CHAPTER 23

Enhancing Web Services

The previous chapter examined how to create and consume basic web services. You learned how to code the StockQuote web service and how to call it from an ASP.NET web client. In this chapter, you'll add a few more frills to the mix, including state management, security, and transactions. As you'll see, not all of these features lend themselves easily to the web services model. Although these features are available, some complicate your web service and don't keep to the cross-platform ideal of SOAP. In many cases the best web services are the simplest web services.

Finally, for a little more fun, you'll see how to access a real, mature web service on the Internet: Microsoft's TerraService, which exposes a vast library of millions of satellite pictures from a database that is 1.5 terabytes in size. You'll build both a web client and a Windows client that perform a few basic tasks with TerraService.

State Management

Web services are usually designed as *stateless* classes. This means a web service provides a collection of utility functions that don't need to be called in any particular order and don't retain any information between requests.

This design is used for a number of reasons. First, retaining any kind of state uses memory on the web server, which reduces performance as the number of clients grows. Another problem is that contacting a web service takes time. If you can retrieve all the information you need at once and process it on the client, it's generally faster than making several separate calls to a web service that maintains state.

Several kinds of state management are available in ASP.NET, including using the Application and Session collections or using custom cookies. All of these techniques are also applicable to web services. However, with web services, session management is disabled by default. Otherwise, the server needs to check for session information each time a request is received, which imposes a slight overhead.

You can enable session state management for a specific method by using the EnableSession property of the WebMethod attribute:

```
<WebMethod(EnableSession:=True)> _
```

Before you implement state management, make sure you review whether you can accomplish your goal in another way. Many ASP.NET gurus believe that maintaining state is at best a performance drag and is, at worst, a contradiction to the philosophy of lightweight stateless web services. For example, instead of using a dedicated function to submit client preferences, you could add extra parameters to all your functions to accommodate the required information. Similarly, instead of allowing a user to manipulate a large DataSet stored in the Session collection on the server, you could return the entire object to the client and then clear it from memory.

■ **Tip** When evaluating state management, you have to consider many application-specific details. But in general, it's always a good idea to reduce the amount of information stored in memory on the web server, especially if you want to create a web service that can scale to serve hundreds of simultaneous users.

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

The StockQuote Service with State Management

The following example introduces state management into the StockQuote web service. It provides two methods: the standard GetStockQuote() method and a GetStockUsage() method that returns a CounterInfo object with information about how many times the web method was used.

```
<WebService(Description:="Methods to get stock information.", _
 Namespace:="http://www.prosetech.com/Stocks")> _
Public Class StockQuote_SessionState
    Inherits WebService

    <WebMethod(EnableSession:=True)> _
    Public Function GetStockQuote(ByVal ticker As String) As Decimal
        ' Increment counters. This function locks the application
        ' collection to prevent synchronization errors.
        Application.Lock()
```

```
        If Application(ticker) Is Nothing Then
            Application(ticker) = 1
        Else
            Application(ticker) = CType(Application(ticker), Integer) + 1
        End If

        Application.UnLock()

        If Session(ticker) Is Nothing
            Session(ticker) = 1
        Else
            Session(ticker) = CType(Session(ticker), Integer) + 1
        End If

        ' Return a value representing the length of the ticker.
        Return ticker.Length
    End Function

    <WebMethod(EnableSession:=True)> _
    Public Function GetStockUsage( ByVal ticker As String) As CounterInfo
        Dim result As New CounterInfo()
        result.GlobalRequests = CType(Application(ticker), Integer)
        result.SessionRequests = CType(Session(ticker), Integer)
        Return result
    End Function
End Class

Public Class CounterInfo
    Public GlobalRequests As Integer
    Public SessionRequests As Integer
End Class
```

This example allows the StockQuote service to record how much it has been used. Every time the GetStockQuote() method is called, two counters are incremented. One is a global counter that tracks the total number of requests for that stock quote from all clients. The other one is a session-specific counter that represents how many times the current client has requested a quote for the specified stock. The ticker string is used to name the stored item. This means if a client requests quotes for 50 different stocks, 50 different counter variables will be created. Clearly, this system wouldn't be practical for large-scale use. One practice that this example does follow correctly is that of locking the Application collection before updating it. This can cause a significant performance slowdown, but it guarantees that the value will be accurate even if multiple clients are accessing the web service simultaneously.

To view the counter information on the client, you can call the GetStockUsage() function with the ticker for the stock you're interested in. A custom object, CounterInfo, is returned to you with both pieces of information.

It's unlikely that you'd design an application to store this much usage information in memory, but it gives you a good indication of how you can use session and application state management in a web service just as easily as in an ASP.NET page. Note that session state automatically expires after a preset amount of time, and both the Session and Application collections will be emptied when the application is stopped. For more information about ASP.NET state management, refer to Chapter 9.

Consuming a Stateful Web Service

If you try to use a stateful web service, you're likely to be frustrated. Every time you recreate the proxy object, a new session ID is automatically assigned, which means you won't be able to access the session information from the previous request. To counter this problem, you need to take extra steps to preserve the cookie that has the session ID.

In a web client, you'll need to store the session ID between requests and then reapply it to the proxy object just before you make a call to the web service. You can use just about any type of storage: a database, a local cookie, the view state, or even the Session collection for the current web page. In other words, you could store information for your web service session in your current (local) page session.

If the previous discussion seems a little confusing, it's because you need to remind yourself that this example really has *two* active sessions. The web service client (which is itself a web application in this scenario) has its own session. Additionally, the web service has its own, separate session. The proxy class bears the burden of holding the web service session information (namely, the cookie). Figure 23-1 illustrates the difference.

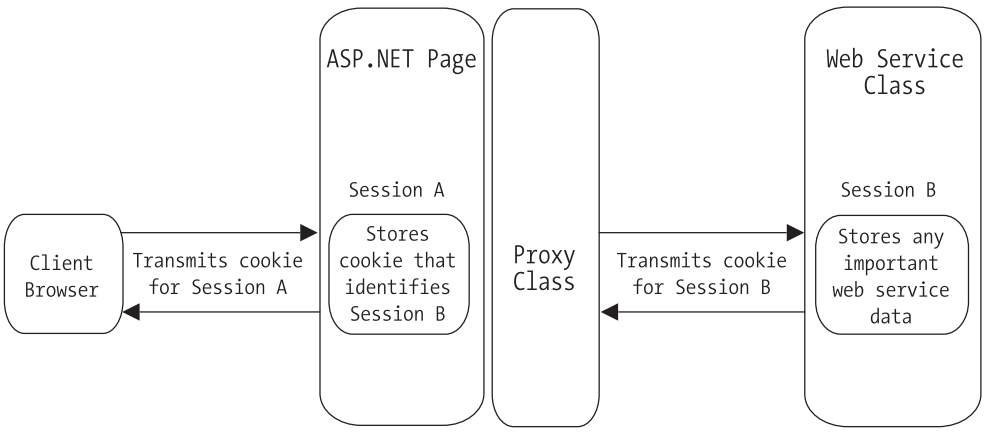

Figure 23-1. *Session cookies with a web client*

The next example sheds some light on this situation. The client is a simple web page with two buttons and a label. Every time the Call Service button is clicked, the GetStockQuote() method is invoked with a default parameter. When the Get Usage Info button is clicked, the GetStockUsage() method is called, which returns the current usage statistics. Information is added to the label, creating a log that records every action.

```
Public Partial Class FailedSessionServiceTest
    Inherits Page

    Private ws As New localhost.StockQuote_SessionState()

    Protected Sub cmdGetCounter_Click(ByVal sender As Object, _
      ByVal e As System.EventArgs) Handles cmdGetCounter.Click
        Dim wsInfo As localhost.CounterInfo = ws.GetStockUsage("MSFT")

        ' Add usage information to the label.
        lblResult.Text &= "<b>Global: " & wsInfo.GlobalRequests.ToString()
        lblResult.Text &= "<br />Session: "
        lblResult.Text &= wsInfo.SessionRequests.ToString() & "<br /></b>"
    End Sub

    Protected Sub cmdCallService_Click(ByVal sender As Object, _
      ByVal e As System.EventArgs) Handles cmdCallService.Click
        Dim result As Decimal = ws.GetStockQuote("MSFT")

        ' Add confirmation message to the label.
        lblResult.Text &= "GetStockQuote With MSFT Returned "
        lblResult.Text &= result.ToString() & "<br />"
    End Sub

End Class
```

Unfortunately, every time this button is clicked, the proxy object is recreated, and a new session is used. The output on the page tells an interesting story: after clicking the Call Service four times and the Get Usage button once, the web service reports the right number of global application requests but the wrong number of session requests, as shown in Figure 23-2.

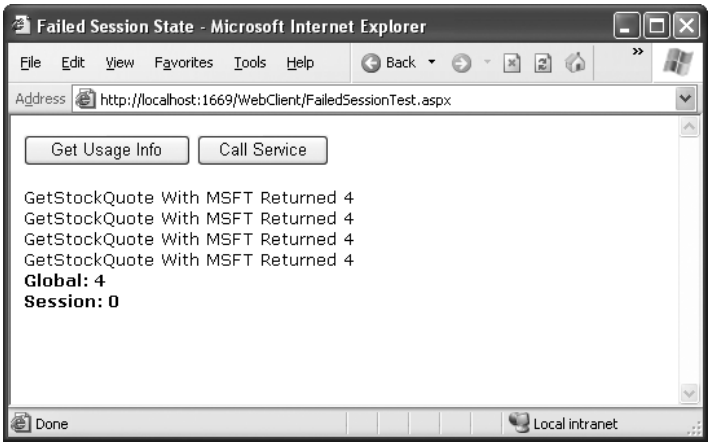

Figure 23-2. *A failed attempt to use session state*

To fix this problem, you need to set the CookieContainer property of the proxy class. First be sure to import the System.Net class in the code-behind for the web page:

```
Imports System.Net
```

This gives you easy access to the CookieContainer class, which is a straightforward container that can hold any number of cookies. In this example, the CookieContainer is used by the proxy class to hold the cookie with the session ID.

The next step is to modify the event handlers. In the following example, the event handlers call two private functions: GetCookie() and SetCookie(). These methods are called immediately before and after the web method call.

```
Protected Sub cmdGetCounter_Click(ByVal sender As Object, _
  ByVal e As System.EventArgs) Handles cmdGetCounter.Click
    GetCookie()
    Dim wsInfo As localhost.CounterInfo = ws.GetStockUsage("MSFT")
    SetCookie()

    ' Add usage information to the label.
    lblResult.Text &= "<b>Global: " & wsInfo.GlobalRequests.ToString()
    lblResult.Text &= "<br />Session: "
    lblResult.Text &= wsInfo.SessionRequests.ToString() & "<br /></b>"
End Sub

Protected Sub cmdCallService_Click(ByVal sender As Object, _
  ByVal e As System.EventArgs) Handles cmdCallService.Click
    GetCookie()
```

```
    Dim result As Decimal = ws.GetStockQuote("MSFT")
    SetCookie()

    ' Add confirmation message to the label.
    lblResult.Text &= "GetStockQuote With MSFT Returned "
    lblResult.Text &= result.ToString() & "<br />"
End Sub
```

The GetCookie() method initializes the CookieContainer for the proxy object, enabling it to send cookies to the web service, and receive them as well. But there's a catch. When the web service returns the cookie, the web page client needs to store that cookie somewhere in order to use it later. In this example, the web page stores it in its session state using the SetCookie() method. The GetCookie() method always checks this location to see if there's a cookie from a previous method invocation waiting to be used.

Here's the complete code for the GetCookie() method:

```
Private Sub GetCookie()
    ' Initialize the proxy object CookieContainer so it can receive cookies.
    ws.CookieContainer = New CookieContainer()

    ' If a cookie exists from a previous call, retrieve it,
    ' and place it in the proxy object.
    ' If this is the first time the button has been clicked,
    ' no cookie will be present.
    If Session("WebServiceCookie") IsNot Nothing Then
        ' Retrieve the cookie object from session state.
        Dim sessionCookie As Cookie
        sessionCookie = CType(Session("WebServiceCookie"), Cookie)
        ws.CookieContainer.Add(sessionCookie)
    End If
End Sub
```

The SetCookie() method explicitly stores the web service session cookie in session state:

```
Private Sub SetCookie()
    ' Retrieve and store the web service cookie for next time.
    ' The session is always stored in a special cookie
    ' called ASP.NET_SessionId.
    Dim wsUri As New Uri("http://localhost")

    Dim cookies As CookieCollection
    cookies = ws.CookieContainer.GetCookies(wsUri)
    Session("WebServiceCookie") = cookies("ASP.NET_SessionId")
End Sub
```

The code could have been simplified somewhat if it stored the whole cookie collection instead of just the session cookie, but this approach is more controlled.

Now the page works correctly, as shown in Figure 23-3.

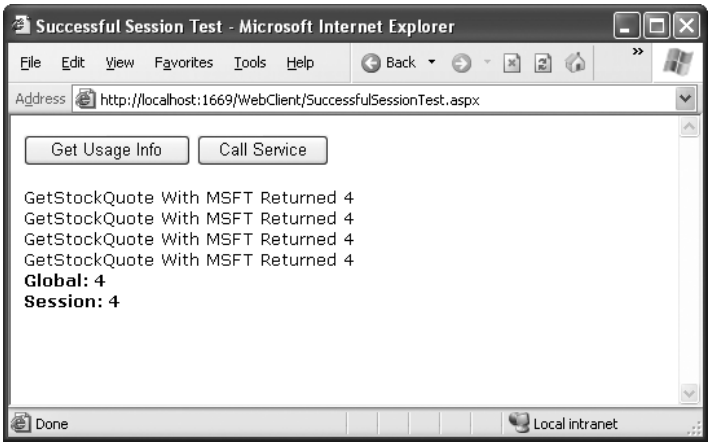

Figure 23-3. *A successful use of session state*

DOES SESSION STATE MANAGEMENT MAKE SENSE WITH A WEB SERVICE?

Because web services are destroyed after every method call, they don't provide a natural mechanism for storing state information. You can use the Session collection to compensate for this limitation, but this approach raises the following complications:

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

Web Service Security

Most of the security issues that relate to web pages also affect web services. For example, web services transmit messages using clear text, which means a malicious user with access to the network could easily spy on confidential information or even modify a message as it's transmitted. To prevent both of these possibilities in a web service that requires complete

confidentiality, your best bet is SSL, as described in Chapter 18. SSL doesn't mesh perfectly with the world of web services—it's a little more heavyweight than is ideal, and it complicates business-to-business transactions with multiple web services—but it's the safest choice for today's web services. Other SOAP-based encryption and signing standards that address these problems are evolving, but they're still in flux. You can get more information and even try them in your web services by downloading Microsoft's WSE (Web Services Enhancements) toolkit at `http://msdn.microsoft.com/webservices/building/wse`. Just expect to adapt as the standards of today continue to change.

Along with the issue of encryption, web services may also need the same ability to authenticate users as web pages. For example, you may want to verify that a user is allowed to access a web service (or a particular web method) before you perform a task.

You have two options for setting up authentication with a web service:

- Your first option is to use the Windows authentication features that are built into IIS and ASP.NET. Chapter 18 describes these standard security features. The only difference between ASP.NET security in a web service and in a web page is that a web service doesn't present any user interface, so it cannot request user credentials on its own. It has to rely on the calling code or rely on having the user logged on under an authorized account.

■ **Note** Windows authentication is a realistic solution only if all your users are on the same network, running Windows, and they have preexisting user accounts.

- For maximum flexibility, you can create and use your own login and authentication code. Sadly, forms authentication won't help you here—its reliance on cookies and its assumption that the end user will log in through a web page mean it won't work in a web service scenario. However, you can use the membership features discussed in Chapter 19 to reduce the code you write for the login process.

In the following sections, you'll consider both approaches.

Windows Authentication with a Web Service

Windows authentication works with a web service in much the same way it works with a web page. The difference is that a web service is executed by another application, not directly by the browser. For that reason, a web service has no built-in way to prompt the user for a user name and password. Instead, the application that's using the web service needs to supply this information. The application might read this information from a configuration file or a database, or it might prompt the user for this information before contacting the web service.

For example, consider the following web service, which provides a single TestAuthenticated() method. This method checks whether the user is authenticated. If the user is authenticated, it returns the full user name.

```
Public Class SecureService
    Inherits System.Web.Services.WebService

    <WebMethod()> _
    Public Function TestAuthenticated() As String
        If Not User.Identity.IsAuthenticated Then
            Return "Not authenticated."
        Else
            Return "Authenticated as: " & User.Identity.Name
        End If
    End Sub
End Class
```

■ **Note** To use this approach, you also need to ensure the <authentication> tag is set to use Windows authentication (the default), and you need to configure the authentication protocols you want to support using IIS Manager. Chapter 18 has the full details about configuring a web application to use Windows authentication.

The first time you test this web service, you'll find that the user isn't authenticated. As with a web page, authentication springs into action only when you explicitly deny anonymous users. To do this, you can configure the virtual directory in IIS Manager (as described in Chapter 18) or add an authorization rule that targets a specific page or folder. For example, here's what you need to require authentication for the SecureService.asmx method:

```
<configuration>
    <location path="SecureService.asmx">
        <system.web>
            <authorization>
                <deny users="?"/>
            </authorization>
        </system.web>
    </location>

    <system.web>
      ...
    </system.web>
</configuration>
```

It's not possible to turn authentication on or off for individual methods. As a result, if you want some methods to use authentication and others to not require it, you'll need to code these methods in separate web services.

If you request SecureService.asmx in a browser, you'll get the expected result. The browser will authenticate you, and then show you the test page. Using the test page you can invoke the TestAuthenticated() method, which will return your full user name as a string in the form [DomainName]\[UserName] or [ComputerName]\[UserName]).

However, if you try to call your web service from a client, you won't be as lucky. Instead, you'll receive an Access Denied error message. This indicates that authentication was required but the client didn't supply the correct authentication credentials.

The problem is that, by default, the proxy class won't supply any authentication information (unlike a browser, which prompts the user to log in). To submit user credentials to this service, the client needs to modify the NetworkCredential property of the proxy class. You have two options:

- You can create a new NetworkCredential object and attach this to the NetworkCredential property of the proxy object. When you create the NetworkCredential object, you'll need to specify the user name and password you want to use. This approach works with all forms of Windows authentication.
- If the web service is using Integrated Windows authentication, you can automatically submit the credentials of the current user by using the DefaultCredentials shared property of the CredentialCache class and applying that to the NetworkCredential property of the proxy object.

Both the CredentialCache and the NetworkCredential classes are found in the System.Net namespace. Thus, before continuing, you should import this namespace:

```
Imports System.Net
```

The following code shows a web page with two buttons. One button performs an unauthenticated call, while the user submits some user credentials. The unauthenticated call will fail if you've disabled anonymous users for the web application. Otherwise, the unauthenticated call will succeed, but the TestAuthenticated() method will return a string informing you that authentication wasn't performed. The authenticated call will always succeed as long as you submit credentials that correspond to a valid user on the web server.

```
Public Partial Class TestSecuredService
    Inherits Page

    Protected Sub cmdUnauthenticated_Click(ByVal sender As Object, _
      ByVal e As System.EventArgs) Handles cmdUnauthenticated.Click
        Dim ws As New localhost.SecureService()
        lblInfo.Text = ws.TestAuthenticated()
    End Sub
```

```
    Protected Sub cmdAuthenticated_Click(ByVal sender As Object, _
      ByVal e As System.EventArgs) Handles cmdAuthenticated.Click
        Dim ws As New localhost.SecureService()

        ' Specify some user credentials for the web service.
        ' This example uses the user account "GuestAccount" with the password
        ' "Guest".
        Dim credentials As New NetworkCredential("GuestAccount", "Guest")
        ws.Credentials = credentials

        lblInfo.Text = ws.TestAuthenticated()
    End Sub
End Class
```

Figure 23-4 shows what the web page looks like after a successfully authenticated call to the computer fariamat.

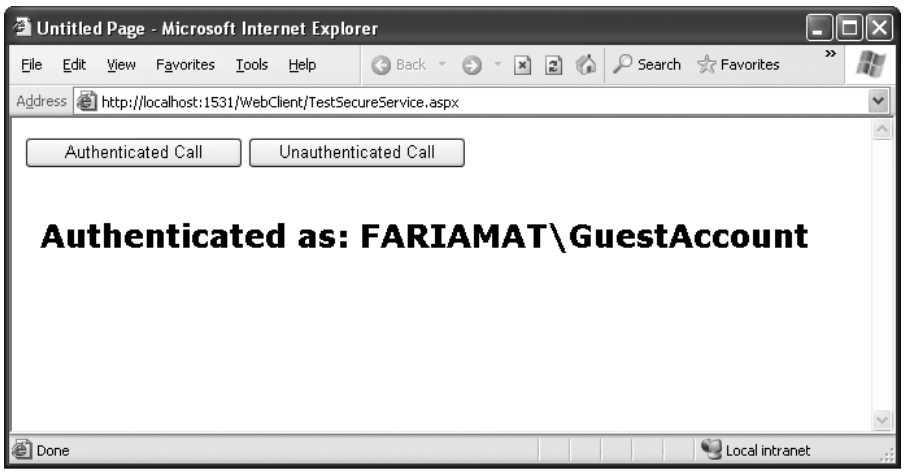

Figure 23-4. *Successful authentication through a web service*

If you want to use the credentials of the currently logged-in account with Integrated Windows authentication, you would use this code instead:

```
Dim ws As New localhost.SecureService()
ws.Credentials = CredentialCache.DefaultCredentials
lblInfo.Text = ws.TestAuthenticated()
```

Keep in mind that this probably won't have the result you want in a deployed web application. On a live web server, the current user account is always the account that ASP.NET is using, not the user account of the remote user who is requesting the web

page. (For example, if you're using IIS 5, the ASP.NET account runs all ASP.NET code.) However, this technique makes sense in a Windows application. In this case, when you retrieve the default credentials, you'll submit the account information of the user who is running the application.

■ **Note** Forms authentication won't work with a web service because there's no way for a web service to direct the user to a web page. In fact, the web service might not even be accessed through a browser—it might be used by a Windows application or even an automated Windows service.

Ticket-Based Authentication

Many web services use their own custom authentication systems. Usually, they use a form of *ticket-based authentication* to increase performance and make the coding model. With ticket-based authentication, clients call a specific web method in the web service to log in, at which point they will supply credentials (such as a user name and password combination). The login method will then create a new, unique ticket and return it to the client. From this point, the client can use any method in the web service, as long as the client supplies the ticket.

The benefit of ticket-based authentication is that you need only one authentication step, which is properly separated from the rest of your code. On subsequent requests, your web service can verify the ticket rather than authenticating the user against the database, which is always faster (in a properly designed ticket system). Finally, ticket-based authentication allows you to take advantage of SOAP headers (which you'll see shortly). SOAP headers make the ticket management and authorization process transparent to the client.

Fortunately, you can get most of the benefits of a ticket-based authentication system without writing your own authentication logic. Instead, you can use the membership and role management features that ASP.NET provides (as discussed in Chapter 19). It's still up to you to transfer the user credentials and keep track of who has logged in by issuing and verifying tickets. However, you don't need to write the authentication code that checks whether a user and password combination is valid.

The following example shows how you could adapt the StockQuote service to use ticket-based authentication. In order to use this code as written, you also need to import the System.Security namespace.

```
<WebService(Description:="Methods to get stock information.", _
 Namespace:="http://www.prosetech.com/Stocks")> _
Public Class StockQuote_Security
    Inherits WebService
```

```
    <WebMethod()> _
    Public Function Login(ByVal username As String, ByVal password As String) _
      As LicenseKey
        If Membership.ValidateUser(userName, password) Then
            ' Generate a license key made up of
            ' some random sequence of characters.
            Dim key As New LicenseKey()
            key.Ticket = Guid.NewGuid().ToString()

            ' Add the key object to application state.
            Application(key.Ticket) = key

            Return key
        Else
            ' Cause an error that will be returned to the client.
            Throw New SecurityException("Unauthorized.")
        End If
    End Function

    <WebMethod()> _
    Public Function GetStockQuote(ByVal ticker As String, _
      ByVal key As LicenseKey) As Decimal
        If Not VerifyKey(key.Ticket) Then
            Throw New SecurityException("Unauthorized.")
        Else
            ' Normal GetStockQuote code goes here.
            Return ticker.Length
        End If
    End Function

    Private Function VerifyKey(ByVal ticket As String) As Boolean
        ' Look up this key to make sure it's valid.
        If Application(ticket) Is Nothing
            Return False
        Else
            Return True
        End If
    End Function
End Class

Public Class LicenseKey
    Public Ticket As String
End Class
```

Dissecting the Code...

- Before using any method in this class, the client must call the Login() web method and store the received license key. Every other web method will require that license key.
- The Login() method uses the Membership class, which performs the user authentication for you. To use membership in a web application, you need to configure the membership database. However, if you're using SQL Server 2005 Express Edition, the default settings will create the required database for you in the App_Data directory automatically. Chapter 19 has the full details.
- The LicenseKey class uses a GUID. This is a randomly generated 128-bit number that is guaranteed to be statistically unique. (In other words, the chance of generating two unique GUIDs in a web service or guessing another user's GUID is astronomically small.) You could also add other information to the LicenseKey class about the current user to retrieve later as needed. For example, you could keep track of the date the license was issued.
- The same private function—VerifyKey()—is used regardless of what web method is invoked. This ensures that the license verification code is written in only one place and is used consistently.
- The Login() method stores the key in server memory using application state. The Application collection has certain limitations, including no support for multiple web servers (web farms). The tickets will also be lost if the web application restarts.

■ **Note** You could add data caching to this design to get around the limitations of the Application class. With data caching, you would store every key in two places—in the database, which would be used as a last resort, and duplicated in the in-memory cache. The VerifyKey() function would perform the database lookup only if the cached item is not found. For more information about caching, refer to Chapter 26. However, even if you make this enhancement, the essential design of this web service remains the same. The only difference is how the LicenseKey is stored and retrieved in between requests.

Ticket-Based Authentication with SOAP Headers

The potential problem with this example is that it requires a license key to be submitted with each method call as a parameter. This can be a bit tedious. To streamline this process, you can use a custom SOAP header.

SOAP headers are separate pieces of information that are sent, when required, in the header section of a SOAP message. These headers can contain all the same data types you use for method parameters and return values. The advantage of SOAP headers is just the

convenience. The client specifies a SOAP header once, and the header is automatically sent to every method that needs it, making the coding clearer and more concise. You can also change the header details without being forced to edit the signature of every web method.

To create a custom SOAP header, you first need to define a class that inherits from the SoapHeader class (which is found in the System.Web.Services.Protocols namespace). You can then add all the additional pieces of information you want it to contain as member variables. In the following example, the SOAP header simply stores the license key:

```
Public Class LicenseKeyHeader
    Inherits System.Web.Services.Protocols.SoapHeader

    Public Ticket As String

    Public Sub New(ByVal ticket As String)
        Me.Ticket = ticket
    End Sub

    Public Sub New()
        Ticket = Guid.NewGuid().ToString()
    End Sub
End Class
```

Notice that this class is slightly enhanced over the LicenseKey class in the previous example. The default constructor now generates a random ticket value automatically.

Next, your web service needs a public member variable to receive the SOAP header:

```
Public Class StockQuote_SoapSecurity
    Inherits WebService

    Public KeyHeader As LicenseKeyHeader

    ' (Other web service code goes here.)
End Sub
```

The web service requires one last ingredient. Each web service method that wants to access the LicenseKeyHeader must explicitly indicate this requirement with a SoapHeader attribute. The attribute indicates the web service member variable where the header should be placed.

```
<WebMethod()> _
<SoapHeader("KeyHeader", Direction:=SoapHeaderDirection.In)> _
Public Function GetStockQuote(ByVal ticker As String) As Decimal
    ...
End Function
```

The Direction parameter indicates that this method *receives* the header with the ticket information. That's different from the Login() method, which creates the header and returns it to the client.

The complete web service is quite similar to the earlier example. The chief difference is that when the client calls the Login() method, no information is provided as a return value. Instead, a LicenseKeyHeader is created and sent silently to the client. The next time the client calls any methods, this header is transmitted automatically, which means those methods don't need to accept key information through their parameters.

Here's the full code:

```
<WebService(Description:="Methods to get stock information.", _
 Namespace:="http://www.prosetech.com/Stocks")> _
Public Class StockQuote_SoapSecurity
    Inherits WebService

    Public KeyHeader As LicenseKeyHeader

    <WebMethod()> _
    <SoapHeader("KeyHeader", Direction:=SoapHeaderDirection.Out)> _
    Public Sub Login(ByVal username As String, ByVal password As String)
        If Membership.ValidateUser(userName, password) Then
            ' Generate a license key, and store it in the SOAP header.
            KeyHeader = New LicenseKeyHeader()

            ' Store this user's KeyHeader object in application state.
            Application(KeyHeader.Ticket) = KeyHeader
        Else
            ' Cause an error that will be returned to the client.
            Throw New SecurityException("Unauthorized.")
        End If
    End Function

    <WebMethod()> _
    <SoapHeader("KeyHeader", Direction:=SoapHeaderDirection.In)> _
    Public Function GetStockQuote(ByVal ticker As String) _
      As Decimal
        If Not VerifyKey(KeyHeader.Ticket) Then
            Throw New SecurityException("Unauthorized.")
        Else
            ' Normal GetStockQuote code goes here.
            Return ticker.Length
        End If
    End Function
```

```
    Private Function VerifyKey(ByVal ticket As String) As Boolean
        ' Look up this key to make sure it's valid.
        If Application(ticket) Is Nothing
            Return False
        Else
            Return True
        End If
    End Function
End Class
```

■ **Caution** SOAP messages are sent over the network as ordinary text. This means if you create an authentication system like this, malicious users could watch network traffic and intercept passwords as they are sent from the client to the web service. To solve this problem, you can use SSL with your web service to encrypt all the messages that are exchanged. Refer to Chapter 18 for more information.

Using SOAP Headers in the Client

Some web services allow information to be specified once and used for multiple methods automatically. This technique works through SOAP headers. The previous example showed a web service that used a SOAP header to receive and maintain security credentials. When you create a client for this service, the custom SoapHeader class is copied into the proxy file.

Using this web service is remarkably easy. First you call the login method and supply your user credentials:

```
' Create the web service proxy class.
Dim ws As New localhost.StockQuote_SoapSecurity()

' Log in.
ws.Login("testUser", "openSesame")
```

If this test succeeds, the web service issues a new ticket, which is attached to the proxy object. From this point, whenever you call a web method that needs the LicenseKeyHeader, it will be transmitted automatically.

```
Dim price As Decimal = ws.GetStockQuote("MSFT")
```

In other words, you set the header once and don't need to worry about supplying any additional security information, as long as you use the same instance of the proxy class. The online examples include a simple example of a web service client that uses a SOAP header for authentication.

Web Service Transactions

Transactions are an important feature in many business applications. They ensure that a series of operations either succeeds or fails as a unit. Transactions also prevent the possibility of inconsistent or corrupted data that can occur if an operation fails midway after committing only some of its changes. The most common example of a transaction is a bank account transfer. When you move $100 from one account to another, two actions take place: a withdrawal in the first account, and a deposit in the second. If these two tasks are a part of a single transaction, they will either both succeed or both fail. It will be impossible for one account to be debited if the other isn't credited.

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

To use a transaction in a web service, you first have to add a reference to the System.EnterpriseServices.dll assembly. To do this, choose Website ➤ Add Reference, and select the System.EnterpriseServices entry on the .NET tab.

You can now import the corresponding namespace so the types you need (TransactionOption and ContextUtil) are at your fingertips:

```
Imports System.EnterpriseServices
```

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

```
<WebMethod(TransactionOption:=TransactionOption.RequiresNew)> _
```

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

```
ContextUtil.SetAbort()
```

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

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

```
<WebMethod(TransactionOption:=TransactionOption.RequiresNew)> _
Public Sub UpdateDatabase()
    ' Create ADO.NET objects.
    Dim connectionString As String = _
      WebConfigurationManager.ConnectionStrings("Northwind").ConnectionString
    Dim con As New SqlConnection(connectionString)
    Dim cmd As New SqlCommand("DELETE * FROM authors", con)

    ' Apply the update. This will be registered as part of the transaction.
    Using con
        con.Open()
        cmd.ExecuteNonQuery()
    End Using

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

    ' (If no errors have occurred, the database changes
    ' are committed here when the method ends).
End Sub
```

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

```
<WebMethod(TransactionOption:=TransactionOption.RequiresNew)> _
Public Sub UpdateDatabase()
    ' Create ADO.NET objects.
    Dim connectionString As String = _
      WebConfigurationManager.ConnectionStrings("Northwind").ConnectionString
    Dim con As New SqlConnection(connectionString)
    Dim cmd As New SqlCommand("DELETE * FROM authors", con)
```

```
    ' Apply the update.
    Try
        con.Open()
        cmd.ExecuteNonQuery()

        Dim fs As New FileStream("does_not_exist.bin", FileMode.Open)
    Catch
        ContextUtil.SetAbort()
    Finally
        con.Close()
    End Try
End Sub
```

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

An Example with TerraService

Now that you know how to use a web service, you aren't limited just to using the web services that you create. In fact, the Internet is brimming with sample services you can use to test your applications or to try new techniques. In the future, you'll even be able to buy prebuilt functionality you can integrate into your ASP.NET applications by calling a web service. Typically, you'll pay for these services using some kind of subscription model. Because they're implemented through .NET web services and WSDL contracts, you won't need to install anything extra on your web server.

The next sections cover how to use one of the more interesting web services: Microsoft's TerraService. TerraService is based on the hugely popular TerraServer site where web surfers can view topographic maps and satellite photographs of most of the globe. The site was developed by Microsoft's research division to test SQL Server and increase Microsoft's boasting ability. Under the hood, a 1.5TB SQL Server 2000 database stores all the pictures that are used as a collection of millions of different tiles. You can find out more about TerraServer at `http://terraservice.net`.

To promote .NET, Microsoft has equipped TerraServer with a web service interface that allows you to access the TerraServer information. Using this interface (called TerraService) isn't difficult, but creating a polished application with it is. Before you can really understand how to stitch together different satellite tiles to create a large picture, you need to have some basic understanding of geography and projection systems. You can find plenty of additional information about it at the TerraServer site. However, because this book is about .NET and not geography, your use of TerraService will be fairly straightforward. Once you understand the basics, you can continue experimenting with additional web methods and create a more extensive application based on TerraService.

Adding the Reference

The first step is to create a new ASP.NET application and add a web reference. In Visual Studio, you start by returning to the Add Web Reference dialog box. The TerraService web service is located at `http://terraservice.net/TerraService.asmx`. Type it into the Address text box, and press Enter. The TerraService text page will appear (see Figure 23-5).

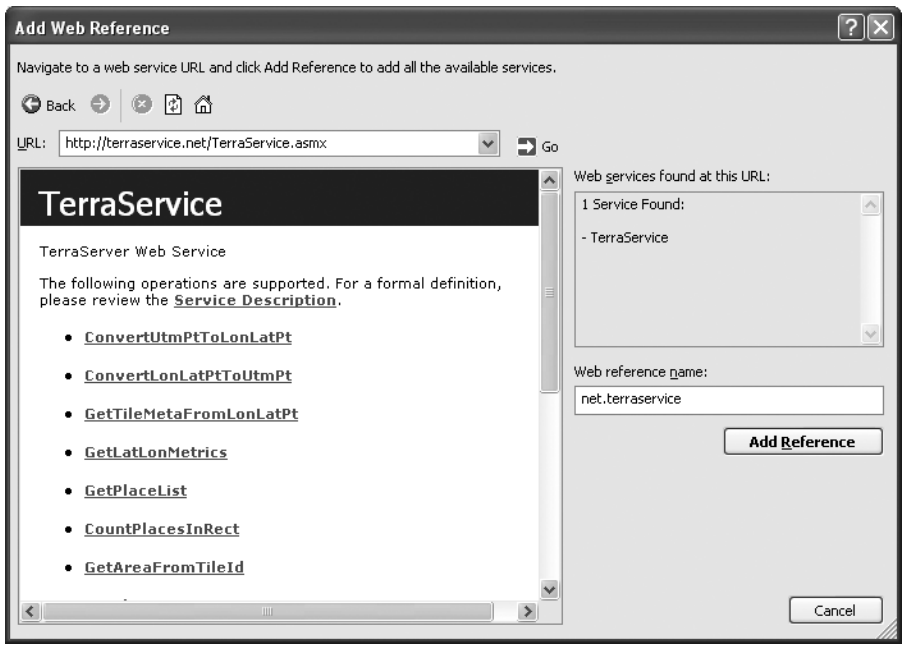

Figure 23-5. *Adding a TerraService web reference*

You can see from the displayed test page that there are approximately 15 functions. At the time of this writing, no additional information was provided on the test page (indicating that Microsoft doesn't always follow its own recommendations). To read about these methods, you'll have to browse the site on your own.

Click Add Reference to create the Web Reference. The WSDL document and discovery files will be created under the namespace net.terraservice, as shown in Figure 23-6.

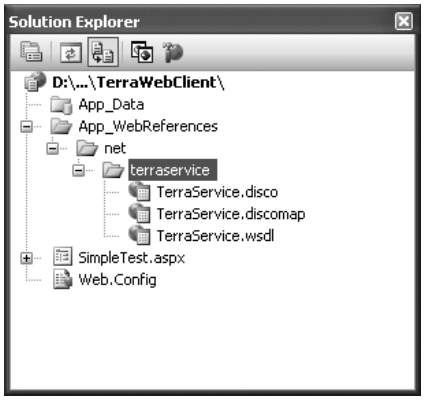

Figure 23-6. *The TerraService files in Visual Studio*

Testing the Client

Before continuing too much further, it makes sense to try a simple method to see whether the web service works as expected. A good choice to start is the simple GetPlaceFacts() method, which retrieves some simple information about a geographic location that you supply. In this case, the TerraService documentation (and the Visual Studio IntelliSense) informs you that this method requires the use of two special classes: a Place class (which specifies what you're searching for) and a PlaceFacts class (which provides you with the information about your place). These classes are available in the net.terraservice namespace.

The following example retrieves information about a place named Seattle when a button is clicked and displays it in a label. The process is split into two separate subroutines for better organization, although this isn't strictly required.

```
Private ts As New net.terraservice.TerraService()

Protected Sub cmdShow_Click(ByVal sender As Object, _
  ByVal e As System.EventArgs) Handles cmdShow.Click
    ' Create the Place object for Seattle.
    Dim searchPlace As New net.terraservice.Place()
    searchPlace.City = "Seattle"
    searchPlace.State = "Washington"
    searchPlace.Country = "US"
```

```
    ' Define the PlaceFacts objects to retrieve your information.
    Dim facts As net.terraservice.PlaceFacts

    ' Call the web service method.
    facts = ts.GetPlaceFacts(searchPlace)

    ' Display the results with the help of a subroutine.
    ShowPlaceFacts(facts)
End Sub

Private Sub ShowPlaceFacts(ByVal facts As net.terraservice.PlaceFacts)
    lblResult.Text &= "<b>Place: " & facts.Place.City & "</b><br /><br />"
    lblResult.Text &= facts.Place.State & ", " & facts.Place.Country
    lblResult.Text &= "<br /> Lat: " & facts.Center.Lat.ToString()
    lblResult.Text &= "<br /> Long: " & facts.Center.Lon.ToString()
    lblResult.Text &= "<br /><br />"
End Sub
```

The result is a successful retrieval of information about the city of Seattle, including its longitude, latitude, and country, as shown in Figure 23-7.

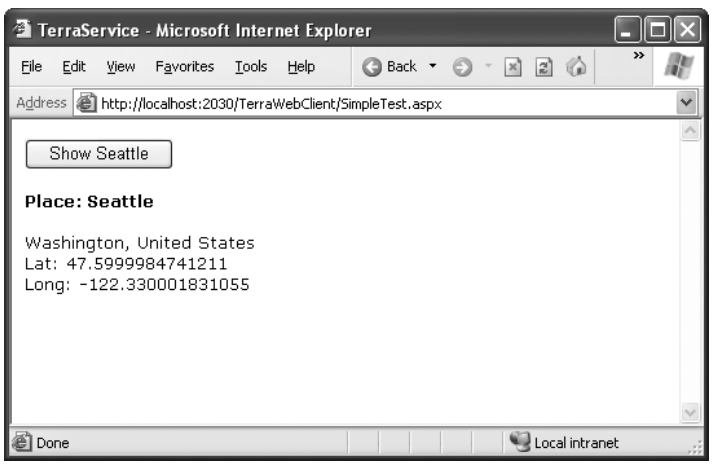

Figure 23-7. *Retrieving information from TerraService*

Searching for More Information

The TerraService documentation doesn't recommend relying on the GetPlaceFacts() method, because it's able to retrieve information only about the first place with the matching name. Typically, many locations share the same name. Even in the case of Seattle, several landmark locations are stored in the TerraServer database along with the city itself. All these places start with the word *Seattle*.

The GetPlaceList() method provides a more useful approach, because it returns an array with all the matches. When using GetPlaceList(), you have to specify a maximum number of allowed matches to prevent your application from becoming tied up with an extremely long query. You also have a third parameter, which you can set to True to retrieve only results that have corresponding picture tiles in the database, or to False to return any matching result. Typically, you would use True if you were creating an application that displays the satellite images for searched locations. You'll also notice that the GetPlaceList() function accepts a place name directly and doesn't use a Place object.

To demonstrate this method, add a second button to the test page, and use the following code:

```
Protected Sub cmdShowAll_Click(ByVal sender As Object, _
  ByVal e As System.EventArgs) Handles cmdShowAll.Click
    ' Retrieve the matching list (for the city Kingston).
    Dim factsArray() As net.terraservice.PlaceFacts
    factsArray = ts.GetPlaceList("Kingston", 100, False)

    ' Loop through all the results, and display them.
    For Each facts As net.terraservice.PlaceFacts In factsArray
        ShowPlaceFacts(facts)
    Next
End Sub
```

The result is a list of about 50 matches for places with the name Kingston, as shown in Figure 23-8.

Figure 23-8. *Retrieving place matches*

Displaying a Tile

By this point, you've realized that using TerraService isn't really different from using any other DLL or code library. The only difference is that you're accessing it over the Internet. In this case, it's no small feat: the TerraServer database contains so much information that it would be extremely difficult to download over the Internet. Providing an organized, carefully limited view through a web service interface solves all these problems.

The next example shows your last trick for a web service. It retrieves the closest matching tile for the city of Seattle and displays it. (A tile is a small, square section of satellite imagery.) To display the tile, the example takes the low-level approach of using Response.WriteBinary().

```
Protected Sub cmdShowPic_Click(ByVal sender As Object, _
  ByVal e As System.EventArgs) Handles cmdShowPic.Click
    ' Define the search.
    Dim searchPlace As New net.terraservice.Place()
    searchPlace.City = "Seattle"
    searchPlace.State = "Washington"
    searchPlace.Country = "US"

    ' Get the PlaceFacts for Seattle.
    Dim facts As net.terraservice.PlaceFacts
    facts = ts.GetPlaceFacts(searchPlace)

    ' Retrieve information about the tile at the center of Seattle, using
    ' the Scale and Theme enumerations from the terraservice namespace.
    Dim tileData As net.terraservice.TileMeta
    tileData = ts.GetTileMetaFromLonLatPt(facts.Center, _
     net.terraservice.Theme.Photo, net.terraservice.Scale.Scale16m)

    ' Retrieve the image.
    Dim image() As Byte = ts.GetTile(tileData.Id)

    ' Display the image.
    Response.BinaryWrite(image)
End Sub
```

In a more advanced application, a significant amount of in-memory graphical manipulation might be required to get the result you want. Figure 23-9 shows the retrieved tile.

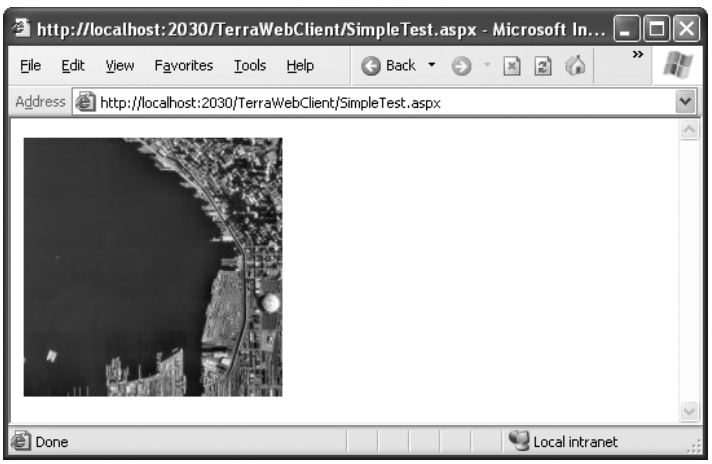

Figure 23-9. *A tile from TerraService*

This example uses two additional TerraService methods. GetTileMetaFromLonLatPt() retrieves information about the tile at a given point. This function finds out what tile contains the center of Seattle. The GetTile() method retrieves the binary information representing the tile by using the TileID provided by GetTileMetaFromLonLatPt().

You'll probably also want to combine several tiles, which requires the use of other TerraService methods. You can find the full set of supported methods documented at `http://terraservice.net/`. Once again, using these methods requires an understanding of graphics and geography, but the methods don't require any unusual use of web services. The core concept—remote method calls through SOAP communication—remains the same.

Windows Clients

Because this book focuses on ASP.NET, you haven't had the chance to see one of the main advantages of web services. Quite simply, they allow desktop applications to use pieces of functionality from the Internet. This allows you to provide a rich, responsive desktop interface that periodically retrieves any real-time information it needs from the Internet. The process is almost entirely transparent. As high-speed access becomes more common, you may not even be aware of which portions of functionality depend on the Internet and which ones don't.

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

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

The following example shows the form code for a simple Windows TerraService client that uses the GetPlaceList() web method. It has been modified to search for a place that the user enters in a text box and display the results in a list box.

```
Public Class TerraWindowsClient
    Private ts As new net.terraservice.TerraService()

    Private Sub cmdShow_Click(ByVal sender As System.Object, _
      ByVal e As System.EventArgs) Handles cmdShow.Click
        ' Retrieve the matching list.
```

```
        Dim factsArray() As net.terraservice.PlaceFacts
        factsArray = ts.GetPlaceList(txtPlace.Text, 100, False)

        ' Loop through all the results, and display them.
        For Each facts As net.terraservice.PlaceFacts in factsArray
            ShowPlaceFacts(facts)
        Next
    End Sub

    Private Sub ShowPlaceFacts(ByVal facts As net.terraservice.PlaceFacts)
        Dim newItem As String
        newItem = "Place: " & facts.Place.City & ", "
        newItem &= facts.Place.State & ", " & facts.Place.Country
        lstPlaces.Items.Add(newItem)
    End Sub
End Class
```

Figure 23-10 shows the interface for this application.

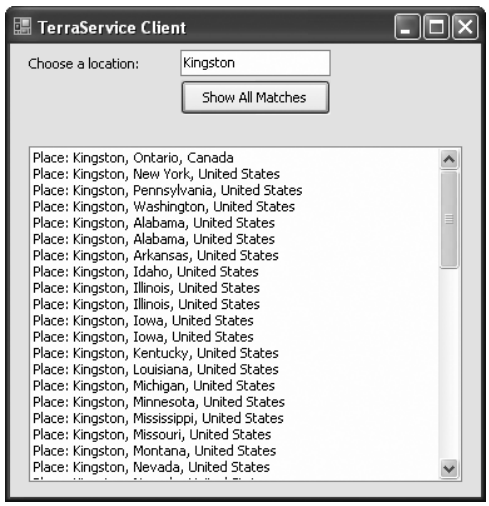

Figure 23-10. *A Windows client for TerraService*

Of course, Windows development contains many other possibilities, which are covered in many other excellent books. The interesting part from your vantage point is that a Windows client can interact with a web service just like an ASP.NET application does. This raises a world of new possibilities for integrated Windows and web applications.

The Last Word

This chapter explored some more ideas for developing your web services. You took an in-depth look at web service security and considered when (and if) it makes sense to use session state and transactions in a web service.

To keep learning about web services, it helps to look at examples on the Web, such as TerraService. For even more experimentation, consider some of the following web services:

- XMethods (`http://www.xmethods.com`) is a more general web service catalog. It includes many web services you can use in .NET applications, such as a currency exchange reporter and a delayed stock quote. Even though most of these web services run on non-.NET platforms, you can use them in your .NET applications seamlessly.

- Microsoft MapPoint (`http://msdn.microsoft.com/library/en-us/dnanchor/html/anch_mappointmain.asp`) is an interesting example that enables you to access high-quality maps and geographical information. MapPoint isn't free, but you can use a free trial of the web service.