7680CH33.qxd 8/25/06 10:51 AM Page 1247

CHAPTER 33

■ ■ ■

Advanced Web Services

In the past two chapters, you took a close look at how web services work with ASP.NET. Using the techniques you've learned already, you can create web services that expose data to other applications and organizations, and you can consume .NET and non-.NET web services on the Internet.

However, the story doesn't end there. In this chapter, you'll learn how to extend your web service skills with specific techniques that are often important in real-world web service scenarios. You'll focus on three topic areas:

Calling web services asynchronously: Web service calls take time, especially if the web server is located across the globe and connected by a slow network connection. By using asynchronous calls, you can keep working while you wait for a response.

Securing web services: In Part 4 you learned how you can secure web pages to prevent anonymous users. You can apply some of the same techniques to protect your web services.

Using SOAP extensions: The web services infrastructure is remarkably extensible, thanks to a SOAP extension model that allows you to create components that plug into the SOAP serialization and deserialization process. In this chapter, you'll see a basic SOAP extension and briefly consider the WSE (Web Services Enhancements) toolkit that uses SOAP extensions to provide support for new and emerging standards.

All of these topics build on the concepts you learned in the past two chapters.

■**Note** Although .NET 2.0 uses the same asynchronous programming model as .NET 1.*x*, the asynchronous support in the proxy class has changed, as you'll see in this chapter. The techniques you use for securing web services and using SOAP extensions haven't changed, although web service standards continue to evolve and are provided in new versions of the separate WSE toolkit.

Asynchronous Calls

As you learned in Chapter 31, the .NET Framework shields the programmer from the complexities of calling a web service by providing your applications with a proxy class. The code that uses the proxy class looks the same whether the web service is on the same computer, on a local network, or across the Internet.

Despite superficial similarities, the underlying plumbing used to invoke a web service is very different from an in-process function call. Not only must the call be packaged into a SOAP message, but it also needs to be transmitted across the network using HTTP. Because of the inherent nature of the Internet, the time it takes to call a web service can vary greatly from one call to the next. Although your client can't speed up a web method invocation, it can choose not to sit idle while waiting for the response. Instead, it can continue to perform calculations, read from the file system or a database, and even call additional web services. This asynchronous design pattern is more difficult to implement, but in certain situations it can reap significant benefits.

In general, asynchronous processing makes sense in two cases:

- If you're creating a Windows application. In this case, an asynchronous call allows the user interface to remain responsive.
- If you have other computationally expensive work to do, or you have to access other resources that have a high degree of latency. By performing the web service call asynchronously, you can carry out this work while waiting for the response. A special case is when you need to call several independent web services. In this situation, you can call them all asynchronously, collapsing your total waiting time.

It's just as important that you realize when you should *not* use the asynchronous pattern. Asynchronous calls won't speed up the time taken to receive a response. In other words, in most web applications that use web services, you'll have no reason to call the web service asynchronously, because your code still needs to wait for the response to be received before it can render the final page and send it back to the client. However, if you need to call a number of web services at once, or you can perform other tasks while waiting, you may shave a few milliseconds off the total request processing time.

In the following sections, you'll see how asynchronous calls can save time with a web client and provide a more responsive user interface with a Windows client.

■**Note** Asynchronous calls work a little differently in ASP.NET 2.0. The proxy class no longer has built-in BeginXxx() and EndXxx() methods. However, you can use an alternate approach for event-based notification of asynchronous operations that is built into the proxy class and makes sense for long-running clients such as Windows applications.

Asynchronous Delegates

You can use asynchronous threads in .NET in several ways. All delegates provide BeginInvoke() and EndInvoke() methods that allow you to trigger them on one of the threads in the CLR thread pool. This technique, which is convenient and scales well, is the one you'll consider in this section. Alternatively, you could use the System.Threading.Thread class to explicitly create a new thread, with complete control over its priority and lifetime.

As you already know, delegates are typesafe function pointers that form the basis for .NET events. You create a delegate that references a specific method, and then you can call that method through the delegate.

The first step is to define the delegate at the namespace level (if it's not already present in the .NET class library). For example, here's a delegate that can point to any method that accepts a single integer parameter and returns an integer:

```
public delegate int DoSomethingDelegate(int input);
```

Now consider a class that has a method that matches this delegate:

```
public class MyClass
{
    public int DoubleNumber(int input)
    {
        return input * 2;
    }
}
```

You can create a delegate variable that points to a method with the same signature. Here's the code:

```
MyOClass myObj = new MyClass();

// Create a delegate that points to the myObj.DoubleNumber() method.
DoSomethingDelegate doSomething = new DoSomethingDelegate(myObj.DoubleNumber);

// Call the myObj.DoubleNumber() method through the delegate.
int doubleValue = doSomething(12);
```

What you may not realize is that delegates also have built-in threading smarts. Every time you define a delegate (such as DoSomethingDelegate in the previous example), a custom delegate class is generated and added to your assembly. (A custom delegate class is needed because the code for each delegate is different, depending on the signature of the method you've defined.) When you call a method through the delegate, you are actually relying on the Invoke() method of the delegate class.

The Invoke() method executes the linked method synchronously. However, the delegate class also includes methods for asynchronous invocation—BeginInvoke() and EndInvoke(). When you use BeginInvoke(), the call returns immediately, but it doesn't provide the return value. Instead, the method is simply queued to start on another thread. When calling BeginInvoke(), you supply all the parameters of the original method, plus two additional parameters for an optional callback and state object. If you don't need these details (described later in this section), simply pass a null reference.

```
IAsyncResult async = doSomething.BeginInvoke(12, null, null);
```

BeginInvoke() doesn't provide the return value of the underlying method. Instead, it returns an IAsyncResult object, which you can examine to determine when the asynchronous operation is complete. To pick up the results later, you submit the IAsyncResult object to the matching EndInvoke() method of the delegate. EndInvoke() waits for the operation to complete if it hasn't already finished and then provides the real return value. If any unhandled errors occurred in the method that you executed asynchronously, they'll bubble up to the rest of your code when you call EndInvoke().

Here's the previous example rewritten to call the delegate asynchronously:

```
MyClass myObj = new MyClass();

// Create a delegate that points to the myObj.DoubleNumber() method.
DoSomethingDelegate doSomething = new DoSomethingDelegate(myObj.DoubleNumber);

// Start the myObj.DoubleNumber() method on another thread.
IAsyncResult handle = doSomething.BeginInvoke(originalValue, null, null);

// (Do something else here while myObj.DoubleNumber() is executing.)

// Retrieve the results, and wait (synchronously) if they're still not ready.
int doubleValue = doSomething.EndInvoke(handle);
```

To gain some of the benefits of multithreading with this technique, you could call several methods asynchronously with BeginInvoke(). You could then call EndInvoke() on all of them before continuing.

A Simple Asynchronous Call

The following example demonstrates the difference between synchronous and asynchronous code. To test the example, you need to slow down your code artificially to simulate heavy load conditions or time-consuming tasks. First, add this line to the GetEmployees() method in the web service to add a delay of four seconds:

```
System.Threading.Thread.Sleep(4000);
```

Next, create a web page that uses the web service. This web page also defines a private method that simulates a time-consuming task, again using the Thread.Sleep() method. Here's the code you need to add to the web page:

```
private void DoSomethingSlow()
{
    System.Threading.Thread.Sleep(3000);
}
```

In your page, you need to execute both methods. Using a simple piece of timing code, you can compare the synchronous approach with the asynchronous approach. Depending on which button the user clicks, you will perform the two operations synchronously (one after the other) or asynchronously at the same time.

Here's how you would execute the two tasks synchronously:

```
protected void cmdSynchronous_Click(object sender, System.EventArgs e)
{
    // Record the start time.
    DateTime startTime = DateTime.Now;

    // Get the web service data.
    EmployeesService proxy = new EmployeesService();
    try
    {
        GridView1.DataSource = proxy.GetEmployees();
    }
    catch (Exception err)
    {
        lblInfo.Text = "Problem contacting web service.";
        return;
    }

    GridView1.DataBind();

    // Perform some other time-consuming tasks.
    DoSomethingSlow();

    // Determine the total time taken.
    TimeSpan timeTaken = DateTime.Now.Subtract(startTime);
    lblInfo.Text = "Synchronous operations took " + timeTaken.TotalSeconds +
      " seconds.";
}
```

To use asynchronous delegates, you need to define a delegate that matches the signature of the method you want to call asynchronously. In this case, it's the GetEmployees() method:

```
public delegate DataSet GetEmployeesDelegate();
```

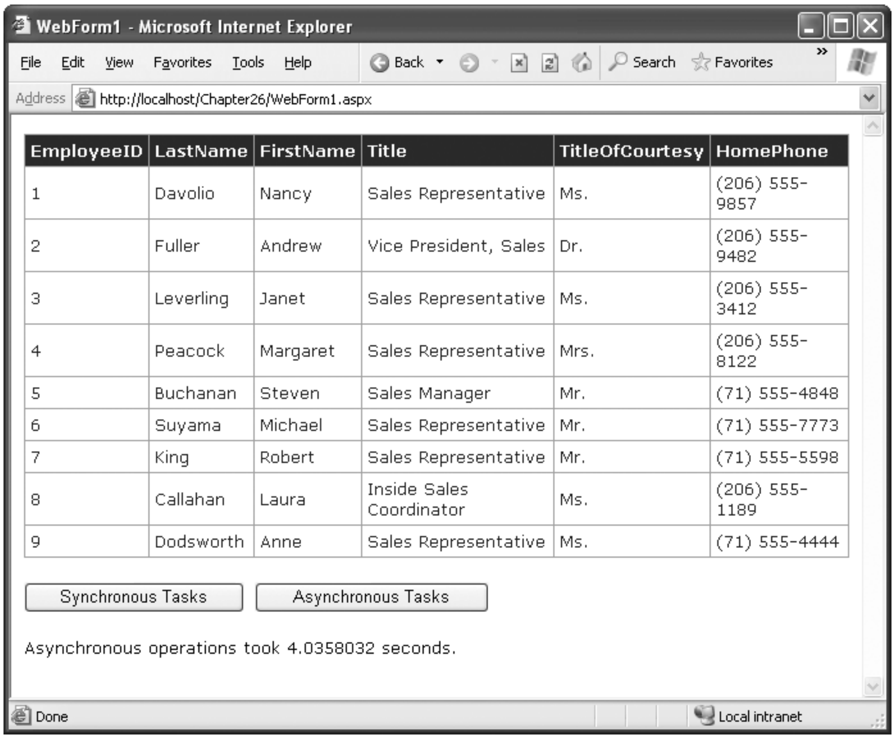
And here's how you could start the web service first so that the operations overlap:

```
protected void cmdAsynchronous_Click(object sender, System.EventArgs e)
{
    // Record the start time.
    DateTime startTime = DateTime.Now;

    // Start the web service on another thread.
    EmployeesService proxy = new EmployeesService();
    GetEmployeesDelegate async = new GetEmployeesDelegate(proxy.GetEmployees);
    IAsyncResult handle = async.BeginInvoke(null, null);

    // Perform some other time-consuming tasks.
    DoSomethingSlow();

    // Retrieve the result. If it isn't ready, wait.
    try
    {
        GridView1.DataSource = async.EndInvoke(handle);
    }
    catch (Exception err)
    {
        lblInfo.Text = "Problem contacting web service.";
        return;
    }
    GridView1.DataBind();

    // Determine the total time taken.
    TimeSpan timeTaken = DateTime.Now.Subtract(startTime);
    lblInfo.Text = "Asynchronous operations took " + timeTaken.TotalSeconds +
      " seconds.";
}
```

Notice that the exception handler wraps the EndInvoke() method but not the BeginInvoke() method. That's because if any errors occur while processing the request (whether because of a network problem or a server-side exception), your code won't receive it until you call the EndInvoke() method.

When you run these two examples, you'll find that the synchronous code takes between 7 and 8 seconds, while the asynchronous code takes only between 4 and 5 seconds. Figure 33-1 shows the web page with the time reading at the bottom.

■ **Tip** Remember, the advantage of threading depends on the type of operations. In this example, the full benefit of threading is realized because the operations aren't CPU bound—they are simply waiting idly. This is similar to the behavior you'll experience contacting external web services or databases. However, if you try to use threading to simultaneously run two tasks that use the CPU on the *same* computer (and that computer has only one CPU), you won't see any advantage, because both tasks will get about half the CPU resources and will take about twice as long to execute. That's why threading is ideal for web services but not nearly as useful for the rest of your business code.


WebForm1 - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back
Search
Favorites
Address
http://localhost/Chapter26/WebForm1.aspx

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

```
Asynchronous operations took 4.0358032 seconds.
```

Figure 33-1. *Testing an asynchronous method call*

Concurrent Asynchronous Calls

The IAsyncState object gives you a few other options that are useful when calling multiple web methods at once. The key is the IAsyncState.WaitHandle object, which returns a System.Threading.WaitHandle object. Using this object, you can call WaitAll() to wait until all your asynchronous operations are complete. The following example uses this technique to call the GetEmployees() method three times at once:

```
protected void cmdMultiple_Click(object sender, System.EventArgs e)
{
    // Record the start time.
    DateTime startTime = DateTime.Now;

    EmployeesService proxy = new EmployeesService();
    GetEmployeesDelegate async = new GetEmployeesDelegate(proxy.GetEmployees);

    // Call three methods asynchronously.
    IAsyncResult handle1 = async.BeginInvoke(null, null);
    IAsyncResult handle2 = async.BeginInvoke(null, null);
    IAsyncResult handle3 = async.BeginInvoke(null, null);

    // Create an array of WaitHandle objects.
    WaitHandle[] waitHandles = {handle1.AsyncWaitHandle,
     handle2.AsyncWaitHandle, handle3.AsyncWaitHandle};
    // Wait for all the calls to finish.
    WaitHandle.WaitAll(waitHandles);

    // You can now retrieve the results.
    DataSet ds1 = async.EndInvoke(handle1);
    DataSet ds2 = async.EndInvoke(handle2);
    DataSet ds3 = async.EndInvoke(handle3);

    // Merge all the results into one table and display it.
    DataSet dsMerge = new DataSet();
    dsMerge.Merge(ds1);
    dsMerge.Merge(ds2);
    dsMerge.Merge(ds3);
    GridView1.DataSource = dsMerge;
    GridView1.DataBind();

    // Determine the total time taken.
    TimeSpan timeTaken = DateTime.Now.Subtract(startTime);
    lblInfo.Text = "Calling three methods took " + timeTaken.TotalSeconds +
      " seconds.";
}
```

Instead of using a wait handle, you could launch the three asynchronous calls by calling BeginInvoke() three times and then call the three EndInvoke() methods immediately after that. In this case, your code would wait if required. However, using a wait handle clarifies your code.

You can also use one of the overloaded versions of the WaitAll() method that accepts a timeout value. If this amount of time passes without the calls completing, an exception will be thrown. However, it's usually best to rely on the Timeout property of the proxy instead, which will end the call if a response isn't received in the designated amount of time.

You can also instruct a wait handle to block the thread until any one of the method calls has finished using the static WaitHandle.WaitAny() method with an array of WaitHandle objects. The WaitAny() method returns as soon as at least one of the asynchronous calls completes. This can be a useful technique if you need to process a batch of data from different sources and the order that you process it is not important. It allows you to begin processing the results from one method before the others are complete. However, it also complicates your code, because you'll need to test the IsCompleted property of each IAsyncResult object and call WaitAny() multiple times (usually in a loop) until all the methods have finished.

■**Note** The asynchronous features of the proxy class also lend themselves to asynchronous pages, an advanced technique that you learned about in Chapter 11. If you need to call one or more slow-responding web services, an asynchronous page can improve scalability. For more information, refer to the "Asynchronous Pages" section at the end of Chapter 11.

Responsive Windows Clients

In a Windows client, the threading code you use is a little different. Typically, you'll want to allow the application to continue unhindered while the operation is taking place. When the call is complete, you might simply want to refresh the display with updated information.

Support for this pattern is built into the proxy class. To understand how it works, it helps to look at the proxy class code. For every web method in your web service, the proxy class actually includes two methods—the synchronous version you've seen so far and an asynchronous version that adds the suffix *Async* to the method.

Here's the code for the synchronous version of the GetEmployees() method. The attributes for XML serialization have been omitted.

```
[SoapDocumentMethod(...)]
public DataSet GetEmployees()
{
    object[] results = this.Invoke("GetEmployees", new object[]{});
    return ((DataSet)(results[0]));
}
```

And here's the asynchronous version of the same method. Notice that the code actually contains two versions of the GetEmployeesAsync() method. The only difference is that one accepts an additional userState parameter, which can be any object you use to identify the call. When the call is completed later, you'll receive this object in the callback. The userState parameter is particularly useful if you have several asynchronous web methods underway at the same time.

```
public void GetEmployeesAsync()
{
    this.GetEmployeesAsync(null);
}

public void GetEmployeesAsync(object userState)
{
    if ((this.GetEmployeesOperationCompleted == null))
    {
        this.GetEmployeesOperationCompleted = new
          System.Threading.SendOrPostCallback(
          this.OnGetEmployeesOperationCompleted);
    }
    this.InvokeAsync("DownloadFile", new object[]{},
      this.GetEmployeesOperationCompleted, userState);
}
```

The idea is that you can call GetEmployeesAsync() to launch the request. This method returns immediately, before the request is even sent over the network, and the proxy class waits on a free thread (just as with the asynchronous delegate example) until it receives the response. As soon as the response is received and deserialized, .NET fires an event to notify your application. You can then retrieve the results.

For this system to work, the proxy class also adds an event for each web method. This event is fired when the asynchronous method is finished. Here's the completion event for the GetEmployees() method:

```
public event GetEmployeesCompletedEventHandler GetEmployeesCompleted;
```

The proxy class code is quite smart here—it simplifies your life by creating a custom EventArgs object for every web method. This EventArgs object exposes the result from the method as a Result property. The class is declared with the partial keyword so that you can add code to it in another (nonautomatically generated) file.

```
// Defines the signature of the completion event.
public delegate void GetEmployeesCompletedEventHandler(object sender,
 GetEmployeesCompletedEventArgs e);

public partial class GetEmployeesCompletedEventArgs :
  System.ComponentModel.AsyncCompletedEventArgs
{
    private object[] results;
    // The constructor is internal, prevent other assemblies from instantiating
    // this class.
    internal GetEmployeesEventArgs(object[] results,
      Exception exception, bool cancelled, object userState) :
      base(exception, cancelled, userState)
    {
        this.results = results;
    }

    public System.Data.DataSet Result
    {
        get
        {
            this.RaiseExceptionIfNecessary();
            return ((System.Data.DataSet)(this.results[0]));
        }
    }
}
```

Notice that if an error occurred on the server side, it won't be thrown until you attempt to retrieve the Result property, at which point the SoapException is wrapped in a TargetInvocation-Exception.

■**Note** Result isn't the only property you might find in the custom EventArgs object. If your web method accepts ref or out parameters, these will also be added. That way, you can receive the modified values of all ref or out parameters when the call is complete.

Along with the Result property, you can also use a few properties that are declared in the base AsyncCompletedEventArgs class, including Cancelled (returns true if the operation was cancelled before it was completed, a feature you'll see shortly), Error (returns an exception object if an unhandled exception occurred during the request), and UserState (returns the state object you supplied when you called the method).

To try this pattern, you can modify the Windows client developed in Chapter 31 to use an asynchronous call. The first step is to create an event handler for the completion event in the form class:

```
private void GetEmployeesCompleted(object sender, GetEmployeesCompletedEventArgs e)
{ ... }
```

When the user clicks the Get Employees button, the code will use the GetEmployeesAsync() method to start the process. But first, it needs to attach an event handler to the GetEmployees-Completed event. Here's the code you need:

```
private void cmdGetEmployees_Click(object sender, System.EventArgs e)
{
    // Disable the button so that only one asynchronous
    // call will be permitted at a time.
    cmdGetEmployees.Enabled = false;

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

    // Create the callback delegate.
    proxy.GetEmployeesCompleted += new
      GetEmployeesCompletedEventHandler(this.GetEmployeesCompleted);
    // Call the web service asynchronously.
    proxy.GetEmployeesAsync();
}
```

When the operation is finished, the proxy class fires the event. When you handle the event, you can bind the result directly to the grid. You don't need to worry about marshaling your call to the user interface thread, because that's handled automatically by the proxy class before the event is raised, which is a significant convenience.

```
private void GetEmployeesCompleted(object sender, GetEmployeesCompletedEventArgs e)
{
    // Get the result.
    try
    {
        dataGridView1.DataSource = e.Result;
    }
    catch (System.Reflection.TargetInvocationException err)
    {
        MessageBox.Show("An error occurred.");
    }
}
```

If you run this example and click the Get Employees button, the button will become disabled, but the application will remain responsive. You can drag and resize the window, click other buttons to execute more code, and so on. Finally, when the results have been received, the callback will be triggered, and the DataGridView will be refreshed automatically.

You can use one other trick. The proxy class has built-in support for cancellation. To use it, you need to supply a state object when you call the asynchronous version of the proxy class method. Then, simply call the CancelAsync() method of the proxy class and supply the same state object. The rest of the process is taken care of automatically.

To implement the cancellation model with the existing application, you first need to declare the proxy class at the form level so it's available to all your event handlers:

```
private EmployeesService proxy = new EmployeesService();
```

Next, attach the event handler when the form loads so it's hooked up only once:

```
private void Form1_Load(object sender, EventArgs e)
{
    proxy.GetEmployeesCompleted += new
      GetEmployeesCompletedEventHandler(this.GetEmployeesCompleted);
}
```

In this example, the state object that you use isn't important, because you will have only a single operation taking place at once. If you're performing multiple operations at once, it makes sense to generate a new GUID to track each one.

First, declare it in the form class:

```
private Guid requestID;
```

Then, generate it, and supply it to the asynchronous web method call:

```
requestID = Guid.NewGuid();
proxy.GetEmployeesAsync(requestID);
```

Now, all you need to do is use the same GUID when calling the CancelAsync() method. Here's the code for a cancel button:

```
private void cmdCancel_Click(object sender, EventArgs e)
{
    proxy.CancelAsync(requestID);
    MessageBox.Show("Operation cancelled.");
}
```

This has one important consideration. As soon as you call CancelAsync(), the completed event fires. This makes sense, because the long-running operation has finished (albeit because of programmatic intervention) and you may need to update the user interface. However, you obviously can't access the method result in the completion event because the code was interrupted. To prevent an error, you need to explicitly test for cancellation, as shown here:

```
private void GetEmployeesCompleted(object sender, GetEmployeesCompletedEventArgs e)
{
    if (!e.Cancelled)
    {
        try
        {
            dataGridView1.DataSource = e.Result;
        }
        catch (System.Reflection.TargetInvocationException err)
        {
            MessageBox.Show("An error occurred.");
        }
    }
}
```

Asynchronous Services

So far, you've seen several examples that allow clients to call web services asynchronously. But in all these examples, the web method still runs synchronously from start to finish. What if you want a different behavior that allows the client to trigger a long-running process and then connect later to pick up the results?

Unfortunately, .NET doesn't directly support this model. Part of the problem is that all web service communication must be initiated by the client. Currently, the web server has no way to initiate a callback to the client to tell them when a task is complete. And even if standards evolve to fill this gap, it's unlikely that this solution will gain widespread use because of the nature of the architecture of the Web. Many clients connect from behind proxy servers or firewalls that don't allow incoming connections or hide location information such as the IP address. As a result, the client needs to initiate every connection.

Of course, certain innovative solutions can provide some of the functionality you want. For example, a client could reconnect periodically to poll the server and see if a task is complete. You could use this design if a web server needs to perform extremely time-consuming tasks, such as rendering complex graphics. However, one ingredient is still missing. In these situations, you need a way for a client to start a web method without waiting for the web method to finish executing. ASP.NET makes this behavior possible with *one-way methods*.

With one-way methods (also known as *fire-and-forget methods*), the client sends a request message, but the web service never responds. This means the web method returns immediately and closes the connection. The client doesn't need to spend any time waiting. However, one-way methods have a few drawbacks. Namely, the web method can't provide a return value or use a ref or out parameter. Similarly, if the web method throws an unhandled exception, it won't be propagated back to the client.

To create a fire-and-forget XML web service method, you need to apply a SoapDocument-Method attribute to the appropriate web method and set the OneWay property to true, as shown here:

```
[SoapDocumentMethod(OneWay = true)]
[WebMethod()]
public DataSet GetEmployees()
{ ... }
```

The client doesn't need to take any special steps to call a one-way method asynchronously. Instead, the method always returns immediately.

Of course, you might not want to use one-way methods. The most important limitation is that you can't return any information. For example, a common asynchronous server-side pattern is for the server to return some sort of unique, automatically generated ticket to the client when the client submits a request. The client can then submit this ticket to other methods to check the status or retrieve the results. With a one-way method, there's no way to return a ticket to the client or notify the client if an error occurs. Another problem is that one-way methods still use ASP.NET worker threads. If the task is extremely long and there are many ongoing tasks, other clients might not be able to submit new requests, which is far from ideal.

The only practical way to deal with long-running, asynchronous tasks in a heavily trafficked website is to combine web services with another .NET technology—*remoting*. For example, you could create an ordinary, synchronous web method that returns a ticket and then calls a method in a server-side component using remoting. The remoting component could then begin its processing task asynchronously. This technique of using a web service as a front-end to a full-fledged, continuously running server-side component is a more complex hallmark of distributed design. To learn more, you may want to consult a dedicated book about distributed programming, such as *Microsoft .NET Distributed Applications* (Microsoft Press, 2003) or a book about .NET remoting, such as *Advanced .NET Remoting* (Apress, 2002). Of course, if you don't need this flexibility, you're better off avoiding it completely, because it introduces significant complexity and extra work.

Securing Web Services

In an ideal world, you could treat a web service as a class library of functionality and not worry about coding user authentication or security logic. However, to create subscription-based or micro-payment web services, you need to determine who is using it. And even if you aren't selling your logic to an audience of eager web developers, you may still need to use authentication to protect sensitive data and lock out malicious users, especially if your web service is exposed over the Internet.

You can use some of the same techniques you use to protect web pages to defend your web services. For example, you can use IIS to enforce SSL (just direct your clients to a web service URL starting with `https://`). You can also use IIS to apply Windows authentication, although you need to apply a few additional steps, as you'll learn in the next section. Finally, you can create your own custom authentication system using SOAP headers.

Windows Authentication

Windows authentication works with a web service in much the same way that 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, there's 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 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 user name (which will be a string in the form DomainName\UserName or ComputerName\UserName).

```
public class SecureService : System.Web.Services.WebService
{
    [WebMethod()]
    public string TestAuthenticated()
    {
        if (!User.Identity.IsAuthenticated)
        {
            return "Not authenticated.";
        }
        else
        {
            return "Authenticated as: " + User.Identity.Name;
        }
    }
}
```

The web service can also examine role membership, although this web service doesn't take this step.

To submit user credentials to this service, the client needs to modify the NetworkCredential property of the proxy class. You have two options:

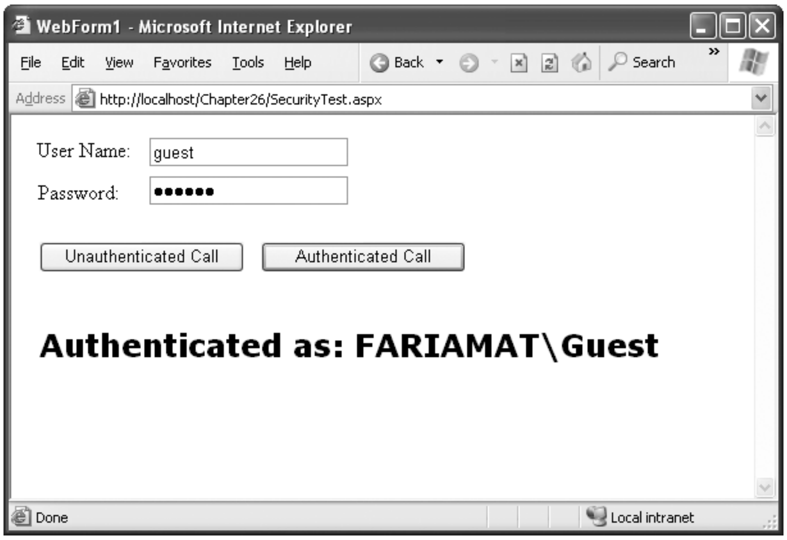
- You can create a new NetworkCredential object and attach this to the NetworkCredential property of the proxy object. When you create the NetworkCredential object, you'll need to specify the user name and password you want to use. This approach works with all forms of Windows authentication.
- If the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

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

```
using System.Net;
```

The following code shows a web page with two text boxes and two buttons (see Figure 33-2). One button performs an unauthenticated call, while the other submits the user name and password that have been entered in the text boxes.

The unauthenticated call will fail if you've disabled anonymous users. 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.


WebForm1 - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back
Search
Address http://localhost/Chapter26/SecurityTest.aspx
User Name: guest
Password: ●●●●●●
Unauthenticated Call
Authenticated Call
Done
Local intranet

Authenticated as: FARIAMAT\Guest

Figure 33-2. *Successful authentication through a web service*

Here's the complete web-page code:

```
public partial class WindowsAuthenticationSecurityTest : Page
{
    protected void cmdUnauthenticated_Click(Object sender, EventArgs e)
    {
        SecureService proxy = new SecureService();
        try
        {
            lblInfo.Text = proxy.TestAuthenticated();
        }
        catch (Exception err)
        {
            lblInfo.Text = err.Message;
        }
    }

    protected void cmdAuthenticated_Click(Object sender, EventArgs e)
    {
        SecureService proxy = new SecureService();

        // Supply some user credentials for the web service.
        NetworkCredential credentials = new NetworkCredential(
          txtUserName.Text, txtPassword.Text);
        proxy.Credentials = credentials;

        lblInfo.Text = proxy.TestAuthenticated();
    }
}
```

To try this, you can add the following <location> tag to the web.config file to restrict access to the SecureService.asmx web service:

```
<configuration>
    <system.web>
        <authorization>
            <allow users="*" />
        </authorization>
        ...
    </system.web>

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

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

```
SecuredService proxy = new SecuredService();
proxy.Credentials = CredentialCache.DefaultCredentials;
lblInfo.Text = proxy.TestAuthenticated();
```

In this example (as in all web pages), the current user account will be the account that ASP.NET is using, not the user account of the remote user who is requesting the web page. If you use the same technique in a Windows application, you'll submit the account information of the user who is running the application.

Custom Ticket-Based Authentication

Windows authentication is a good solution for web services when you have a small set of users who have existing Windows accounts. However, it doesn't work as well for large-scale public web services. When working with ASP.NET web pages, you usually turn to forms authentication to fill the gaps. However, forms authentication won't work with a web service because a web service has no way 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. Forms authentication is also cookie-based, which is an unnecessary restriction to place on web services, which might use protocols that don't support cookies or clients that don't expect them.

A common solution is to roll your own authentication system. In this model, users will 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 register the user session and create a new, unique ticket. From this point on, the user can reconnect to the web service by supplying the ticket to every other method.

A properly designed ticket system has a number of benefits. As with forms authentication, it provides complete flexibility. It also optimizes performance and ensures scalability, because you can cache the ticket in memory. On subsequent requests, you can verify the ticket rather than authenticating the user against the database. Finally, it allows you to take advantage of SOAP headers, which make the ticket management and authorization process transparent to the client.

With ASP.NET 2.0, it becomes possible to simplify custom authentication in a web service. Although it's still up to you to transfer the user credentials and keep track of who has logged in by issuing and verifying tickets, you can use the membership and role manager features discussed in Chapter 21 and Chapter 23 to handle the authentication and authorization. In the following sections, you'll see how to create a custom ticket-based authentication system that leverages membership and role management in this way.

Tracking the User Identity

To use custom security, the first step is to decide what user-specific information you want to cache in memory. You need to create a custom class that represents this information. This class can include information about the user (name, e-mail address, and so on) and the user's permissions. It should also include the ticket.

Here's a basic example that stores a user name and ticket:

```
public class TicketIdentity
{
    private string userName;
    public string UserName
    {
        get { return userName; }
    }

    private string ticket;
    public string Ticket
    {
        get { return ticket; }
    }

    public TicketIdentity(string userName)
    {
        this.userName = userName;

        // Create the ticket GUID.
        Ticket = Guid.NewGuid().ToString();
    }
}
```

■**Note** You've probably noticed that this identity class doesn't implement IIdentity. That's because this approach doesn't allow you to plug into the .NET security model in the same way you could with custom web-page authentication. Essentially, the problem is that you need to perform the authentication *after* the User object has already been created. And you can't get around this problem using the global.asax class, because the application event handlers won't have access to the web method parameters and SOAP header you need in order to perform the authentication and authorization.

Once you have the user identity class, you need to create a SOAP header. This header tracks a single piece of information—the user ticket. Because the ticket is a randomly generated GUID, it's not practically possible for a malicious user to "guess" what ticket value another user has been issued.

```
public class TicketHeader : SoapHeader
{
    public string Ticket;

    public TicketHeader(string ticket)
    {
        Ticket = ticket;
    }

    public TicketHeader()
    {}
}
```

You must then add a member variable for the TicketHeader to your web service:

```
public class SoapSecurityService : WebService
{
    public TicketHeader Ticket;
    ...
}
```

Authenticating the User

The next step is to create a dedicated web method that logs the user in. The user needs to submit user credentials to this method (such as a login and password). Then, the method will retrieve the user information, create the TicketIdentity object, and issue the ticket.

In this example, a Login() web method checks the user credentials using the static Membership.ValidateUser() method. A new ticket is constructed with the user information and stored in a user-specific slot in the Application collection. At the same time, a new SOAP header is issued with the ticket so that the user can access other methods.

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

```
[WebMethod()]
[SoapHeader("Ticket", Direction = SoapHeaderDirection.Out)]
public void Login(string userName, string password)
{
    if (Membership.ValidateUser(username, password))
    {
        // Create a new ticket.
        TicketIdentity ticket = new TicketIdentity(username);

        // Add this ticket to Application state.
        Application[ticket.Ticket] = ticket;

        // Create the SOAP header.
        Ticket = new TicketHeader(ticket.Ticket);
    }
    else
    {
        throw new SecurityException("Invalid credentials.");
    }
}
```

Note that in this example, the TicketIdentity object is stored in the Application collection, which is global to all users. However, you don't need to worry about one user's ticket overwriting another. That's because the tickets are indexed using the GUID. Every user has a separate ticket GUID and hence a separate slot in the Application collection.

The Application collection has certain limitations, including no support for web farms and poor scalability to large numbers of users. The tickets will also be lost if the web application restarts. To improve this solution, you could store the information in two places: in the Cache object and in a back-end database. That way, your code can check the Cache first, and if a matching TicketIdentity is found, no database call is required. But if the TicketIdentity isn't present, the information can still be retrieved from the database. It's important to understand that this enhancement still uses the same SOAP header with the ticket and the same TicketIdentity object. The only difference is how the TicketIdentity is stored and retrieved between requests.

Authorizing the User

Once you have the Login() method in place, it makes sense to create a private method that can be called to verify that a user is present. You can then call this method from other web methods in your web service.

The following AuthorizeUser() method checks for a matching ticket and returns the Ticket-Identity if it's found. If not, an exception is thrown, which will be returned to the client.

```
private TicketIdentity AuthorizeUser(string ticket)
{
    TicketIdentity ticketIdentity = (TicketIdentity)Application[ticket];
    if (ticket != null)
    {
        return ticketIdentity;
    }
    else
    {
        throw new SecurityException("Invalid ticket.");
    }
}
```

In addition, this overloaded version of AuthorizeUser() verifies that the user has a ticket and is a member of a specific role. The ASP.NET role management provider handles the role-checking work.

```
private TicketIdentity AuthorizeUser(string ticket, string role)
{
    TicketIdentity ticketIdentity = AuthorizeUser(ticket);
    if (Roles.IsUserInRole(ticketIdentity.UserName, role))
    {
        throw new SecurityException("Insufficient permissions.");
    }
    else
    {
        return ticketIdentity;
    }
}
```

Using these two helper methods, you can build other web service methods that test a user's permissions before performing certain tasks or returning privileged information.

Testing the SOAP Authentication System

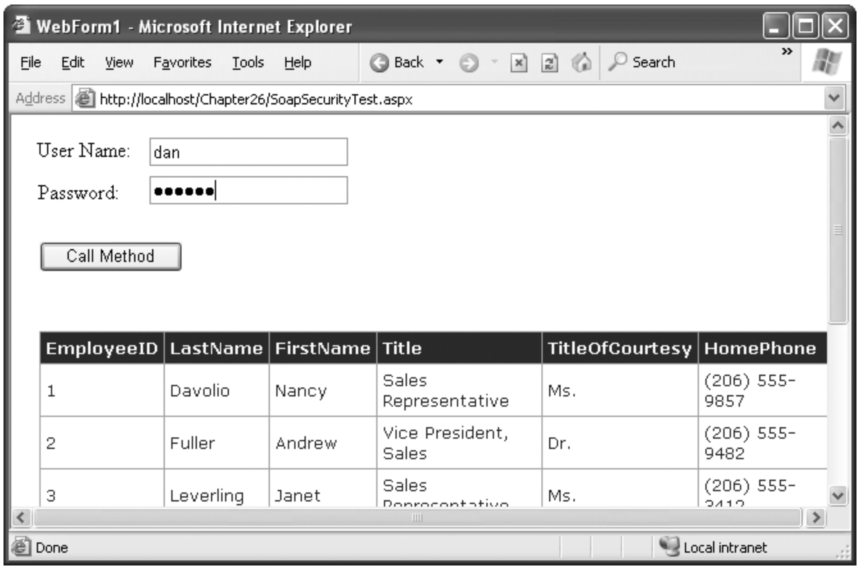
Now you simply need to create a test web method that uses the AuthorizeUser() method to check that the user has the required permissions. Here's an example that checks that the client is an administrator before allowing the client to retrieve the DataSet with the employee list:

```
[WebMethod()]
[SoapHeader("Ticket", Direction = SoapHeaderDirection.In)]
public DataSet GetEmployees()
{
    AuthorizeUser(Ticket.Ticket, "Administrator");
    ...
}
```

To make the test even easier to set up, the sample code for this chapter includes a CreateTest-User() web method that generates a specific user and makes that user part of the Administrators role:

```
[WebMethod()]
public void CreateTestUser(string username, string password)
{
    // Delete the user if the user already exists.
    if (Membership.GetUser(username) != null)
    {
        Membership.DeleteUser(username);
    }
    // Create the user.
    Membership.CreateUser(username, password);

    // Make this user an administrator
    // and create the role if it doesn't exist.
    string role = "Administrator";
    if (!Roles.RoleExists(role))
    {
        Roles.CreateRole(role);
    }
    Roles.AddUserToRole(username, role);
}
```

Now you can create a client that tests this. In this case, a web page provides two text boxes for the user to supply a user name and password (see Figure 33-3). This information is passed to the Login() method, and then the GetEmployees() method is called to retrieve the data. This method succeeds for a user with the Administrator role but fails for everyone else.


WebForm1 - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back
Search
Address
http://localhost/Chapter26/SoapSecurityTest.aspx
Call Method
Done
Local intranet

User Name: dan

Password: ••••••

EmployeeID	LastName	FirstName	Title	TitleOfCourtesy	HomePhone
1	Davolio	Nancy	Sales Representative	Ms.	(206) 555-9857
2	Fuller	Andrew	Vice President, Sales	Dr.	(206) 555-9482
3	Leverling	Janet	Sales	Ms.	(206) 555-

Figure 33-3. *Testing a web method that uses ticket-based authorization*

Here's the web-page code:

```
protected void cmdCall_Click(object sender, System.EventArgs e)
{
    SoapSecurityService proxy = new SoapSecurityService();
    try
    {
        proxy.Login(txtUserName.Text, txtPassword.Text);
        GridView1.DataSource = proxy.GetEmployees();
        GridView1.DataBind();
    }
    catch (Exception err)
    {
        lblInfo.Text = err.Message;
    }
}
```

The best part is that the client doesn't need to be aware of the ticket management. That's because the Login() method issues the ticket, and the proxy class maintains it. As long as the client uses the same instance of the proxy class, the same ticket value will be submitted automatically, and the user will be authenticated.

You can do quite a bit to enhance this authentication system. For example, you might want to record additional details with the TicketIdentity, including the time the ticket was created and last accessed and the network address of the user who owns the ticket. This way, you can incorporate additional checks into the AuthorizeUser() method. You could have tickets time out after a long period of disuse, or you could reject a ticket if the IP address of the client has changed.

SOAP Extensions

ASP.NET web services provide high-level access to SOAP. As you've seen, you don't need to know much about SOAP in order to create and call web services. If, however, you are a developer with a good understanding of SOAP and you want to get your hands dirty with low-level access to SOAP messages, the .NET Framework allows that too. In this section, you'll see how to intercept SOAP messages and manipulate them.

■**Note** Even if you don't want to manipulate SOAP messages directly, it's worth learning about SOAP extensions because they are part of the infrastructure that supports the WSE, which you'll learn about later in the section "The Web Services Enhancements."

SOAP extensions are an extensibility mechanism. They allow third-party developers to create components that plug into the web service model and provide other services. For example, you could create a SOAP extension to selectively encrypt or compress portions of a SOAP message before it is sent from the client. Of course, you would also need to run a matching SOAP extension on the server to decrypt or decompress the message after it has been received but before it's deserialized.

To create a SOAP extension, you need to create a class that derives from the System.Web.Services.Protocols.SoapExtension class. The SoapExtension class includes a ProcessMessage() method that's triggered automatically as the SOAP message passes through several stages. For example, if you run a SOAP extension on a web server, the following four stages will occur:

- **SoapMessageStage.BeforeDeserialize** occurs immediately after the web server receives the SOAP request message.
- **SoapMessageStage.AfterDeserialize** occurs after the raw SOAP message is translated to .NET data types but just before the web method code runs.

- **SoapMessageStage.BeforeSerialize** occurs after the web method code runs but before the return value is translated into a SOAP message.
- **SoapMessageStage.AfterSerialize** occurs after the return data is serialized into a SOAP response message but before it is sent to the client application.

At each stage, you can retrieve various bits of information about the SOAP message. In the BeforeDeserialize or AfterSerialize stage, you can retrieve the full SOAP message text.

You can also implement a SOAP extension on the client. In this case, the same four stages occur. Except now, the message is being received, deserialized, and acted upon by the proxy class, not the web service. Figure 33-4 shows the full process.

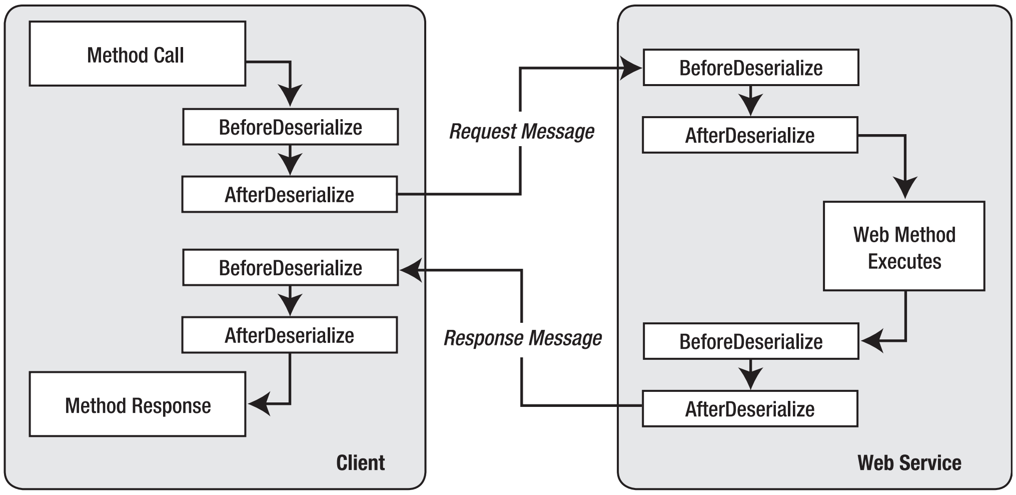


Figure 33-4. *SOAP processing on the client and server*

Creating production-level SOAP extensions is not easy. This has a number of serious challenges:

- SOAP extensions often need to be executed on both the server and client. That means you need to worry about distributing and managing another component, which can be exceedingly complex in a large distributed system.
- SOAP extensions make your web services less compatible. Third-party clients may not realize they need to install and run a SOAP extension to use your service, as this information isn't included in the WSDL document. If the clients are running non-.NET platforms, they won't be able to use the SoapExtension class you've created, and you may need to find another way to extend the SOAP processing pipeline, which can range from difficult to impossible depending on the environment.
- The Internet abounds with examples of SOAP extensions that apply encryption in a poor, insecure fashion. The flaws include insecure key management, no ability to perform a key exchange, and slow performance because of a reliance on asymmetric encryption instead of symmetric encryption. A much better choice is to use a SSL through IIS, which is bulletproof.

In most cases, creating SOAP extensions is a task that's best left to the architects at Microsoft, who are focused on the issues and challenges of developing enterprise-level plumbing. In fact, Microsoft implements several newer, less mature web service standards using SOAP extensions in the freely downloadable WSE component you'll learn about later in the section "The Web Services Enhancements."

■Note SOAP extensions work only over SOAP. When you test a web method through the browser test page, they won't be invoked. Also, SOAP messages won't work if you set the BufferResponse attribute of the WebMethod attribute to false. In this case, the SOAP extension can't act on the SOAP data, because ASP.NET begins sending it immediately, before it has been completely generated.

Creating a SOAP Extension

In the following example, you'll see a sample SOAP extension that logs SOAP messages to the Windows event log. SOAP faults will be logged as errors in the event log.

■Tip Some developers use SOAP extensions for tracing purposes. However, as you already learned in Chapter 32, that's not necessary—the trace utility included with the Microsoft SOAP Toolkit is far more convenient. However, one advantage of the SOAP logger you'll see in this example is that you could use it to capture and store SOAP messages permanently, even if the trace utility isn't running. You also don't need to change the port in the client code to route messages through the trace utility.

All SOAP extensions consist of two ingredients: a custom class that derives from System.Web.Services.Protocols.SoapExtension and a custom attribute that you apply to a web method to indicate that your SOAP extension should be used. The custom attribute is the simpler of the two ingredients.

The SoapExtension Attribute

The SoapExtension attribute allows you to link specific SOAP extensions to the methods in a web class. When you create your SoapExtension attribute, you derive from the System.Web.Services.Protocols.SoapExtensionAttribute, as shown here:

```
[AttributeUsage(AttributeTargets.Method)]
public class SoapLogAttribute : System.Web.Services.Protocols.SoapExtensionAttribute
{ ... }
```

Note that the attribute class bears another attribute—the AttributeUsage attribute. It indicates where you can use your custom attribute. SOAP extension attributes are always applied to individual method declarations, much like the SoapHeader and WebMethod attributes. Thus, you should use AttributeTargets.Method to prevent the user from applying it to some other code construct (such as a class declaration). You should use AttributeUsage anytime you need to create a custom attribute—it isn't limited to web service scenarios.

Every SoapExtension attribute needs to override two abstract properties: Priority and ExtensionType. Priority sets the order that SOAP extensions work if you have multiple extensions configured. However, it's not needed in simpler extensions such as the one in this example. The ExtensionType property returns a Type object that represents your custom SoapExtension class, and it allows .NET to attach your SOAP extension to the method. In this example, the class name of the SOAP Extension is SoapLog, although we haven't explained the code yet.

```
private int priority;
public override int Priority
{
    get { return priority; }
    set { priority = value; }
}
```

7680CH33.qxd 8/25/06 10:51 AM Page 1269

```
public override Type ExtensionType
{
    get { return typeof(SoapLog); }
}
```

In addition, you can add properties that will supply extra bits of initialization information to your SOAP extension. The following example adds a Name property, which stores the source string that will be used when writing event log entries, and a Level property, which configures what types of messages will be logged. If the level is 1, the SoapLog extension will log only error messages. If the level is 2 or greater, the SoapLog extension will write all types of messages. If the level is 3 or greater, the SoapLog extension will add an extra piece of information to each message that records the stage when the log entry was written.

```
private string name = "SoapLog" ;
public string Name
{
    get { return name;}
    set { name = value; }
}

private int level = 1;
public int Level
{
    get { return level;}
    set { level = value; }
}
```

You can now apply this custom attribute to a web method and set the Name and Level properties. Here's an example that uses the log source name EmployeesService.GetEmployeesCount and a log level of 3:

```
[SoapLog(Name="EmployeesService.GetEmployeesCount", Level=3)]
[WebMethod()]
public int GetEmployeesCount()
{ ... }
```

Now, whenever the GetEmployeesCount() method is called with a SOAP message, the SoapLog class is created, initialized, and executed. It has the chance to process the SOAP request message before the GetEmployeesCount() method receives it, and it has the chance to process the SOAP response message after the GetEmployeesCount() method returns a result.

The SoapExtension

The SoapExtension class we'll use to log messages is fairly long, although much of the code is basic boilerplate that every SOAP extension uses. We'll examine it piece by piece.

The first detail to notice is that the class derives from the abstract base class SoapExtension. This is a requirement. The SoapExtension class provides many of the methods you need to override, including the following:

- **GetInitializer() and Initialize()**: These methods pass initial information to the SOAP extension when it's first created.
- **ProcessMessage()**: This is where the actual processing takes place, allowing your extension to take a look at (and modify) the raw SOAP.
- **ChainStream()**: This method is a basic piece of infrastructure that every web service should provide. It allows you to gain access to the SOAP stream without disrupting other extensions.

Here's the class definition:

```
public class SoapLog : System.Web.Services.Protocols.SoapExtension
{ ... }
```

ASP.NET calls the GetInitializer() method the first time your extension is used for a particular web method. It gives you the chance to initialize and store some data that will be used when processing SOAP messages. You store this information by passing it back as the return value from the GetInitializer() method.

When the GetInitializer() method is called, you receive one important piece of information—the custom attribute that was applied to the corresponding web method. In the case of the SoapLog, this is an instance of the SoapLogAttribute class, which provides the Name and Level property. To store this information for future use, you can return this attribute from the GetInitializer() method, as shown here:

```
public override object GetInitializer(LogicalMethodInfo methodInfo,
 SoapExtensionAttribute attribute)
{
    return attribute;
}
```

Actually, the GetInitializer() method has two versions. Only one is invoked, and it depends on whether the SOAP extension is configured through an attribute (as in this example) or through a configuration file. If applied through a configuration file, the SOAP extension automatically runs for every method of every web service.

Even if you don't plan to use the configuration file to initialize a SOAP extension, you still need to implement the other version of GetInitializer(). In this case, it makes sense to return a new SoapLogAttribute instance so that the default Name and Level settings are available later:

```
public override object GetInitializer(Type obj)
{
    return new SoapLogAttribute();
}
```

GetInitializer() is called only the first time your SOAP extension is executed for a method. However, every time the method is invoked, the Initialize() method is triggered. If you returned an object from the GetInitializer() method, ASP.NET provides this object to the Initialize() method every time it's called. In the SoapLog extension, this is a good place to extract the Name and Level information and store it in member variables so it will be available for the remainder of the SOAP processing work. (You couldn't store this information in the GetInitialize() method, because that method won't be called every time the SOAP extension is executed.)

```
private int level;
private string name;

public override void Initialize(object initializer)
{
    name = ((SoapLogAttribute)initializer).Name;
    level = ((SoapLogAttribute)initializer).Level;
}
```

The workhorse of the extension is the ProcessMessage() method, which ASP.NET calls at various stages of the serialization process. A SoapMessage object is passed to the ProcessMessage() method, and you can examine this method to retrieve information about the message, such as its stage and the message text. The SoapLog extension reads the full message only in the AfterSerialize and BeforeDeserialize stages, because these are the only stages when you can retrieve the full XML of the SOAP message. However, if the level is 3 or greater, a basic log entry will be created in the BeforeSerialize and AfterDeserialize stages that simply records the name of the stage.

Here's the full ProcessMessage() code:

```
public override void ProcessMessage(SoapMessage message)
{
    switch (message.Stage)
    {
        case System.Web.Services.Protocols.SoapMessageStage.BeforeSerialize:
            if (level > 2 )
               WriteToLog(message.Stage.ToString(),
                 EventLogEntryType.Information);
            break;
        case System.Web.Services.Protocols.SoapMessageStage.AfterSerialize:
            LogOutputMessage(message);
            break;
        case System.Web.Services.Protocols.SoapMessageStage.BeforeDeserialize:
            LogInputMessage(message);
            break;
        case System.Web.Services.Protocols.SoapMessageStage.AfterDeserialize:
            if (level > 2 )
               WriteToLog(message.Stage.ToString(),
                 EventLogEntryType.Information);
            break;
    }
}
```

The ProcessMessage() method doesn't contain the actual logging code. Instead, it calls other private methods such as WriteLogLog(), LogOutputMessage(), and LogInputMessage(). The Write-ToLog() is the final point at which the log entry is created using the System.Diagnostics.EventLog class. If needed, this code creates a new event log and a new log source using the name that was set in the Name property of the custom extension attribute.

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

```
private void WriteToLog(string message, EventLogEntryType type)
{
    // Create a new log named Web Service Log, with the event source
    // specified in the attribute.
    EventLog log;
    if (!EventLog.SourceExists(name))
      EventLog.CreateEventSource(name, "Web Service Log");

    log = new EventLog();
    log.Source = name;
    log.WriteEntry(message, type);
}
```

When the SOAP message is in the BeforeSerialize or AfterDeserialize stage, the WriteToLog() method is called directly, and the name of the stage is written. When the SOAP message is in the AfterSerialize or BeforeDeserialize stage, you need to perform a little more work to retrieve the SOAP message.

Before you can build these methods, you need another ingredient—the CopyStream() method. That's because the XML in the SOAP message is contained in a stream. The stream has a pointer that indicates the current position in the stream. The problem is that as you read the message data from the stream (for example, to log it), you move the pointer. This means that if the log extension reads a stream that is about to be deserialized, it will move the pointer to the end of the stream. For ASP.NET to properly deserialize the SOAP message, the pointer must be set back to the beginning of the stream. If you don't take this step, a deserialization error will occur.

To make this process easier, you can use a private CopyStream() method. This method copies the contents of one stream to another stream. After this method is executed, both streams will be positioned at the end.

```
private void CopyStream(Stream fromstream, Stream tostream)
{
    StreamReader reader = new StreamReader(fromstream);
    StreamWriter writer = new StreamWriter(tostream);
    writer.WriteLine(reader.ReadToEnd());
    writer.Flush();
}
```

Another ingredient you need is the ChainStream() method, which the ASP.NET plumbing calls before serialization or deserialization takes place. Your SOAP extension can override the ChainStream() method to insert itself into the processing pipeline. At this point, the extension can cache a reference to the original stream and create a new in-memory stream, which is then returned to the next extension in the chain.

```
private Stream oldStream;
private Stream newStream;

public override Stream ChainStream(Stream stream)
{
    oldStream = stream;
    newStream = new MemoryStream();
    return newStream;
}
```

Of course, this is only part of the story. It's up to the other methods to either read data out of the old stream or write data into the new stream, depending on what stage the message is in. You do this by calling the CopyStream() method. Once you've implemented this somewhat confusing design, the end result is that every SOAP extension has a chance to modify the SOAP stream without overwriting each other's changes. For the most part, the ChainStream() and CopyStream() methods are basic pieces of SOAP extension architecture that are identical in every SOAP extension you'll see.

The LogInputMessage() and LogOutputMessage() methods have the task of extracting the message information and logging it. Both methods use the CopyStream() method. When deserializing, the input stream contains the XML to deserialize, and the pointer is at the beginning of the stream. The LogInputMessage() method copies the input stream into the memory stream buffer and logs the contents of the stream. It sets the pointer to the beginning of the memory stream buffer so that the next extension can get access to the stream.

```
private void LogInputMessage(SoapMessage message)
{
    CopyStream(oldStream, newStream);
    message.Stream.Seek(0, SeekOrigin.Begin);
    LogMessage(message, newStream);
    message.Stream.Seek(0, SeekOrigin.Begin);
}
```

When serializing, the serializer writes to the memory stream created in ChainStream(). When the LogOutputMessage() function is called after serializing, the pointer is at the end of the stream. The LogOutputMessage() function sets the pointer to the beginning of the stream so that the extension can log the contents of the stream. Before returning, the content of the memory stream is copied to the outgoing stream, and the pointer is then back at the end of both streams.

```
private void LogOutputMessage(SoapMessage message)
{
    message.Stream.Seek(0, SeekOrigin.Begin);
    LogMessage(message, newStream);
    message.Stream.Seek(0, SeekOrigin.Begin);
    CopyStream(newStream, oldStream);
}
```

Once they've moved the stream to the right position, both LogInputMessage() and LogOutput-Message() extract the message data from the SOAP stream and write a log message entry with that information. The function also checks whether the SOAP message contains a fault. In that case, the message is logged in the event log as an error.

```
private void LogMessage(SoapMessage message, Stream stream)
{
    StreamReader reader = new StreamReader(stream);
    eventMessage = reader.ReadToEnd();

    string eventMessage;
    if (level > 2)
        eventMessage = message.Stage.ToString() +"\n" + eventMessage;
    if (eventMessage.IndexOf("<soap:Fault>") > 0)
    {
        // The SOAP body contains a fault.
        if (level > 0)
            WriteToLog(eventMessage, EventLogEntryType.Error);
    }
    else
    {
        // The SOAP body contains a message.
        if (level > 1)
            WriteToLog(eventMessage, EventLogEntryType.Information);
    }
}
```

This completes the code for the SoapLog extension.

Using the SoapLog Extension

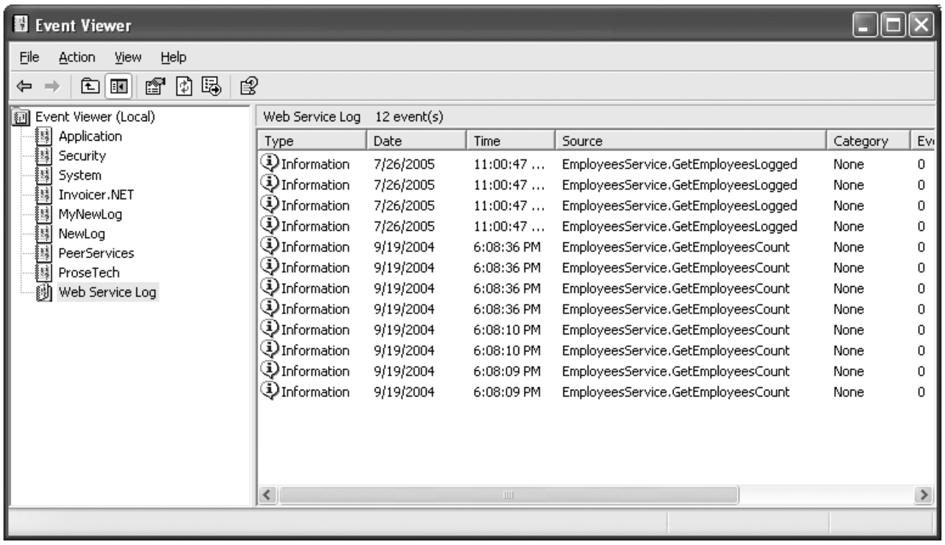
To test the SoapLog extension, you need to apply the SoapLogAttribute to a web method, as shown here:

```
[SoapLog(Name="EmployeesService.GetEmployeesLogged", Level=3)]
[WebMethod()]
public int GetEmployeesLogged()
{ ... }
```

You then need to create a client application that calls that method. When you run the client and call the method, the SoapLog extension will run and create the event log entries.

■ **Note** For the SoapLog extension to successfully write to the event log, the ASP.NET worker process (typically, the account ASPNET) must have permission to access the Windows event log. Otherwise, no entries will be written. Note that if a SOAP extension fails and generates an exception at any point, it will simply be ignored. Your client code and web service methods will not be notified.

To verify that the entries appear, run the Event Viewer (choose Programs ➤ Administrative Tools ➤ Event Viewer from the Start menu). Look for the log named Web Service Log. Figure 33-5 shows the event log entries that you'll see after calling GetEmployeesCount() twice with a log level of 3.



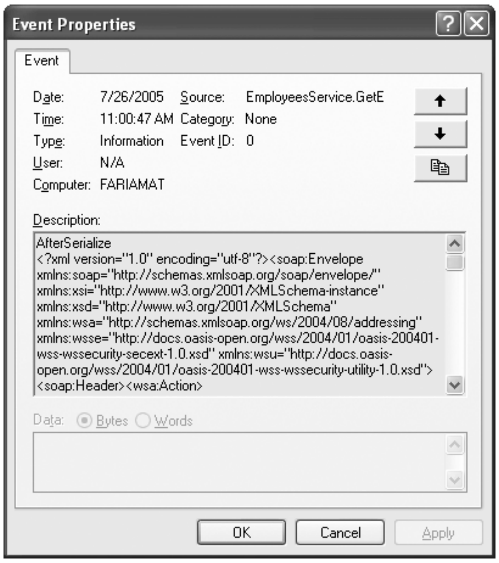
Event Viewer (Local)
- Application
- Security
- System
- Invoicer.NET
- MyNewLog
- NewLog
- PeerServices
- ProseTech
- Web Service Log

Web Service Log 12 event(s)

Type	Date	Time	Source	Category	Ev
Information	7/26/2005	11:00:47 ...	EmployeesService.GetEmployeesLogged	None	0
Information	7/26/2005	11:00:47 ...	EmployeesService.GetEmployeesLogged	None	0
Information	7/26/2005	11:00:47 ...	EmployeesService.GetEmployeesLogged	None	0
Information	7/26/2005	11:00:47 ...	EmployeesService.GetEmployeesLogged	None	0
Information	9/19/2004	6:08:36 PM	EmployeesService.GetEmployeesCount	None	0
Information	9/19/2004	6:08:36 PM	EmployeesService.GetEmployeesCount	None	0
Information	9/19/2004	6:08:36 PM	EmployeesService.GetEmployeesCount	None	0
Information	9/19/2004	6:08:36 PM	EmployeesService.GetEmployeesCount	None	0
Information	9/19/2004	6:08:10 PM	EmployeesService.GetEmployeesCount	None	0
Information	9/19/2004	6:08:10 PM	EmployeesService.GetEmployeesCount	None	0
Information	9/19/2004	6:08:09 PM	EmployeesService.GetEmployeesCount	None	0
Information	9/19/2004	6:08:09 PM	EmployeesService.GetEmployeesCount	None	0

Figure 33-5. *The event log entries for the SOAP extension*

You can look at individual entries by double-clicking them. The Description field shows the full event log message, with the XML data from the SOAP message, as shown in Figure 33-6.



Date:	7/26/2005	Source:	EmployeesService.GetE
Time:	11:00:47 AM	Category:	None
Type:	Information	Event ID:	0
User:	N/A		
Computer:	FARIAMAT		

Description:

```
AfterSerialize
<?xml version="1.0" encoding="utf-8"?><soap:Envelope
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing"
xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
wss-wssecurity-secext-1.0.xsd" xmlns:wsu="http://docs.oasis-
open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd">
<soap:Header><wsa:Action>
```

Figure 33-6. *A SOAP message in an event log entry*

The SoapLog extension is a useful tool when developing or monitoring web services. However, you should use it judiciously. If you track even a single method, the number of event log entries could quickly grow into the thousands. As the event log fills, old messages are automatically discarded. You can configure these properties by right-clicking an event log in the Event Viewer and choosing Properties.

The Web Services Enhancements

Since .NET first appeared, the world of web service standards hasn't been quiet. In fact, numerous standards, in various degrees of testing, revision, and standardization, are continuously being developed and used in the developer community. In future versions of .NET, many of these additions will be fused into the class library. As these standards are still fairly new and subject to change, they aren't ready yet. However, you can download another Microsoft tool—the free WSE toolkit—to gain support for a slew of new web service standards today.

■**Note** The general rule of thumb is that SOAP, WSDL, and UDDI are incorporated into .NET, and recent enhancements (such as SOAP 1.2) are incorporated into .NET 2.0. However, standards that are built on top of these three basics (such as those for SOAP-based encryption, security, and so on) are not yet part of .NET and are the exclusive territory of the WSE.

To install the WSE (or just read about it), browse to `http://msdn.microsoft.com/webservices/building/wse`. The WSE provides a class library assembly with a set of useful .NET classes. Behind the scenes, the WSE uses SOAP extensions to adjust the web service messages. This means your code interacts with a set of helper objects, and behind the scenes the WSE implements the appropriate SOAP standards. Using the WSE has two key disadvantages. First, the toolkit is subject to change. The version that works with Visual Studio 2005 (version 3.0) is not compatible with earlier versions. Second, you need to use the WSE on both the client and the web server. In other words, if you want to use a web service that uses the WSE to gain a new feature, your application must also use the WSE (if it's a .NET client) or another toolkit that supports the same standards (if it's not).

Many of the features in the WSE are implemented through a special SoapContext class. Your code interacts with the SoapContext object. Then, when you send the message, various filters (SOAP extensions) examine the properties of the SoapContext and, if warranted, add SOAP headers to the outgoing message. Figure 33-7 illustrates this process.

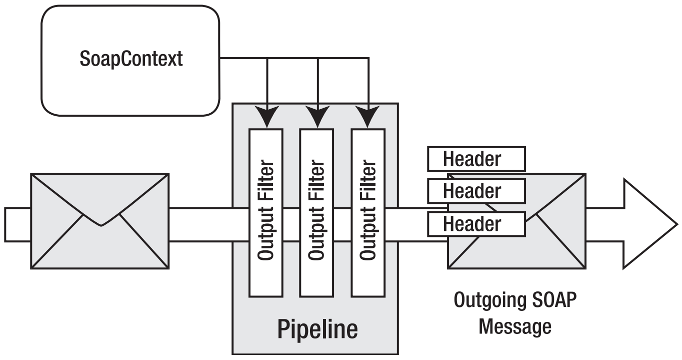


Figure 33-7. *Processing an outgoing SOAP message*

The idea behind the filters is that they can be plugged in only when they are needed. For example, if you don't need a security header, then you can leave the security header filter out of the processing pipeline by tweaking the configuration settings. Additionally, because each filter works by adding a distinct SOAP header, you have the freedom to combine as many (or as few) extensions as you need at once.

When a SOAP message is received, the same process happens in reverse. In this case, the filters look for specific SOAP headers and use the information in them to configure the SoapContext object. Figure 33-8 shows this processing model.

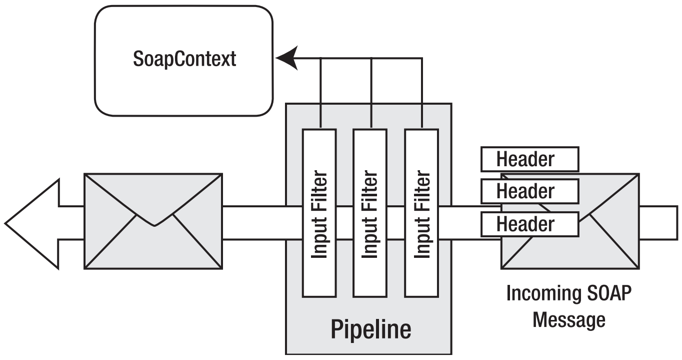


Figure 33-8. *Processing an incoming SOAP message*

So, what are the standards supported by the WSE? The full list is available with the WSE documentation, but it includes support for authentication and message encryption, establishing trust, message routing, and interoperability. The WSE also allows you to use SOAP messages for direct communication over a TCP connection (no HTTP required). In the following sections, you'll take a look at using the WSE for a simple security scenario. For a complete examination of the WSE, refer to Jeffrey Hasan's book *Expert Service-Oriented Architecture in C#: Using the Web Services Enhancements 2.0* (Apress, 2004) or the online documentation included with the WSE toolkit.

Installing the WSE

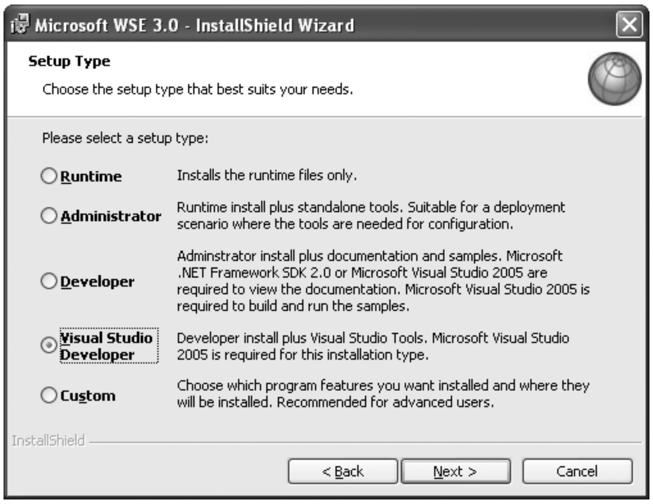
Before you can go any further, you need to download and install the WSE. When you run the setup program, choose Visual Studio Developer if you have Visual Studio 2003 installed, as this enables project support (see Figure 33-9).

To use the WSE in a project, you need to take an additional step. In Visual Studio, right-click the project name in the Solution Explorer, and select WSE Settings from the bottom of the menu. You'll see two check boxes. If you're creating a web service, select both settings, as shown in Figure 33-10. If you're creating a client application, select only the first setting, Enable This Project for Web Services Enhancements.

When you select the first option, Enable This Project for Web Services Enhancements, Visual Studio automatically adds a reference to the Microsoft.Web.Services2.dll assembly and modifies the web.config for the application to add support for the WSE configuration handler. In addition, any web references that are created from this point on will include WSE support in the proxy class. However, the web references you've already created won't have WSE support until you update them.

When you enable the second option, Enable Microsoft Web Services Enhancements SOAP Extensions, Visual Studio modifies the web.config file to register the SOAP extension that adds support for your web services. This option is required only for ASP.NET web services that use the WSE.

7680CH33.qxd 8/25/06 10:51 AM Page 1277


Microsoft WSE 3.0 - InstallShield Wizard

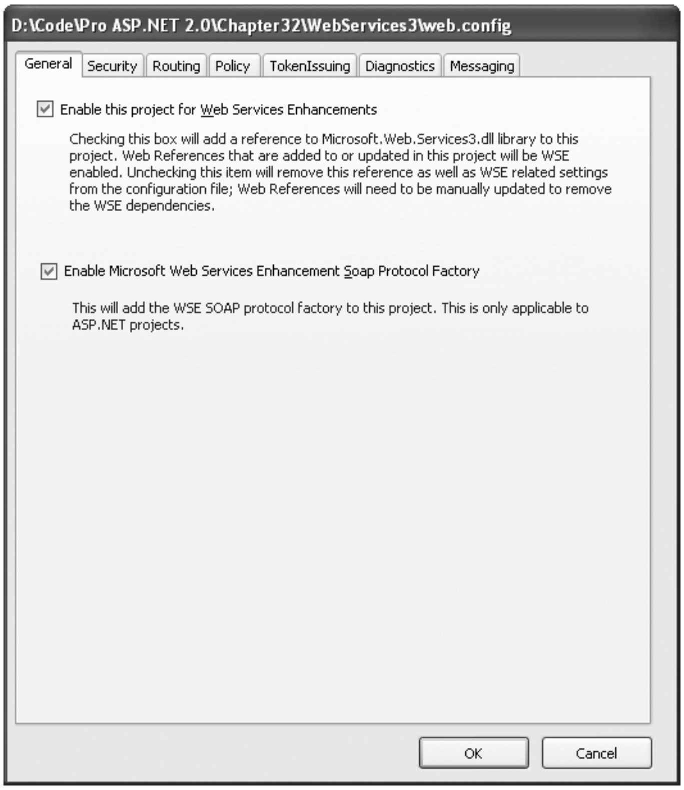
Setup Type

Choose the setup type that best suits your needs.

Please select a setup type:

○ Runtime	Installs the runtime files only.
○ Administrator	Runtime install plus standalone tools. Suitable for a deployment scenario where the tools are needed for configuration.
○ Developer	Adminstrator install plus documentation and samples. Microsoft .NET Framework SDK 2.0 or Microsoft Visual Studio 2005 are required to view the documentation. Microsoft Visual Studio 2005 is required to build and run the samples.
◉ Visual Studio Developer	Developer install plus Visual Studio Tools. Microsoft Visual Studio 2005 is required for this installation type.
○ Custom	Choose which program features you want installed and where they will be installed. Recommended for advanced users.

Figure 33-9. *Installing the WSE*


D:\Code\Pro ASP.NET 2.0\Chapter32\WebServices3\web.config

☑ Enable this project for Web Services Enhancements

Checking this box will add a reference to Microsoft.Web.Services3.dll library to this project. Web References that are added to or updated in this project will be WSE enabled. Unchecking this item will remove this reference as well as WSE related settings from the configuration file; Web References will need to be manually updated to remove the WSE dependencies.

☑ Enable Microsoft Web Services Enhancement Soap Protocol Factory

This will add the WSE SOAP protocol factory to this project. This is only applicable to ASP.NET projects.

Figure 33-10. *Enabling the WSE for a project*

Performing Authentication with the WSE

Many of the WSE classes support security standards. One of the most simple and straightforward of these classes is the UsernameToken class, which represents user credentials.

The UsernameToken information is added to the message as a SOAP header. However, it's added in a way that conforms to the WS-Security standard, which would be quite laborious to implement on your own. The benefit is that by implementing this general security model, you can add authentication without developing a proprietary approach, which could make it more difficult for third-party and cross-platform use of your web service. Also, it's likely that ASP.NET and WS-Security can provide a more secure, robust approach than one an individual developer or organization could develop (without investing considerable time and effort).

Currently, the security infrastructure in the WSE has a few holes and doesn't fully deliver on its promise. However, it's easy to use and extend. To see how it works, it's helpful to consider a simple example.

Setting Credentials in the Client

In the next example, you'll see how you can use the WSE to perform a secure call to the EmployeesService. The first step is to add the web reference. When you add a reference to the EmployeesService in a WSE-enabled project, Visual Studio will actually create *two* proxy classes. The first proxy class (which has the same name as the web service class) is the same as the web service generated in non-WSE projects. The second proxy class has the suffix *WSE* appended to its class name. This class comes from the Microsoft.Web.Services3.WebServicesClientProtocol class, and it includes support for WSE features (in this case, adding WS-Security tokens).

Thus, to use WS-Security with the EmployeesService, you need to create a WSE-enabled project, add or refresh the web reference, and then modify your code so that it uses the EmployeeServiceWse proxy. Provided you've taken those steps, you can add a new UsernameToken with your credentials using a single code statement:

```
// Create the proxy.
EmployeesServiceWse proxy = new EmployeesServiceWse();

// Add the WS-Security token.
proxy.RequestSoapContext.Security.Tokens.Add(
  new UsernameToken(userName, password, PasswordOption.SendPlainText));

// Bind the results.
GridView1.DataSource = proxy.GetEmployees().Tables[0];
```

As this code demonstrates, the security token is added as a UsernameToken object. It's inserted in the Security.Tokens collection of the RequestSoapContext, which is a SoapContext object that represents the request message you're about to send.

To use this code as written, you need to import the following namespaces:

```
using Microsoft.Web.Services3;
using Microsoft.Web.Services3.Security;
using Microsoft.Web.Services3.Security.Tokens;
```

■ **Note** Notice that the WSE namespaces incorporate the number 3, which indicates the third version of the WSE toolkit. This is because the third version is not backward compatible with the first two. To prevent a conflict with partially upgraded applications, the WSE classes are separated into distinct namespaces by version. This is part of the messy reality of working with emerging web service standards.

The constructor for the UsernameToken class accepts three parameters: a string with the user name, a string with the password, and the hashing option you would like to use. Unfortunately, if you want to use the default authentication provider in the WSE (which uses Windows authentication), you *must* choose PasswordOption.SendPlainText. As a result, this code is extremely insecure and subject to network spying unless you send the request over an SSL connection.

Although this example adds only two extra details to the request, the SOAP message actually becomes much more complex because of the way the WS-Security standard is structured. It defines additional details such as an expiration date (used to prevent replay attacks) and a *nonce* (a random value that can be incorporated in the hash to increase security). Here's a slightly shortened example of the SOAP request message:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/03/addressing"
 mlns:wsse="http://docs.oasis-open.org/wss/2004/01/..."
 xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/...">
  <soap:Header>
    <wsa:Action>http://www.apress.com/ProASP.NET/GetEmployees</wsa:Action>
    <wsa:MessageID>uuid:5b1bc235-7f81-40c4-ac1e-e4ea81ade319</wsa:MessageID>
      <wsa:ReplyTo>
        <wsa:Address>http://schemas.xmlsoap.org/ws/2004/03/...</wsa:Address>
      </wsa:ReplyTo>
      <wsa:To>http://localhost:8080/WebServices3/EmployeesService.asmx</wsa:To>
      <wsse:Security soap:mustUnderstand="1">
      <wsu:Timestamp wsu:Id="Timestamp-dc0d8d9a-e385-438f-9ff1-2cb0b699c90f">
        <wsu:Created>2004-09-21T01:49:33Z</wsu:Created>
        <wsu:Expires>2004-09-21T01:54:33Z</wsu:Expires>
      </wsu:Timestamp>
      <wsse:UsernameToken xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/..."
       wsu:Id="SecurityToken-8b663245-30ac-4178-b2c8-724f43fc27be">
        <wsse:Username>guest</wsse:Username>
        <wsse:Password
         Type="http://docs.oasis-open.org/wss/2004/01/...">secret</wsse:Password>
        <wsse:Nonce>9m8UofSBhw+XWIqfO83NiQ==</wsse:Nonce>
        <wsu:Created>2004-09-21T01:49:33Z</wsu:Created>
      </wsse:UsernameToken>
    </wsse:Security>
  </soap:Header>
  <soap:Body>
    <GetEmployees xmlns="http://www.apress.com/ProASP.NET/" />
  </soap:Body>
</soap:Envelope>
```

Reading Credentials in the Web Service

The WSE-enabled service examines the supplied token and validates it immediately. The default authentication provider that's included with the WSE uses Windows authentication, which means it extracts the user name and password from the SOAP header and uses it to log the user in under a Windows account. If the token doesn't map to a valid Windows account, an error message is returned to the client. However, if no token is supplied, no error occurs. It's up to you to check for this condition on the web server in order to restrict access to specific web methods.

Unfortunately, the WSE isn't integrated in such a way that it can use the User object. Instead, you need to retrieve the tokens from the current context. The WSE provides a RequestSoapContext. Using the RequestSoapContext.Current property, you can retrieve an instance of the SoapContext class that represents the last received message. You can then examine the SoapContext.Security.Tokens collection.

To simplify this task, it helps to create a private method like the one shown here. It checks that a token exists and throws an exception if it doesn't. Otherwise, it returns the user name.

```
private string GetUsernameToken()
{
    // Although there may be many tokens, only one of these
    // will be a UsernameToken.
    foreach (UsernameToken token in RequestSoapContext.Current.Security.Tokens)
    {
        return token.Username;
    }
    throw new SecurityException("Missing security token");
}
```

You could call the GetUsernameToken() method at the beginning of a web method to ensure that security is in effect. Overall, this is a good approach to enforce security. However, it's important to keep its limitations in mind. First, it doesn't support hashing or encrypting the user credentials. Also, it doesn't support more advanced Windows authentication protocols such as Digest authentication and Integrated Windows authentication. That's because the authentication is implemented by the WSE extensions, not by IIS. Similarly, the client always needs to submit a password and user name. The client has no way to automatically submit the credentials of the current user, as demonstrated earlier in this chapter with the CredentialCache object. In fact, the Credentials property of the proxy is ignored completely.

Fortunately, you aren't limited to the scaled-down form of Windows authentication provided by the default WSE authentication service. You can also create your own authentication logic, as described in the next section.

Custom Authentication

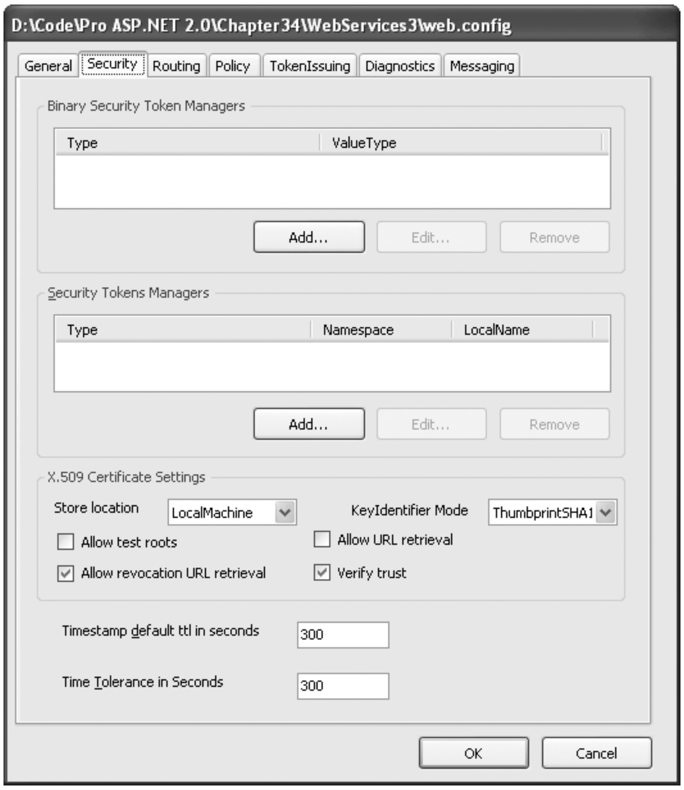
By creating your own authentication class, you can perform authentication against any data source, including an XML file or a database. To create your own authenticator, you simply need to create a class that derives from UsernameTokenManager and overrides the AuthenticateToken() method. In this method, your code needs to look up the user who is trying to become authenticated and return the password for that user. ASP.NET will then compare this password against the user credentials and decide whether authentication fails or succeeds.

Creating this class is quite straightforward. Here's an example that simply returns hard-coded passwords for two users. This provides a quick-and-easy test, although a real-world example would probably use ADO.NET code to get the same information.

```
public class CustomAuthenticator : UsernameTokenManager
{
    protected override string AuthenticateToken(UsernameToken token)
    {
        string username = token.Username;

        if (username == "dan")
            return "secret";
        else if (username == "jenny")
            return "opensesame";
        else
            return "";
    }
}
```

The reason you don't perform the password test on your own is because the type of comparison depends on how the credentials are encoded. For example, if they are passed in clear text, you need to perform a simple string comparison. If they are hashed, you need to create a new password hash using the same standardized algorithm, which must take the same data into account (including the random nonce from the client message). However, the WSE can perform this comparison task for you automatically, which dramatically simplifies your logic. The only potential problem is that you need to have the user's password stored in a retrievable form on the web server. If you're storing only password hashes in a back-end database, you won't be able to pass the original password to ASP.NET, and it won't be able to re-create the credential hash it needs to authenticate the user.

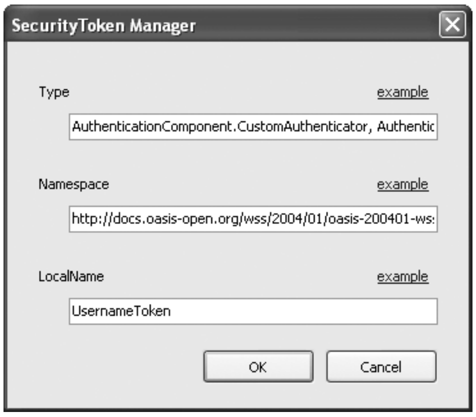
Once you've created your authentication class, you still need to tell the WSE to use it for authenticating user tokens by registering your class in the web.config file. To accomplish this, right-click the project name in the Solution Explorer, and select WSE Settings. Next, select the Security tab (shown in Figure 33-11).


D:\Code\Pro ASP.NET 2.0\Chapter34\WebServices3\web.config

Store location: LocalMachine
KeyIdentifier Mode: ThumbprintSHA1
☐ Allow test roots
☐ Allow URL retrieval
☑ Allow revocation URL retrieval
☑ Verify trust
Timestamp default ttl in seconds: 300
Time Tolerance in Seconds: 300

Figure 33-11. *Security settings*

In the Security Tokens Managers section, click the Add button. This displays the SecurityToken Manager dialog box (see Figure 33-12).



Type
AuthenticationComponent.CustomAuthenticator, Authentic

Namespace
http://docs.oasis-open.org/wss/2004/01/oasis-200401-ws:

LocalName
UsernameToken

Figure 33-12. *Configuring a new UsernameTokenManager*

In the SecurityToken Manager dialog box, you need to specify three pieces of information:

- **Type**: Enter the fully qualified class name, followed by a comma, followed by the assembly name (without the .dll extension). For example, if you have a web project named MyProject with an authentication class named CustomAuthenticator, enter **MyProject.Custom-Authenticator, MyProject**.
- **Namespace**: Enter the following hard-coded string: `http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd`.
- **QName**: Enter the following hard-code string: `UsernameToken`.

You build the client in essentially the same way, no matter how your authentication is performed on the server. However, when you create a custom authentication class, you can choose to use hashed passwords, as shown here:

```
// Create the proxy.
EmployeesServiceWse proxy = new EmployeesServiceWse();

// Add the WS-Security token.
proxy.RequestSoapContext.Security.Tokens.Add(
  new UsernameToken("dan", "secret", PasswordOption.SendHashed));

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

The best part is that you don't need to manually hash or encrypt the user password. Instead, the WSE hashes it automatically based on your instructions and performs the hash comparison on the server side. It even incorporates a random nonce value to prevent replay attacks.

Summary

In this chapter, you learned a variety of advanced SOAP techniques, including how to call web services asynchronously, how to enforce security, and how to use SOAP extensions. The world of web services is sure to continue to evolve, so look for new standards and increasingly powerful capabilities to appear in future versions of the .NET Framework.