

55

Web Services with ASP.NET

WHAT'S IN THIS CHAPTER

- The syntax of SOAP and WSDL
- How SOAP and WSDL are used by web services
- Exposing and consuming web services
- The use of web services
- Exchanging data by using SOAP headers

Web services are a way of performing remote method calls over HTTP that can make use of *Simple Object Access Protocol* (SOAP). In the past, this issue was fraught with difficulty, as anyone who has any DCOM (Distributed COM) experience knows. Using older technologies, the act of instantiating an object on a remote server, calling a method, and obtaining the result is far from simple, and the necessary configuration is even trickier.

SOAP simplifies matters immensely. This technology is an XML-based standard that details how method calls can be made over HTTP in a reproducible manner. A remote SOAP server is capable of understanding these calls and performing all the hard work for you, such as instantiating the required object, making the call, and returning a SOAP-formatted response to the client.

The .NET Framework makes it very easy for you to do this. As with ASP.NET, you are able to use the full array of C# and .NET techniques on the server, but (perhaps more importantly) the simple consumption of web services can be achieved from any platform with HTTP access to the server. In other words, it is conceivable that Linux code could, for example, use ASP.NET web services, or even Internet-enabled fridges. To quote a real-world example, in the past I have had great success combining ASP.NET web services with Macromedia Flash to create data-enabled flash content.

In addition, web services can be completely described using *Web Service Description Language* (WSDL), allowing the dynamic discovery of web services at runtime. WSDL provides descriptions of all methods (along with the types required to call them) using XML with XML schemas. A wide variety of types are available to web services, which range from simple primitive types to full `DataSet` objects; this makes it possible to marshal full in-memory databases to a client. This can result in a dramatic reduction in load on a database server.



Note that this chapter deals with ASP.NET web services and not WCF web services, which are a more recent addition to .NET. ASP.NET web services are simpler to use and perfectly adequate for most situations, while Windows Communication Foundation (WCF) web services encompass all of ASP.NET web service functionality and add additional capabilities. For more information on WCF, see Chapter 43.

SOAP

As mentioned, one way to exchange data with web services is SOAP. This technology had a lot of press when it was first released, especially as Microsoft decided to adopt it for use in the .NET Framework. When you think about it, finding out exactly how SOAP works is a bit like finding out about how HTTP works — interesting, but not essential. Most of the time you don't have to worry about the format of the exchanges made with web services; they just happen, you get the results you want, and everyone is happy.

For this reason, this section won't go into a huge amount of depth, but you will see some simple SOAP requests and responses so you can get a feel for what is going on under the hood.

Imagine that you want to call a method in a web service with the following signature:

```
int DoSomething(string stringParam, int intParam)
```

The SOAP headers and body required for this are shown in the following code, with the address of the web service (more on this later) at the top:

```
POST /SomeLocation/myWebService.asmx HTTP/1.1
Host: hostname
Content-Type: text/xml; charset=utf-8
Content-Length: length
SOAPAction: "http://tempuri.org/DoSomething"
<?xml version="1.0" encoding="utf-8"?>
<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>
    <DoSomething xmlns="http://tempuri.org/">
      <stringParam>string</stringParam>
      <intParam>int</intParam>
    </DoSomething>
  </soap:Body>
</soap:Envelope>
```

The length parameter here specifies the total byte size of the content and varies depending on the values sent in the *string* and *int* parameters. Host will also vary, depending on where the web service is located.

The soap namespace referenced here defines various elements that you use to build your message. When you send this over HTTP, the actual data sent is slightly different (but related). For example, you could call the preceding method using the simple GET method:

```
GET /SomeLocation/myWebService.asmx/DoSomething?stringParam=string&intParam=int
HTTP/1.1
Host: hostname
```

The SOAP response of this method is as follows:

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <DoSomethingResponse xmlns="http://tempuri.org/">
```

```

        <DoSomethingResult>int</DoSomethingResult>
    </DoSomethingResponse>
</soap:Body>
</soap:Envelope>

```

where *length* varies depending on the contents, in this case *int*.

The actual response over HTTP is simpler, as shown in this example:

```

HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length
<?xml version="1.0"?>
<int xmlns="http://tempuri.org/">int</int>

```

This is a far simpler XML format.

As discussed at the start of this section, the beauty of all this is that you can ignore it completely. Only if you want to do something really odd does the exact syntax become important.

WSDL

WSDL completely describes web services, the methods available, and the various ways of calling these methods. The exact details of this process won't really benefit you that much, but a general understanding is useful.

WSDL is another fully XML-compliant syntax, and specifies web services by the methods available, the types used by these methods, the formats of request and response messages sent to and from methods via various protocols (pure SOAP, HTTP GET, and so on), and various bindings between these specifications. WSDL is understood by a variety of clients — not just .NET ones, but others such as Macromedia Flash, as mentioned in the introduction to this chapter.

Perhaps the most important part of a WSDL file is the type-definition section. This section uses XML schemas to describe the format for data exchange with the XML elements that can be used and their relationships.

For example, the web service method used as an example in the last section:

```
int DoSomething(string stringParam, int intParam)
```

would have types declared for the request as follows:

```

<?xml version="1.0" encoding="utf-8"?>
<definitions xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:s="http://www.w3.org/2001/XMLSchema"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl"
    ...other namespaces...>
    <wsdl:types>
        <s:schema elementFormDefault="qualified"
            targetNamespace="http://tempuri.org/">
            <s:element name="DoSomething">
                <s:complexType>
                    <s:sequence>
                        <s:element minOccurs="0" maxOccurs="1" name="stringParam"
                            type="s:string" />
                        <s:element minOccurs="1" maxOccurs="1" name="intParam"
                            type="s:int" />
                    </s:sequence>
                </s:complexType>
            </s:element>
            <s:element name="DoSomethingResponse">
                <s:complexType>
                    <s:sequence>
                        <s:element minOccurs="1" maxOccurs="1" name="DoSomethingResult"
                            type="s:int" />
                    </s:sequence>
                </s:complexType>
            </s:element>
        </s:schema>
    </wsdl:types>

```

```

</s:schema>
</wsdl:types>
...other definitions...
</definitions>

```

These types are all that are required for the SOAP and HTTP requests and responses you saw earlier, and are bound to these operations later in the file. All the types are specified using standard XML schema syntax, for example:

```

<s:element name="DoSomethingResponse">
  <s:complexType>
    <s:sequence>
      <s:element minOccurs="1" maxOccurs="1" name="DoSomethingResult"
        type="s:int" />
    </s:sequence>
  </s:complexType>
</s:element>

```

This specifies that an element called `DoSomethingResponse` has a child element called `DoSomethingResult` that contains an integer. This integer must occur between 1 and 1 times, meaning that it must be included.

If you have access to the WSDL for a web service, you can use it. As you see shortly, this isn't that difficult to do.

After this brief look at SOAP and WSDL, it's time to move on to discuss how you create and consume web services.

WEB SERVICES

The information about web services in this chapter is divided into two subsections:

- “Exposing Web Services,” which addresses writing web services and placing them on web servers.
- “Consuming Web Services,” which covers using web services in a client application.

Exposing Web Services

Web services are exposed by placing code either directly into `.asmx` files or by referencing web service classes from these files. As with ASP.NET pages, creating a web service in Visual Studio .NET uses the latter method, and you will too for demonstration purposes.

Create a web service project (via File ⇨ New ⇨ Web Site . . .) in the `C:\ProCSharp\Chapter55` directory and call it `PCWebService1` (see Figure 55-1). Creating a web service project generates a similar set of files as creating a web application project, and you have the same location options for creating the project. In fact, the only difference is that instead of a file called `Default.aspx`, a file called `Service.aspx` is created, with code-behind in `App_Code/Service.cs`.

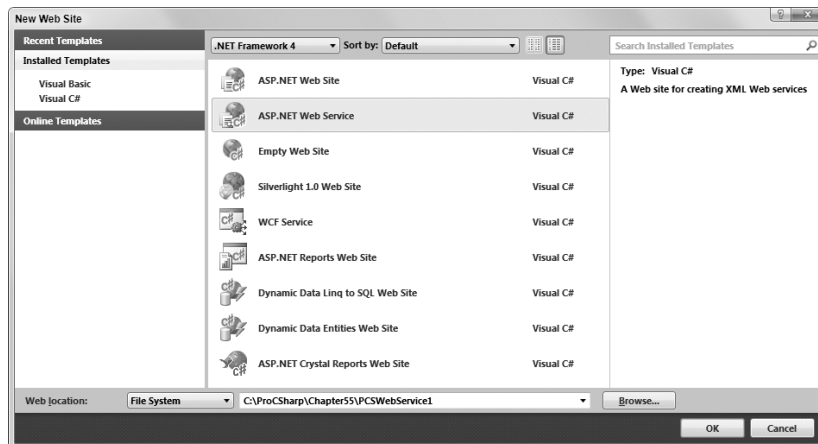


FIGURE 55-1

The code in `Service.asmx` is as follows:

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

This references the code file `/App_Code/Service.cs`. The following listing shows the generated code:



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Services;

[WebService(Namespace = "http://tempuri.org/")]
[WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
// To allow this Web Service to be called from script, using ASP.NET AJAX,
// uncomment the following line.
// [System.Web.Script.Services.ScriptService]
public class Service : System.Web.Services.WebService
{
    public Service()
    {
        //Uncomment the following line if using designed components
        //InitializeComponent();
    }

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

code snippet PCSWebService1\App_Code\Service.cs

This code contains several standard namespace references, and defines the web service class `Service` (referenced in `Service.asmx`), which inherits from `System.Web.Services.WebService`. The `WebService` attribute specifies the namespace for the web service, which enables client applications to differentiate between web service methods with the same name, but on different web services. The `WebServiceBinding` attribute relates to web service interoperability, as defined in the WS-I Basic Profile 1.1 specification. Put simply, this attribute can declare that a web service supports a standard WSDL description for one or more of its web methods, or, as is the case here, defines a new set of WSDL definitions. There is also a commented out `ScriptService` attribute, which if uncommented makes it possible to call web methods using ASP.NET AJAX script. It is now up to you to provide additional methods on this web service class.

Adding a method accessible through the web service simply requires defining the method as `public` and giving it the `WebMethod` attribute. This attribute simply labels the methods you want to be accessible through the web service. You look at the types you can use for the return type and parameters shortly, but for now replace the autogenerated `HelloWorld()` method with the following one:

```
[WebMethod]
public string CanWeFixIt()
{
    return "Yes we can!";
}
```

Now compile the project.

To see whether everything works, run the application with `Ctrl+F5` and you'll be taken straight to the test interface for the web service, as shown in Figure 55-2.

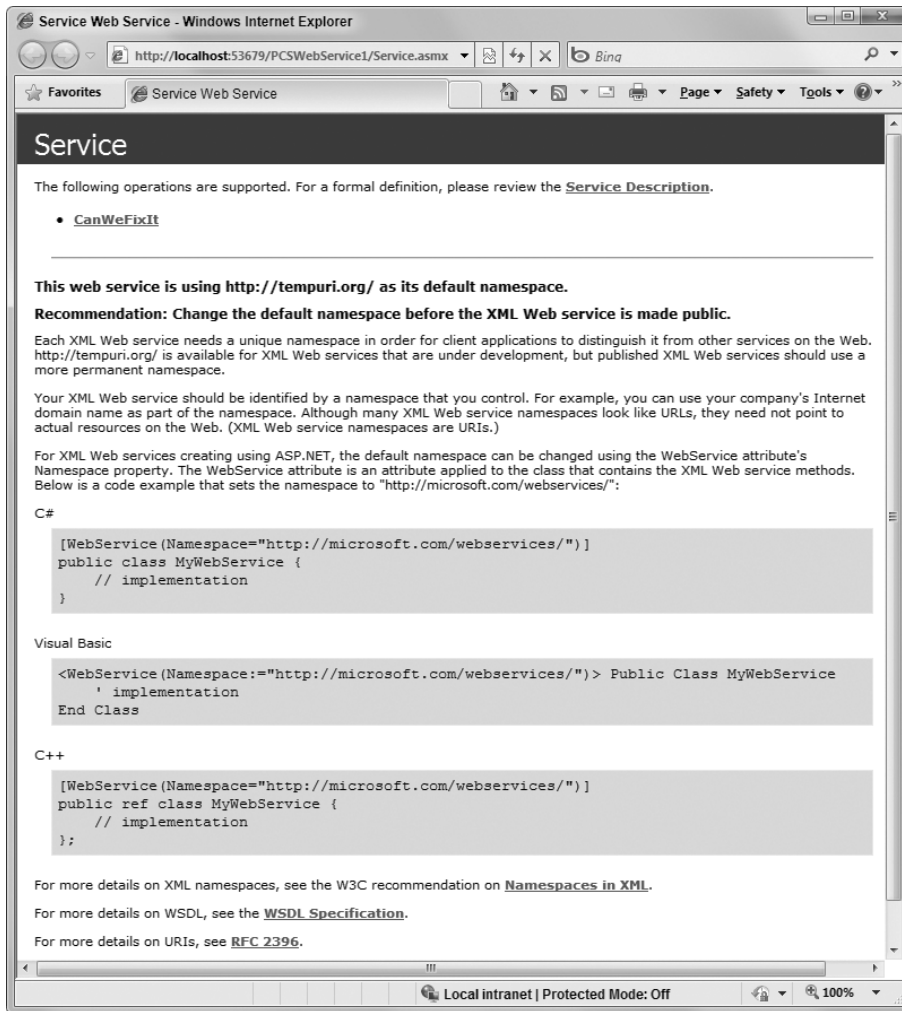


FIGURE 55-2



Note that, by default, this test page is only available to callers from the local computer, even when your web service is hosted in Internet Information Services (IIS).

Most of the text shown in the browser concerns the fact that the web service namespace is set to `http://tempuri.org`. This isn't a problem during development, although (as the text on the web page says) it should be changed later on. This can be done using the `WebService` attribute as shown. For now, though, you can leave things as they are.

Clicking the method name gives you information about the SOAP request and response, as well as examples of how the request and response will look using the HTTP GET and HTTP POST methods. You can also test the method by clicking the Invoke button. If the method requires simple parameters, you can enter these

on this form as well (for more complex parameters, this form won't allow you to test the method in this way). If you do this you will see XML returned by the method call:

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

This demonstrates that your method is working perfectly.

Following the Service Description link, from the browser screen shown in Figure 55-2, allows you to view the WSDL description of the web service. The most important part is the description of the element types for requests and responses:

```
<wsdl:types>
  <s:schema elementFormDefault="qualified" targetNamespace="http://tempuri.org/">
    <s:element name="CanWeFixIt">
      <s:complexType />
    </s:element>
    <s:element name="CanWeFixItResponse">
      <s:complexType>
        <s:sequence>
          <s:element minOccurs="0" maxOccurs="1" name="CanWeFixItResult"
            type="s:string" />
        </s:sequence>
      </s:complexType>
    </s:element>
  </s:schema>
</wsdl:types>
```

The description also contains descriptions of the types required for requests and responses, as well as various bindings for the service, making it quite a long file.

Types Available for Web Services

Web services can be used to exchange any of the following types:

String	Double
Char	Guid
Byte	Decimal
Boolean	DateTime
Int16	XmlQualifiedName
Int32	class
Int64	struct
UInt16	XmlNode
UInt32	DataSet
UInt64	enum
Single	

Arrays of all these types are also allowed, as are generic collection types such as `List<string>`. Note also that only public properties and fields of `class` and `struct` types are marshalled.

Consuming Web Services

Now that you know how to create web services, in this section you look at how to use them. To do this, you need to generate a proxy class in your code that knows how to communicate with a given web service. Any calls from your code to the web service will go through this proxy, which looks identical to the web

service, giving your code the illusion that you have a local copy of it. In actual fact there is a lot of HTTP communication going on, but you are shielded from the details. There are two ways of doing this. You can either use the WSDL.exe command-line tool or the Add Web Reference menu option in Visual Studio .NET.

Using WSDL.exe from the command-line generates a .cs file containing a proxy class, based on the WSDL description of the web service. You specify this using the URL of the web service, for example:

```
WSDL http://localhost:53679/PCSWebService1/Service.asmx?WSDL
```



Note that both here and in the example that follows you are using the default file system hosting for web applications. For the preceding URL to work the Visual Web Developer Web Server for the web service must be running. There is also no guarantee that the port number for the web service (in this case 53679) will stay the same. Although this is fine for illustration, typically you want your web services to reside on a fixed web server such as IIS — otherwise, you'll have to continually remake proxy classes. One way to make sure that the web service is available for testing is to include multiple web sites in one solution.

This generates a proxy class for the example from the previous section in a file called `Service.cs`. The class will be named after the web service, in this case `Service`, and contain methods that call identically named methods of the service. To use this class, you simply add the .cs file generated to a project and use code along the lines of this:

```
Service myService = new Service();
string result = myService.CanWeFixIt();
```

By default, the class generated is placed in the root namespace, so no using statement is necessary, but you can specify a different namespace to use with the `/n:<namespace>` command-line option of WSDL.exe.

This technique works fine but can be annoying to continually redo if the service is being developed and changing continuously. Of course, it could be executed in the build options for a project to automatically update the generated proxy before each compile, but there is a better way.

This better way is illustrated by creating a client for the example in the previous section, in a new empty web site called `PCSWebClient1` (in the `C:\ProCSharp\Chapter55` directory). Create this project now, add a `Default.aspx` page, and add the following code to `Default.aspx`:



Available for
download on
Wrox.com

```
<form id="form1" runat="server">
  <div>
    <asp:Label Runat="server" ID="resultLabel" />
    <br />
    <asp:Button Runat="server" ID="triggerButton" Text="Invoke CanWeFixIt()" />
  </div>
</form>
```

code snippet PCSWebClient1\Default.aspx

You'll bind the button-click event handler to the web service shortly. First, you must add a reference to the web service to your project. To do this, right-click the new client project in the Solution Explorer and select the Add Web Reference... option. In the window that appears, type the URL of the web service `Service.asmx` file, or use the web services on the Local Machine link to find it automatically, as shown in Figure 55-3.

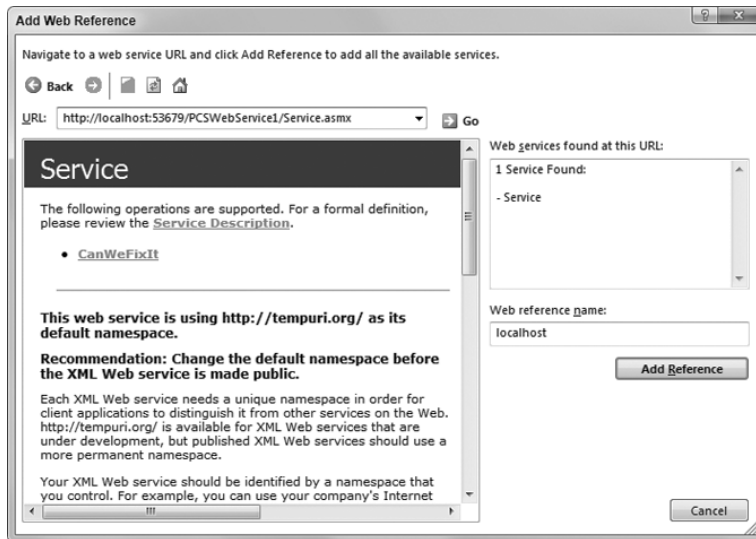


FIGURE 55-3

From here you can add a reference with the Add Reference button. First, though, change the default entry for the web reference name from localhost to myWebService. Pressing the Add Reference button now adds myWebService to the App_WebReferences folder of the project in Solution Explorer. If you examine this folder in the Solution Explorer window, you can see that the files Service.disco, Service.discomap, and Service.wsdl have been added to the project.

The web reference name, myWebService, is also the namespace you need to reference to use the proxy class that has been created for you. Add the following using statement to your code in Default.aspx.cs:

```
using myWebService;
```

Now you can use the service in your class without fully qualifying its name.

Add an event handler to the button on the form by double-clicking the button in design view, and adding the following code:

```
protected void triggerButton_Click(object sender, EventArgs e)
{
    Service myService = new Service();
    resultLabel.Text = myService.CanWeFixIt();
}
```

Running the application and clicking the button displays the result of CanWeFixIt() in the browser window.



Note that if you are using the ASP.NET Development Server (that is, your web applications are hosted on the local file system, not IIS) you may get a 401: Unauthorized error. This is because this server is configured to require NTLM authentication by default. To fix this, you can either disable this setting by unchecking the NTLM Authentication box on the Start Options page of the property pages for PCSWebService1 or pass default credentials when calling the web service method. This latter option requires this code:

```
myService.Credentials = System.Net.CredentialCache.DefaultCredentials;
```

This web service might change later, but with this method you can simply right-click the App_WebReference folder in the Server Explorer and select Update Web/Service References. This generates a new proxy class for you to use.

Also, the web service may move later, perhaps when it is deployed. If you look in `web.config` for the client application you will see the following:



```
<appSettings>
  <add key="myWebService.Service"
    value="http://localhost:53679/PCWebService1/Service.asmx" />
</appSettings>
```

code snippet PCWebClient1\web.config

This setting configures where web requests will be sent, so you must ensure that it matches the web service location, or use some other means to record this information if you prefer.

EXTENDING THE EVENT-BOOKING EXAMPLE

Now that you know the basics of creating and consuming web services, you can apply your knowledge to extending the meeting room booker application from Chapters 40, “Core ASP.NET” and 41, “ASP.NET Features.” Specifically, you extract the database access aspects from the application and place them into a web service. This web service has two methods:

- `GetData()`, which returns a `DataSet` object containing all three tables in the `PCSDemoSite` database.
- `AddEvent()`, which adds an event and returns the number of rows affected so the client application can check that a change has been made.

In addition, you’ll design the web service with load reduction in mind. Specifically, you’ll store a `DataSet` containing the meeting room booker data at the application level in the web service application. This means that multiple requests for the data won’t require additional database requests. The data in this application-level `DataSet` object will only be refreshed when new data is added to the database. This means that changes made to the database by other means, such as manual editing, will *not* be reflected in the `DataSet`. Still, as long as you know that your web service is the only application with direct access to the data, you have nothing to worry about.

The Event-Booking Web Service

Create a new web service project in Visual Studio in the `C:\ProCSharp\Chapter55` directory and call it `PCWebService2`. The first thing to do is to copy the database file (`MeetingRoomBooker.mdf`) from `PCSDemoSite` in the code for Chapter 41 into the `App_Data` directory for the web service. Next, you need to add a `Global.asax` file to the project, then modify the code in its `Application_Start()` event handler. You want to load all the data in the `MeetingRoomBooker` database into a data set and store it. This mostly involves code that you’ve already seen, because getting the database into a `DataSet` is something you’ve already done. You’ll also use a connection string stored in `web.config` as you’ve seen in earlier chapters. The code for the `web.config` is as follows (the connection string should be placed on a single line):



```
<?xml version="1.0" ?>
<configuration>
  ...
  <connectionStrings>
    <add name="MRBConnectionString"
      connectionString="Data Source=.\SQLEXPRESS;Integrated
        Security=True;AttachDBFilename=|DataDirectory|MeetingRoomBooker.mdf;
        User Instance=True"
      providerName="System.Data.SqlClient" />
    </connectionStrings>
  </configuration>
```

code snippet PCWebService2\web.config

And the code for the `Application_Start()` event handler in `Global.asax` is:



```
void Application_Start(Object sender, EventArgs e)
{
    System.Data.DataSet ds;
    System.Data.SqlClient.SqlConnection sqlConnection1;
    System.Data.SqlClient.SqlDataAdapter daAttendees;
    System.Data.SqlClient.SqlDataAdapter daRooms;
    System.Data.SqlClient.SqlDataAdapter daEvents;

    using (sqlConnection1 = new System.Data.SqlClient.SqlConnection())
    {
        sqlConnection1.ConnectionString =
            ConfigurationManager.ConnectionStrings["MRBConnectionString"]
                .ConnectionString;

        sqlConnection1.Open();

        ds = new System.Data.DataSet();
        daAttendees = new System.Data.SqlClient.SqlDataAdapter(
            "SELECT * FROM Attendees", sqlConnection1);
        daRooms = new System.Data.SqlClient.SqlDataAdapter(
            "SELECT * FROM Rooms", sqlConnection1);
        daEvents = new System.Data.SqlClient.SqlDataAdapter(
            "SELECT * FROM Events", sqlConnection1);

        daAttendees.Fill(ds, "Attendees");
        daRooms.Fill(ds, "Rooms");
        daEvents.Fill(ds, "Events");
    }

    Application["ds"] = ds;
}
```

code snippet PCSWebService2\Global.asax

The important code to note here is in the last line. `Application` objects (just like `Session` objects) have a collection of name-value pairs that you can use to store data. Here you are creating a name in the `Application` store called `ds`, which takes the serialized value of `ds` containing the `Attendees`, `Rooms`, and `Events` tables from your database. This value will be accessible to all instances of the web service at any time.

This technique is very useful for read-only data because multiple threads are able to access it, reducing the load on your database. Note, however, that the `Events` table is likely to change, and you'll have to update the application-level `DataSet` class when this happens. You look at this shortly.

Next you can replace the default `Service` service with a new service, called `MRBService`. To do this, delete the existing `Service.asmx` and `Service.cs` files and add a new web service to the project called `MRBService` (make sure you check the option to place code in a separate file). You can then add the `GetData()` method to your service in `MRBService.cs`:



```
[WebMethod]
public DataSet GetData()
{
    return (DataSet)Application["ds"];
}
```

code snippet PCSWebService2\App_Code\MRBService.cs

This uses the same syntax as `Application_Load()` to access the stored `DataSet`, which you simply cast to the correct type and return.

Note that for this to work, and to make life easier in the other web method you'll be adding, you can add the following using statements:

```
using System;
using System.Collections.Generic;
using System.Configuration;
using System.Data;
using System.Linq;
using System.Web;
using System.Web.Services;
```

The `AddEvent()` method is slightly more complicated. Conceptually, you need to do the following:

1. Accept event data from the client
2. Create a SQL `INSERT` command using this data
3. Connect to the database and execute the SQL statement
4. Refresh the data in `Application["ds"]` if the addition is successful
5. Return a success or failure notification to the client (the client will refresh its `DataSet` if required)

Starting from the top, you'll accept all fields in their correct data types:

```
[WebMethod]
public int AddEvent(string eventName, int eventRoom,
                   string eventAttendees, DateTime eventDate)
{
    ...
}
```

Next, you declare the objects you'll need for database access, connect to the database, and execute your query, all using similar code to that in `PCSDemoSite` (remember, you need the connection string here, taken from `web.config`):

```
[WebMethod]
public int AddEvent(string eventName, int eventRoom,
                   string eventAttendees, DateTime eventDate)
{
    System.Data.SqlClient.SqlConnection sqlConnection1;
    System.Data.SqlClient.SqlDataAdapter daEvents;
    DataSet ds;

    using (sqlConnection1 = new System.Data.SqlClient.SqlConnection())
    {
        sqlConnection1.ConnectionString =
            ConfigurationManager.ConnectionStrings["MRBConnectionString"]
                .ConnectionString;

        System.Data.SqlClient.SqlCommand insertCommand =
            new System.Data.SqlClient.SqlCommand(
                "INSERT INTO [Events] (Name, Room, "
                + "AttendeeList, EventDate) VALUES (@Name, @Room, @AttendeeList, "
                + "@EventDate)", sqlConnection1);
        insertCommand.Parameters.Add("Name", SqlDbType.VarChar, 255).Value
            = eventName;
        insertCommand.Parameters.Add("Room", SqlDbType.Int, 4).Value
            = eventRoom;
        insertCommand.Parameters.Add("AttendeeList", SqlDbType.Text, 16).Value
            = eventAttendees;
        insertCommand.Parameters.Add("EventDate", SqlDbType.DateTime, 8).Value
            = eventDate;

        sqlConnection1.Open();
```

```

        int queryResult = insertCommand.ExecuteNonQuery();
    }
}

```

You use `queryResult` to store the number of rows affected by the query as before. You can check this to see whether it is 1 to gauge your success. If you are successful, you execute a new query on the database to refresh the `Events` table in your `DataSet`. It is vital to lock the application data while you perform updates to ensure that no other threads can access `Application["ds"]` while you update it. You can do this using the `Lock()` and `UnLock()` methods of the `Application` object:

```

[WebMethod]
public int AddEvent(string eventName, int eventRoom,
                  string eventAttendees, DateTime eventDate)
{
    ...
    int queryResult = insertCommand.ExecuteNonQuery();
    if (queryResult == 1)
    {
        daEvents = new System.Data.SqlClient.SqlDataAdapter(
            "SELECT * FROM Events", sqlConnection1);
        ds = (DataSet)Application["ds"];
        ds.Tables["Events"].Clear();
        daEvents.Fill(ds, "Events");
        Application.Lock();
        Application["ds"] = ds;
        Application.UnLock();
    }
}
}

```

Finally, you return `queryResult`, allowing the client to know whether the query was successful:

```

[WebMethod]
public int AddEvent(string eventName, int eventRoom,
                  string eventAttendees, DateTime eventDate)
{
    ...
    return queryResult;
}

```

And with that, you have completed your web service. As before, you can test this service simply by viewing the `.asmx` file in a web browser, so you can add records and look at the XML representation of the `DataSet` returned by `GetData()` without writing any client code.

Before moving on, it's worth discussing the use of `DataSet` objects with web services. At first glance this seems like a fantastic way of exchanging data, and indeed it is an extremely powerful technique. However, the fact that the `DataSet` class is so versatile does have implications. If you examine the WSDL generated for the `GetData()` method, you'll see the following:

```

<s:element name="GetDataResponse">
  <s:complexType>
    <s:sequence>
      <s:element minOccurs="0" maxOccurs="1" name="GetDataResult">
        <s:complexType>
          <s:sequence>
            <s:element ref="s:schema" />
            <s:any />
          </s:sequence>
        </s:complexType>
      </s:element>
    </s:sequence>
  </s:complexType>
</s:element>

```

As you can see, this is very generic code, which allows the `DataSet` object passed to contain any data specified with an inline schema. Unfortunately, this means that the WSDL does not completely describe the web service. For .NET clients this isn't a problem, and things progress as naturally as they did when passing a simple string in the earlier example, the only difference being that you exchange a `DataSet` object. However, non-.NET clients must have prior knowledge of the data that will be passed or some equivalent of a `DataSet` class to access the data. This includes script clients, such as those that use client-side ASP.NET AJAX code to process retrieved data.

A workaround to this requirement is to repackage the data into a different format — an array of structs, for example. If you were to do this, you could customize the XML produced in any way you want, and the XML could be completely described by the schema for the web service. This can also have an impact in terms of performance, because passing a `DataSet` object can result in an awful lot of XML — far more than is necessary in most cases. The overhead resulting from repackaging data is likely to be much less than that associated with sending the data over the web, and because there'll probably be less data, the serialization and deserialization is also likely to be quicker. Therefore, if performance is an issue, you probably should avoid using `DataSet` objects in this way — unless of course you make use of the additional functionality that `DataSet` objects make available to you.

For the purposes of this example, though, using a `DataSet` object is not a problem and greatly simplifies other code.

The Event-Booking Client

The client you use in this section is a development of the `PCSDemoSite` web site from Chapter 41. Call this application `PCSDemoSite2`, in the directory `C:\ProCSharp\Chapter55`, and use the code from `PCSDemoSite` as a starting point.

You'll make two major modifications to the project. First, you'll remove all direct database access from this application and use the web service instead. Second, you'll introduce an application-level store of the `DataSet` object returned from the web service that is updated only when necessary, meaning that even less of a load is placed on the database.

The first thing to do to your new web application is to add a web reference to the `PCSWebService2/MRBService.asmx` service. You can do this in the same way you saw earlier in this chapter through right-clicking the project in Server Explorer, locating the `.asmx` file, calling the web reference `MRBService`, and clicking Add Reference. You may need to start the ASP.NET Development Server before you can do this, which you can do by viewing the `.asmx` file of the web service in a browser. Because you aren't using the local database anymore, you can also delete that from the `App_Data` directory, and remove the `MRBConnectionString` entry from `web.config`. All the rest of the modifications are made to `MRB.ascx` and `MRB.ascx.cs`.

To start with, you can delete all the data sources on `MRB.ascx`, and remove the `DataSourceID` entries on all the currently data-bound controls. This is because you'll be handling the data binding yourself from the code-behind file.



Note that when you change or remove the `DataSourceID` property of a web server control, you may be asked if you want to remove the templates you have defined, because there is no guarantee that the data the control will work with will be valid for those templates. In this case you'll be using the same data, but from a different source, so be sure to keep the templates. If you do delete them, the HTML layout of the result will revert to the default, which won't look very nice, so you'll have to add them again from scratch or rewrite them.

Next, you'll need to add a property to `MRB.ascx.cs` to store the `DataSet` returned by the web service. This property actually uses Application state storage, in much the same way as `Global.asax` in the web service. The code is:



```
public DataSet MRBData
{
    get
    {
        if (Application["mrbData"] == null)
        {
            Application.Lock();
            MRBService.MRBService service = new MRBService.MRBService();
            service.Credentials = System.Net.CredentialCache.DefaultCredentials;
            Application["mrbData"] = service.GetData();
            Application.Unlock();
        }
        return Application["mrbData"] as DataSet;
    }
    set
    {
        Application.Lock();
        if (value == null && Application["mrbData"] != null)
        {
            Application.Remove("mrbData");
        }
        else
        {
            Application["mrbData"] = value;
        }
        Application.Unlock();
    }
}
```

code snippet PCSDemoSite2\MRB\MRB.ascx.cs

Note that you need to lock and unlock the Application state, just as in the web service. Also, note that the `Application["mrbData"]` storage is filled only when necessary, that is, when it is empty. This `DataSet` object is now available to all instances of `PCSDemoSite2`, meaning that multiple users can read data without any calls to the web service or indeed to the database. The credentials are also set here, which as noted earlier, is necessary for using web services hosted using the ASP.NET Development Server. You can comment out this line if you don't need it.

To bind to the controls on the web page, you can supply `DataView` properties that map to data stored in this property, as follows:

```
private DataView EventData
{
    get
    {
        return MRBData.Tables["Events"].DefaultView;
    }
}

private DataView RoomData
{
    get
    {
        return MRBData.Tables["Rooms"].DefaultView;
    }
}
```

```

private DataView AttendeeData
{
    get
    {
        return MRBData.Tables["Attendees"].DefaultView;
    }
}

private DataView EventDetailData
{
    get
    {
        if (EventList != null && EventList.SelectedValue != null)
        {
            return new DataView(MRBData.Tables["Events"], "ID=" +
                EventList.SelectedValue.ToString(), "",
                DataViewRowState.CurrentRows);
        }
        else
        {
            return null;
        }
    }
}

```

You can also remove the existing `EventData` field and `EventData` property.

Most of these properties are simple; it's only the last that does anything new. In this case, you are filtering the data in the `Events` table to obtain just one event — ready to display in the detail view `FormView` control.

Now that you aren't using data source controls, you have to bind data yourself. A call to the `DataBind()` method of the page will achieve this, but you also need to set the data source `DataView` properties for the various data-bound controls on the page. One good way to do this is to do it during an override of the `OnDataBinding()` event handler, as follows:

```

protected override void OnDataBinding(EventArgs e)
{
    roomList.DataSource = RoomData;
    attendeeList.DataSource = AttendeeData;
    EventList.DataSource = EventData;
    FormView1.DataSource = EventDetailData;
    base.OnDataBinding(e);
}

```

Here you are setting the `DataSource` properties of `roomList`, `attendeeList`, `EventList`, and `FormView1` to the properties defined earlier. Next, you can add the `DataBind()` call to `Page_Load()`:

```

void Page_Load(object sender, EventArgs e)
{
    if (!this.IsPostBack)
    {
        nameBox.Text = Context.User.Identity.Name;
        DateTime trialDate = DateTime.Now;
        calendar.SelectedDate = GetFreeDate(trialDate);
        DataBind();
    }
}

```

Also, you must change `submitButton_Click()` to use the web service `AddData()` method. Again, much of the code can remain unchanged; only the data addition code needs changing:

```

void submitButton_Click(object sender, EventArgs e)
{
    if (Page.IsValid)
    {
        ...
    }
}

```



```

    try
    {
        MRBService.MRBService service = new MRBService.MRBService();
        if (service.AddEvent(eventBox.Text, int.Parse(roomList.SelectedValue),
            attendees, calendar.SelectedDate) == 1)
        {
            MRBData = null;
            DataBind();
            calendar.SelectedDate =
                GetFreeDate(calendar.SelectedDate.AddDays(1));
        }
    }
    catch
    {
    }
}

```

In fact, all you've really done here is simplify things a great deal. This is often the case when using well-designed web services — you can forget about much of the inner workings and instead concentrate on the user experience.

There isn't a huge amount to comment on in this code. Continuing to make use of `queryResult` is a bonus, and locking the application is essential as already noted.

Two final modifications are required. First is `EventList_SelectedIndexChanged()`:

```

protected void EventList_SelectedIndexChanged(object sender, EventArgs e)
{
    FormView1.DataSource = EventDetailData;
    EventList.DataSource = EventData;
    EventList.DataBind();
    FormView1.DataBind();
}

```

This is simply to make sure that the data sources for the event list and detail views are refreshed properly.

You also need to add an `EventList_SelectedIndexChanging()` handler, which must be attached to the `SelectedIndexChanging` event of the `EventList` control:

```

protected void EventList_SelectedIndexChanging(object sender,
    ListViewSelectEventArgs e)
{
    EventList.SelectedIndex = e.NewSelectedIndex;
}

```

Without adding this, the code will fail as the changes you have made require a handler for this event.

The meeting room booker in the `PCSDemoSite2` web site should look and function exactly as the one in `PCSDemoSite`, but perform substantially better. You can also use the same web service very easily for other applications — simply displaying events on a page, for example, or even editing events, attendee names, and rooms if you add some more methods. Doing this won't break `PCSDemoSite2` because it simply ignores any new methods created. You will, however, have to introduce some kind of trigger mechanism to update the data cached in the event list, because modifying this data elsewhere will cause data to become out of date.

EXCHANGING DATA USING SOAP HEADERS

One final topic to look at in this chapter is using SOAP headers to exchange information, rather than including information in method parameters. The reason for covering this is that it is a very nice system to use for maintaining a user login. This section won't go into detail about setting up your server for SSL connections, or the various methods of authentication that can be configured using IIS, because these do not affect the web service code you need to get this behavior.

Say that you have a service that contains a simple authentication method with a signature as follows:

```
AuthenticationToken AuthenticateUser(string userName, string password);
```

where `AuthenticationToken` is a type you define that can be used by the user in later method calls, for example:

```
void DoSomething(AuthenticationToken token, OtherParamType param);
```

After logging in, the user has access to other methods using the token received from `AuthenticateUser()`. This technique is typical of secure web systems, although it is often implemented in a far more complex way.

You can simplify this process further by using a SOAP header to exchange tokens (or any other data). You can restrict methods so they are only called if a specified SOAP header is included in the method call. This simplifies their structure as follows:

```
void DoSomething(OtherParamType param);
```

The advantage here is that, after you have set the header on the client, it persists. After an initial bit of setting up, you can ignore authentication tokens in all further web method calls.

To see this in action, create a new web service project called `PCSWebService3` in the directory `C:\ProCSharp\Chapter55\`, and add a new class to the `App_Code` directory called `AuthenticationToken`, as follows:



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

public class AuthenticationToken : SoapHeader
{
    public Guid InnerToken;
}
```

code snippet PCSWebService3\App_Code\AuthenticationToken.cs

You'll use a GUID to identify the token, a common procedure, because you can be sure that it is unique.

To declare that the web service can have a custom SOAP header, simply add a public member to the service class of your new type:



```
public class Service : System.Web.Services.WebService
{
    public AuthenticationToken AuthenticationTokenHeader;
```

code snippet PCSWebService3\App_Code\Service.cs

You will also need to use the `System.Web.Services.Protocols.SoapHeaderAttribute` attribute to mark those web methods that require the extra SOAP header to work. However, before you add such a method, you can add a very simple `Login()` method that clients can use to obtain an authentication token:

```
[WebMethod(true)]
public Guid Login(string userName, string password)
{
    if ((userName == "Karli") && (password == "Cheese"))
    {
        Guid currentUser = Guid.NewGuid();
        Session["currentUser"] = currentUser;
        return currentUser;
    }
    else
    {
        Session["currentUser"] = null;
        return Guid.Empty;
    }
}
```

If the correct username and password are used, then a new `Guid` object is generated, stored in a session-level variable, and returned to the user. If authentication fails, an empty `Guid` instance is returned and stored at the session level. The `true` parameter enables session state for this web method, because it is disabled by default in web services and it is required for this functionality.

Next, you have a method that accepts the header, as specified by the `SoapHeaderAttribute` attribute:

```
[WebMethod(true)]
[SoapHeaderAttribute("AuthenticationTokenHeader",
    Direction = SoapHeaderDirection.In)]
public string DoSomething()
{
    if (Session["currentUser"] != null &&
        AuthenticationTokenHeader != null &&
        AuthenticationTokenHeader.InnerToken
            == (Guid)Session["currentUser"])
    {
        return "Authentication OK.";
    }
    else
    {
        return "Authentication failed.";
    }
}
```

This returns one of two strings, depending on whether the `AuthenticationTokenHeader` header exists, isn't an empty `Guid`, and matches the one stored in `Session["currentUser"]` (if this `Session` variable exists).

Next, you must create a quick client to test this service. Add a new empty web site called `PCSWebClient2` to the solution, with a `Default.aspx` page containing the following simple code for the user interface:



```
<form id="form1" runat="server">
  <div>
    User Name:
    <asp:TextBox Runat="server" ID="userNameBox" /><br />
    Password:
    <asp:TextBox Runat="server" ID="passwordBox" /><br />
    <asp:Button Runat="server" ID="loginButton" Text="Log in" /><br />
    <asp:Label Runat="server" ID="tokenLabel" /><br />
    <asp:Button Runat="server" ID="invokeButton"
      Text="Invoke DoSomething()" /><br />
    <asp:Label Runat="server" ID="resultLabel" /><br />
  </div>
</form>
```

code snippet PCSWebClient2\Default.aspx

Add the `PCSWebService3` service as a web reference (because the web service is local to the solution you can click the `Web Services in This Solution` link to get a reference quickly) with the name `authenticateService`, and add the following using statements to `Default.aspx.cs`:



```
using System.Net;
using authenticateService;
```

code snippet PCSWebClient2\Default.aspx.cs

You need to use the `System.Net` namespace because it includes the `CookieContainer` class. This is used to store a reference to a cookie, and you require this if you are working with web services that use session state. This is because you need some way for the web service to retrieve the correct session state across multiple calls from the client, where the proxy for the service is re-created on each postback. By retrieving the cookie used by the web service to store session state, storing it between web service calls, and then using

it in later calls, you can maintain the correct session state in the web service. Without doing this, the web service would lose its session state, and therefore the login information required in this scenario.

Back to the code, where you use a protected member to store the web reference proxy, and another to store a Boolean value indicating whether the user is authenticated or not:

```
public partial class _Default : System.Web.UI.Page
{
    protected Service myService;
    protected bool authenticated;

```

Page_Load() starts by initializing the myService service, as well as preparing a CookieContainer instance for use with the service:

```
protected void Page_Load(object sender, EventArgs e)
{
    myService = new Service();
    myService.Credentials = CredentialCache.DefaultCredentials;
    CookieContainer serviceCookie;

```

Next, you check for a stored CookieContainer instance or create a new one. Either way you assign the CookieContainer to the web service proxy, ready to receive the cookie information from the web service after a call is made. The storage used here is the ViewState collection of the form (a useful way to persist information between postbacks, which works in a similar way to storing information at the application or session level):

```
    if (ViewState["serviceCookie"] == null)
    {
        serviceCookie = new CookieContainer();
    }
    else
    {
        serviceCookie = (CookieContainer)ViewState["serviceCookie"];
    }
    myService.CookieContainer = serviceCookie;

```

Page_Load() then looks to see if there is a stored header and assigns the header to the proxy accordingly (assigning the header in this way is the only step you must take for the data to be sent as a SOAP header). This way any event handlers that are being called (such as the one for the web method-invoking button) don't have to assign a header — that step has already been taken:

```
    AuthenticationToken header = new AuthenticationToken();
    if (ViewState["AuthenticationTokenHeader"] != null)
    {
        header.InnerToken = (Guid)ViewState["AuthenticationTokenHeader"];
    }
    else
    {
        header.InnerToken = Guid.Empty;
    }
    myService.AuthenticationTokenValue = header;
}

```

Next, you add an event handler for the Login button by double-clicking it in the Designer:

```
protected void loginButton_Click(object sender, EventArgs e)
{
    Guid authenticationTokenHeader = myService.Login(userNameBox.Text,
        passwordBox.Text);
    tokenLabel.Text = authenticationTokenHeader.ToString();
    if (ViewState["AuthenticationTokenHeader"] != null)
    {
        ViewState.Remove("AuthenticationTokenHeader");
    }
}

```

```

ViewState.Add("AuthenticationTokenHeader", authenticationTokenHeader);
if (ViewState["serviceCookie"] != null)
{
    ViewState.Remove("serviceCookie");
}
ViewState.Add("serviceCookie", myService.CookieContainer);
}

```

This handler uses any data entered in the two text boxes to call `Login()`, displays the `Guid` returned, and stores the `Guid` in the `ViewState` collection. It also updates the `CookieContainer` stored in the `ViewState` collection, ready for reuse.

Finally, you have to add a handler in the same way for the `Invoke DoSomething()` button:

```

protected void invokeButton_Click(object sender, EventArgs e)
{
    resultLabel.Text = myService.DoSomething();
    if (ViewState["serviceCookie"] != null)
    {
        ViewState.Remove("serviceCookie");
    }
    ViewState.Add("serviceCookie", myService.CookieContainer);
}

```

This handler simply outputs the text returned by `DoSomething()` and updates the `CookieContainer` storage just like `loginButton_Click()`.

When you run this application, you can click the `Invoke DoSomething()` button straight away, because `Page_Load()` has assigned a header for you to use (if no header is assigned, an exception will be thrown because you have specified that the header is required for this method). This results in a failure message, returned from `DoSomething()`, as shown in Figure 55-4.

If you try to log in with any username and password except “Karli” and “Cheese” you will get the same result. If, on the other hand, you log in using these credentials and then call `DoSomething()`, you get the success message, as shown in Figure 55-5.

You can also see a string representation of the `Guid` used for validation.

Of course, applications that use this technique of exchanging data via SOAP headers are likely to be far more complicated. You may decide to store login tokens in a more scalable way than just using session storage, perhaps in a database. For completeness you can also implement your own expiration scheme for these tokens when a certain amount of time has passed and provide the option for users to log out, which would simply mean removing the token. Session state expires after a certain amount of time (20 minutes by default), but more complicated and powerful schemes are possible. You could even validate the token against the IP address used by the user for further security. The key points here though are that the username and password of the user are only sent once, and that using a SOAP header simplifies later method calls.

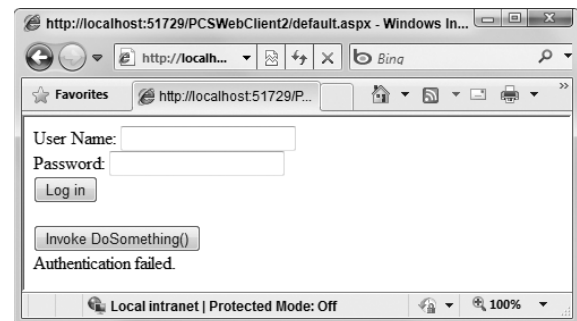


FIGURE 55-4

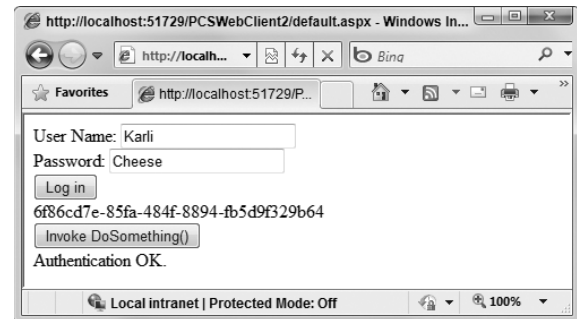


FIGURE 55-5

SUMMARY

In this chapter, you have seen how to create and consume web services using C# and the Visual Studio .NET development platform. Doing this is perhaps surprisingly simple, but is instantly recognizable as something that could prove to be incredibly useful.

It has also been pointed out that web services may be accessed from any platform. This is the result of using the SOAP protocol, which doesn't limit you to .NET.

The main example developed in this chapter illustrates how you can create .NET-distributed applications with ease. I have assumed here that you are using a single server to test things, but there is no reason why the web service shouldn't be completely separate from the client. It may even be on a separate server from the database if an additional data tier is required.

The use of data caching throughout is another important technique to master for use in large-scale applications, which might have thousands of users connecting simultaneously.

Exchanging data via SOAP headers, introduced in the last example, is another useful technique that can be worked into your applications. The example uses the exchange of a login token, but there is no reason why more complex data shouldn't be exchanged in this way. Perhaps this could be used for simple password protection of web services, without having to resort to imposing more complex security.

Remember that web service consumers don't necessarily have to be web applications. There is no reason why you can't use web services from Windows Forms or WPF applications — which certainly seems to be an attractive option for a corporate intranet. One added bonus concerning web services in ASP.NET is the ability to call web services asynchronously, using a simple, event-driven system. This fits in perfectly with Windows Forms applications, and means that you can keep your applications responsive while complicated web services carry out hard work for you.

As a final note, it is worth mentioning that web services can be seen as a stripped down subset of the more recent WCF technology, covered in Chapter 43, "Windows Communication Foundation." This doesn't mean, though, that you should always use WCF services rather than web services — web services are much simpler to implement and the simplicity is a great thing.