Web Services using .NET

Barry Cornelius
Computing Services, University of Oxford
Date: 6th May 2004; first created: 31st January 2002
http://users.ox.ac.uk/~barry/papers/
mailto:barry.cornelius@oucs.ox.ac.uk

---

1 Introduction

James Snell (coauthor of the book ‘Programming Web Services with SOAP’ [36]) says:

- ‘Web services are not just a passing fad. While there is a lot of hype out there, there is also a fundamental change going on in the way the Internet works, and the various web services technologies are part of this change. It will be important for people to understand these technologies and to know how they work.’
- ‘Web services technologies represent a fundamental shift in the way web applications will be written for e-businesses. Sure, there will still be web pages and HTML, and JavaScript development, but many e-business applications will increasingly rely on programmatic interfaces that tie Internet-based applications together on a more functional level. This is what web services do for us.’

This paper:

- introduces the idea of a Web Service;
- demonstrates how the .NET Framework makes it easy to provide a Web Service;
- demonstrates how the .NET Framework makes it easy to access a Web Service;
- examines how you communicate with a Web Service (HTTP using SOAP);
- looks at a tool for monitoring the HTTP traffic;
- introduces other specifications for providing Web Services;
- considers the security features offered by the WS-Security specification;
- introduces a language (WSDL) that is used to describe the services offered by a Web Service;
- indicates what non-Microsoft products can be used for Web Services;
- demonstrates how an Apache Axis client can be used to access a .NET Web Service;
- mentions how businesses can publish/discover services (UDDI);
- demonstrates with examples how easy it is to use Web Services provided around the world.

2 What is a Web Service?

2.1 A definition

A Web Service is the provision of one or more methods that can be invoked by some external site. The external site contacts the webserver of the site providing the Web Service to get a method of the Web Service executed. It is usually a program running on the external site that wants the result of a call of the method. A site might be providing Web Services in several different areas.

2.2 An example

In their book [35] about the .NET Framework, Thai and Lam provide the following answer to the question *How can software be viewed as services?*:

- ‘The example we are about to describe might seem far-fetched; however, it is possible with current technology. Imagine the following. As you grow more attached to the Internet, you might choose to replace your computer at home with something like an Internet Device, specially designed for use with the Internet. Let’s call it an `iDev`. With this device, you can be on the Internet immediately. If you want to do word processing, you can point your `iDev` to a Microsoft Word service somewhere in Redmond and type away without the need to install word processing software. When you are done, the document can be saved at an `iStore` server where you can later retrieve it. Notice that for you to do this, the `iStore` server must host a software service to allow you to store documents. Microsoft would charge you a service fee based on the amount of time your word processor is running and which features you use (such as the grammar and spell checkers). The `iStore` service charges vary based on the size of your document and how long it is stored. Of course, all these charges won’t come in the mail, but rather through an escrow service where the money can be piped from and to your bank account or credit card.’
- ‘All of these things can be done today with Web Services.’

3 Using .NET to provide a Web Service

3.1 Providing the code to be executed

If the Web Service is provided using the .NET Framework, the methods of each Web Service may be written in any .NET language. Although we could provide all of the code ourselves, Visual Studio.NET (Microsoft’s IDE for the .NET Framework) has a wizard that generates the two files necessary to provide a Web Service. If we tell Visual Studio.NET to use C# and tell it to provide a Web Service class called `Service1`, it will produce the files `Service1.asmx` and `Service1.asmx.cs`. It will produce these files in a directory accessible by IIS (Internet Information Server (Microsoft’s webserver software)). We will assume that this webserver is at `www.a.com`.

The file `Service1.asmx.cs` contains some code in C#. All we have to do is to add the code of each method of our Web Service to the `Service1.asmx.cs` file. So we could add the method:

```
[WebMethod]
public double ToFahrenheit(double pCentigrade)
{
    return 32 + pCentigrade*9/5;
}
```

As shown above, we have to supply a `WebMethod` *attribute* for each method of the class that we wish to be accessible through the Web Service.

Such methods are declared in a class that is derived from the class `WebService` of the `System.Web.Services` namespace. It is also usual to include a `WebService` attribute:

```
0006: [WebService]
0007: public class Service1 : System.Web.Services.WebService
0008: {
0009:     ...
0010: }
```

Note: if this class provides other methods that do not have a `WebMethod` attribute, they will not be (directly) accessible by an external site.

Here is an example of the code that is generated by Visual Studio.NET's wizard. I only typed in lines 24–25 and lines 61–65:

```
0011: using System;
0012: using System.Collections;
0013: using System.ComponentModel;
0014: using System.Data;
0015: using System.Diagnostics;
0016: using System.Web;
0017: using System.Web.Services;
0018:
0019: namespace ServerConvert
0020: {
0021:     /// <summary>
0022:     /// Summary description for Service1.
0023:     /// </summary>
0024:     [WebService(Namespace="http://www.a.com/webservices/",
0025:          Description="This Web Service provides temperature conversion services.")]
0026:     public class Service1 : System.Web.Services.WebService
0027:     {
0028:         public Service1()
0029:         {
0030:             //CODEGEN: This call is required by the ASP.NET Web Services Designer
0031:             InitializeComponent();
0032:         }
0033:
0034:         #region Component Designer generated code
0035:
0036:         //Required by the Web Services Designer
0037:         private IContainer components = null;
0038:
0039:         /// <summary>
0040:         /// Required method for Designer support - do not modify
0041:         /// the contents of this method with the code editor.
0042:         /// </summary>
0043:         private void InitializeComponent()
0044:         {
0045:         }
0046:
0047:         /// <summary>
0048:         /// Clean up any resources being used.
0049:         /// </summary>
0050:         protected override void Dispose( bool disposing )
0051:         {
0052:             if(disposing && components != null)
0053:             {
0054:                 components.Dispose();
0055:             }
0056:             base.Dispose(disposing);
0057:         }
0058:
0059:         #endregion
0060:
0061:         [WebMethod(Description="This converts from Centigrade to Fahrenheit.")]
0062:         public double ToFahrenheit(double pCentigrade)
0063:         {
0064:             return 32 + pCentigrade*9/5;
0065:         }
0066:     }
0067: }
```

The above code gives more complicated forms of the `WebMethod` and `WebService` attributes.

3.2 Providing a WWW page for accessing the code

As the webmethods of the `Service1` class are to be accessed through a webserver, we must also provide a WWW page that (a) indicates that a Web Service is available and (b) gives the location of the file containing the code.

If you are using the wizard of Visual Studio.NET, this is the second file that is automatically generated for you. It will create the WWW page in a file with a `.asmx` extension. For example, the file `Service1.asmx` could contain:

```
0068: <%@ WebService Language="c#" Codebehind="Service1.asmx.cs"
0069:                Class="ServerConvert.Service1" %>
```

As you can see, this WWW page does not contain HTML instructions. Instead, it is just providing a directive to the webserver telling it that there is a Web Service in C# in the file `Service1.asmx.cs` and that it is provided by the class `ServerConvert.Service1`.

3.3 Waiting for requests to come in

So, assuming that IIS is running on `www.a.com`, the Web Service is now available at a URL like:

```
0070: http://www.a.com/wsud/cs/ServerConvert/Service1.asmx
```

3.4 Testing the Web Service

An easy way of testing a .NET Web Service is to use a browser to go to a URL like the one given above. A WWW page containing a list of the methods offered by the Web Service will be dynamically generated. You can then click on the name of the method you want to be executed. Another WWW page will be dynamically generated. It will contain a web form that has textboxes for the method's arguments and a submit button that is labelled *INVOKE*. When the button is clicked, an appropriate HTTP request is sent to the Web Service's webserver. The appropriate method will then be executed, and the webserver will send a reply back to the browser.

Instead of sending back HTML, the body of the reply is usually coded in a simple XML language. For example:

```
0071: <?xml version="1.0" encoding="utf-8"?>
0072: <double xmlns="http://www.a.com/webservices/">
0073: 32
0074: </double>
```

What the browser does when it receives this XML document depends on the browser being used. Some browsers will display the XML nicely in the pane of the browser's window; others will ask you what you want to do with the XML; or you may get a blank screen and you have to click on *View Source Code* to see the XML.

4 Writing .NET programs to access a Web Service

4.1 Introduction

We now look at how we can provide a .NET program that accesses a Web Service. Note: elsewhere, *accessing* a Web Service is called *consuming* a Web Service.

Suppose a program running on an external site wants to call the `ToFahrenheit` method provided by the Web Service that was given earlier. What we can do is to get the code of the program to generate an HTTP request, send it to the webserver providing the Web Service, and then decode the reply that gets sent back.

Once again, there are tools in the .NET Framework that help us to do this.

4.2 Using a wizard/tool to generate a proxy class

For example, Visual Studio.NET has a wizard that automatically generates the code of a class that looks similar to the class of the Web Service. This class is called a *proxy class*. When you use the wizard, you have to supply it with the URL of the Web Service. It then goes away to the appropriate webserver to query the Web Service to discover the methods that the Web Service is providing together with their parameters and return types. For each method of the Web Service that has a WebMethod attribute, the wizard will generate the code of a method that has the same parameters and the same return type.

So if you want to call a method of the Web Service, you instead call the method of the proxy class that has the same name. Behind the scenes, the call of the method of the proxy class creates the HTTP request; sends it to the webserver providing the Web Service; waits for the reply; and then decodes the reply that is returned by the Web Service.

If you have not got Visual Studio.NET, you can instead generate a proxy class with a tool (called `wsdl.exe`) that is part of the .NET Framework (which is free). This tool lives in the directory:

```
C:\Program Files\Microsoft.NET\FrameworkSDK\Bin
```

4.3 Writing a program that uses the proxy class

When we are providing a client of a Web Service, all we have to do is to create an instance of the proxy class that the wizard/tool has generated, and then apply the appropriate method to that object:

```
            Service1 tService1 = new Service1();
            double tFahrenheit = tService1.ToFahrenheit(tCentigrade);
```

4.4 Providing a console application that calls `ToFahrenheit`

So, here is a console application that calls the `ToFahrenheit` method:

```
using Console  = System.Console;
using Service1 = ConsoleConvert.Proxy.Service1;
namespace ConsoleConvert
{
    public class Class1
    {
        public static void Main()
        {
            Console.Write("Type in a Centigrade value: ");
            string tCentigradeString = Console.ReadLine();
            double tCentigrade = double.Parse(tCentigradeString);
            Console.WriteLine("Centigrade value is: " + tCentigrade);
            Service1 tService1 = new Service1();
            double tFahrenheit = tService1.ToFahrenheit(tCentigrade);
            Console.WriteLine("Fahrenheit value is: " + tFahrenheit);
        }
    }
}
```

4.5 Looking at the code of a method of the proxy class

As mentioned above, the wizard/tool automatically generates the code of a class that has similar methods. So if we gave it the URL of the Web Service that is providing the `ToFahrenheit` method it would generate code like:

```
namespace ConsoleConvert.Proxy
{
    public class Service1 : System.Web.Services.Protocols.SoapHttpClientProtocol
    {
        ...
        public double ToFahrenheit(double pCentigrade)
        {
            object[] results = this.Invoke("ToFahrenheit", new object[]{pCentigrade});
            return (double) results[0];
        }
        ...
    }
}
```

When the `ToFahrenheit` method of the proxy class is called, its call of `Invoke` sends the appropriate HTTP request to the Web Service's site and decodes the reply that is returned. Then, an appropriate value is returned to the caller of the `ToFahrenheit` method.

4.6 The full code of the proxy class

Here is the full code of the proxy class that was generated by Visual Studio.NET:

```
//------------------------------------------------------------------------------
// <autogenerated>
//     This code was generated by a tool.
//     Runtime Version: 1.1.4322.573
//
//     Changes to this file may cause incorrect behavior and will be lost if
//     the code is regenerated.
// </autogenerated>
//------------------------------------------------------------------------------

//
// This source code was auto-generated by Microsoft.VSDesigner, Version 1.1.4322.573.
//
namespace ConsoleConvert.Proxy {
    using System.Diagnostics;
    using System.Xml.Serialization;
    using System;
    using System.Web.Services.Protocols;
    using System.ComponentModel;
    using System.Web.Services;

    /// <remarks/>
    [System.Diagnostics.DebuggerStepThroughAttribute()]
    [System.ComponentModel.DesignerCategoryAttribute("code")]
    [System.Web.Services.WebServiceBindingAttribute(Name="Service1Soap",
        Namespace="http://www.a.com/webservices/")]
    public class Service1 : System.Web.Services.Protocols.SoapHttpClientProtocol {

        /// <remarks/>
        public Service1() {
            this.Url =
            "http://www.a.com/wsud/cs/ServerConvert/Service1.asmx";
        }

        /// <remarks/>
        [System.Web.Services.Protocols.SoapDocumentMethodAttribute
            ("http://www.a.com/webservices/ToFahrenheit",
            RequestNamespace="http://www.a.com/webservices/",
            ResponseNamespace="http://www.a.com/webservices/",
            Use=System.Web.Services.Description.SoapBindingUse.Literal,
            ParameterStyle=System.Web.Services.Protocols.SoapParameterStyle.Wrapped)]
        public System.Double ToFahrenheit(System.Double pCentigrade) {
            object[] results = this.Invoke("ToFahrenheit", new object[] {
                        pCentigrade});
            return ((System.Double)(results[0]));
        }

        /// <remarks/>
        public System.IAsyncResult BeginToFahrenheit(System.Double pCentigrade,
                System.AsyncCallback callback, object asyncState) {
            return this.BeginInvoke("ToFahrenheit", new object[] {
                        pCentigrade}, callback, asyncState);
        }

        /// <remarks/>
        public System.Double EndToFahrenheit(System.IAsyncResult asyncResult) {
            object[] results = this.EndInvoke(asyncResult);
            return ((System.Double)(results[0]));
        }
    }
}
```

4.7 Asynchronous calls

You will see that the proxy class provides two other methods besides `ToFahrenheit`. If a client uses the proxy class's `ToFahrenheit` method, it will sit there waiting for the Web Service's webserver to reply. If, instead, the client uses the `BeginToFahrenheit` method of the proxy class, the call of `BeginToFahrenheit` will return almost immediately. In the client, the call of `BeginToFahrenheit` has to supply (in an argument) the method that it wants to be called when the webserver delivers its reply. That method will need to call `EndToFahrenheit` to get the result of the call of `ToFahrenheit`.

5 Communicating with a .NET Web Service

5.1 HTTP GET and HTTP POST requests

When an external site wishes to call a method of a Web Service, it needs to indicate to the Web Service the name of the method to call and the arguments to be passed to the method. And the external site is expecting the Web Service to return a value. This communication is done by sending an HTTP request to the webserver of the Web Service's site.

Most of the billions of queries sent every day through the WWW are HTTP GET requests. For example, when you type the following URL into the location box of a browser:

```
http://www.a.com/pictures/index.htm
```

the browser will contact port 80 of `www.a.com` and send the following lines to it:

```
GET /pictures/index.htm HTTP/1.1
Host: www.a.com
Content-Type: text/html

```

`www.a.com` will return lines like the following to the browser:

```
HTTP/1.1 200 OK
Date: Mon, 29 Mar 2004 14:40:58 GMT
Server: Apache/1.3.27 (Unix) PHP/4.3.2 mod_ssl/2.8.12 OpenSSL/0.9.6g
Last-Modified: Mon, 29 Mar 2004 14:37:27 GMT
Accept-Ranges: bytes
Content-Type: text/html; charset=iso-8859-1
Content-Length: 31

<html>
<p>
hello world!
</p>
</html>
```

Although most WWW pages are produced by means of an HTTP GET request, some WWW pages are the result of an HTTP POST request. HTTP POST requests often occur when you click on the *submit* button of a web form. For example, suppose a WWW page provides the following web form:

```
<form method="POST" action="http://www.a.com/pictures/query.php">
    <input type="text" name="last"/>
    <input type="text" name="first"/>
    <input type="text" name="age"/>
    <input type="submit" value="submit/>
</form>
```

Suppose the visitor to this WWW page types some values (say `Bloggs`, `Fred` and `27`) into the three textboxes and then clicks the *submit* button. Because the form has a method attribute of `"POST"`, an HTTP POST request will be generated:

```
POST /pictures/query.php HTTP/1.1
Host: www.a.com
Content-Type: application/x-www-form-urlencoded
Content-Length: XXXX

last=Bloggs&first=Fred&age=27
```

5.2 Communicating with a Web Service

The .NET Framework provides three mechanisms for allowing a Web Service to be contacted:

1. an HTTP GET request;
2. a classical HTTP POST request;
3. an HTTP POST request where the body of the request is some XML coded using SOAP (the Simple Object Access Protocol [40]).

The reply from the Web Service is sent as an HTTP reply usually with a body coded in XML.

5.3 Communicating with a Web Service using HTTP POST and SOAP

Although the first two mechanisms can deal with simple cases, they are inadequate for describing more complicated arguments. To cater for more complex cases, a Web Service normally allows itself to be contacted by an HTTP POST request where the body of the request is in XML coded using SOAP.

An example is:

```
POST /wsud/cs/ServerConvert/Service1.asmx HTTP/1.1
Host: www.a.com
Content-Type: text/xml; charset=utf-8
Content-Length: XXXX
SOAPAction: "http://www.a.com/webservices/ToFahrenheit"

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ToFahrenheit xmlns="http://www.a.com/webservices/">
      <pCentigrade>0</pCentigrade>
    </ToFahrenheit>
  </soap:Body>
</soap:Envelope>
```

Hiding in all this detail is the fact that we want to contact a webserver on `www.a.com` to visit the page at `/wsud/cs/ServerConvert/Service1.asmx` to execute the `ToFahrenheit` method with an argument (`pCentigrade`) of `0`.

The reply from the webserver (after the appropriate method has been executed) is likely to be XML coded using SOAP. An example is:

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: YYYY

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <ToFahrenheitResponse xmlns="http://www.a.com/webservices/">
      <ToFahrenheitResult>32</ToFahrenheitResult>
    </ToFahrenheitResponse>
  </soap:Body>
</soap:Envelope>
```

Hidden away in all this XML is the fact that the result of the call of the method is the value 32.

5.4 Don't panic

Although the webserver handling a request receives the request coded in some way (e.g., using one of the above three mechanisms), when providing a Web Service we do not have to provide any code to decode this HTTP request. For example, we do not have to parse the XML if the third mechanism is being used.

Similarly, we do not need to generate the XML of the reply.

So, when we are providing a Web Service, we need not be concerned about this: all the hard work of decoding the request and encoding the reply is done elsewhere for us.

Similarly, when we are providing a client of a Web Service, we do not have to write code that generates the HTTP request with a body coded in SOAP and we do not have to write the code to parse the XML that gets returned by the Web Service: instead, all of that is done by the call of `Invoke` that appears in the body of each method of the proxy class, e.g.:

```
            object[] results = this.Invoke("ToFahrenheit", new object[]{pCentigrade});
```

6 Monitoring the HTTP traffic

The .NET Framework is not the only way of providing a Web Service or accessing a Web Service. There are many other products that can be used. The Apache Software Foundation provide a product called *Axis*, and we will look at how this can be used later. Included with Axis is a program called `tcpmon` that can be used to monitor the HTTP traffic that we are generating.

So, instead of the client program (e.g., `ConsoleConvert`) contacting the Web Service directly, we can arrange for it to go through the `tcpmon` program. So the client program contacts `tcpmon` and `tcpmon` contacts the Web Service.

In order to run `tcpmon`, we need to make a change to the code of the proxy class. In `Service1`'s constructor, change:

```
            this.Url =
            "http://www.a.com/wsud/cs/ServerConvert/Service1.asmx";
```

to:

```
            this.Url =
            "http://localhost:8080/wsud/cs/ServerConvert/Service1.asmx";
```

where 8080 is an arbitrarily chosen port. Then compile the `ConsoleConvert` program again, in order to get it to use this new version of the proxy class. Now, when you want to run `ConsoleClient`, you first need to ensure that `tcpmon` is running. As `tcpmon` is a part of Axis, it requires a classpath to be set up before it can be started. This can be done using commands like the following in a Command Prompt window:

```
set AXIS_HOME=E:\axis-1_1RC2
set CLASSPATH=.
set CLASSPATH=%AXIS_HOME%\lib\axis.jar;%CLASSPATH%
set CLASSPATH=%AXIS_HOME%\lib\commons-discovery.jar;%CLASSPATH%
set CLASSPATH=%AXIS_HOME%\lib\commons-logging.jar;%CLASSPATH%
set CLASSPATH=%AXIS_HOME%\lib\jaxrpc.jar;%CLASSPATH%
set CLASSPATH=%AXIS_HOME%\lib\saaj.jar;%CLASSPATH%
set CLASSPATH=%AXIS_HOME%\lib\log4j-1.2.4.jar;%CLASSPATH%
set CLASSPATH=%AXIS_HOME%\lib\wsdl4j.jar;%CLASSPATH%
```

As you can see, it is a long classpath. The `tcpmon` program can then be started using the command:

```
java org.apache.axis.utils.tcpmon 8080 www.a.com 80
```

After the `tcpmon` window appears, run the `ConsoleConvert` program in the usual way. The `tcpmon` window should show you the HTTP request that goes to `www.a.com` and the HTTP response that comes back from that site.

7 Providing more than just the basics

Of course, writing a program in terms of a Web Service means that the program is dependent on a connection to the Internet, the availability of the computer and the webserver of the Web Service, whether the Web Service still maintains the same contract as when the program was written, ... Although the facilities described so far enable us to provide and access Web Services, it only establishes the basics: a lot more has to be done if Web Services are to be used to provide a reliable software service.

Various companies have been working on producing specifications for a number of areas that build on the basics of Web Services. These include:

- Security: WS-Security, WS-Trust, WS-SecureConversation, WS-Federation
- Reliable Messaging: WS-ReliableMessaging
- Transactions: WS-Coordination, WS-AtomicTransaction, WS-BusinessActivity

These specifications are examined by the papers at [9, 11]. Other papers that look at the bigger picture include those at [10, 16, 29].

In the next section of this document, we just consider the first of these: WS-Security.

8 Adding security to a Web Service

8.1 More about WS-Security

Although you could use `https` instead of `http` to ensure SOAP messages are encrypted when they pass from A to B, this is regarded as inadequate. One of the problems with using `https` is that if B decides to send the SOAP message to some other machine C, A has no control over whether `https` gets used by B. So, can we do something different about security without using `https`?

IBM, Microsoft and Verisign have released a specification called 'WS-Security' [12, 22, 37] that provides a number of ways in which SOAP messages can be transferred securely.

For example, you can:

- arrange for the SOAP message to include in its header a username and a password, and the server rejects unrecognised usernames and passwords;
- arrange for the SOAP message to include in its header a digital signature formed from its body using the sender's private key;
- arrange for the body of the SOAP message to be encrypted.

In this document, we will only look at the first of these: providing a username and a password.

The WS-Security specification provides three ways in which a username/password can be communicated in a `UsernameToken` element of the header of a SOAP message:

- only a username is provided;
- both a username and a password are provided in plain text;
- the password is encrypted.

In this document, we will only be looking at this last possibility.

However, even if the password is encrypted, 'an evil intermediary could [capture the information in a SOAP header and later] send the hashed password and then could be authenticated as the original sender' [27].

So in a later proposal (called the 'WS-Security Addendum' [13]), 'instead of sending just a hash of the password, the addendum specifies that a digest version of the password should be sent [containing] a combination of the password, a *Nonce* (functionally a unique string that identifies this request), and the creation time' [27]. A server could use the Nonce and the timestamp values to ensure that replays do not take place.

8.2 Web Services Enhancements (WSE)

In December 2002, Microsoft released the 'Web Services Enhancements' (*WSE*) [23, 25, 26, 27, 28, 29]. This includes an implementation of the WS-Security specification (and its *Addendum*).

In order to use the WSE, you will need to download it and install it. The latest version is available from [24].

8.3 Modifying a server to check a username and a password

We now suppose that the WSE has been downloaded and installed, and that we wish to alter the server and the client of a Web Service so that they use a username and a password for authentication.

On the server, the checking of the username and the password of any `UsernameToken` elements of the header of a SOAP message is done by a method called `AuthenticateToken`. By default, the `AuthenticateToken` method provided by the `UsernameTokenManager` class checks the username of the `UsernameToken` element against those known to Microsoft Windows and returns the associated password. However, a Web Service can derive a class from the `UsernameTokenManager` class and override the `AuthenticateToken` method. This method can either have the information about valid usernames and passwords embedded in its code or it can check the information in some external source such as a database. The method should return the password associated with the username.

Here is an example where the valid usernames and passwords are embedded:

```
    public class MyUsernameTokenManager : UsernameTokenManager
    {
        protected override string AuthenticateToken(
                                UsernameToken pUsernameToken)
        {
            switch(pUsernameToken.Username)
            {
                case "superman":
                    return "mansuper";
                default:
                    throw new SoapException("Unrecognised username",
                        SoapException.ClientFaultCode);
            }
        }
    }
```

Because the `switch` has one `case`, only the username `superman` and the password `mansuper` will be accepted. In the following text, it will be assumed that this code appears as part of a namespace called `ServerUsername`.

In order to use the WSE, you will need to make some changes to the `web.config` file that is used by the Web Service. This file is stored in the same directory as the other files of the project (e.g., `Service.asmx.cs`). It contains an XML document. You will need to add the following to its `configuration` element:

```
<configSections>
    <section
        name="microsoft.web.services"
        type="Microsoft.Web.Services.Configuration.WebServicesConfiguration,
                 Microsoft.Web.Services, Version=2.0.0.0, Culture=neutral,
                 PublicKeyToken=31bf3856ad364e35" />
</configSections>
<microsoft.web.services>
    <security>
        <securityTokenManager
            qname="wsse:UsernameToken"
            xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/12/secext"
            type="ServerUsername.MyUsernameTokenManager, ServerUsername" />
    </security>
</microsoft.web.services>
```

and the following needs to be added to the `system.web` element:

```
<webServices>
    <soapExtensionTypes>
        <add
            type="Microsoft.Web.Services.WebServicesExtension,
                     Microsoft.Web.Services, Version=2.0.0.0, Culture=neutral,
                     PublicKeyToken=31bf3856ad364e35"
            priority="1"
            group="0"/>
    </soapExtensionTypes>
</webServices>
```

Note: the text of each of the two `type` elements given above needs to be typed on one line: above, they are presented on several lines in order to help you read them.

One of the changes to the `web.config` file arranges for it to include a `securityTokenManager` element. The contents of this element ensure that `ServerUsername.MyUsernameTokenManager`'s `AuthenticateToken` method gets used.

Now, whenever the Web Service is contacted, automatically behind the scenes, the WSE SOAP Extension will authenticate the usernames/passwords of any `UsernameToken` elements of the header of the SOAP message.

But we have to ensure that an incoming SOAP message actually has one `UsernameToken` element. In the following code for a Web Service providing a `ToFahrenheit` method, this is done by a subsidiary method called `iGetUsernameToken`:

```
0281: using PasswordOption       = Microsoft.Web.Services.Security.PasswordOption;
0282: using RequestSoapContext   = Microsoft.Web.Services.RequestSoapContext;
0283: using SecurityToken        = Microsoft.Web.Services.Security.SecurityToken;
0284: using SoapContext          = Microsoft.Web.Services.SoapContext;
0285: using SoapException        = System.Web.Services.Protocols.SoapException;
0286: using UsernameToken        = Microsoft.Web.Services.Security.UsernameToken;
0287: using UsernameTokenManager = Microsoft.Web.Services.Security.Tokens.UsernameTokenManager;
0288: using System;
0289: using System.Collections;
0290: using System.ComponentModel;
0291: using System.Data;
0292: using System.Diagnostics;
0293: using System.Web;
0294: using System.Web.Services;
0295: namespace ServerUsername
0296: {
0297:     [WebService(Namespace="http://www.a.com/webservices/",
0298:          Description="This Web Service provides temperature conversion services.")]
0299:     public class Service1 : System.Web.Services.WebService
0300:     {
0301:         ...  <<< as lines 28 to 60 on page 3 of this document >>>
0302:         [WebMethod(Description="This converts from Centigrade to Fahrenheit.")]
0303:         public double ToFahrenheit(double pCentigrade)
0304:         {
0305:             UsernameToken tUsernameToken = iGetUsernameToken();
0306:             return 32 + pCentigrade*9/5;
0307:         }
0308:         private static UsernameToken iGetUsernameToken()
0309:         {
0310:             SoapContext tSoapContext = RequestSoapContext.Current;
0311:             if (tSoapContext==null)
0312:             {
0313:                 throw new Exception("Only SOAP requests are permitted");
0314:             }
0315:             if (tSoapContext.Security.Tokens.Count!=1)
0316:             {
0317:                 throw new SoapException("There must be one security token",
0318:                     SoapException.ClientFaultCode);
0319:             }
0320:             foreach (SecurityToken tSecurityToken in tSoapContext.Security.Tokens)
0321:             {
0322:                 if (tSecurityToken is UsernameToken)
0323:                 {
0324:                     UsernameToken tUsernameToken = (UsernameToken)tSecurityToken;
0325:                     if (tUsernameToken.PasswordOption==PasswordOption.SendHashed)
0326:                     {
0327:                         return tUsernameToken;
0328:                     }
0329:                     else
0330:                     {
0331:                         throw new SoapException("The PasswordOption has the wrong value",
0332:                             SoapException.ClientFaultCode);
0333:                     }
0334:                 }
0335:                 else
0336:                 {
0337:                     throw new SoapException("A UsernameToken was expected",
0338:                         SoapException.ClientFaultCode);
0339:                 }
0340:             }
0341:             throw new SoapException("A SecurityToken has not been found",
0342:                 SoapException.ClientFaultCode);
0343:         }
0344:     }
0345:     ...  <<< as lines 241 to 255 on page 11 of this document >>>
0346: }
```

You will see that this code uses some types from the `Microsoft.Web.Services` namespace (and from some of its subnamespaces). These namespaces are a part of the WSE. In order to get Visual Studio.NET to find these types, you will need to go to the *Add Reference* option of its *Project* menu, and then use this option to add a reference to `Microsoft.Web.Services.dll`. This file is one of those that will have been created by installing the WSE.

Although the above code is checking that there is a `UsernameToken` element, it does not prevent replay attacks: it is being lazy in that it does not include code to check the Nonce (`tUsernameToken.Nonce`) and the timestamp value (`tUsernameToken.Created`) of the incoming SOAP message.

8.4 Getting a client to present a username and a password

We also have to alter the client so that it arranges for a username and a password to be present in the header of the SOAP message that it generates.

Currently, the proxy class (`Service1`) is derived from the class:

```
System.Web.Services.Protocols.SoapHttpClientProtocol
```

`Service1` needs to be altered so that it is derived from the class:

```
Microsoft.Web.Services.WebServicesClientProtocol
```

The latter class is itself derived from:

```
System.Web.Services.Protocols.SoapHttpClientProtocol
```

and so `Service1` can be used as before but it now has access to the other facilities that are in its new base class: these facilities are there to support WS-Security.

We can then add the username and the password to the header of the SOAP message by applying the appropriate method to the `Service1` object:

```
using Console        = System.Console;
using Exception      = System.Exception;
using PasswordOption = Microsoft.Web.Services.Security.PasswordOption;
using Service1       = ConsoleUsername.Proxy.Service1;
using UsernameToken  = Microsoft.Web.Services.Security.UsernameToken;
namespace ConsoleUsername
{
    public class Class1
    {
        public static void Main()
        {
            Console.Write("Type in a Centigrade value: ");
            string tCentigradeString = Console.ReadLine();
            double tCentigrade = double.Parse(tCentigradeString);
            Console.WriteLine("Centigrade value is: " + tCentigrade);
            Console.Write("Type in a Username: ");
            string tUsername = Console.ReadLine();
            Console.WriteLine("Username is: " + tUsername);
            Console.Write("Type in a Password: ");
            string tPassword = Console.ReadLine();
            Console.WriteLine("Password is: " + tPassword);
            UsernameToken tUsernameToken =
                new UsernameToken(tUsername, tPassword, PasswordOption.SendHashed);
            Service1 tService1 = new Service1();
            tService1.RequestSoapContext.Security.Tokens.Add(tUsernameToken);
            try
            {
                double tFahrenheit = tService1.ToFahrenheit(tCentigrade);
                Console.WriteLine("Fahrenheit value is: " + tFahrenheit);
            }
            catch(Exception pException)
            {
                Console.WriteLine(pException.Message);
            }
        }
    }
}
```

Once again, the above code uses types from a subnamespace of the `Microsoft.Web.Services` namespace. As before, in order to get Visual Studio.NET to find these types, you will need to go to the *Add Reference* option of its *Project* menu, and then use this to add a reference to `Microsoft.Web.Services.dll`.

9 Using a language other than C#

Although all the above code is given in C#, the wizards of Visual Studio.NET generate code in any language supported by Visual Studio.NET. So, for example, we could get it to generate a Web Service written in Visual Basic.NET and use that from a proxy class whose code is provided in C#, or vice-versa. Or, if we are using a Web Service that is provided by someone else, we may not know whether the Web Service is written in the same language as our proxy class.

10 Describing a Web Service using WSDL

If a site provides a Web Service, it needs to provide some means for an external site to find out the details of the Web Service, i.e., what methods are provided and how they are called.

For example, earlier we saw that, when we use the wizard of Visual Studio.NET that generates a proxy class, we supply the wizard with the URL of a Web Service. Now, the information it gets back from the Web Service describes the services being offered.

These facilities are described using the notation of an XML language called *WSDL* (*Web Services Description Language*).

A complete WSDL file is given below. Here is an explanation of some parts of the file:

1. The `service` element of a WSDL file gives basic information about the Web Service such as the location of the `.asmx` file and details about which *ports* (HTTP GET, HTTP POST and/or SOAP) are supported.
2. There is a `binding` element for each of the ports. A binding element describes for each method (provided by the Web Service) the way in which a request is coded and the way in which the reply is coded. So it might say the request is in SOAP and the reply is in SOAP. Or it might say the request is url-encoded and the reply is given using some XML coding.
3. There are two `message` elements for each of the ports. One of the message elements is used to give the name and the type of each parameter, and the other message element is used to describe the type of the result. So an *In* message element might say that there is one parameter called `pCentigrade` that is a `double` and the corresponding *Out* message element might say that a `double` is returned.

A fuller description of the purpose of the various parts of a WSDL file is provided by [21].

Here is a WSDL file that could be used to describe the Web Service being offered by `Service1.asmx`:

```
<?xml version="1.0" encoding="utf-8"?>
<definitions
 xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
 xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
 xmlns:s="http://www.w3.org/2001/XMLSchema"
 xmlns:s0="http://www.a.com/webservices/"
 xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
 xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"
 xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
 targetNamespace="http://www.a.com/webservices/"
 xmlns="http://schemas.xmlsoap.org/wsdl/">
  <types>
    <s:schema
      elementFormDefault="qualified"
      targetNamespace="http://www.a.com/webservices/">
      <s:element name="ToFahrenheit">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="1" maxOccurs="1" name="pCentigrade"
              type="s:double" />
          </s:sequence>
        </s:complexType>
      </s:element>
      <s:element name="ToFahrenheitResponse">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="1" maxOccurs="1" name="ToFahrenheitResult"
              type="s:double" />
          </s:sequence>
        </s:complexType>
      </s:element>
    </s:schema>
  </types>
  <message name="ToFahrenheitSoapIn">
    <part name="parameters" element="s0:ToFahrenheit" />
  </message>
  <message name="ToFahrenheitSoapOut">
    <part name="parameters" element="s0:ToFahrenheitResponse" />
  </message>
  <portType name="Service1Soap">
    <operation name="ToFahrenheit">
```

```
      <documentation>This converts from Centigrade to Fahrenheit.</documentation>
      <input message="s0:ToFahrenheitSoapIn" />
      <output message="s0:ToFahrenheitSoapOut" />
    </operation>
  </portType>
  <binding name="Service1Soap" type="s0:Service1Soap">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
      style="document" />
    <operation name="ToFahrenheit">
      <soap:operation
        soapAction="http://www.a.com/webservices/ToFahrenheit"
        style="document" />
      <input>
        <soap:body use="literal" />
      </input>
      <output>
        <soap:body use="literal" />
      </output>
    </operation>
  </binding>
  <service name="Service1">
    <documentation>This Web Service provides temperature conversion
      services.</documentation>
    <port name="Service1Soap" binding="s0:Service1Soap">
      <soap:address
        location="http://www.a.com/wsud/cs/ServerConvert/Service1.asmx" />
    </port>
  </service>
</definitions>
```

Although the .NET Framework provides a tool (called `disco.exe`) that can be used to generate a WSDL file, usually when using the .NET Framework you instead supply a URL to the webserver (hosting the Web Service) that causes the WSDL to be generated automatically. You do this by appending `?wsdl` to the URL for a `.asmx` file:

```
http://www.a.com/wsud/cs/ServerConvert/Service1.asmx?wsdl
```

11 Other products for Web Services

11.1 Introduction

As mentioned earlier, the .NET Framework is not the only way of providing a Web Service or accessing a Web Service. There are many other products that can be used. Here are two examples:

- Sun provide JAX-RPC. This is included in Sun's *Java Web Services Development Pack* [34].
- Apache have two products: there is *Apache SOAP* and a newer product called *Axis* [1, 3].

I have successfully installed each of the above products; used them to create Web Services; and used them to access Web Services. I have also successfully interoperated between the above products and .NET.

The next section of this document looks at how Axis can be used to access the .NET Web Service that was developed earlier in this document.

11.2 Accessing a Web Service from an Apache SOAP client

I have written a document entitled ‘Web Services using Axis’ that goes in great detail about using Axis to create Web Services and access Web Services. It is available at [3].

Here, we will only look at using the client side of Axis. The instructions that follow are paraphrased from the Axis User's Guide [1].

Earlier, details were given of how to obtain the WSDL description of a Web Service. Suppose the WSDL for the Web Service offering the `ToFahrenheit` method has been stored in the file `Service1.wsdl`.

Having downloaded and installed Axis, a proxy class for this Web Service can be created using the command:

```
java org.apache.axis.wsdl.WSDL2Java Service1.wsdl
```

As `WSDL2Java` is a program that comes with Axis, in order to execute the above command you will need to set up the long classpath that was given in Section 6 of this document.

The WSDL2Java program looks at the WSDL document to discover things about the Web Service. It then generates a proxy class.

If, having executed the above `WSDL2Java` command, you then execute the command:

```
dir com\a\www\webservices
```

you will see that the WSDL2Java program has created the files:

```
Service1.java
Service1Locator.java
Service1Soap.java
Service1SoapStub.java
```

The file `Service1SoapStub.java` contains the proxy class; the other files are supporting files.

The interface `Service1` declares headers for `get` methods for each port that is listed in the `service` element of the WSDL document. In our example, it declares a header for a method called `getService1Soap`. The class `Service1Locator` is an implementation of this interface, and so it implements the `getService1Soap` method. This method returns an instance of the proxy class `Service1SoapStub`. This class implements the interface `Service1Soap`: this interface just declares headers for each method that is provided by the Web Service. In our example, it just provides the header of the `toFahrenheit` method.

These interfaces and classes are used in the following Java program:

```
import java.rmi.              RemoteException;         // ConsoleClient.java
import javax.xml.rpc.         ServiceException;
import com.a.www.webservices. Service1;
import com.a.www.webservices. Service1Soap;
import com.a.www.webservices. Service1Locator;
public class ConsoleClient
{
    public static void main(String[] pArgs)
            throws RemoteException, ServiceException
    {
        double tCentigrade = Double.parseDouble(pArgs[0]);
        System.out.println("Centigrade: " + tCentigrade);
        Service1 tService1 = new Service1Locator();
        Service1Soap tService1Soap = tService1.getService1Soap();
        double tFahrenheit = tService1Soap.toFahrenheit(tCentigrade);
        System.out.println("Fahrenheit: " + tFahrenheit);
    }
}
```

This program expects a Centigrade value as an argument on the `java` command line:

```
javac ConsoleClient.java
java ConsoleClient 0.0
```

The `getService1Soap` method creates an instance of the proxy class `Service1SoapStub`. Having created the object, the program applies the `toFahrenheit` method to the object:

```
        double tFahrenheit = tService1Soap.toFahrenheit(tCentigrade);
```

As with proxy classes in .NET, when a method of an Axis proxy class is called, the code of the method sends the appropriate SOAP request to the Web Service; decodes the XML that is returned by the Web Service; and returns an appropriate value to the caller of the method.

12 Using Web Services provided by other sites

12.1 Accessing the Web Service provided by Amazon

12.1.1 Accessing and using Amazon's WSDL document

Amazon provide a number of web sites around the world (including `www.amazon.co.uk`). Many of these Amazon sites provide a Web Service.

There is information about Amazon Web Services at:

`http://www.amazon.com/webservices/`

The WSDL document for accessing the Web Service provided by the Amazon sites in UK and Germany is located at:

`http://soap-eu.amazon.com/schemas3/AmazonWebServices.wsdl`

and the WSDL document for the Amazon Web Services at its USA and Japan sites is located at:

`http://soap.amazon.com/schemas3/AmazonWebServices.wsdl`

12.1.2 Providing a console application that is a client of Amazon's Web Services

We can use one of the above WSDL documents to generate a proxy class for accessing the information stored at an Amazon site. You will get a proxy class called `AmazonSearchService` together with a number of supporting classes such as `AsinRequest`. The latter is used to generate a search for a particular product.

The following console application accesses the information stored at the `www.amazon.co.uk` site. An *affliate* of Amazon's site should set the `tag` field (of an `AsinRequest` object) to their affliate name. If you are not an Amazon affliate, you should set this field to `"webservices-20"`. Whether or not you are an affliate, you still have to register with Amazon to use their Web Service. In the following code, my registration id has been replaced by `"XXXXXXXXXXXXXX"`. Besides using the WSDL document that is appropriate for the particular Amazon site you wish to access, it is also necessary to set the `locale` field. In the following code, this field is set to `uk`. If your query only requires access to a small number of the available fields, you can set the `type` field to `"lite"`: if instead you want all of the values, set this field to `"heavy"`.

It seems it is part of the design of Amazon's Web Service to return `null` if a value is not available in time: it seems that their view is that it is better to be fast in providing some of the information rather than hanging around for all of it to become available.

```
using AmazonSearchService = ConsoleAmazon.Proxy.AmazonSearchService;
using AsinRequest         = ConsoleAmazon.Proxy.AsinRequest;
using Console             = System.Console;
using Details             = ConsoleAmazon.Proxy.Details;
using ProductInfo         = ConsoleAmazon.Proxy.ProductInfo;
namespace ConsoleAmazon
{
    public class Class1
    {
        public static void Main()
        {
            Console.Write("Type in an ISBN: ");
            string tIsbnString = Console.ReadLine();
            AmazonSearchService tAmazonSearchService =
                new AmazonSearchService();
            AsinRequest tAsinRequest = new AsinRequest();
            tAsinRequest.tag = "webservices-20";
            tAsinRequest.devtag = "XXXXXXXXXXXXXX";
            tAsinRequest.asin = tIsbnString;
            tAsinRequest.locale = "uk";
            tAsinRequest.type = "heavy";
            ProductInfo tProductInfo  =
                tAmazonSearchService.AsinSearchRequest(tAsinRequest);
            Details[] tDetailsArray = tProductInfo.Details;
            Details tDetails = tDetailsArray[0];
            Console.WriteLine(tDetails.ProductName);
            Console.WriteLine(tDetails.ReleaseDate);
            Console.WriteLine(tDetails.OurPrice);
            Console.WriteLine(tDetails.ListPrice);
            Console.WriteLine(tDetails.UsedPrice);
            Console.WriteLine(tDetails.SalesRank);
            Console.WriteLine(tDetails.Availability);
            Console.WriteLine(tDetails.ImageUrlSmall);
            Console.WriteLine(tDetails.ImageUrlMedium);
            Console.WriteLine(tDetails.ImageUrlLarge);
        }
    }
}
```

12.2 Sites providing Web Services

One useful site for Web Services is `www.xmethods.net`: it gives a list of sites providing Web Services. But this list is not ordered in a way that makes it easy to find a Web Service.

UDDI is an initative that aims to make it easier to locate Web Services. UDDI means *Universal Description, Discovery and Integration*. UDDI is targetted at two kinds of customers: those businesses who want to publish what services they offer and those businesses who need to locate a service and access it from some program. More details are available at `http://www.uddi.org/`.

If you use a browser to go to `http://www.xmethods.net/`, you will get a WWW page listing the 30 WWW Services that were most recently added to this website. However, half way down this WWW page, there is a link labelled `FULL LIST`. If you click on this link, you will get the full list of Web Services that are known to `www.xmethods.net`.

Towards the bottom of the full list, there is a Web Service described as:

```
Publisher:       dpchiesa
Service Name:    ZipToCityState
Description:     Retrieves valid City+State pairs for a given
                 US Zip Code, or longitude/latitude for a zipcode.
                 Also retrieves zipcodes for city/state pairs
Implementation:  MS .NET
```

If you click on the `ZipToCityState` link, the `www.xmethods.net` site gives you more details about this Web Service. In particular, it says that a WSDL document is available at:

```
http://www.winisp.net/cheeso/zips/ZipService.asmx?WSDL
```

The above URL ends in `.asmx?WSDL`. The `.asmx` is a sure sign that this Web Service has been implemented using Microsoft's .NET Framework. As explained earlier, one of the benefits of using .NET for Web Services is that WWW pages that can be used to test a Web Service are dynamically generated. You can go to the first of these WWW pages by using the above URL without the last 5 characters, i.e., by going to:

```
http://www.winisp.net/cheeso/zips/ZipService.asmx
```

You should get a WWW page saying:

```
The following operations are supported. For a formal definition,
please review the Service Description.
   * ZipToLatLong
   * CityToZip
   * ZipToCityAndState
   * CityTo1Zip
   * ZipTo1CityAndState
```

This is a list of the names of the methods provided by the Web Service. If you now click on one of these names, e.g., `ZipTo1CityAndState`, another WWW page is dynamically generated. It is the WWW page at:

```
http://www.winisp.net/cheeso/zips/ZipService.asmx?op=ZipTo1CityAndState
```

This WWW page provides a textbox and an *Invoke* button.

If you type a zipcode like `94042` into the textbox and click on the *Invoke* button, the browser will contact the webserver at `www.winisp.net` to get it to execute the `ZipTo1CityAndState` method. After a little time, your browser will give you the following WWW page:

```
<?xml version="1.0" encoding="utf-8"?>
<string xmlns="http://dinoch.dyndns.org/webservices/">
MOUNTAIN VIEW CA
</string>
```

You may have to click on *View Page Source* to see this XML.

You could use the techniques described in Section 4 of this document to provide a program that accesses this Web Service.

13 References

1. Apache, ‘Axis User’s Guide’,
   http://ws.apache.org/axis/
2. Mike Clark, Peter Fletcher, J. Jeffrey Hanson, Romin Irani, Mark Waterhouse, Jorgen Theli,
   ‘Web Services Business Strategies and Architectures’, Expert Press, 2002, 1904284132.
3. Barry Cornelius, ‘Web Services using Axis’,
   http://www.dur.ac.uk/barry.cornelius/papers/web.services.using.axis/
4. Barry Cornelius, ‘A Taste of C#’,
   http://www.dur.ac.uk/barry.cornelius/papers/a.taste.of.csharp/
5. Barry Cornelius, ‘Comparing .NET with Java’,
   http://www.dur.ac.uk/barry.cornelius/papers/comparing.dotnet.with.java/
6. Ray Djajadinata, ‘Yes, you can secure your Web services documents’,
   http://www.javaworld.com/javaworld/jw-08-2002/jw-0823-securexml.html
   http://www.javaworld.com/javaworld/jw-10-2002/jw-1011-securexml.html
7. Chrisina Draganova, ‘Web Services and Java’,
   http://www.ics.ltsn.ac.uk/pub/jicc7/draganova2.doc
8. Gil Hansen, ‘Gil Hansen’s XML & WebServices URLs’,
   http://www.javamug.org/mainpages/XML.html#WebServices
9. IBM and Microsoft, ‘Federation of Identities in a Web Services World’,
   http://msdn.microsoft.com/webservices/understanding/advancedwebservices/default.aspx?pull=/library/en-us/dnglobspec/html/ws-
10. IBM and Microsoft, ‘Reliable Message Delivery in a Web Services World’,
    ftp://www6.software.ibm.com/software/developer/library/ws-rm-exec-summary.pdf
11. IBM, Microsoft and Verisign, ‘Secure, Reliable, Transacted Web Services: Architecture and Composition’,
    http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnwebsrv/html/wsoverview.asp
12. IBM, Microsoft and Verisign, ‘Web Services Security (WS-Security)’,
    http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnglobspec/html/ws-security.asp
13. IBM, Microsoft and Verisign, ‘WS-Security Addendum’,
    http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnglobspec/html/ws-security-addendum.asp
14. Jonathan Lurie and R. Jason Belanger, ‘The great debate: .Net vs. J2EE’,
    http://www.javaworld.com/javaworld/jw-03-2002/jw-0308-j2eenet.html
15. Anbazhagan Mani and Arun Nagarajan, ‘Understanding quality of service for Web services’,
    http://www-106.ibm.com/developerworks/library/ws-quality.html
16. Joseph Mayo, ‘C# Unleashed’,
    SAMS, 2001, 0-672-32122-X.
17. Klaus Michelsen, ‘C# Primer Plus’,
    SAMS, 2001, 0-672-32152-1.
18. Microsoft, ‘Visual Studio.NET and .NET Framework Reviewers Guide’,
    http://download.microsoft.com/download/VisualStudioNET/Utility/7.0/W9X2K/EN-US/frameworkevalguide.doc
19. Microsoft, ‘Walkthrough: Creating a Web Service Using Visual Basic or C#’,
    Help system with Visual Studio.NET.
20. Microsoft, ‘Walkthrough: Accessing a Web Service Using Visual Basic or C#’,
    Help system with Visual Studio.NET.
21. Microsoft, ‘Web Services Description Language (WSDL) Explained’,
    http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnwebsrv/html/wsdlexplained.asp
22. Microsoft (Scott Seely), ‘Understanding WS-Security’,
    http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnwebsrv/html/understwspol.asp
23. Microsoft, ‘Web Services Enhancements (WSE)’,
    http://msdn.microsoft.com/webservices/building/wse/default.aspx

24. Microsoft, ‘Web Services Enhancements (WSE) 2.0 Technology Preview’,
http://www.microsoft.com/downloads/details.aspx?FamilyId=21FB9B9A-C5F6-4C95-87B7-FC7AB49B3EDD&displaylang=en

25. Microsoft (Matt Powell), ‘Programming with Web Services Enhancements 2.0’,
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnwse/html/programwse2.asp

26. Microsoft (Don Smith), ‘WS-Security Drilldown in Web Services Enhancements 2.0’,
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnwse/html/wssecdrill.asp

27. Microsoft (Matt Powell),
‘WS-Security Authentication and Digital Signatures with Web Services Enhancements’,
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnwse/html/wssecauthwse.asp

28. Microsoft (Jeannine Hall Gailey), ‘Encrypting SOAP Messages Using Web Services Enhancements’,
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnwse/html/wseencryption.asp

29. Benjamin Mitchell, ‘.NET Enterprise Web Services: WSE and Indigo’,
http://noops.prismtechnologies.com/talks/50/NOOPSDec11_Edit.ppt

30. Chris Peiris, ‘Creating a .NET Web Service’,
http://www.15seconds.com/issue/010430.htm

31. Jeffrey Richter, ‘Applied Microsoft .NET Framework Programming’,
Microsoft Press, 2002, 0-7356-1442-9.

32. John Sharp and Jon Jagger, ‘Microsoft Visual C# .NET Step by Step’,
Microsoft Press, 2003, 0-7356-1909-3.

33. Aaron Skonnard, ‘Publishing/Discovering Web Services via DISCO/UDDI’,
http://www.develop.com/conferences/conferencedotnet/materials/W4.pdf

34. Sun Microsystems, ‘Java Web Services Developer Pack 1.1’,
http://java.sun.com/webservices/webservicespack.html

35. Thuan Thai and Hoang Q. Lam, ‘.NET Framework Essentials (3rd edition)’,
O’Reilly, 2003, 0-596-00505-9.

36. Doug Tidwell, James Snell, Pavel Kulchenko, ‘Programming Web Services with SOAP’,
O’Reilly, 2001, 0-596-00095-2.

37. Paul Townend, ‘Web Service Security’,
http://www.dur.ac.uk/p.m.townend/webServiceSecurity.ppt

38. ‘uddi.org’,
http://www.uddi.org/

39. Venu Vasudevan, ‘A Web Services Primer’,
http://webservices.xml.com/pub/a/ws/2001/04/04/webservices/

40. W3C, ‘Simple Object Access Protocol (SOAP)’,
http://www.w3.org/2000/xp/Group/

41. W3C, ‘Web Services Activity’,
http://www.w3.org/2002/ws/

42. ‘WebServices.org’,
http://www.webservices.org/

43. ‘XMethods’,
http://www.xmethods.com/