CHAPTER 25

Understanding XML Web Services

Chapter 18 introduced you to the .NET remoting layer. As you have seen, this technology allows any number of .NET-savvy computers to exchange information across machine boundaries. While this is all well and good, one possible limitation of the .NET remoting layer is the fact that each machine involved in the exchange must have the .NET Framework installed, must understand the CTS, and must speak the same wire format (such as TCP).

XML web services offer a more flexible alternative to distributed application development. Simply put, an *XML web service* is a unit of code hosted by a web server that can be accessed using industry standards such as HTTP and XML. As you would guess, using neutral technologies, XML web services offer an unprecedented level of operating system, platform, and language interoperability.

In this final chapter, you will learn how to build XML web services using the .NET platform. Along the way, you will examine a number of related topics, such as discovery services (UDDI and DISCO), the Web Service Description Language (WSDL), and the Simple Object Access Protocol (SOAP). Once you understand how to build an XML web service, you will examine various approaches to generate client-side proxies that are capable of invoking "web methods" in a synchronous and asynchronous fashion.

The Role of XML Web Services

From the highest level, you can define an XML web service as a unit of code that can be invoked via HTTP requests. Unlike a traditional web application, however, XML web services are not (necessarily) used to emit HTML back to a browser for display purposes. Rather, an XML web service often exposes the same sort of functionality found in a standard .NET code library (e.g., crunch some numbers, fetch a `DataSet`, return stock quotes, etc.).

Benefits of XML Web Services

At first glance, an XML web services may seem to be little more than just another remoting technology. While this is true, there is more to the story. Historically speaking, accessing remote objects required platform-specific (and often language-specific) protocols (DCOM, Java RMI, etc.). The problem with this approach is not the underlying technology, but the fact that each is locked into a specific (often proprietary) wire format. Thus, if you are attempting to build a distributed system that involves numerous operating systems, each machine must agree upon the packet format, transmission protocol, and so forth. To simplify matters, XML web services allow you to invoke methods and properties of a remote object using standard HTTP requests. To be sure, of all the protocols in existence today, HTTP is the one specific wire protocol that all platforms can agree on (after all, HTTP is the backbone of the World Wide Web).

Another fundamental problem with proprietary remoting architectures is that they require the sender and receiver to understand the same underlying type system. However, as I am sure you can agree, a Java `arrayList` has little to do with a .NET `ArrayList`, which has nothing to do with a C++ array. XML web services provide a way for unrelated platforms, operating systems, and programming languages to exchange information in harmony. Rather than forcing the caller to understand a specific type system, information is passed between systems via XML data representation (which is little more than a well-formatted string). The short answer is, if your operating system can go online and parse character data, it can interact with an XML web service.

■ **Note** A production-level Microsoft .NET XML web service is hosted under IIS using a unique virtual directory. As explained in Chapter 23, however, as of .NET 2.0 it is now possible to load web content from a local directory (for development and testing purposes) using `WebDev.WebServer.exe`.

Defining an XML Web Service Client

One aspect of XML web services that might not be readily understood from the onset is the fact that an XML web service consumer is not limited to a web page. Console-based and Windows Forms–based clients can use a web service just as easily. In each case, the XML web service consumer indirectly interacts with the distant XML web service through an intervening proxy type.

An XML web service proxy looks and feels like the actual remote object and exposes the same set of members. Under the hood, however, the proxy's implementation code forwards requests to the XML web service using standard HTTP. The proxy also maps the incoming stream of XML back into .NET-specific data types (or whatever type system is required by the consumer application). Figure 25-1 illustrates the fundamental nature of XML web services.

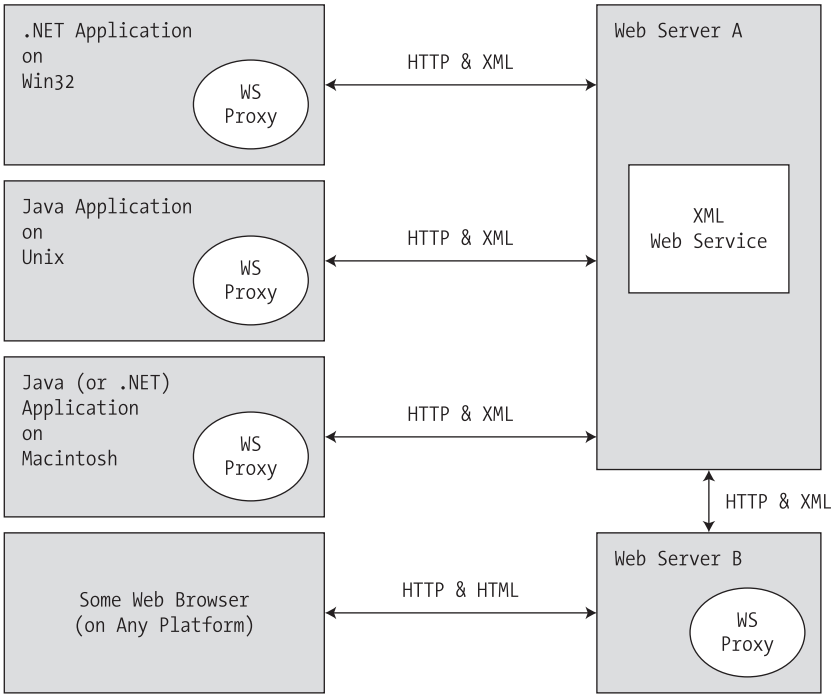

Figure 25-1. *XML web services in action*

The Building Blocks of an XML Web Service

In addition to the managed code library that constitutes the exposed functionality, an XML web service requires some supporting infrastructure. Specifically, an XML web service involves the following core technologies:

- A discovery service (so clients can resolve the location of the XML web service)
- A description service (so clients know what the XML web service can do)
- A transport protocol (to pass the information between the client and the XML web service)

We'll examine details behind each piece of infrastructure throughout this chapter. However, to get into the proper frame of mind, here is a brief overview of each supporting technology.

Previewing XML Web Service Discovery

Before a client can invoke the functionality of a web service, it must first know of its existence and location. Now, if you are the individual (or company) who is building the client and XML web service, the discovery phase is quite simple given that you already know the location of the web service in question. However, what if you wish to share the functionality of your web service with the world at large?

To do this, you have the option of registering your XML web service with a Universal Description, Discovery, and Integration (UDDI) server. Clients may submit request to a UDDI catalog to find a list of all web services that match some search criteria (e.g., "Find me all web services having to do real time weather updates"). Once you have identified a specific web server from the list returned via the UDDI query, you are then able to investigate its overall functionality. If you like, consider UDDI to be the white pages for XML web services.

In addition to UDDI discovery, an XML web service built using .NET can be located using DISCO, which is a somewhat forced acronym standing for *Discovery of Web Services*. Using static discovery (via a `*.disco` file) or dynamic discovery (via a `*.vsdisco` file), you are able to advertise the set of XML web services that are located at a specific URL. Potential web service clients can navigate to a web server's `*.disco` file to see links to all the published XML web services.

Understand, however, that dynamic discovery is disabled by default, given the potential security risk of allowing IIS to expose the set of all XML web services to any interested individual. Given this, I will not comment on DISCO services for the remainder of this text.

■ **Note** If you wish to activate dynamic discovery support for a given web server, look up the Microsoft Knowledge Base article Q307303 on `http://support.microsoft.com`.

Previewing XML Web Service Description

Once a client knows the location of a given XML web service, the client in question must fully understand the exposed functionality. For example, the client must know that there is a method named `GetWeatherReport()` that takes some set of parameters and sends back a given return value before the client can invoke the method. As you may be thinking, this is a job for a platform-, language-, and operating system–neutral metalanguage. Specifically speaking, the XML-based metadata used to describe a XML web service is termed the *Web Service Description Language* (*WSDL*).

In a good number of cases, the WSDL description of an XML web service will be automatically generated by Microsoft IIS when the incoming request has a `?wsdl` suffix. As you will see, the primary consumers of WSDL contracts are proxy generation tools. For example, the `wsdl.exe` command-line utility (explained in detail later in this chapter) will generate a client-side C# proxy class from a WSDL document.

For more complex cases (typically for the purposes of interoperability), many developers take a "WSDL first" approach and begin building their web services by defining the WSDL document manually. As luck would have it, the `wsdl.exe` command-line tool is also able to generate interface descriptions for an XML web service based on a WSDL definition.

Previewing the Transport Protocol

Once the client has created a proxy type to communicate with the XML web service, it is able to invoke the exposed web methods. As mentioned, HTTP is the wire protocol that transmits this data. Specifically, however, you can use HTTP `GET`, HTTP `POST`, or SOAP to move information between consumers and web services.

By and large, SOAP will be your first choice, for as you will see, SOAP messages can contain XML descriptions of complex types (including your custom types as well as types within the .NET base class libraries). On the other hand, if you make use of the HTTP `GET` or HTTP `POST` protocols, you are restricted to a more limited set of core data XML schema types.

The .NET XML Web Service Namespaces

Now that you have a basic understanding of XML web services, we can get down to the business of building such a creature using the .NET platform. As you would imagine, the base class libraries define a number of namespaces that allow you to interact with each web service technology (see Table 25-1).

Table 25-1. *XML Web Service–centric Namespaces*

Namespace	Meaning in Life
`System.Web.Services`	This namespace contains the core types needed to build an XML web service (including the all-important `[WebMethod]` attribute).
`System.Web.Services.Configuration`	These types allow you configure the runtime behavior of an ASP.NET XML web service.
`System.Web.Services.Description`	These types allow you to programmatically interact with the WSDL document that describes a given web service.
`System.Web.Services.Discovery`	These types allow a web consumer to programmatically discover the web services installed on a given machine.
`System.Web.Services.Protocols`	This namespace defines a number of types that represent the atoms of the various XML web service wire protocols (HTTP `GET`, HTTP `POST`, and SOAP).

■ **Note** All XML web service–centric namespaces are contained within the `System.Web.Services.dll` assembly.

Examining the System.Web.Services Namespace

Despite the rich functionality provided by the .NET XML web service namespaces, the vast majority of your applications will only require you to directly interact with the types defined in `System.Web.Services`. As you can see from Table 25-2, the number of types is quite small (which is a good thing).

Table 25-2. *Members of the* `System.Web.Services` *Namespace*

Type	Meaning in Life
`WebMethodAttribute`	Adding the `[WebMethod]` attribute to a method or property in a web service class type marks the member as invokable via HTTP and serializable as XML.
`WebService`	This is an optional base class for XML web services built using .NET. If you choose to derive from this base type, your XML web service will have the ability to retain stateful information (e.g., session and application variables).
`WebServiceAttribute`	The `[WebService]` attribute may be used to add information to a web service, such as a string describing its functionality and underlying XML namespace.
`WebServiceBindingAttribute`	This attribute (new .NET 2.0) declares the binding protocol a given web service method is implementing (HTTP `GET`, HTTP `POST`, or SOAP) and advertises the level of web services interoperability (WSI) conformity.
`WsiProfiles`	This enumeration (new to .NET 2.0) is used to describe the web services interoperability (WSI) specification to which a web service claims to conform.

The remaining namespaces shown in Table 25-1 are typically only of direct interest to you if you are interested in manually interacting with a WSDL document, discovery services, or the underlying wire protocols. Consult the .NET Framework 2.0 SDK documentation for further details.

Building an XML Web Service by Hand

Like any .NET application, XML web services can be developed manually, without the use of an IDE such as Visual Studio 2005. In an effort to demystify XML web services, let's build a simple XML web service by hand. Using your text editor of choice, create a new file named `HelloWorldWebService.asmx` (by convention, `*.asmx` is the extension used to mark .NET web service files). Save it to a convenient location on your hard drive (e.g., C:\HelloWebService) and enter the following type definition:

```
<%@ WebService Language="C#" Class="HelloWebService.HelloService" %>
using System;
using System.Web.Services;

namespace HelloWebService
{
    public class HelloService
    {
        [WebMethod]
        public string HelloWorld()
        {
            return "Hello!";
        }
    }
}
```

For the most part, this `*.asmx` file looks like any other C# namespace definition. The first noticeable difference is the use of the `<%@WebService%>` directive, which at minimum must specify the name of the managed language used to build the contained class definition and the fully qualified

name of the class. In addition to the `Language` and `Class` attributes, the `<%@WebService%>` directive may also take a `Debug` attribute (to inform the ASP.NET compiler to emit debugging symbols) and an optional `CodeBehind` value that identifies the associated code file within the optional App_Code directory (see Chapter 23). In this example, you have avoided the use of a code-behind file and embedded all required logic directly within a single `*.asmx` file.

Beyond the use of the `<%@WebService%>` directive, the only other distinguishing characteristic of this `*.asmx` file is the use of the `[WebMethod]` attribute, which informs the ASP.NET runtime that this method is reachable via incoming HTTP requests and should serialize any return value as XML.

■ **Note** Only members that are adorned with `[WebMethod]` are reachable by HTTP. Members not marked with the `[WebMethod]` attribute cannot be called by the client-side proxy.

Testing Your XML Web Service Using WebDev.WebServer.exe

Recall (again, from Chapter 23) that `WebDev.WebServer.exe` is a development ASP.NET web server that ships with the .NET platform 2.0 SDK. While `WebDev.WebServer.exe` would never be used to host a production-level XML web service, this tool does allow you to run web content directly from a local directory. To test your service using this tool, open a Visual Studio 2005 command prompt and specify an unused port number and physical path to the directory containing your `*.asmx` file:

```
WebDev.WebServer /port:4000 /path:"C:\HelloWebService"
```

Once the web server has started, open your browser of choice and specify the name of your `*.asmx` file exposed from the specified port:

```
http://localhost:4000/HelloWorldWebService.asmx
```

At this point, you are presented with a list of all web methods exposed from this URL (see Figure 25-2).

Figure 25-2. *Testing the XML web service*

If you click the HelloWorld link, you will be passed to another page that allows you to invoke the `[WebMethod]` you just selected. Once you invoke `HelloWorld()`, you will be returned not a literal .NET-centric `System.String`, but rather the XML data representation of the textual data returned from the `HelloWorld()` web method:

```
<?xml version="1.0" encoding="utf-8" ?>
<string xmlns="http://tempuri.org/">Hello!</string>
```

Testing Your Web Service Using IIS

Now that you have tested your XML web service using `WebDev.WebServer.exe`, you'll transfer your `*.asmx` file into an IIS virtual directory. Using the information presented in Chapter 23, create a new virtual directory named HelloWS that maps to the physical folder containing the `HelloWorldWebService.asmx` file. Once you do, you are able to test your web service by entering the following URL in your web browser:

```
http://localhost/HelloWS/HelloWorldWebService.asmx
```

Viewing the WSDL Contract

As mentioned, WSDL is a metalanguage that describes numerous characteristics of the web methods at a particular URL. Notice that when you test an XML web service, the autogenerated test page supplies a link named "Service Description." Clicking this link will append the token `?wsdl` to the current request. When the ASP.NET runtime receives a request for an `*.asmx` file tagged with this suffix, it will automatically return the underlying WSDL that describes each web method.

At this point, don't be alarmed with the verbose nature of WSDL or concern yourself with the format of a WSDL document. For the time being, just understand that WSDL describes how web methods can be invoked using each of the current XML web service wire protocols.

The Autogenerated Test Page

As you have just witnessed, XML web services can be tested within a web browser using an autogenerated HTML page. When an HTTP request comes in that maps to a given `*.asmx` file, the ASP.NET runtime makes use of a file named `DefaultWsdlHelpGenerator.aspx` to create an HTML display that allows you to invoke the web methods at a given URL. You can find this `*.aspx` file under the following directory (substitute `<version>` with your current version of the .NET Framework, of course):

```
C:\Windows\Microsoft.NET\Framework\<version>\CONFIG
```

Providing a Custom Test Page

If you wish to instruct the ASP.NET runtime to make use of a custom `*.aspx` file for the purposes of testing your XML web services, you are free to customize this page with additional information (add your company logo, additional descriptions of the service, links to a help document, etc.). To simplify matters, most developers copy the existing `DefaultWsdlHelpGenerator.aspx` to their current project as a starting point and modify the original HTML and C# code.

As a simple test, copy the `DefaultWsdlHelpGenerator.aspx` file into the directory containing `HelloWorldWebService.asmx` (e.g., C:\HelloWebService). Rename this copy to `MyCustomWsdlHelpGenerator.aspx` and update the some aspect of the HTML, such as the `<title>` tag. For example, change the following existing markup:

```
<title><%#ServiceName + " " + GetLocalizedText("WebService")%></title>
```

to the following:

```
<title>My Rocking <%#ServiceName + " " + GetLocalizedText("WebService")%></title>
```

Once you have modified the HTML content, create a `Web.config` file and save it to your current directory. The following XML elements instruct the runtime to make use of your custom `*.aspx` file, rather than `DefaultWsdlhelpGenerator.aspx`:

```
<!--  Here you are specifying a custom *.aspx file -->
<configuration>
  <system.web>
    <webServices>
      <wsdlHelpGenerator href="MyCustomWsdlHelpGenerator.aspx" />
    </webServices>
  </system.web>
</configuration>
```

When you request your web service, you should see that the browser's title has been updated with your custom content. On a related note, if you wish to disable help page generation for a given web service, you can do so using the following `<remove>` element within the `Web.config` file:

```
<!--  Disable help page generation -->
<configuration>
  <system.web>
    <webServices>
      <protocols>
        <!-- This element also disables WSDL generation -->
        <remove name="Documentation"/>
      </protocols>
    </webServices>
  </system.web>
</configuration>
```

■ **Source Code** The HelloWorldWebService files are included under the Chapter 25 subdirectory.

Building an XML Web Service Using Visual Studio 2005

Now that you have created an XML web service by hand, let's see how Visual Studio 2005 helps get you up and running. Using the File ➤ New ➤ Web Site menu option, create a new C# XML web service project named MagicEightBallWebService and save it to your local file system (see Figure 25-3).

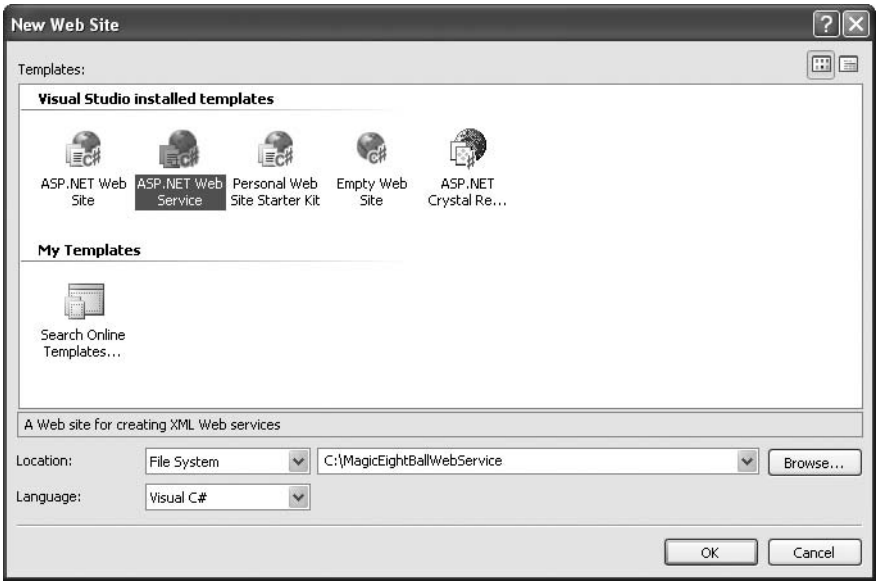

Figure 25-3. *Visual Studio 2005 XML Web Service project*

■ **Note** Like an ASP.NET website, XML web service projects created with Visual Studio 2005 place the `*.sln` file under My Documents\Visual Studio 2005\Projects.

Once you click the OK button, Visual Studio 2005 responds by generating a `Service.asmx` file that defines the following `<%@WebService%>` directive:

```
<%@ WebService Language="C#"
CodeBehind="~/App_Code/Service.cs" Class="Service" %>
```

Note that the `CodeBehind` attribute is used to specify the name of the C# code file (placed by default in your project's App_Code directory) that defines the related class type. By default, `Service.cs` is defined as so:

```
using System;
using System.Web;
using System.Web.Services;
using System.Web.Services.Protocols;

[WebService(Namespace = "http://tempuri.org/")]
[WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
public class Service : System.Web.Services.WebService
{
    public Service () {
    }

    [WebMethod]
    public string HelloWorld() {
        return "Hello World";
    }
}
```

Unlike the previous HelloWorldWebService example, notice that the `Service` class now derives from the `System.Web.Services.WebService` base class. You'll examine the members defined by this type in just a moment, but know for now that deriving from this base class is entirely optional.

Also notice that the `Service` class is adorned with two (also optional) attributes named `[WebService]` and `[WebServiceBinding]`. Again, you'll examine the role of these attributes a bit later in this chapter.

Implementing the TellFortune() Web Method

Your MagicEightBall XML web service will mimic the classic fortune-telling toy. To do so, add the following new method to your `Service` class (feel free to delete the existing `HelloWorld()` web method):

```
[WebMethod]
public string TellFortune(string userQuestion)
{
    string[] answers =  { "Future Uncertain", "Yes", "No",
        "Hazy", "Ask again later", "Definitely" };

    // Return a random response to the question.
    Random r = new Random();
    return string.Format("{0}? {1}",
        userQuestion, answers[r.Next(answers.Length)]);
}
```

To test your new XML web service, simply run (or debug) the project using Visual Studio 2005. Given that the `TellFortune()` method requires a single input parameter, the autogenerated HTML test page provides the required input field (see Figure 25-4).

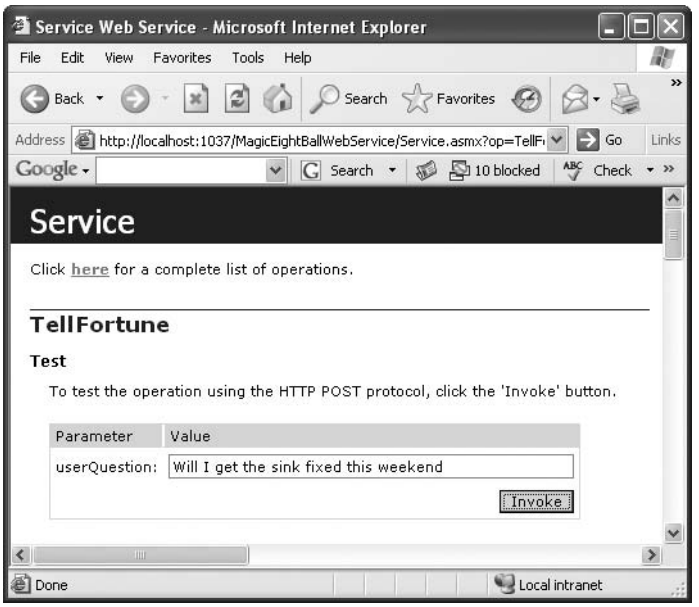

Figure 25-4. *Invoking the* `TellFortune()` *web method*

Here is a possible response to the question "Will I get the sink fixed this weekend":

```
<?xml version="1.0" encoding="utf-8" ?>
<string xmlns="http://tempuri.org/">
Will I get the sink fixed this weekend? Hazy
</string>
```

So, at this point you have created two simple XML web services: one by hand and the other using Visual Studio 2005. Now that you know the basics, we can dig into the specifics, beginning with the role of the `WebService` base class.

■ **Source Code** The MagicEightBallWebService files are included under the Chapter 25 subdirectory.

The Role of the WebService Base Class

As you saw during the development of the HelloWorldWebService service, a web service can derive directly from `System.Object`. However, by default, web services developed using Visual Studio 2005 automatically derive from the `System.Web.Service.WebService` base class. Table 25-3 documents the core members of this class type.

Table 25-3. *Key Members of the* `System.Web.Services.WebService` *Type*

Property	Meaning in Life
`Application`	Provides access to the `HttpApplicationState` object for the current HTTP request
`Context`	Provides access to the `HttpContext` type that encapsulates all HTTP-specific context used by the HTTP server to process web requests
`Server`	Provides access to the `HttpServerUtility` object for the current request
`Session`	Provides access to the `HttpSessionState` type for the current request
`SoapVersion`	Retrieves the version of the SOAP protocol used to make the SOAP request to the XML web service; new to .NET 2.0

As you may be able to gather, if you wish to build a *stateful* web service using application and session variables (see Chapter 24), you are required to derive from `WebService`, given that this type defines the `Application` and `Session` properties. On the other hand, if you are building an XML web service that does not require the ability to "remember" information about the external users, extending `WebService` is not required. We will revisit the process of building stateful XML web services during our examination of the `EnableSession` property of the `[WebMethod]` attribute.

Understanding the [WebService] Attribute

An XML web service class may optionally be qualified using the `[WebService]` attribute (not to be confused with the `WebService` base class). This attribute supports a few named properties, the first of which is `Namespace`. This property can be used to establish the name of the XML namespace to use within the WSDL document.

As you may already know, XML namespaces are used to scope custom XML elements within a specific group (just like .NET namespaces). By default, the ASP.NET runtime will assign a dummy XML namespace of `http://tempuri.org` for a given `*.asmx` file. As well, Visual Studio 2005 assigns the `Namespace` value to `http://tempuri.org` by default.

Assume you have created a new XML web service project with Visual Studio 2005 named CalculatorService that defines the following two web methods, named `Add()` and `Subtract()`:

```
[WebService(Namespace = "http://tempuri.org/")]
[WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
public class Service : System.Web.Services.WebService
{
    [WebMethod]
    public int Subtract(int x, int y) { return x - y; }

    [WebMethod]
    public int Add(int x, int y) { return x + y; }
}
```

Before you publish your XML web service to the world at large, you should supply a proper namespace that reflects the point of origin, which is typically the URL of the site hosting the XML web service. In the following code update, note that the `[WebService]` attribute also allows you to set a named property termed `Description` that describes the overall nature of your web service:

```
[WebService(Description = "The Amazing Calculator Web Service",
    Namespace ="http://www.IntertechTraining.com/")]
[WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
public class Service : System.Web.Services.WebService
{ ... }
```

The Effect of the Namespace and Description Properties

If you run the project, you will find that the warning to replace `http://tempuri.org` is no longer displayed in the autogenerated test page. Furthermore, if you click the Service Description link to view the underlying WSDL, you will find that the `TargetNamespace` attribute has now been updated with your custom XML namespace. Finally, the WSDL file now contains a `<documentation>` element that is based on your `Description` value:

```
<wsdl:documentation xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
    The Amazing Calculator Web Service
</wsdl:documentation>
```

As you might guess, it would be completely possible to build a custom utility that reads the value contained within the `<documentation>` element (e.g., an XML web service–centric object browser). In most cases, however, this value will be used by `DefaultWsdlHelpGenerator.aspx`.

The Name Property

The final property of the `WebServiceAttribute` type is `Name`, which is used to establish the name of the XML web service exposed to the outside world. By default, the external name of a web service is identical to the name of the class type itself (`Service` by default). However, if you wish to decouple the .NET class name from the underlying WSDL name, you can update the `[WebService]` attribute as follows:

```
[WebService(Description = "The Amazing Calculator Web Service",
    Namespace ="http://www.IntertechTraining.com/",
    Name = "CalculatorWebService")]
[WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
public class Service : System.Web.Services.WebService
{ ... }
```

Figure 25-5 shows the test page generated by `DefaultWsdlHelpGenerator.aspx` based on the `[WebService]` attribute.

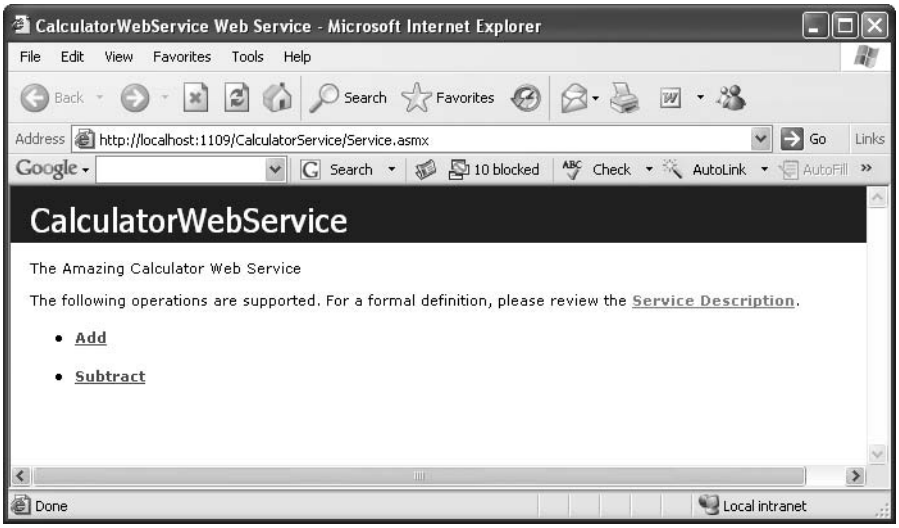

Figure 25-5. *The CalculatorWebService*

Understanding the [WebServiceBinding] Attribute

As of .NET 2.0, an XML web service can be attributed with `[WebServiceBinding]`. Among other things, this new attribute is used to specify if the XML web service conforms to "Web services interoperability (WSI) basic profile 1.1". So, what exactly does that mean? Well, if you have been actively working with XML web services, you may know firsthand that one of the frustrating aspects of this technology is that early on, WSDL was an evolving specification. Given this fact, it was not uncommon for the same WSDL element (or attribute) to be interpreted in different manners across development tools (IIS, WSAD), web servers (IIS, Apache), and architectures (.NET, J2EE).

Clearly this is problematic for an XML web service, as one of the motivating factors is to simplify the way in which information can be processed in a multiplatform, multi-architecture, and multilanguage universe. To rectify the problem, the WSI initiative offers a nonproprietary web services specification to promote the interoperability of web services across platforms. Under .NET 2.0, the `ConformsTo` property of `[WebServiceBinding]` can be set to any value of the `WsiProfiles` enumeration:

```
public enum WsiProfiles
{
    // The web service makes no conformance claims.
    None,
    // The web service claims to conform to the
    // WSI Basic Profile version 1.1.
    BasicProfile1_1
}
```

By default, XML web services generated using Visual Studio 2005 are assumed to conform to the WSI basic profile 1.1. Of course, simply setting the `ConformsTo` named property to `WsiProfiles.BasicProfile1_1` does not guarantee each web method is truly compliant. For example, one rule of BP 1.1 states that every method in a WSDL document must have a unique name (overloading of exposed web methods is not permitted under BP 1.1). The good news is that the ASP.NET runtime is able to determine various BP 1.1 validations and will report the issue at runtime.

Ignoring BP 1.1 Conformance Verification

As of .NET 2.0, XML web services are automatically checked against the WSI basic profile (BP) 1.1. In most cases, this is a good thing, given that you are able to build software that has the greatest reach as possible. In some cases, however, you may wish to ignore BP 1.1 conformance (e.g., if you are building in-house XML web services where interoperability is not much of an issue). To instruct the runtime to ignore BP 1.1 violations, set the `ConformsTo` property to `WsiProfiles.None` and the `EmitConformanceClaims` property to false:

```
[WebService(Description = "The Amazing Calculator Web Service",
    Namespace ="http://www.IntertechTraining.com/",
    Name = "CalculatorWebService")]
[WebServiceBinding(ConformsTo = WsiProfiles.None ,
    EmitConformanceClaims = false)]
public class Service : System.Web.Services.WebService
{...}
```

As you might suspect, the value assigned to `EmitConformanceClaims` controls whether the conformance claims expressed by the `ConformsTo` property are provided when a WSDL description of the web service is published. With this, BP 1.1 violations will be permitted, although the auto-generated test page will still display warnings.

Disabling BP 1.1 Conformance Verification

If you wish to completely disable BP 1.1 verification for your XML web service, you may do so by defining the following `<conformanceWarnings>` element within a proper `Web.config` file:

```
<configuration>
  <system.web>
    <webServices>
      <conformanceWarnings>
        <remove name='BasicProfile1_1'/>
      </conformanceWarnings>
    </webServices>
  </system.web>
</configuration>
```

■ **Note** The `[WebServiceBinding]` attribute can also be used to define the intended binding for specific methods via the `Name` property. Consult the .NET Framework 2.0 SDK documentation for further details.

Understanding the [WebMethod] Attribute

The `[WebMethod]` attribute must be applied to each method you wish to expose from an XML web service. Like most attributes, the `WebMethodAttribute` type may take a number of optional named properties. Let's walk through each possibility in turn.

Documenting a Web Method via the Description Property

Like the `[WebService]` attribute, the `Description` property of the `[WebMethod]` attribute allows you to describe the functionality of a particular web method:

```
public class Service : System.Web.Services.WebService
{
    [WebMethod(Description = "Subtracts two integers.")]
    public int Subtract(int x, int y) { return x - y; }

    [WebMethod(Description = "Adds two integers.")]
    public int Add(int x, int y) { return x + y; }
}
```

Under the hood, when you specify the `Description` property within a `[WebMethod]` attribute, the WSDL contract is updated with a new `<documentation>` element scoped at the method name level:

```
<wsdl:operation name="Add">
  <wsdl:documentation xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
    Adds two integers.
  </wsdl:documentation>
  <wsdl:input message="tns:AddSoapIn" />
  <wsdl:output message="tns:AddSoapOut" />
</wsdl:operation>
```

Avoiding WSDL Name Clashes via the MessageName Property

One of the rules of WSI BP 1.1 is that each method within a WSDL document must be unique. Therefore, if you wish your XML web services to conform to BP 1.1, you should not overload methods in your implementation logic. For the sake of argument, however, assume that you have overloaded the `Add()` method so that the caller can pass two integer or float data types. You would find the following runtime error:

```
Both Single Add(Single, Single) and Int32 Add(Int32, Int32)
use the message name 'Add'. Use the MessageName property
attribute to specify unique of the WebMethod
custom message names for the methods.
```

Again, the best approach is to simply not overload the `Add()` method in the first place. If you must do so, the `MessageName` property of the `[WebMethod]` attribute can be used to resolve name clashes in your WSDL documents:

```
public class Service : System.Web.Services.WebService
{
...
    [WebMethod(Description = "Adds two float.",
        MessageName = "AddFloats")]
    public float Add(float x, float y) { return x + y; }

    [WebMethod(Description = "Adds two integers.",
        MessageName = "AddInts")]
    public int Add(int x, int y) { return x + y; }
}
```

Once you have done so, the generated WSDL document will internally refer to each overloaded version of `Add()` uniquely (`AddFloats` and `AddInts`). As far as the client-side proxy is concerned, however, there is only a single overloaded `Add()` method.

Building Stateful Web Services via the EnableSession Property

As you may recall from Chapter 24, the `Application` and `Session` properties allow an ASP.NET web application to maintain stateful data. XML web services gain the exact same functionality via the

System.Web.Services.WebService base class. For example, assume your CalculatorService maintains an application-level variable (and is thus available to each session) that holds the value of PI, as shown here:

```
public class CalcWebService: System.Web.Services.WebService
{
     // This web method provides access to an app-level variable
     // named SimplePI.
     [WebMethod(Description = "Get the simple value of PI.")]
     public float GetSimplePI()
     {  return (float)Application["SimplePI"]; }
     ...
}
```

The initial value of the SimplePI application variable could be established with the Application_Start() event handler defined in the Global.asax file. Insert a new global application class to your project (by right-clicking your project icon within Solution Explorer and selecting Add New Item) and implement Application_Start() as so:

```
<%@ Application Language="C#" %>
<script runat="server">
    void Application_Start(Object sender, EventArgs e)
    {
        Application["SimplePI"] = 3.14F;
    }
...
</script>
```

In addition to maintaining applicationwide variables, you may also make use of Session to maintain session-centric information. For the sake of illustration, implement the Session_Start() method in your Global.asax to assign a random number to each user who is logged on:

```
<%@ Application Language="C#" %>
<script runat="server">
...
    void Session_Start(Object sender, EventArgs e)
    {
        // To prove session state data is available from a web service,
        // simply assign a random number to each user.
        Random r = new Random();
        Session["SessionRandomNumber"] = r.Next(1000);
    }
...
</script>
```

For testing purposes, create a new web method in your Service class that returns the user's randomly assigned value:

```
public class Service : System.Web.Services.WebService
{
...
    [WebMethod(EnableSession = true,
     Description = "Get your random number!")]
    public int GetMyRandomNumber()
    { return (int)Session["SessionRandomNumber"]; }
}
```

Note that the [WebMethod] attribute has explicitly set the EnableSession property to true. This step is not optional, given that by default each web method has session state *disabled*. If you were

now to launch two or three browsers (to generate a set of session IDs), you would find that each logged-on user is returned a unique numerical token. For example, the first caller may receive the following XML:

```
<?xml version="1.0" encoding="utf-8" ?>
<int xmlns="http://www.IntertechTraining.com/WebServers">931</int>
```

while the second caller may find her value is 472:

```
<?xml version="1.0" encoding="utf-8" ?>
<int xmlns="http://www.IntertechTraining.com/WebServers">472</int>
```

Configuring Session State via Web.config

Finally, recall that a `Web.config` file may be updated to specify where state should be stored for the XML web service using the `<sessionState>` element (described in the previous chapter).

```
<sessionState
     mode="InProc"
     stateConnectionString="tcpip=127.0.0.1:42424"
     sqlConnectionString="data source=127.0.0.1;Trusted_Connection=yes"
     cookieless="false"
     timeout="20"
/>
```

■ **Source Code** The CalculatorService files are included under the Chapter 25 subdirectory.

Exploring the Web Service Description Language (WSDL)

Over the last several examples, you have been exposed to partial WSDL snippets. Recall that WSDL is an XML-based grammar that describes how external clients can interact with the web methods at a given URL, using each of the supported wire protocols. In many ways, a WSDL document can be viewed as a contract between the web service client and the web service itself. To this end, it is yet another metalanguage. Specifically, WSDL is used to describe the following characteristics for each exposed web method:

- The name of the XML web methods
- The number of, type of, and ordering of parameters (if any)
- The type of return value (if any)
- The HTTP `GET`, HTTP `POST`, and SOAP calling conventions

In most cases, WSDL documents are generated automatically by the hosting web server. Recall that when you append the `?wsdl` suffix to a URL that points to an `*.asmx` file, the hosting web server will emit the WSDL document for the specified XML web service:

```
http://localhost/SomeWS/theWS.asmx?wsdl
```

Given that IIS will automatically generate WSDL for a given XML web service, you may wonder if you are required to deeply understand the syntax of the generated WSDL data. The answer typically depends on how your service is to be consumed by external applications. For in-house XML web services, the WSDL generated by your XML web server will be sufficient most of the time.

However, it is also possible to begin an XML web service project by authoring the WSDL document by hand (as mentioned earlier, this is termed the *WSDL first* approach). The biggest selling point for WSDL first has to do with interoperability concerns. Recall that prior to the WSI specification, it was not uncommon for various web service tools to generate incompatible WSDL descriptions. If you take a WSDL first approach, you can craft the document as required.

As you might imagine, taking a WSDL first approach would require you to have a very intimate view of the WSDL grammar, which is beyond the scope of this chapter. Nevertheless, let's get to know the basic structure of a valid WSDL document. Once you understand the basics, you'll better understand the usefulness of the `wsdl.exe` command-line utility.

■ **Note** To see the most recent information on WSDL, visit `http://www.w3.org/tr/wsdl`.

Defining a WSDL Document

A valid WSDL document is opened and closed using the root `<definitions>` element. The opening tag typically defines various `xmlns` attributes. These qualify the XML namespaces that define various subelements. At a minimum, the `<definitions>` element will specify the namespace where the WSDL elements themselves are defined (`http://schemas.xmlsoap.org/wsdl`). To be useful, the opening `<definitions>` tag will also specify numerous XML namespaces that define simple data WSDL types, XML schema types, SOAP elements, and the target namespace. For example, here is the `<definitions>` section for CalculatorService:

```
<?xml version="1.0" encoding="utf-8"?>
<wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
xmlns:tns="http://www.IntertechTraining.com/"
xmlns:s="http://www.w3.org/2001/XMLSchema"
xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
targetNamespace="http://www.IntertechTraining.com/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
...
</wsdl:definitions>
```

Within the scope of the root element, you will find five possible subelements. Thus, a bare-bones WSDL document would look something like the following:

```
<?xml version="1.0" encoding="utf-8"?>
<wsdl:definitions ...>
     <wsdl:types>
          <!-- List of types exposed from WS ->
     <wsdl:/types>
     <wsdl:message>
          <!-- Format of the messages ->
     <wsdl:/message>
     <wsdl:portType>
          <!-- Port information ->
     <wsdl:/portType>
     <wsdl:binding>
          <!-- Binding information ->
     <wsdl:/binding>
     <wsdl:service>
          <!-- Information about the XML web service itself ->
```

```
     <wsdl:/service>
< wsdl:/definitions>
```

As you would guess, each of these subelements will contain additional elements and attributes to further describe the intended functionality. Let's check out the key nodes in turn.

The <types> Element

First, we have the `<types>` element, which contains descriptions of any and all data types exposed from the web service. As you may know, XML itself defines a number of "core" data types, all of which are defined within the XML namespace: `http://www.w3.org/2001/XMLSchema` (which appears in your `<definitions>` root element). For example, recall the `Subtract()` method of CalculatorService took two integer parameters. In terms of WSDL, the CLR `System.Int32` is described within a `<complexType>` element:

```
<s:element name="Subtract">
  <s:complexType>
    <s:sequence>
      <s:element minOccurs="1" maxOccurs="1" name="x" type="s:int" />
      <s:element minOccurs="1" maxOccurs="1" name="y" type="s:int" />
    </s:sequence>
  </s:complexType>
</s:element>
```

The integer that is returned from the `Subtract()` method is also described within the `<types>` element:

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

If you have a web method that returns or receives custom data types, they will also appear within a `<complexType>` element. You will see the details of how to expose custom .NET data types via a given web method a bit later in this chapter. For the sake of illustration, assume you have defined a web method that returns a structure named `Point`:

```
public struct Point
{
     public int x;
     public int y;
     public string pointName;
}
```

The WSDL description of this "complex type" would look like the following:

```
<s:complexType name="Point">
  <s:sequence>
    <s:element minOccurs="1" maxOccurs="1" name="x" type="s:int" />
    <s:element minOccurs="1" maxOccurs="1" name="y" type="s:int" />
    <s:element minOccurs="0" maxOccurs="1" name="pointName" type="s:string" />
  </s:sequence>
</s:complexType>
```

The <message> Element

The <message> element is used to define the format of the request and response exchange for a given web method. Given that a single web service allows multiple messages to be transmitted between the sender and receiver, it is permissible for a single WSDL document to define multiple <message> elements. Typically, these message definitions use the types defined in the <types> element.

Regardless of how many <message> elements are defined within a WSDL document, they tend to occur in pairs. The first definition represents the input-centric format of the message, while the second defines the output-centric format of the same message. For example, the Subtract() method of CalculatorService is defined by the following <message> element:

```
<wsdl:message name="SubtractSoapIn">
  <wsdl:part name="parameters" element="tns:Subtract" />
</wsdl:message>
<wsdl:message name="SubtractSoapOut">
  <wsdl:part name="parameters" element="tns:SubtractResponse" />
</wsdl:message>
```

Here, you are only viewing the SOAP binding of the service. As you may recall from the beginning of this chapter, XML web services can be invoked via SOAP, HTTP GET, and HTTP POST. Thus, if you were to enable HTTP POST bindings (explained later), the generated WSDL would also show the following <message> data:

```
<wsdl:message name="SubtractHttpPostIn">
  <part name="n1" type="s:string" />
  <part name="n2" type="s:string" />
<wsdl:/message>
<wsdl:message name="SubtractHttpPostOut">
  <part name="Body" element="s0:int" />
<wsdl:/message>
```

In reality, <message> elements are not all that useful in and of themselves. However, these message definitions are referenced by other aspects of a WSDL document.

■ **Note** Not all web methods require both a request and response. If a web method is a one-way method, then only a request <message> element is necessary. You can mark a web method as a one-way method by applying the [SoapDocumentMethod] attribute.

The <portType> Element

The <portType> element defines the characteristics of the various correspondences that can occur between the client and server, each of which is represented by an <operation> subelement. As you might guess, the most common operations would be SOAP, HTTP GET, and HTTP POST. Additional operations do exist, however. For example, the one-way operation allows a client to send a message to a given web server but does not receive a response (sort of a fire-and-forget method invocation). The solicit/response operation allows the server to issue a request while the client responds (which is the exact opposite of the request/response operation).

To illustrate the format of a possible <operation> subelement, here is the WSDL definition for the Subtract() method:

```
<wsdl:portType name="CalculatorWebServiceSoap">
  <wsdl:operation name="Subtract">
    <wsdl:input message="tns:SubtractSoapIn" />
    <wsdl:output message="tns:SubtractSoapOut" />
  </wsdl:operation>
<wsdl:/portType>
```

Note how the <input> and <output> elements make reference to the related message name defined within the <message> element. If HTTP POST were enabled for the Subtract() method, you would find the following additional <operation> element:

```
<wsdl:portType name="CalculatorWebServiceHttpPost">
  <wsdl:operation name="Subtract">
    <wsdl:input message="s0:SubtractHttpPostIn" />
    <wsdl:output message="s0:SubtractHttpPostOut" />
  <wsdl:/operation>
<wsdl:/portType>
```

Finally, be aware that if a given web method has been described using the Description property, the <operation> element will contain an embedded <documentation> element.

The <binding> Element

This element specifies the exact format of the HTTP GET, HTTP POST, and SOAP exchanges. By far and away, this is the most verbose of all the subelements contained in the <definition> root. For example, here is the <binding> element definition that describes how a caller may interact with the MyMethod() web method using SOAP:

```
<wsdl:binding name="CalculatorWebServiceSoap12"
  type="tns:CalculatorWebServiceSoap">
  <soap12:binding transport="http://schemas.xmlsoap.org/soap/http" />
  <wsdl:operation name="Subtract">
    <soap12:operation soapAction="http://www.IntertechTraining.com/Subtract"
      style="document" />
    <wsdl:input>
      <soap12:body use="literal" />
    </wsdl:input>
    <wsdl:output>
      <soap12:body use="literal" />
    </wsdl:output>
  </wsdl:operation>
</wsdl:binding>
```

The <service> Element

Finally we have the <service> element, which specifies the characteristics of the web service itself (such as its URL). The chief duty of this element is to describe the set of ports exposed from a given web server. To do so, the <services> element makes use of any number of <port> subelements (not to be confused with the <portType> element). Here is the <service> element for CalculatorService:

```
<wsdl:service name="CalculatorWebService">
  <wsdl:documentation xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
    The Amazing Calculator Web Service
  </wsdl:documentation>
  <wsdl:port name="CalculatorWebServiceSoap"
    binding="tns:CalculatorWebServiceSoap">
    <soap:address location="http://localhost:1109/CalculatorService/Service.asmx" />
  </wsdl:port>
  <wsdl:port name="CalculatorWebServiceSoap12"
    binding="tns:CalculatorWebServiceSoap12">
    <soap12:address location=
      "http://localhost:1109/CalculatorService/Service.asmx" />
  </wsdl:port>
</wsdl:service>
```

So, as you can see, the WSDL automatically returned by IIS is not rocket science, but given that WSDL is an XML-based grammar, it is a bit on the verbose side. Nevertheless, now that you have a better understanding of WSDL's place in the world, let's dig a bit deeper into the XML web service wire protocols.

■ **Note** Recall that the `System.Web.Services.Description` namespace contains a plethora of types that allow you to programmatically read and manipulate raw WSDL (so check it out if you are so interested).

Revisiting the XML Web Service Wire Protocols

Technically, XML web services can use any RPC protocol to facilitate communication (such as DCOM or CORBA). However, most web servers bundle this data into the body of an HTTP request and transmits it to the consumer using one of three core bindings (see Table 25-4).

Table 25-4. *XML Web Service Bindings*

Transmission Binding	Meaning in Life
HTTP `GET`	`GET` submissions append parameters to the query string of the URL.
HTTP `POST`	`POST` transmissions embed the data points into the header of the HTTP message rather than appending them to the query string.
SOAP	SOAP is a wire protocol that specifies how to submit data and invoke methods across the wire using XML.

While each approach leads to the same result (invoking a web method), your choice of wire protocol determines the types of parameters (and return types) that can be sent between each interested party. The SOAP protocol offers you the greatest flexibility, given that SOAP messages allow you to pass complex data types (as well as binary files) between the caller and XML web service. However, for completeness, let's check out the role of standard HTTP `GET` and `POST`.

HTTP GET and HTTP POST Bindings

Although `GET` and `POST` verbs may be familiar constructs, you must be aware that this method of transportation is not rich enough to represent such complex items as structures or classes. When you use `GET` and `POST` verbs, you can interact with web methods using only the types listed in Table 25-5.

Table 25-5. *Supported* `POST` *and* `GET` *Data Types*

Data Types	Meaning in Life
Enumerations	`GET` and `POST` verbs support the transmission of .NET `System.Enum` types, given that these types are represented as a static constant string.
Simple arrays	You can construct arrays of any primitive type.
Strings	`GET` and `POST` transmit all numerical data as a string token. *String* really refers to the string representation of CLR primitives such as `Int16`, `Int32`, `Int64`, `Boolean`, `Single`, `Double`, `Decimal`, and so forth.

By default, HTTP `GET` and HTTP `POST` bindings are not enabled for remote XML web service invocation. However, HTTP `POST` is enabled to allow a machine to invoke local web services (in fact, this is exactly what the autogenerated help page is leveraging behind the scenes). These settings are established in the `machine.config` file using the `<protocols>` element. Here is a partial snapshot:

```
<!--  In the machine.config file! -->
<webServices>
  <protocols>
    <add name="HttpSoap1.2" />
    <add name="HttpSoap" />
    <add name="Documentation" />
    <!-- HTTP GET/POST disabled! -->
    <!-- <add name="HttpPost"/> -->
    <!-- <add name="HttpGet"/> -->
    <!-- Used by the web service test page -->
    <add name="HttpPostLocalhost" />
  </protocols>
</webServices>
```

To re-enable HTTP `GET` or HTTP `POST` for a given web service, explicitly add in the `HttpPost` and `HttpGet` names within a local `Web.config` file:

```
<configuration>
  <system.web>
    <webServices>
      <protocols>
        <add name="HttpPost"/>
        <add name="HttpGet"/>
      </protocols>
    </webServices>
  </system.web>
</configuration>
```

Again, recall that if you make use of standard HTTP `GET` or HTTP `POST`, you are not able to build web methods that take complex types as parameters or return values (e.g., an ADO.NET `DataSet` or custom structure type). For simple web services, this limitation may be acceptable. However, if you make use of SOAP bindings, you are able to build much more elaborate XML web services.

SOAP Bindings

Although a complete examination of SOAP is beyond the scope of this text, understand that SOAP itself does not define a specific protocol and can thus be used with any number of existing Internet protocols (HTTP, SMTP, and others). The general role of SOAP, however, remains the same: provide a mechanism to invoke methods using complex types in a language- and platform-neutral manner. To do so, SOAP encodes each complex method with a SOAP message.

A SOAP message defines two core sections. First, we have the SOAP envelope, which can be understood as the conceptual container for the relevant information. Second, we have the rules that are used to describe the information in said message (placed into the SOAP body). An optional third section (the SOAP header) may be used to specify general information regarding the message itself, such as security or transactional information.

```
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
     <soap:Header>
          <!-- Optional header information -->
     </soap:Header>
     <soap:Body>
          <!-- Method invocation information -->
     </soap:Body>
</soap:Envelope>
```

Viewing a SOAP Message

Although you are not required to understand the gory details of SOAP to build XML web services with the .NET platform, you are able to view the format of the SOAP message for each exposed web method using the autogenerated test page. For example, if you were to click the link for the `Add()` method of CalculatorService, you would find the following SOAP 1.1 request:

```
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <Add xmlns="http://www.IntertechTraining.com ">
      <x>int</x>
      <y>int</y>
    </Add>
  </soap:Body>
</soap:Envelope>
```

The corresponding SOAP 1.1 response looks like this:

```
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <AddResponse xmlns="http://www.IntertechTraining.com ">
      <AddResult>int</AddResult>
    </AddResponse>
  </soap:Body>
</soap:Envelope>
```

The wsdl.exe Command-Line Utility

Now that you've completed a primer on WSDL and SOAP, let's begin to examine how to build client programs that communicate with remote XML web services using the `wsdl.exe` command-line tool. In a nutshell, `wsdl.exe` performs two major tasks:

- Generates a server-side file that functions as a skeleton for implementing an XML web service
- Generates a client-side file that functions as the proxy to a remote XML web service

`wsdl.exe` supports a number of command-line flags, all of which can be viewed at the command prompt by specifying the `-?` option. Table 25-6 points out some of the more common arguments.

Table 25-6. *Select Options of* `wsdl.exe`

Command-Line Flag	Meaning in Life
`/appsettingurlkey`	Instructs `wsdl.exe` to build a proxy that does not make use of hard-coded URLs. Instead, the proxy class will be configured to read the URL from a client-side `*.config` file.
`/language`	Specifies the language to use for the generated proxy class: `CS` (C#; default) `VB` (Visual Basic .NET) `JS` (JScript) `VJS` (Visual J#) The default is C#.
`/namespace`	Specifies the namespace for the generated proxy or template. By default, your type will not be defined within a namespace definition.

Command-Line Flag	Meaning in Life
`/out`	Specifies the file in which to save the generated proxy code. If the file is not specified, the file name is based on the XML web service name.
`/protocol`	Specifies the protocol to use within the proxy code; SOAP is the default. However, you can also specify `HttpGet` or `HttpPost` to create a proxy that communicates using simple HTTP `GET` or `POST` verbs.
`/serverInterface`	Generates server-side interface bindings for an XML web service based on the WSDL document.

■ **Note** The `/server` flag of `wsdl.exe` has been deprecated under .NET 2.0. `/serverInterface` is now the preferred method to generate server-side skeleton code.

Transforming WSDL into a Server-Side XML Web Service Skeleton

One interesting use of the `wsdl.exe` utility is to generate server-side skeleton code (via the `/serverInterface` option) based on a WSDL document. Clearly, if you are interested in taking a WSDL first approach to building XML web services, this would be a very important option. Once this source code file has been generated, you have a solid starting point to provide the actual implementation of each web method.

Assume you have created WSDL document (`CarBizObject.wsdl`) that describes a single method named `DeleteCar()` that takes a single integer as input and returns nothing. This method is exposed from an XML web service named CarBizObject, which can be invoked using SOAP bindings.

To generate a server-side C# code file from this WSDL document, open a .NET-aware command window and specify the `/serverInterface` flag, followed by the name of the WSDL document you wish to process. Note that the WSDL document may be contained in a local `*.wsdl` file:

```
wsdl /serverInterface CarBizObject.wsdl
```

or it can be obtained dynamically from a given URL via the `?wsdl` suffix:

```
wsdl /serverInterface http://localhost/CarService/CarBizObject.asmx?wsdl
```

Once `wsdl.exe` has processed the XML elements, you are presented with interface descriptions for each web method:

```
[System.Web.Services.WebServiceBindingAttribute(Name="CarBizObjectSoap",
  Namespace="http://IntertechTraining.com/")]
public partial interface ICarBizObjectSoap
{
...
    void RemoveCar(int carID);
}
```

Using these interfaces, you can define a class that implements the various methods of the XML web service.

■ **Source Code** The `CarBizObject.wsdl` file is included under the Chapter 25 subdirectory.

Transforming WSDL into a Client-Side Proxy

Although undesirable, it is completely possible to construct a client-side code base that manually opens an HTTP connection, builds the SOAP message, invokes the web method, and translates the incoming stream of XML back into CTS data types. A much-preferred approach is to leverage `wsdl.exe` to generate a proxy class that maps to the web methods defined by a given `*.asmx` file.

To do so, you will specify (at a minimum) the name of the proxy file to be generated (via the `/out` flag) and the location of the WSDL document. By default, `wsdl.exe` will generate proxy code written in C#. However, if you wish to obtain proxy code in an alternative .NET language, make use of the `/language` flag. You should also be aware that by default, `wsdl.exe` generates a proxy that communicates with the remote XML web service using SOAP bindings. If you wish to build a proxy that leverages straight HTTP `GET` or HTTP `POST`, you may make use of the `/protocol` flag.

Another important point to be made regarding generating proxy code via `wsdl.exe` is that this tool truly needs the *WSDL* of the XML web service, not simply the name of the `*.asmx` file. Given this, understand that if you make use of `WebDev.WebServer.exe` to develop and test your services, you will most likely want to copy your project's content to an IIS virtual directory before generating a client-side proxy.

For the sake of illustration, assume that you have created a new IIS virtual directory (CalcService), which contains the content for the CalculatorService project. Once you have done so, you can generate the client proxy code as so:

```
wsdl /out:proxy.cs http://localhost/CalcService/Service.asmx?wsdl
```

As a side note, be aware that `wsdl.exe` will not define a .NET namespace to wrap the generated C# types unless you specify the `/n` flag at the command prompt:

```
wsdl /out:proxy.cs /n:CalculatorClient
    http://localhost/CalcService/Service.asmx?wsdl
```

Examining the Proxy Code

If you open up the generated proxy file, you'll find a type that derives from `System.Web.Services.Protocols.SoapHttpClientProtocol` (unless, of course, you specified an alternative binding via the `/protocols` option):

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

This base class defines a number of members leveraged within the implementation of the proxy type. Table 25-7 describes some (but not all) of these members.

Table 25-7. *Core Members of the* `SoapHttpClientProtocol` *Type*

Inherited Members	Meaning in Life
`BeginInvoke()`	This method starts an asynchronous invocation of the web method.
`CancelAsync()`	This method (new to .NET 2.0) cancels an asynchronous call to an XML web service method, unless the call has already completed.
`EndInvoke()`	This method ends an asynchronous invocation of the web method.
`Invoke()`	This method synchronously invokes a method of the web service.

Inherited Members	Meaning in Life
`InvokeAsync()`	This method (new to .NET 2.0) is the preferred way to asynchronously invoke a method of the web service.
`Proxy`	This property gets or sets proxy information for making a web service request through a firewall.
`Timeout`	This property gets or sets the timeout (in milliseconds) used for synchronous calls.
`Url`	This property gets or sets the base URL to the server to use for requests.
`UserAgent`	This property gets or sets the value for the user agent header sent with each request.

The Default Constructor

The default constructor of the proxy hard-codes the URL of the remote web service and stores it in the inherited `Url` property:

```
public CalculatorWebService()
{
    this.Url = "http://localhost/CalcService/Service.asmx";
}
```

The obvious drawback to this situation is that if the XML web service is renamed or relocated, the proxy class must be updated and recompiled. To build a more flexible proxy type, `wsdl.exe` provides the `/appsettingurlkey` flag (which may be abbreviated to `/urlkey`). When you specify this flag at the command line, the proxy's constructor will contain logic that reads the URL using a key contained within a client-side `*.config` file.

```
wsdl /out:proxy.cs /n:CalcClient /urlkey:CalcUrl
    http://localhost/CalcService/Service.asmx?wsdl
```

If you now check out the default constructor of the proxy, you will find the following logic (note that if the correct key cannot be found, the hard-coded URL will be used as a backup):

```
public CalculatorWebService()
{
  string urlSetting =
    System.Configuration.ConfigurationManager.AppSettings["CalcUrl"];
    if ((urlSetting != null))
    {
        this.Url = urlSetting;
    }
    else
    {
        this.Url = "http://localhost/CalcService/Service.asmx";
    }
}
```

The corresponding client-side `app.config` file will look like this:

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

Synchronous Invocation Support

The generated proxy also defines synchronous support for each web method. For example, the synchronous implementation of the `Subtract()` method is implemented as so:

```
public int Subtract(int x, int y)
{
    object[] results = this.Invoke("Subtract", new object[] {x, y});
    return ((int)(results[0]));
}
```

Notice that the caller passes in two `System.Int32` parameters that are packaged as an array of `System.Objects`. Using late binding, the `Invoke()` method will pass these arguments to the Subtract method located at the established URL. Once this (blocking) call completes, the incoming XML is processed, and the result is cast back to the caller as `System.Int32`.

Asynchronous Invocation Support

Support for invoking a given web method asynchronously has changed quite a bit from .NET 1.*x*. As you might recall from previous experience, .NET 1.1 proxies made use of `BeginXXX()/EndXXX()` methods to invoke a web method on a secondary thread of execution. For example, consider the following `BeginSubtract()` and `EndSubtract()` methods:

```
public System.IAsyncResult BeginSubtract(int x, int y,
    System.AsyncCallback callback, object asyncState)
{
    return this.BeginInvoke("Subtract", new object[] {x, y},
    callback, asyncState);
}
public int EndSubtract(System.IAsyncResult asyncResult)
{
    object[] results = this.EndInvoke(asyncResult);
    return ((int)(results[0]));
}
```

While `wsdl.exe` still generates these familiar `Begin/End` methods, under .NET 2.0 they have been deprecated and are replaced by the new `XXXAsync()` methods:

```
public void SubtractAsync(int x, int y)
{
    this.SubtractAsync(x, y, null);
}
```

These new `XXXAsync()` methods (as well as a related `CancelAsync()` method) work in conjunction with an autogenerated helper method (being an overloaded version of a specific `XXXAsync()` method) which handles the asynchronous operation using C# event syntax. If you examine the proxy code, you will see that `wsdl.exe` has generated (for each web method) a custom delegate, custom event, and custom "event args" class to obtain the result.

Building the Client Application

Now that you better understand the internal composition of the generated proxy, let's put it to use. Create a new console application named CalculatorClient, insert your `proxy.cs` file into the project using Project ➤ Add Existing Item, and add a reference to the `System.Web.Services.dll` assembly. Next, update your `Main()` method as so:

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Fun with WS Proxies *****\n");
        // Make the proxy.
        CalculatorWebService ws = new CalculatorWebService();

        // Call the Add() method synchronously.
        Console.WriteLine("10 + 10 = {0}", ws.Add(10, 10));

        // Call the Subtract method asynchronously
        // using the new .NET 2.0 event approach.
        ws.SubtractCompleted += new
            SubtractCompletedEventHandler(ws_SubtractCompleted);
        ws.SubtractAsync(50, 45);

        // Keep console running to make sure we get our subtraction result.
        Console.ReadLine();
    }

    static void ws_SubtractCompleted(object sender, SubtractCompletedEventArgs e)
    {
        Console.WriteLine("Your answer is: {0}", e.Result);
    }
}
```

Notice that the new .NET 2.0 asynchronous invocation logic does indeed directly map to the C# event syntax, which as you might agree is cleaner than needing to work with `BeginXXX()/EndXXX()` method calls, the `IAsyncResult` interface, and the `AsyncCallback` delegate.

Source Code The CalculatorClient project can be found under the Chapter 25 subdirectory.

Generating Proxy Code Using Visual Studio 2005

Although `wsdl.exe` provides a number of command-line arguments that give you ultimate control over how a proxy class will be generated, Visual Studio 2005 also allows you to quickly generate a proxy file using the Add Web Reference dialog box (which you can activated from the Project menu). As you can see from Figure 25-6, you are able to obtain references to existing XML web services located in a variety of places.

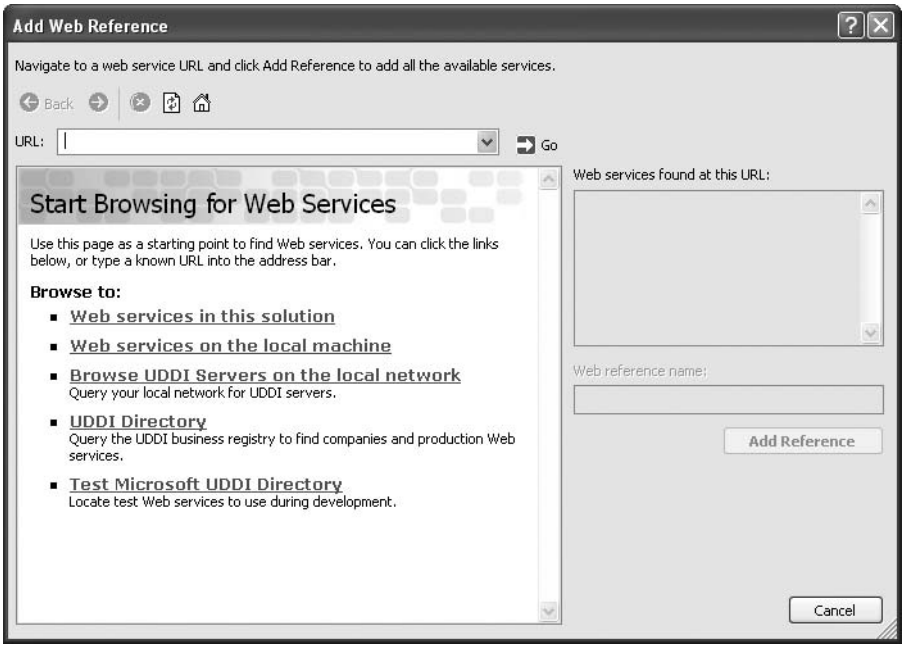

Figure 25-6. *The Add Web Reference dialog box*

■ **Note** The Add Web Reference dialog box cannot reference XML web services hosted with `WebDev.WebServer.exe`.

Notice that not only are you able to obtain a list of XML web services on your local development machine, but you may also query various UDDI catalogs (which you'll do at the end of this chapter). In any case, once you type a valid URL that points to a given `*.wsdl` or `*.asmx` file, your project will contain a new proxy class. Do note that the proxy's namespace (which is based on the URL of origin) will be nested within your client's .NET namespace. Thus, if you have a client named MyClientApp that added a reference to a web service on your local machine you would need to specify the following C# `using` directive:

```
using MyClientApp.localhost;
```

■ **Note** As of Visual Studio 2005, the Add Web Reference dialog box automatically adds an `app.config` file to your project that contains the URL of the referenced XML web service or updates an existing `app.config` file.

Exposing Custom Types from Web Methods

In the final example of this chapter, you'll examine how to build web services that expose custom types as well as more exotic types from the .NET base class libraries. To illustrate this, you'll create a new XML web service that is capable of processing arrays, custom types, and ADO.NET `DataSets`. To begin, create a new XML web service named CarSalesInfoWS that is hosted under an IIS virtual directory.

Exposing Arrays

Create a web method named `GetSalesTagLines()`, which returns an array of strings that represent the current sales for various automobiles, and another named `SortCarMakes()`, which allows the caller to pass in an array of unsorted strings and obtain a new array of sorted strings:

```
[WebService(Namespace = "http://IntertechTraining.com/",
  Description = "A car-centric web service",
  Name = "CarSalesInfoWS")]
[WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
public class Service : System.Web.Services.WebService
{
    [WebMethod(Description = "Get current discount blurbs")]
    public string[] GetSalesTagLines()
    {
        string[] currentDeals = {"Colt prices slashed 50%!",
               "All BMWs come with standard 8-track",
               "Free Pink Caravans...just ask me!"};
        return currentDeals;
    }

    [WebMethod(Description = "Sorts a list of car makes")]
    public string[] SortCarMakes(string[] theCarsToSort)
    {
        Array.Sort(theCarsToSort);
        return theCarsToSort;
    }
}
```

■ **Note** The default test page generated by `DefaultWsdlHelpGenerator.aspx` cannot invoke methods that take arrays of types as parameters.

Exposing Structures

The SOAP protocol is also able to transport XML representations of custom data types (both classes and structures). XML web services make use of the `XmlSerializer` type to encode the type as XML (see Chapter 17 for details). Recall that the `XmlSerializer`

- Cannot serialize private data. It serializes only public fields and properties.
- Requires that each serialized class provide a default constructor.
- Does not require the use of the `[Serializable]` attribute.

This being said, our next web method will return an array of `SalesInfoDetails` structures, defined as so:

```
// A custom type.
public struct SalesInfoDetails
{
     public string info;
     public DateTime dateExpired;
     public string Url;
}
```

Another point of interest regarding the `XmlSerializer` is the fact that it allows you to have fine-grained control over how the type is represented. By default, the `SalesInfoDetails` structure is serialized by encoding each piece of field data as a unique XML element:

```
<SalesInfoDetails>
  <info>Colt prices slashed 50%!</info>
  <dateExpired>2004-12-02T00:00:00.0000000-06:00</dateExpired>
  <Url>http://www.CarsRUs.com</Url>
</SalesInfoDetails>
```

If you wish to change this default behavior, you can adorn your type definitions using attributes found within the `System.Xml.Serialization` namespace (again, see Chapter 17 for full details):

```
public struct SalesInfoDetails
{
    public string info;
    [XmlAttribute]
    public DateTime dateExpired;
    public string Url;
}
```

This yields the following XML data representation:

```
<SalesInfoDetails dateExpired="2004-12-02T00:00:00">
  <info>Colt prices slashed 50%!</info>
  <Url>http://www.CarsRUs.com</Url>
</SalesInfoDetails>
```

The implementation of `GetSalesInfoDetails()` returns a populated array of this custom structure as follows:

```
[WebMethod(Description="Get details of current sales")]
public SalesInfoDetails[] GetSalesInfoDetails()
{
     SalesInfoDetails[] theInfo = new SalesInfoDetails[3];
     theInfo[0].info = "Colt prices slashed 50%!";
     theInfo[0].dateExpired = DateTime.Parse("12/02/04");
     theInfo[0].Url= "http://www.CarsRUs.com";
     theInfo[1].info = "All BMWs come with standard 8-track";
     theInfo[1].dateExpired = DateTime.Parse("8/11/03");
     theInfo[1].Url= "http://www.Bmws4U.com";
     theInfo[2].info = "Free Pink Caravans...just ask me!";
     theInfo[2].dateExpired = DateTime.Parse("12/01/09");
     theInfo[2].Url= "http://www.AllPinkVans.com";
     return theInfo;
}
```

Exposing ADO.NET DataSets

To wrap up your XML web service, here is one final web method that returns a `DataSet` populated with the Inventory table the Cars database you created during our examination of ADO.NET in Chapter 22:

```
// Return all cars in inventory table.
[WebMethod(Description =
  "Returns all autos in the Inventory table of the Cars database")]
public DataSet GetCurrentInventory()
{
     // Fill the DataSet with the Inventory table.
```

```
    SqlConnection sqlConn = new SqlConnection();
    sqlConn.ConnectionString = "data source=localhost; initial catalog=Cars;" +
        "uid=sa; pwd=";
    SqlDataAdapter myDA=
        new SqlDataAdapter("Select * from Inventory", sqlConn);
    DataSet ds = new DataSet();
    myDA.Fill(ds, "Inventory");
    return ds;
}
```

Source Code The CarsSalesInfoWS files can be found under the Chapter 25 subdirectory.

A Windows Forms Client

To test your new XML web service, create a Windows Forms application and reference CarsSalesInfoWS using the Visual Studio 2005 Add Web References dialog box (see Figure 25-7).

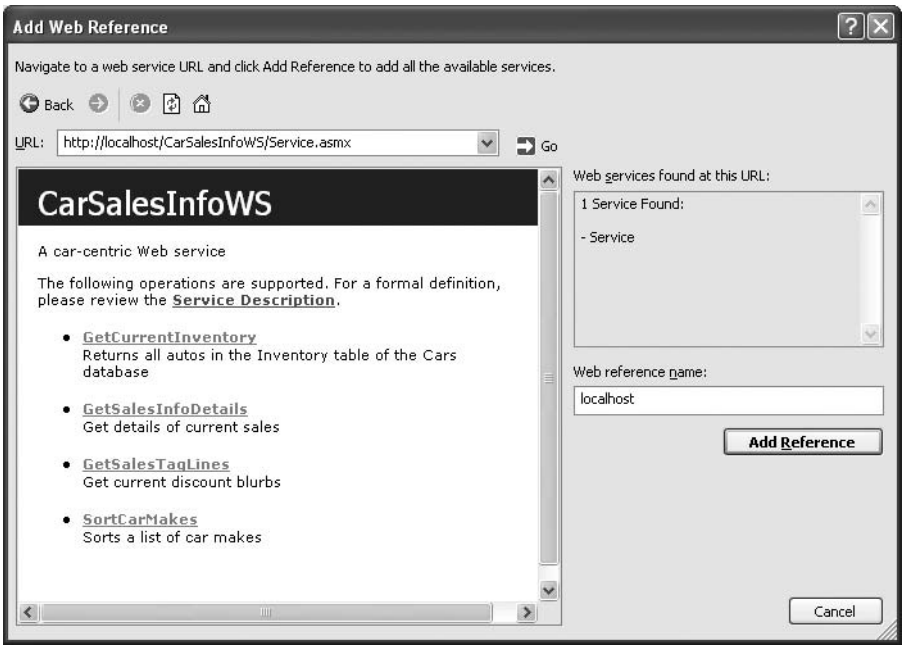

Figure 25-7. *Referencing CarsSalesInfoWS*

At this point, simply make use of the generated proxy to invoke the exposed web methods. Here is one possible Form implementation:

```
using CarsSalesInfoClient.localhost;
...
namespace CarsSalesInfoClient
{
    public partial class MainWindow : Form
    {
```

```
        private CarSalesInfoWS ws = new CarSalesInfoWS();
...
        private void MainWindow_Load(object sender, EventArgs e)
        {
            // Bind DataSet to grid.
            inventoryDataGridView.DataSource
                = ws.GetCurrentInventory().Tables[0];
        }

        private void btnGetTagLines_Click(object sender, EventArgs e)
        {
            string[] tagLines = ws.GetSalesTagLines();
            foreach (string tag in tagLines)
                listBoxTags.Items.Add(tag);
        }
        private void btnGetAllDetails_Click(object sender, EventArgs e)
        {
            SalesInfoDetails[] theSkinny = ws.GetSalesInfoDetails();
            foreach (SalesInfoDetails s in theSkinny)
            {
                string d = string.Format("Info: {0}\nURL:{1}\nExpiration Date:{2}",
                    s.info, s.Url, s.dateExpired);
                MessageBox.Show(d, "Details");
            }
        }
    }
}
```

Figure 25-8 shows a possible test run.

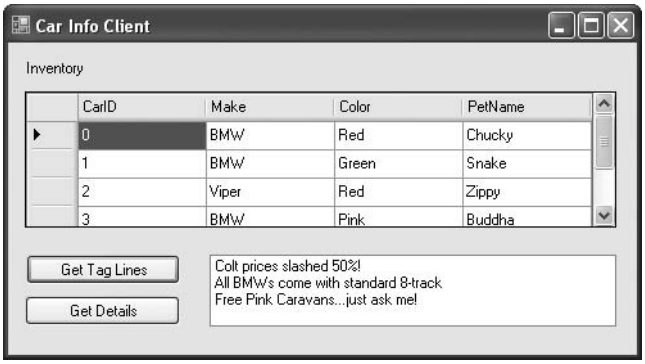

Figure 25-8. *The CarsSalesInfo client*

Client-Side Type Representation

When clients set a reference to a web service that exposes custom types, the proxy class file also contains language definitions for each custom public type. Thus, if you were to examine the client-side representation of SalesInfoDetails (within the generated `Reference.cs` file), you would see that each field has been encapsulated by a strongly typed property (also note that this type is now defined as a *class* rather than a structure):

```
[System.SerializableAttribute()]
[System.Xml.Serialization.XmlTypeAttribute
  (Namespace="http://IntertechTraining.com/")]
public partial class SalesInfoDetails {
    private string infoField;
    private string urlField;
    private System.DateTime dateExpiredField;

    public string info
    {
        get { return this.infoField; }
        set { this.infoField = value; }
    }
    public string Url
    {
        get { return this.urlField; }
        set { this.urlField = value; }
    }
    [System.Xml.Serialization.XmlAttributeAttribute()]
    public System.DateTime dateExpired
    {
        get { return this.dateExpiredField; }
        set { this.dateExpiredField = value; }
    }
}
```

Now, understand, of course, that like .NET remoting, types that are serialized across the wire as XML do not retain implementation logic. Thus, if the `SalesInfoDetails` structure supported a set of public methods, the proxy generator will fail to account for them (as they are not expressed in the WSDL document in the first place!). However, if you were to distribute a client-side assembly that contained the implementation code of the client-side type, you would be able to leverage the type-specific logic. Doing so would require a .NET-aware machine, though.

■ **Source Code** The CarSalesInfoClient projects can be found under the Chapter 25 subdirectory.

Understanding the Discovery Service Protocol (UDDI)

It is a bit ironic that the typical first step taken by a client to chat with a remote web service is the final topic of this chapter. The reason for such an oddball flow is the fact that the process of identifying whether or not a given web service exists using UDDI is not only optional, but also unnecessary in a vast majority of cases.

Until XML web services becomes the de facto standard of distributed computing, most web services will be leveraged by companies tightly coupled with a given vendor. Given this, the company and vendor at large already know about each other, and therefore have no need to query a UDDI server to see if the web service in question exists. However, if the creator of an XML web service wishes to allow the world at large to access the exposed functionality to any number of external developers, the web service may be posted to a UDDI catalog.

UDDI is an initiative that allows web service developers to post a commercial web service to a well-known repository. Despite what you might be thinking, UDDI is not a Microsoft-specific technology. In fact, IBM and Sun Microsystems have an equal interest in the success of the UDDI initiative. As you would expect, numerous vendors host UDDI catalogs. For example, Microsoft's

official UDDI website can be found at `http://uddi.microsoft.com`. The official website of UDDI (`http://www.uddi.org`) provides numerous white papers and SDKs that allow you to build internal UDDI servers.

Interacting with UDDI via Visual Studio 2005

Recall that the Add Web Reference dialog box allows you not only to obtain a list of all XML web services located on your current development machine (as well as a well-known URL) but also to submit queries to UDDI servers. Basically, you have the following options:

- Browse for a UDDI server on your company intranet.
- Browse the Microsoft-sponsored UDDI production server.
- Browse the Microsoft-sponsored UDDI test server.

Assume that you are building an application that needs to discover the current weather forecast on a per–zip code basis. Your first step would be to query a UDDI catalog with the following question:

- "Do you know of any web services that pertain to weather data?"

If it is the case that the UDDI server has a list of weather-aware web services, you are returned a list of all registered URLs that export the functionality of your query. Referencing this list, you are able to pick the specific web service you wish to communicate with and eventually obtain the WSDL document that describes the functionality of the weather-centric functionality.

As a quick example, create a brand-new console application project and activate the Add Web Reference dialog box. Next, select the Test Microsoft UDDI Directory link, which will bring you to the Microsoft UDDI test server. At this point, enter **weather** as a search criterion. Once the UDDI catalog has been queried, you will receive a list of all relevant XML web services. When you find an XML web service you are interested in programming against, add a reference to your current project. As you would expect, the raw WSDL will be parsed by the tool to provide you with a C# proxy.

■ **Note** Understand that the UDDI test center is just that: a test center. Don't be too surprised if you find a number of broken links. When you query production-level UDDI servers, URLs tend to be much more reliable, given that companies typically need to pay some sort of fee to be listed.

Summary

This chapter exposed you to the core building blocks of .NET web services. The chapter began by examining the core namespaces (and core types in these namespaces) used during web service development. As you learned, web services developed using the .NET platform require little more than applying the `[WebMethod]` attribute to each member you wish to expose from the XML web service type. Optionally, your types may derive from `System.Web.Services.WebService` to obtain access to the `Application` and `Session` properties (among other things). This chapter also examined three key related technologies: a lookup mechanism (UDDI), a description language (WSDL), and a wire protocol (`GET`, `POST`, or SOAP).

Once you have created any number of `[WebMethod]`-enabled members, you can interact with a web service through an intervening proxy. The `wsdl.exe` utility generates such a proxy, which can be used by the client like any other C# type. As an alternative to the `wsdl.exe` command-line tool, Visual Studio 2005 offers similar functionality via the Add Web Reference dialog box.