on a particular day. You could add more methods to the web service to supply this information, but that would require multiple similar function calls, which would slow down performance, because more time would be spent sending messages back and forth over the Internet. The client code would also become more tedious.

A better solution is to use a data object that encapsulates all the information you need. You can define the class for this object in the same file and then use it as a parameter or return value for any of your functions. The data class is a completely ordinary VB class, and it shouldn't derive from System.Web.Services.WebService. It can contain public member variables that use any of the data types supported for web services. It shouldn't contain methods—if it does, they will simply be ignored when the WSDL document is generated, and they won't be available to the client.

The client will receive the data object and be able to work with it exactly as though it were defined locally. In fact, it will be—the automatically generated proxy class will contain a copy of the class definition. This definition may not match the server-side copy exactly—after all, the client and the server might be written in different programming languages or hosted on different platforms. However, the client-side copy will contain the same data members, which means it can be serialized to the same XML representation.

Here's how the StockQuote service would look with the addition of a convenient data object:

```
Public Class StockInfo
    Public Price As Decimal
    Public Symbol As String
    Public High_52Week As Decimal
    Public Low_52Week As Decimal
    Public CompanyName As String
End Class

<WebServiceBinding(ConformsTo:=WsiProfiles.BasicProfile1_1)> _
<WebService(Description:="Methods to get stock information.", _
 Namespace:="http://www.prosetech.com/Stocks")> _
Public Class StockQuote
    Inherits WebService

    <WebMethod(Description:="Gets a price quote for a stock.")> _
    Public Function GetStockQuote( ByVal ticker As String) _
      As StockInfo
        Dim quote As New StockInfo()
        quote.Symbol = ticker
        quote = FillQuoteFromDB(quote)
        Return quote
    End Function

    Private Function FillQuoteFromDB(ByVal lookup As StockInfo) _
      As StockInfo
        ' You can add the appropriate database code here.
        ' For test purposes this function hard-codes
        ' some sample information.
        lookup.CompanyName = "Trapezoid"
        lookup.Price = 400
        lookup.High_52Week = 410
        lookup.Low_52Week = 20
        Return lookup
    End Function
End Class
```

Dissecting the Code...

Here's what happens in the GetStockQuote() function:

1. A new StockInfo object is created.
2. The corresponding Symbol is specified for the StockInfo object.
3. The StockInfo object is passed to another function that fills it with information. This function is called FillQuoteFromDB().

The FillQuoteFromDB() function isn't visible to remote clients because it lacks the WebMethod attribute. It isn't even visible to other classes or code modules, because it's defined with the Private keyword. Typically, this function will perform some type of database lookup. It might return information that is confidential and should not be made available to all clients. By putting this logic into a separate function, you can separate the code that determines what information the client should receive and still have the ability to retrieve the full record if your web service code needs to examine or modify other information. Generally, most of the work that goes into creating a web service—once you understand the basics—will be spent trying to decide the best way to divide its functionality into separate procedures.

You might wonder how the client will understand the StockInfo object. In fact, the object is really just returned as a block of XML or SOAP data. If you invoke this method through the test page, you'll see the result shown in Figure 22-7.

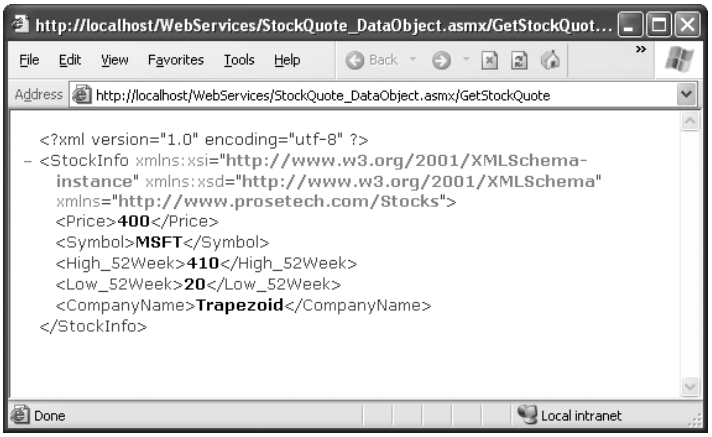

Figure 22-7. *The result from GetStockQuote() as a data object*

But ASP.NET is extremely clever about custom objects. When you use a class like StockInfo in your web service, it adds the definition directly into the WSDL document. When you generate the proxy class for your client, .NET will also add a definition for the

StockInfo class. This definition won't necessarily look exactly the same as your server-side copy, but it will contain all the same data members. (Any code you've added is lost.) The end result is that you'll be able to call the web service and receive a StockInfo object, just like you'd expect if you were calling a local function.

Incidentally, it's up to you whether you use full property procedures or just public variables—the effect is the same. This means you could just as easily rewrite the StockInfo class using property procedures like this:

```
Public Class StockInfo
    Private _price As Decimal
    Private _symbol As String
    Private _high_52Week As Decimal
    Private _low_52Week As Decimal
    Private _companyName As String

    Public Property Price() As Decimal
        Get
            Return _price
        End Get
        Set(ByVal value As Decimal)
            _price = value
        End Set
    End Property

    Public Property Symbol() As String
        Get
            Return _symbol
        End Get
        Set(ByVal value As String)
            _symbol = value
        End Set
    End Property

    ' (Remainder of property procedures omitted.)
End Class
```

From the standpoint of the web service client, this version of the StockInfo class is completely equivalent to the version shown earlier. That's because any code you place in a property procedure (other than the code that sets or gets the value) is ignored when the class is generated on the client side. If you think about it, this makes sense. The StockInfo class definition is used to create the WSDL definition, and the client-side proxy class is generated based on the WSDL document. The WSDL document never contains code.

Consuming a Web Service

Microsoft has repeatedly declared that its ambition with programming tools such as Visual Studio, the CLR, and the .NET class library is to provide a common infrastructure for application developers, who will then need to create only the upper layer of business-specific logic. Web services hold true to that promise. In the following sections, you'll see how you can call a web service method as easily as a method in a local class.

Configuring a Web Service Client in Visual Studio

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 work with both projects at once in Visual Studio, follow these steps:

1. Open the web service application (in this case, the StockQuote service).

2. Select File ➤ Add ➤ New Web Site (assuming you want to create a web client).

3. Choose ASP.NET Web Site, give it a title (for example, enter **WebClient**), and click OK. You can create your client as a File System site or an HTTP site—either way works fine.

4. You should set the new project as the start-up project (otherwise, you'll just see the web service test page when you click the Start button). To make this adjustment, right-click the new project in the Solution Explorer, and choose Set As StartUp Project.

Your Solution Explorer should now look like Figure 22-8.

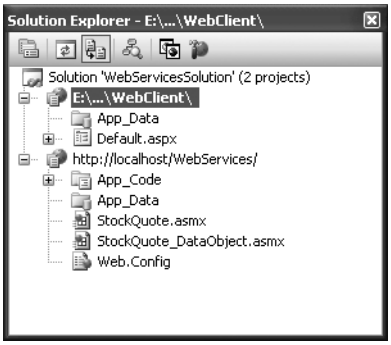

Figure 22-8. *Two projects in Solution Explorer*

Once you've created a solution that has both the web service and the client, you might want to save a solution file so you can quickly load this combination again. To do this, select the first item in the Solution Explorer (the solution name). Then, choose File ➤ Save [SolutionName].sln As.

By default, Visual Studio saves the solution file in a user-specific temporary location. However, you can save it somewhere that's more easily accessible for future use. Once you've saved the .sln solution file, you can double-click it in Windows Explorer to launch Visual Studio with both projects.

■ **Note** You don't need to create a virtual directory in advance for your web client project. You do need to create a virtual directory for your web service project (as described in the previous chapter), because the client needs to know where to find the web service.

The Role of the Proxy Class

Web services communicate with other .NET applications through a special ingredient called the *proxy class*. The proxy class sits between the code in your client application and the web service.

When you (the client) want to call a web method, you call the corresponding method of the proxy object. The proxy object then silently fetches the results for you using SOAP calls. Strictly speaking, you don't need to use a proxy class—you could create and receive the SOAP messages on your own. However, this process is quite difficult and involves a degree of low-level understanding and complexity that would make web services much less useful. The proxy class simplifies life because it takes care of details such as communicating over the network, waiting for a response, parsing the result out of the SOAP message that's returned from the web service, and so on.

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 the following sections, you'll try both approaches.

Creating a Web Reference in Visual Studio

Even when two projects are added to the same solution, they still have no way to communicate with each other. To set up this layer of interaction, you need to create a special proxy class. In Visual Studio, you create this proxy class by adding a web reference. Web references are similar to ordinary references, but instead of pointing to assemblies with ordinary .NET types, they point to a URL with a WSDL contract for a web service.

■ **Note** Before you can add a web reference, you should save and compile the web service application (right-click the project, and choose Build Web Site in Visual Studio). Otherwise, the client might get outdated information about the web services you provide or be unable to see them.

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 22-9. This dialog box provides options for searching web registries or entering a URL directly.

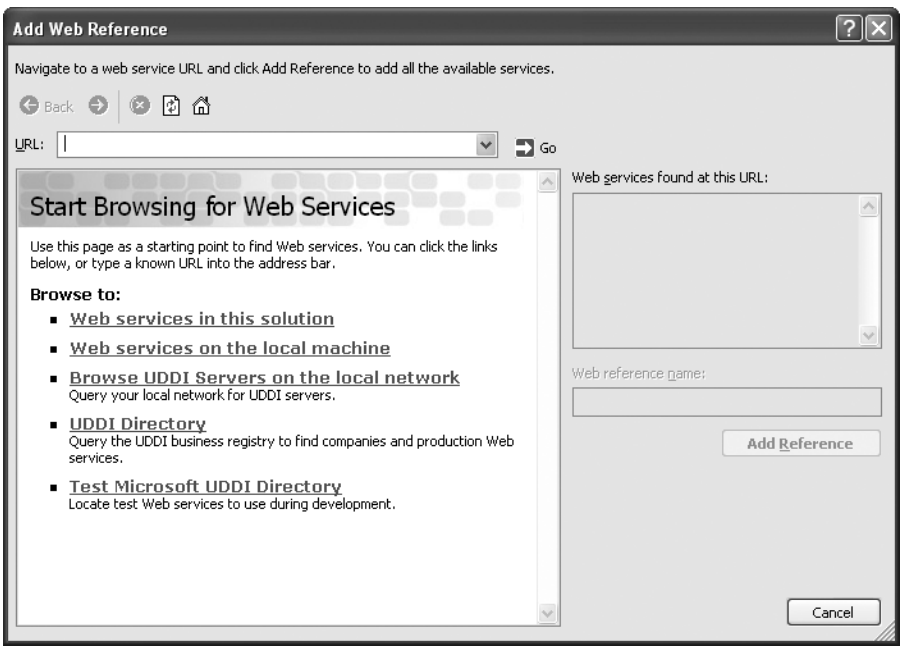

Figure 22-9. *The Add Web Reference dialog box*

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 preview window (as shown in Figure 22-10), and the Add Reference button will be enabled.

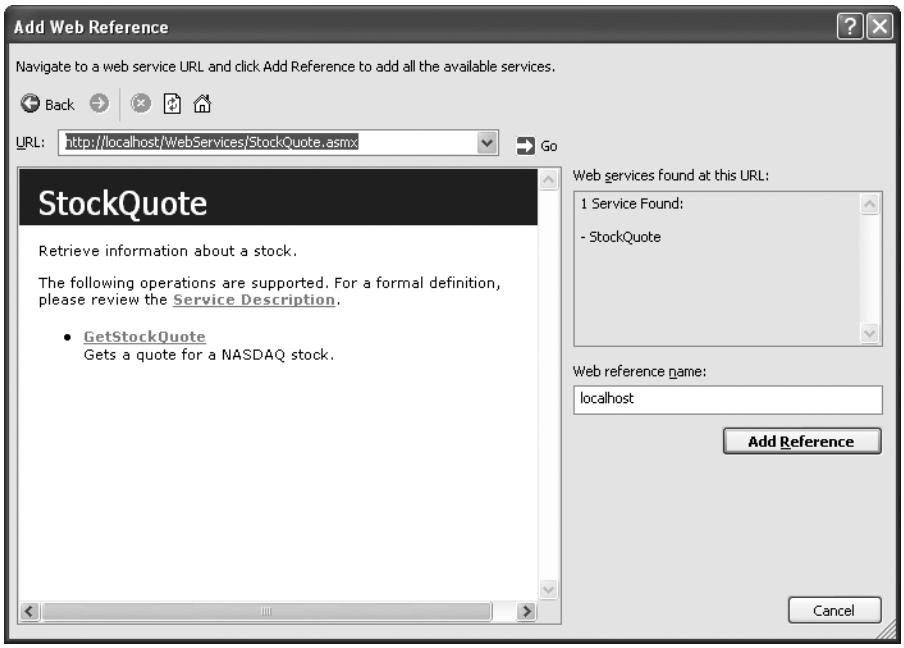

Figure 22-10. *Adding a web reference*

4. Change the value for the web reference name if you want to put your web reference in a different namespace. By default, the namespace is chosen to match the name of the server where the web service resides (and is localhost for web services on the current computer).

5. To add the reference to this web service, click Add Reference.

6. Now your computer (the web server for this service) will appear in the Web References group for your project in the Solution Explorer (see Figure 22-11).

■ **Note** The web reference you create uses the WSDL contract and information that exists at the time you add the reference. If the web service changes, you'll need to update your proxy class by right-clicking the server name (localhost, in this case) and choosing Update Web Reference.

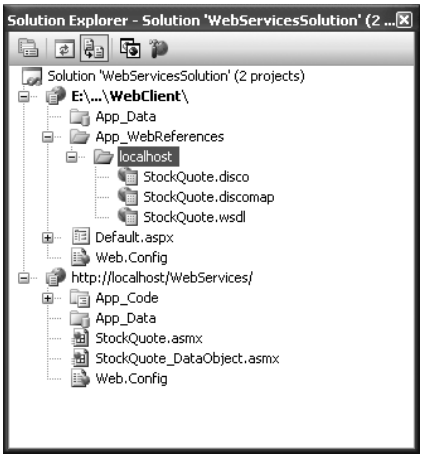

Figure 22-11. *The web reference*

You can add a web reference only to a single .asmx file at a time. If you have more than one web service in the same web application, you need to add a separate web service for each one you want to use. For example, the downloadable examples have two versions of the StockQuote web service—one that uses a data object and one that doesn't. If you decide to use both, you'll need to add two web references.

Creating a Proxy with WSDL.exe

You can also generate the proxy class by hand; you just use a utility called WSDL.exe that is included with the .NET Framework. (You can find this file in the .NET Framework directory, which is typically in a path similar to c:\Program Files\Microsoft.NET\SDK\v2.0\Bin. Visual Studio users have the WSDL.exe utility in the c:\Program Files\Microsoft Visual Studio 8\SDK\v2.0\Bin directory.)

In most cases, you'll use the Visual Studio web reference feature instead of WSDL.exe. But sometimes WSDL.exe makes more sense:

- You need to configure advanced options that Visual Studio doesn't provide. For example, you can use WSDL.exe to generate proxies for two different web services that use the same set of objects. (In this case, you need the /sharetypes parameter.)
- You want to look at the proxy class code. With an ASP.NET client, the proxy class code is created when the application executes for the first time, and you never get to see it.
- You aren't using Visual Studio.

WSDL.exe is a command-line utility, so it's easiest to use by opening a command prompt window. (Click the Start button, and choose Programs ➤ Visual Studio 2005 ➤ Visual Studio Tools ➤ Visual Studio 2005 Command Prompt.)

The syntax for WSDL.exe is as follows:

```
wsdl /language:language  /protocol:protocol /namespace:myNameSpace /out:filename
     /username:username /password:password /domain:domain <url or path>
```

Table 22-2 describes these parameters, along with a few additional details.

Table 22-2. *WSDL.exe Command-Line Parameters*

Parameter	Meaning
language	This is the language in which the class is written. If you don't make a choice, the default is C#.
protocol	Usually, you'll omit this option and use the default (SOAP 1.1). However, you could also specify HttpGet and HttpPost for slightly more limiting protocols, or you could opt for SOAP 1.2 by specifying SOAP12.
namespace	This is the .NET namespace that your proxy class will use. If you omit this parameter, no namespace is used, and the classes in this file are available globally. For better organization, you should probably choose a logical namespace.
out	This allows you to specify the name for the generated file. By default, this is the name of the service followed by an extension indicating the language (such as StockQuote.vb). You can always rename this file after it's generated.
username, password, and domain	You should specify these values if the web server requires authentication to access the discovery and WSDL documents.
url or path	This is the last portion of the WSDL.exe command line, and it specifies the location of the WSDL file for the web service.
appsettingurlkey	The is the configuration setting that's used to store the web service URL. You can change this configuration setting if your web service moves to another server.
fields	If you use this option, any data objects that the web service uses will be generated with public member variables instead of public properties.
sharetypes	This allows you to add a reference to two or more web services that use the same data objects. This ensures that any data object classes are created only once. More important, you can use the same data objects with both web services.

A typical WSDL.exe command might look something like this (split over two lines to fit the bounds of the page):

```
wsdl /namespace:localhost /language:VB
  http://localhost/WebServices/StockQuote.asmx?WSDL
```

In this case, a StockQuote.vb file is created with the proxy class code. The proxy class is defined in a .NET namespace named localhost. You can add this class to your ASP.NET application (in the App_Code folder) and use it to communicate with the web service.

■ **Note** You can use a web service without WSDL.exe or Visual Studio. In fact, you don't even need a proxy class. All you need is a program that can send and receive SOAP messages. After all, web services are designed to be cross-platform. However, unless you have significant experience with another tool (such as the Microsoft SOAP Toolkit), the details are generally frustrating and unpleasant. .NET provides the prebuilt infrastructure you need to easily communicate with a web service, without errors.

Dissecting the Proxy Class

On the surface, Visual Studio makes it seem like all you need is a simple link to your web service. In reality, whenever you add a web reference, Visual Studio creates a proxy class for you automatically. To really understand how web service clients work, you need to take a look at this class.

Unfortunately, you won't see the file for the proxy class in your client project. That's because ASP.NET generates it automatically when you run your application. If you want to study the proxy class code, you'll need to create the proxy class using WSDL.exe (as described in the previous section), or you'll need to add a web reference to another type of application that doesn't use the same compile-on-demand model (such as a Windows application).

■ **Note** You'll learn how to create a Windows client in Chapter 23. For now, you can experiment with the WSDL.exe tool or just review the code that's shown next.

The proxy class has the same name as the web service class. It inherits from SoapHttpClientProtocol, which has a fair bit of built-in functionality. Here's the declaration for the proxy class that provides communication with the StockQuote service:

```
Public Class StockQuote
    Inherits System.Web.Services.Protocols.SoapHttpClientProtocol
    ...
End Class
```

The StockQuote class contains the same methods as the StockQuote web service. This class acts as a stand-in for the remote StockQuote web service. When you call a StockQuote method, you're really calling a method of this local class. This class then performs the SOAP communication as required to contact the "real" remote web service. The proxy class acquires its ability to communicate in SOAP through the .NET class library. It inherits from the SoapHttpClientProtocol class and binds its local methods to web service methods with prebuilt .NET attributes. In other words, the low-level SOAP details are hidden not only from you but also from the proxy class, which relies on ready-made .NET components from the System.Web.Services.Protocols namespace.

For example, here's the GetStockQuote() method as it's defined in the proxy class:

```
<System.Web.Services.Protocols.SoapDocumentMethodAttribute()> _
Public Function GetStockQuote(ByVal ticker As String) As Decimal
     Dim results() As Object = Me.Invoke("GetStockQuote", _
       New Object() {ticker})
    Return CType(results(0), Decimal)
End Function
```

You'll notice that this method doesn't contain any of the business code you created in the web service. (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. In this case, the method calls the base SoapHttpClientProcotol.Invoke() to create the SOAP message and start waiting for the response. The final line of code converts the returned object into a decimal value.

If you're using the version of the StockService that uses the StockInfo data object, you'll see a similar version of the GetStockQuote() method, with one key difference:

```
<System.Web.Services.Protocols.SoapDocumentMethodAttribute()> _
Public Function GetStockQuote(ByVal ticker As String) As StockInfo
     Dim results() As Object = Me.Invoke("GetStockQuote", _
       New Object() {ticker})
    Return CType(results(0), StockInfo)
End Function
```

As you can see, the proxy class not only handles the SOAP communication layer but also handles the conversion from XML to .NET objects and data types as needed. If .NET types were used directly with web services, it would be extremely difficult to use them from other non-Microsoft platforms.

Of course, you might wonder how the client can manipulate a StockInfo object—after all, you defined this object in the web service, *not* the client. This is another web service trick. When Visual Studio builds the proxy class, it automatically checks the

parameter and return types of each method that's defined in the WSDL document. If it determines that a custom class is required, it creates a definition for that class in the proxy file.

You'll see this class definition after the proxy class in the same file:

```
Public Partial Class StockInfo
    Private priceField As Decimal
    Private symbolField As String
    Private high_52WeekField As Decimal
    Private low_52WeekField As Decimal
    Private companyNameField As String

    Public Property Price() As Decimal
        Get
            Return Me.priceField
        End Get
        Set
            Me.priceField = value
        End Set
    End Property

    ' (Remainder or property procedures omitted.)
End Class
```

Because you now have a definition for the StockInfo class, you can create your own StockInfo objects directly and work with them locally. Unlike the StockQuote proxy class, the StockInfo class doesn't participate in any SOAP messages or Internet communication; it's just a simple data class.

■ **Note** Remember, the data class on the client won't necessarily match the data class in the web service. Visual Studio simply looks for all public properties and member variables in the web service version and creates a client-side version that consists entirely of public properties. (Visual Studio uses properties instead of public variables because that gives you support for data binding.) Any code you've placed in the web service version of the class (whether it's in methods, property procedures, or constructors) is ignored completely.

The proxy class also contains some additional code you haven't seen for implementing asynchronous functionality, which allows a client to initiate a web service request and continue working, without waiting for the response. The client will be notified later when the response is received.

This is a useful technique in desktop applications where you don't want to become bogged down waiting for a slow Internet connection. However, the examples in this chapter focus on ASP.NET clients, which don't benefit as much from asynchronous consumption because the page isn't returned to the client until all the code has finished processing. Asynchronous web services are discussed in *Pro ASP.NET 2.0 in C# 2005* (Apress, 2005).

Dynamic Web Service URLs

When you create a web reference with Visual Studio, the location is 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.StockService"
     value="http://localhost/WebServices/StockService.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.

■ **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 use WSDL.exe to generate your proxy class, the default URL isn't stored in a configuration file—it's hard-coded in the constructor. To change this behavior, just use /appsettingurlkey. For example, you could use this command line:

```
wsdl /language:VB /appsettingurlkey:WsUrl
  http://localhost/WebServices/StockQuote.asmx
```

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

Using the Proxy Class

Using the proxy class is easy. In most respects, it isn't any different from using a local class. The following sample page uses the StockQuote service and displays the information it retrieves in a label. You could place this snippet of code into a Page.Load event handler.

```
' Define a StockInfo variable for your results.
Dim wsInfo As localhost.StockInfo

' Create the actual web service proxy object.
Dim ws As New localhost.StockQuote()

' Call the web service method.
wsInfo = ws.GetStockQuote("MSFT")

lblResult.Text = wsInfo.CompanyName & " is at: " & wsInfo.Price.ToString()
```

The whole process is quite straightforward. First, the code creates a StockInfo object to hold the results. Then the code creates an instance of the proxy class, which allows access to all the web service functionality. Finally, the code calls the web service method using the proxy object and assigns the returned data in the StockInfo object. Notice that the proxy class is placed in the localhost namespace, because this proxy class was created through the Visual Studio web reference feature with a web reference name of localhost. If you create a proxy class using WSDL.exe, this isn't the case—in fact, the proxy class isn't placed into any namespace by default, which means it's in the global namespace and is always available.

Figure 22-12 shows the result in a test web page.

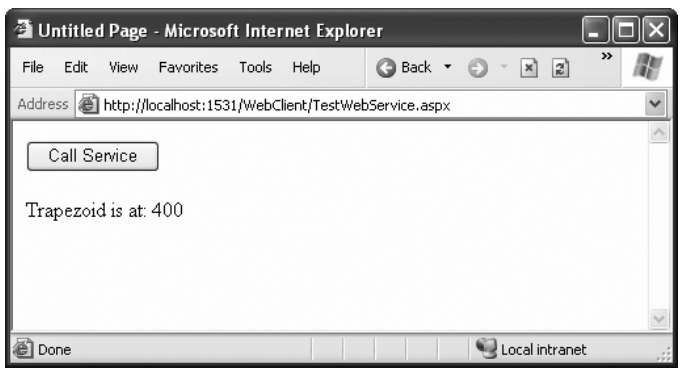

Figure 22-12. *Calling a web service in a web page*

As you experiment with your project, remember that it doesn't have a direct connection to your web service. Whenever you change the web service, you'll have to rebuild it (right-click the web service project, and select Build), and then update the web reference (right-click the client project's localhost reference, and select Update Web Reference). Until you perform these two steps, your client will not be able to access any new methods or methods that have modified signatures (different parameter lists).

Waiting and Timeouts

You might have noticed that the proxy class (StockQuote) really contains many more members than just the three methods shown in the source code. In fact, it acquires a substantial amount of extra functionality because it inherits from the SoapHttpClientProtocol class. In many scenarios, you won't need to use any of these additional features. In some cases, however, they will become useful. One example is with the Timeout property.

The Timeout property allows you to specify the maximum amount of time you're willing to wait, in milliseconds. The default (–1) indicates that you'll wait as long as it takes, which could make your web application unacceptably slow if you attempt to perform a number of operations with an unresponsive web service.

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. By default, the timeout is 100,000 milliseconds (10 seconds).

In the following example, the simple StockQuote client has been rewritten to use a timeout:

```
Dim ws As New localhost.StockQuote()

' This timeout will apply to all web method calls until it's changed.
ws.Timeout = 3000   ' 3,000 milliseconds is 3 seconds.

Try
    ' Call the web service method.
    Dim wsInfo As localhost.StockInfo = ws.GetStockQuote("MSFT")
    lblResult.Text = wsInfo.CompanyName & " is at: " & wsInfo.Price.ToString()

Catch err As System.Net.WebException
    If err.Status = System.Net.WebExceptionStatus.Timeout Then
        lblResult.Text = "Web service timed out after 3 seconds."
    Else
        lblResult.Text = "Another type of problem occurred."
    End If
End Try
```

Web Service Errors

Of course, a typical web service call could lead to other types of errors. For example, the code in your web method could generate an error. To try this, you can create the following error-prone web method:

```
<WebMethod()> _
Public Sub CauseAnError()
    Throw New DivideByZeroException()
End Sub
```

You might assume (quite reasonably) that when the client calls this method, it will receive a DivideByZeroException. However, this actually isn't the case. That's because web services are designed to be thoroughly interoperable, and as a result they don't support the idea of .NET exceptions. And that makes sense—after all, you might call a .NET web service, or you might call a web service created with some other programming framework with a different set of exception objects or a different way of handling errors. (And even if ASP.NET supported .NET exceptions in SOAP messages, you can imagine how a problem could occur if a web method threw a custom exception. The client might not have the required custom exception class, leaving it unable to process the error.)

Instead, when an unhandled exception occurs in a web method, ASP.NET catches the exception and sends a SOAP fault message to the client. When the proxy class receives this fault message, it throws a System.Web.Services.Protocols.SoapException. In other words, no matter what caused the error condition, your code will receive a SoapException. The SoapException.Message property will reveal more details, including the original exception name.

Here's how you can catch this exception on the client:

```
Dim ws As New localhost.ErrorService

Try
    ' Call the web service method.
    ws.CauseAnError()
Catch err As System.Web.Services.Protocols.SoapException
    lblResult.Text = "An error occurred in the web method code.<br />"
    lblResult.Text &= "The error is " & err.Message
End Try
```

Tip Whenever you make a web service call, you should add exception handlers for WebException and SoapException.

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 term *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:

```
Dim connectionProxy As New WebProxy("ProxyServer", 80)

Dim ws As New localhost.StockQuote()
ws.Proxy = connectionProxy
```

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

The Last Word

In ASP.NET, designing a web service is almost as easy as creating an ordinary business class. But to use web services *well,* you need to understand the role they play in enterprise applications. Web services aren't the best way to share functionality between different web pages or web applications on a web server, because of the overhead needed to send SOAP messages over the network. However, they are an excellent way to connect different software packages or glue together the internal systems of separate companies.

In the next chapter, you'll learn how to go further with web services. You'll use more advanced features such as state management, security, and transactions, and you'll learn how to call a web service from a Windows application.