



Article by: Darshan Singh (Managing Editor, PerfectXML.com)

ASP.NET Web Services Techniques

The Microsoft .NET Framework is comprised of two primary technologies: The **Common Language Runtime (CLR)** and **XML Web Services**. The CLR is the engine used for running .NET managed code and offers services such as cross-language integration, object lifetime management, thread management, etc., and it also enforces strict type safety and other forms of code accuracy that ensure security and robustness. The second .NET core technology offering, Web Services facilitate integrating applications over Internet, using open standards such as XML, SOAP and WSDL. ASP.NET, the revolutionary successor of ASP for Web development, adds support for creating and invoking XML Web services.

In this article I will make the assumption that, you being a .NET developer, at the minimum have tried out writing and calling a simple "Hello World" or "a+b" XML Web service; and are familiar with the .NET Framework, C#, VB.NET, and have some knowledge of the .NET Framework Class Library. With these assumptions, in this article I'll present some techniques that I think will be useful while you are working on "real world" XML Web services development using ASP.NET.

More specifically, you'll learn about the following ASP.NET Web Services concepts:

- **Working with SOAP Headers**
- **Asynchronous Web Services Clients**
- **SOAP Faults**
- **Passing Binary Data**
- **Session and Application State Management**
- **Caching**
- **Hooking into HTTP Pipeline**
- **WSDL.exe and web.config Tips**

You'll need the following to try out code examples from this article:

- [Microsoft Windows 2000](#), [Windows XP](#), or [Windows Server 2003](#),
- [Microsoft .NET Framework 1.0 SP2](#) or higher,
- [IIS 5.0](#) or higher; or [ASP.NET Cassini Sample Web Server](#),
- [SQL Server 7/2000](#), or [MSDE 2000](#), and
- [Microsoft Visual Studio .NET](#).

Working with SOAP Headers

SOAP makes use of XML for messaging. This means that the client sends a XML request *"envelope"* to the Web service, which processes the message, generates the XML response *"envelope"* to be sent to the client. SOAP request and response XML documents contain a root node named `Envelope`, which must contain a child node named `Body`. The `Body` tag then contains the actual message exchange details (request method/document, parameters and the response). The `Body` element might also contain `Fault` child element to report any errors. The `Envelope` root node may optionally contain a node named `Header`.

The notion of SOAP headers offers an excellent mechanism to extend the basic SOAP messaging architecture. For instance SOAP headers can be used for implementing routing, security/authentication/digital signature, or for any such "out of band" data.

A SOAP `Header` can be part of request and or response. If the `Envelope` root node contains `Header` (in addition to the required `Body`) node, the `Header` must be the immediate child to the root `Envelope` node. The `Header` node may have two optional attributes: `mustUnderstand` (to indicate if the header entry is mandatory or optional for the recipient to

process), and `actor` (used for indicate the ultimate recipient of the message for a hop-by-hop routed message).

Figure 1 shows a sample SOAP request envelope that contains the authentication details as part of the SOAP Header:

```
<?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">

  <soap:Header>
    <SQLLoginDetails Token="123" xmlns="uri:PerfectXML-TestWS">
      <UserID>sa</UserID>
      <Password>xml+net</Password>
    </SQLLoginDetails>
  </soap:Header>

  <soap:Body>
    <GetData xmlns="uri:PerfectXML-TestWS">
      <TableOrSPName>Customers</TableOrSPName>
      <IsItTableOrSP>eTableQuery</IsItTableOrSP>
    </GetData>
  </soap:Body>

</soap:Envelope>
```

Figure 1 - Sample SOAP Request Envelope

ASP.NET simplifies working with SOAP headers. Adding support for SOAP Headers to a ASP.NET Web service involves following steps:

- Creating a new class that derives from the `SoapHeader` class. The `SoapHeader` class is part of `System.Web.Services.Protocols` namespace,
- Adding an instance of the above class as a member variable to our Web service class, and
- Making use of the `[SoapHeader]` Web method attribute.

Once the above is done, working with SOAP headers is like working with any other class.

Without any further due, let's look at an example of using SOAP Headers with ASP.NET Web services. For brevity, we'll not make use of code-behind feature, and write the Web service logic directly into the `.asmx` file. We'll use C# to code the actual Web service, while the client will be written using Visual Basic .NET.

SOAP Headers Demo

In this example, we'll create a Web service that has just a single method named `GetData` that can be used to get data from a SQL Server database. The Web method takes two parameters – name of the table or a stored procedure and type of first parameter (an enum to indicate if first parameter is name of a table or a stored procedure). The Web method connects to **Northwind** database on local SQL Server runs the query and returns the dataset. Ideally you would pull the connection string from `web.config` – but here just for the sake of simplicity, we have hard-coded the connection string – **except the value of SQL Server user id and password, which is passed as SOAP Header values**.

The `GetData` Web method in the following Web service first checks for `Token` header value and if it is not "123" – raises the exception and hence sending the **SOAP Fault** back to the client. Next, based on the query type (second

parameter), it formulates the SQL statement, which it then executes using `SqlDataAdapter` and gets back a `DataSet` to be sent to the client. The connection string passed to `SqlDataAdapter` makes use of the login id and password values passed as SOAP header.

SOAPHeaders.asmx

```
<%@ WebService Language="C#" Class="SQLTableData"%>
using System;
using System.Xml;
using System.Xml.Serialization;
using System.Web.Services;
using System.Web.Services.Protocols;

using System.Data;
using System.Data.SqlTypes;
using System.Data.SqlClient;

//Overriding SoapHeader class - to create a class that represents
//the content of a SOAP Header

public class SQLLoginDetails : SoapHeader
{
    public string UserID;
    public string Password;

    [XmlAttribute]
    public string Token;
}

[WebService (Namespace="uri:PerfectXML-TestWS", Description="SOAP Headers Demo")]
public class SQLTableData
{
    public SQLLoginDetails sqlAuthentication;

    public enum eQueryType {eTableQuery, eStoredProcedure};

    [WebMethod (Description="Use this method to run a SELECT * query on a table " +
    "or to execute a stored procedure. Method returns DataSet. Requires SQL Server " +
    "authentication details passed as the SOAP Header values.")]

    [SoapHeader ("sqlAuthentication", Required=true)]
    public DataSet GetData(string TableOrSPName, eQueryType IsItTableOrSP)
    {
        DataSet dsResult = new DataSet();
        if(sqlAuthentication.Token != "123")
            throw new Exception("Missing Token.");

        string strQuery;
        if (IsItTableOrSP == eQueryType.eTableQuery)
            strQuery = "SELECT * FROM [" + TableOrSPName + "]";
        else
            strQuery = "EXEC " + TableOrSPName;

        SqlDataAdapter adapter = new SqlDataAdapter(strQuery,
            "SERVER=.;UID=" + sqlAuthentication.UserID +
            ";PWD=" + sqlAuthentication.Password + ";INITIAL CATALOG=NorthWind;");

        adapter.Fill(dsResult);
    }
}
```

```

    return dsResult;
}
}

```

The above Web service defines a class named `SQLLoginDetails` that is derived from the `System.Web.Services.Protocols.SoapHeader` class. This class is the basis for SOAP header functionality in this example – an instance of this class is added to our Web service class `SQLTableData`. The instance member variable (`sqlAuthentication`) is then used with the `soapHeader` Web method `attribute`. Also note how the `[XmlAttribute]` attribute is used in the `SoapHeader` derived class. The above Web service expects a SOAP request message similar to one shown in **Figure 1** above. As part of the SOAP header, it expects a string "Token" passed as an attribute and two child nodes named "UserID" and "Password" (in real life, these parameters would be encrypted).

The following table shows the sample request and response SOAP messages:

SOAP Request with Header

```

<?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:Header>
    <SQLLoginDetails Token="123"
      xmlns="uri:PerfectXML-TestWS">
      <UserID>sa</UserID>
      <Password></Password>
    </SQLLoginDetails>
  </soap:Header>
  <soap:Body>
    <GetData xmlns="uri:PerfectXML-TestWS">
      <TableOrSPName>Customers</TableOrSPName>
      <IsItTableOrSP>eTableQuery</IsItTableOrSP>
    </GetData>
  </soap:Body>
</soap:Envelope>

```

SOAP Response Message

```

<?xml version="1.0" encoding="utf-
<soap:Envelope
  xmlns:soap="http://schemas.xml
  xmlns:xsi="http://www.w3.org/2
  xmlns:xsd="http://www.w3.org/2

  <soap:Body>
    <GetDataResponse xmlns="uri:Perfe
    <GetDataResult>
      <xs:schema id="NewDataSet" xmlr
        xmlns:xs="http://www.w3.c
        xmlns:msdata="urn:schemas
          ...
      </xs:schema>
    <diffgr:diffgram
      xmlns:msdata="urn:schemas-
      xmlns:diffgr="urn:schemas-
        ...
    </diffgr:diffgram>
  </GetDataResult>
</GetDataResponse>
</soap:Body>
</soap:Envelope>

```

If the `Token` header value is not passed or is not "123" the Web service raises the exception – which sends the SOAP **Fault** message back to client, which looks like:

SOAP Fault when `Token` Header Value is not "123"

```

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <soap:Fault>
      <faultcode>soap:Server</faultcode>
      <faultstring>System.Web.Services.Protocols.SoapException:
        Server was unable to process request. ---
        System.Exception: Missing Token.
        at SQLTableData.GetData(String TableOrSPName, eQueryType IsItTableOrSP)
        --- End of inner exception stack trace ---</faultstring>
      <detail />
    </soap:Fault>
  </soap:Body>
</soap:Envelope>

```

Now that our Web service is ready let's write the client for this Web service and see how SOAP Headers are passed from the client. As mentioned, we'll use VB .NET to write the client applications. The client for the above `SQLTableData` Web service is a Windows Forms application which looks like:



Figure 2: SOAP Headers Demo Client Applications

Create an IIS virtual directory named `RealWorldWebSvc`s and then save the above Web service as `SOAPHeaders.asmx` under this virtual directory. To try out the client application ([Figure 2](#)) for this Web service, [download](#) the source code for this article and open the `HeadersClient.sln` solution from the `HeadersClient` folder. This Windows Forms applications refers (`Project | Add Web Reference`) to the above SOAP Headers demo Web service. Let's look at the client code:

Form1.vb

```
Imports System

Imports System.Data
Imports System.Data.SqlClient
Imports System.Data.SqlTypes

Imports System.Web
Imports System.Web.Services
Imports System.Web.Services.Protocols

Imports HeadersClient.localhost

Public Class Form1
    Inherits System.Windows.Forms.Form

    Private WebSvcClient As New SQLTableData()
    Private HeaderData As New SQLLoginDetails()

    Private Sub btnGetData_Click(ByVal sender As System.Object,
        ByVal e As System.EventArgs) Handles btnGetData.Click

        Try
            StatusBar1.Text = "Sending Web Service Request..."
            Cursor = Windows.Forms.Cursors.WaitCursor
            Dim ResultDS As DataSet
            dgResults.SetDataBinding(ResultDS, "")

            HeaderData.UserID = txtUserID.Text
            HeaderData.Password = txtPassword.Text
            HeaderData.Token = txtToken.Text
```

```

WebSvcClient.SQLLoginDetailsValue = HeaderData

If IsItSP.Checked Then
    ResultDS = WebSvcClient.GetData (cbTableOrSPName.Text,
        eQueryType.eStoredProcedure)
Else
    ResultDS = WebSvcClient.GetData (cbTableOrSPName.Text,
        eQueryType.eTableQuery)
End If

dgResults.SetDataBinding(ResultDS, "Table")

StatusBar1.Text = "Ready"
Catch se As SoapException
    StatusBar1.Text = "SOAP Exception: " & se.Message
    MessageBox.Show(se.ToString())
Catch oe As Exception
    StatusBar1.Text = "Exception: " & oe.Message
    MessageBox.Show(oe.ToString())
Finally
    Cursor = Windows.Forms.Cursors.Default
End Try

End Sub
End Class

```

After few `Imports` statements (including the one to import the referred Web service namespace), the Form class declares instances of Web service and the header classes. The button click handler than gets the values (token, user id, password, etc.) from the form controls, populates the header class members, and finally calls the Web service method. If successful, the Web service returns the `DataSet` which is then bound to the data grid using `SetDataBinding` method. Note how the exception handling is implemented in the client code.

The above client code sends the Web service request synchronously and hence it blocks till the Web service method returns. However, it would be a good idea to call the Web service **asynchronously** so that the client application user interface is not blocked while the Web service request is being processed.

Let's add another button to the above form and call it "Get Data Async" and here is the additional code required to support the **Asynchronous Web service invocation**:

```

Private Sub bgnGetDataAsync_Click(ByVal sender As System.Object,
    ByVal e As System.EventArgs) Handles bgnGetDataAsync.Click
    Try
        StatusBar1.Text = "Sending Web Service Request..."

        Dim ResultDS As DataSet
        dgResults.SetDataBinding(ResultDS, "")

        HeaderData.UserID = txtUserID.Text
        HeaderData.Password = txtPassword.Text
        HeaderData.Token = txtToken.Text

        WebSvcClient.SQLLoginDetailsValue = HeaderData

        Dim CallbackFn As New AsyncCallback(AddressOf GetDataCallback)

        If IsItSP.Checked Then
            WebSvcClient.BeginGetData(cbTableOrSPName.Text,
                eQueryType.eStoredProcedure, CallbackFn, WebSvcClient)
        Else

```

```

        WebSvcClient.BeginGetData(cbTableOrSPName.Text,
                                eQueryType.eTableQuery, CallbackFn, WebSvcClient)
    End If
Catch se As SoapException
    StatusBar1.Text = "SOAP Exception: " & se.Message
    MessageBox.Show(se.ToString())
Catch oe As Exception
    StatusBar1.Text = "Exception: " & oe.Message
    MessageBox.Show(oe.ToString())
End Try
End Sub

Public Sub GetDataCallback(ByVal ar As IAsyncResult)
    Try
        Dim WebSvcClient As SQLTableData = CType(ar.AsyncState, SQLTableData)
        AAsyncResultDS = WebSvcClient.EndGetData(ar)

        'See KB Article 318604
        Dim CallDataBindToDataGrid As New MethodInvoker(AddressOf RefreshGrid)
        BeginInvoke(CallDataBindToDataGrid)

    Catch se As SoapException
        StatusBar1.Text = "SOAP Exception: " & se.Message
        MessageBox.Show(se.ToString())
    Catch oe As Exception
        StatusBar1.Text = "Exception: " & oe.Message
        MessageBox.Show(oe.ToString())
    End Try
End Sub

Public Sub RefreshGrid()
    dgResults.SetDataBinding(AAsyncResultDS, "Table")
    StatusBar1.Text = "Ready"
End Sub

```

In the client application, when we add the Web Service Reference, Visual Studio .NET runs the **wsdl.exe** tool to generate the Web service **proxy** that is used by the client and that forwards the request to the actual Web service. In addition to generating the actual Web method proxies, it also generates the asynchronous version of Web methods – by adding two more methods for each Web method – prefixing using `Begin` and `End`, for example: `BeginGetData` and `EndGetData` for the Web method `GetData`. These methods are used for calling the Web service asynchronously.

To the `BeginGetData` method, address of a callback function is passed, which is called when the Web method eventually returns. In this callback method implementation (`GetDataCallback` in the above example), we call the `EndGetData` method and get the resultant value (`DataSet` in this case). In the callback method, we get back the dataset but we can not bind to the data grid control on the form – as both are on different threads (See KB Article [318604](#)) and hence we make use of `MethodInvoker` and `BeginInvoke` to call another member function `RefreshGrid`, which then binds to the grid.

So now the above Web service client application has two buttons: One to call Web service method synchronously and other to send asynchronous Web method request. Type the correct login parameters, select or type table or stored procedure name from the combo box, and click on the "Get Data" button and then on "Get Data Async" button and you'll notice that with synchronous request the UI becomes unresponsive till the Web method finishes (try moving the window); however with Asynchronous Web method invocation, the client is not blocked.

So far you have learned:

- How to create a Web service that accepts SOAP Headers,
- How to generate SOAP Faults,
- Passing SOAP Headers from the client,

- Handling SOAP Faults on the client side, and
- Sending Asynchronous Web service requests

Let's now look at how to pass binary data using Web Services.

Passing Binary Data

If you need to pass binary data as part of SOAP messages, there are two choices available with ASP.NET today:

- Using the "Web Services Enhancements for Microsoft .NET" toolkit that supports WS-Attachment with DIME to send attachments with SOAP. [Click here](#) to read an article on MSDN for more details on this. DIME for sending attachments with SOAP messages is also supported by Microsoft SOAP Toolkit 3.0. [Click here](#) for more information on this.
- Encoding the binary data using **Base64** or **hex** encoding and including that directly as part of the XML message (instead of separate attachment).

In this article we'll explore the second option and write an ASP. NET Web service that loads a image file from the Web server, encodes that using Base64 and Hex and sends that to the client.

Binary Data Demo

The following Web service has two methods: One that returns data encoded using Base64 encoding, while other returns hex encoded binary data.

BinaryData.asmx

```
<%@ WebService Language="C#" Class="BinaryData"%>
using System;
using System.Xml;
using System.Xml.Serialization;
using System.Web.Services;

using System.IO;

[WebService (Namespace="uri:PerfectXML-TestWS", Description="Binary Data Demo")]
public class BinaryData
{
    [WebMethod (Description="Binary Data returned using <b>base64</b> encoding.")]
    public byte[] GetImageAsBase64()
    {
        FileStream ImageFS = new FileStream(@"c:\PXML.JPG", FileMode.Open, FileAccess.Read);

        int ImageSize = Convert.ToInt32(ImageFS.Length);
        byte[] ImageBytes = new byte[ImageSize];
        ImageFS.Read(ImageBytes, 0, ImageSize);
        ImageFS.Close();
        return ImageBytes;
    }

    [WebMethod (Description="Binary Data returned using <b>Hex</b> encoding.")]
    public string GetImageAsHex()
    {
        return System.BitConverter.ToString(GetImageAsBase64()).Replace("-", "");
    }
}
```

The two key points to note in the above Web services are:

- If the Web service method return value is declares as `byte[]`, the ASP.NET Framework would serialize that byte data using Base64 encoding.
- We use `System.BitConverter.ToString` method to convert the binary data into hex.

Save the above .asmx file in the same virtual directory (`RealWorldWebSvcs`) - Make sure the file referred in the code (`c:\PXML.JPG` in above code) exists and **browse to the Web service page using Internet Explorer and see how binary data is serialized using Base64 and hex encoding.**

Let's write the client application that accesses the above Web service. Again, we'll write a VB .NET Windows Forms application. The client application refers (Project | Add Web Reference) to the above `BinaryData.asmx` Web service.

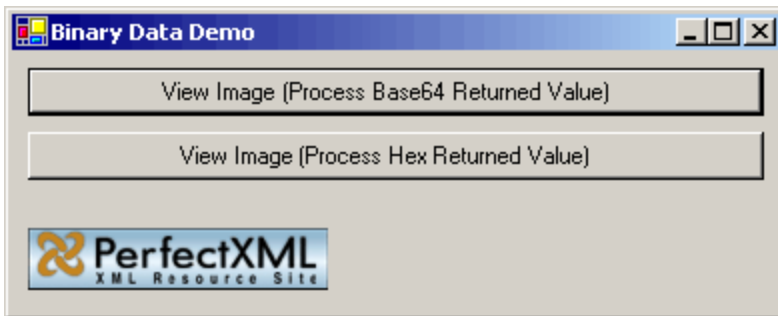


Figure 3 - Binary Data Web Service Client Application

To try out the above client application (**Figure 3**), [download](#) the source code for this article and open the `BinaryDataClient.sln` solution from the `BinaryDataClient` folder.

For Base64 encoded data, we really don't have to do much if it is the .NET client – since the Framework automatically de-serializes the Base64 data once again as byte array, which then we load into memory stream and set that as the bitmap control image. For hex data, we have written a small utility function that converts the hex encoded string into byte array.

Form1.vb

```
Imports BinaryDataClient.localhost
```

```
Imports System.IO
```

```
Imports System.Text
```

```
Public Class Form1
```

```
    Inherits System.Windows.Forms.Form
```

```
    Private Sub btnBase64_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles :
```

```
        Cursor = Cursors.WaitCursor
```

```
        Try
```

```
            Dim binDataHelper As New BinaryData()
```

```
            Dim PictureBytes() As Byte = binDataHelper.GetImageAsBase64()
```

```
            EmpPictureBox.Image = New System.Drawing.Bitmap(New MemoryStream(PictureBytes))
```

```
        Catch e1 As Exception
```

```
            MessageBox.Show(e1.ToString())
```

```
        Finally
```

```
            Cursor = Cursors.Default
```

```
        End Try
```

```
    End Sub
```

```
    Private Sub btnHex_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btn
```

```

Cursor = Cursors.WaitCursor

Try
    Dim binDataHelper As New BinaryData()
    Dim PictureBytes() As Byte = HexStringToByteArray(binDataHelper.GetImageAsHex())
    EmpPictureBox.Image = New System.Drawing.Bitmap(New MemoryStream(PictureBytes))

Catch e1 As Exception
    MessageBox.Show(e1.ToString())
Finally
    Cursor = Cursors.Default
End Try
End Sub

Private Function HexStringToByteArray(ByVal hexString As String) As Byte()
    Dim StringLen As Integer

    StringLen = hexString.Length

    If StringLen Mod 2 <> 0 Then
        Throw New ArgumentException()
    End If

    Dim ByteArrLen As Integer = (StringLen / 2) - 1
    Dim resultByteArray(ByteArrLen) As Byte
    Dim iIndex As Integer

    For iIndex = 0 To ByteArrLen
        resultByteArray(iIndex) = Convert.ToByte(hexString.Substring(iIndex * 2, 2), 16)
    Next

    Return resultByteArray
End Function

End Class

```

The above example illustrates how the .NET Framework simplifies the task of sending and receiving binary data as part of SOAP message. The Web method returns **byte array** and the .NET Framework takes care of serializing into **Base64** and de-serializing into byte array on the receiving side. If on the receiving side you can not use the .NET Framework there are other options, such as using MSXML DOM to convert Base64 encoded data into binary data. [Click here](#) for an example of using MSXML to work with XML and binary data.

Stateful Web Services

HTTP by its nature is stateless. The HTTP client sends the request; the server returns the response and forgets about the client. However, many Web applications pose the requirement to remember the client. ASP and ASP.NET support the notion of **Session** and **Application** state that can be used to maintain essentially the session-level or application-level data on the Web server, respectively. When the user first browses the Web site, ASP and ASP.NET sends a session cookie to the client – this session cookie is then used for associating the user session data on the server.

Now we know that while creating ASP.NET Web application, we can remember the user-specific details (session state) and the entire application settings (application state), is it possible to use these features while writing Web services? That is, to maintain data between `WebMethod` calls? Or remember settings for the entire Web service for all the clients? The answer is yes. If you are familiar with ASP/ASP.NET state management, the same **session** and **Application** collection properties are available to ASP.NET Web services also.

If the Web service class derives from `System.Web.Services.WebService` class, it can directly access the `Session` and `Application` collection properties; however if the Web service does **not** derive from the `System.Web.Services.WebService` class, it can still access the ASP.NET state management features by using

`HttpContext.Current.Session` and `HttpContext.Current.Application` objects, as shown in the following sample Web service.

The following sample Web service does not derive from `System.Web.Services.WebService` class and hence uses `HttpContext.Current.Session` and `HttpContext.Current.Application` (instead of directly using `Session` and `Application` objects) to store and retrieve session and application data, respectively. The Web service has four methods: `Login`, `Logout`, `GetLoggedInUsersCount`, and `ResetLoggedInUsersCount`. The `Login` method checks if user is already logged in current session: if yes, raises the exception (and hence returns SOAP Fault), else sets the session value `Customer` as the input `CustomerID` and increments the application level "logged-in users" count. The `Logout` method makes sure that user is logged into current session, else raises the exception. If the user is logged in (`Session["Customer"] != null`), the `Logout` method resets the session variable to `null` and decrements the application level "logged-in users" count. The other two methods provide access to application level "logged-in users" count and allow it to reset its value to 0. Note that you can also do session and application level processing by defining the session and application event handlers in `Global.asax` file.

StateDemo.asmx

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

[WebService (Namespace="uri:PerfectXML-TestWS", Description="Application and Session State Demo")]
public class StateDemo
{
    [WebMethod (
        EnableSession=true,
        Description="Login by passing the Customer ID, such as ALFKI, VINET, etc.")]
    public bool Login(string CustomerID)
    {
        if (CustomerID == null)
            throw new Exception("Customer ID is required.");

        if (CustomerID.Trim().Length <= 0)
            throw new Exception("Customer ID is required.");

        if(HttpContext.Current.Session["Customer"] == null)
        {
            HttpContext.Current.Session["Customer"] = CustomerID;

            if(HttpContext.Current.Application["LoggedInUsersCount"] == null)
                HttpContext.Current.Application["LoggedInUsersCount"] = 1;
            else
                HttpContext.Current.Application["LoggedInUsersCount"] =
                    Convert.ToInt32(HttpContext.Current.Application["LoggedInUsersCount"]) + 1;

            return true;
        }
        else
        {
            throw new Exception("Already logged in as " + HttpContext.Current.Session["Customer"]);
        }

        return false;
    }

    [WebMethod (
        EnableSession=true,
```

```

        Description="Logout and end the session.")]
public bool Logout()
{
    if(HttpContext.Current.Session["Customer"] == null)
    {
        throw new Exception("Not logged in.");
    }
    else
    {
        HttpContext.Current.Session["Customer"] = null;

        if(HttpContext.Current.Application["LoggedInUsersCount"] == null)
            HttpContext.Current.Application["LoggedInUsersCount"] = 0;
        else
            HttpContext.Current.Application["LoggedInUsersCount"] =
                Convert.ToInt32(HttpContext.Current.Application["LoggedInUsersCount"]) - 1;

        return true;
    }
    return false;
}

[WebMethod (
    EnableSession=true,
    Description="To find out how many users are logged in.")]
public int GetLoggedInUsersCount()
{
    if(HttpContext.Current.Application["LoggedInUsersCount"] == null)
        HttpContext.Current.Application["LoggedInUsersCount"] = 0;

    return Convert.ToInt32(HttpContext.Current.Application["LoggedInUsersCount"]);
}

[WebMethod (
    EnableSession=true,
    Description="To reset the logged in users count.")]
public void ResetLoggedInUsersCount()
{
    HttpContext.Current.Application["LoggedInUsersCount"] = 0;
}
}

```

Save the above .asmx file under a virtual directory (or a Web site) and browse to it using the Internet Explorer browser. Try executing the `Logout` method and it should return back the exception saying "Not logged in". Try `GetLoggedInUsersCount` method and it should return 0. Try `Login` method and pass any string as `CustomerID` value, it should return `True`. Try the `Login` method again with the same `CustomerID` and it should raise an exception saying "Already logged in". Start another browser instance and Try logging in using the same `CustomerID`. It should return `True` this time – as it is a different session. Try `GetLoggedInUsersCount` method and it should return 2.

Note that as ASP.NET processes multiple requests simultaneously, it is generally a good idea to synchronize access to the `Application` object – to make sure that always the correct value is returned and that you are not running into concurrency issues. You can do this by calling `Application.Lock()` before updating the value and `Application.UnLock()` after the update statement. However, if you are holding objects in `Application` scope, it generally makes sense to make the classes thread-safe instead of locking and unlocking.

[Click here](#) for more information on ASP.NET State Management.

Caching

In my opinion every ASP.NET developer (or any Web developer for that matter) should have knowledge of what Caching is, how to implement it, and how does it work. If you really want to build high-performance and scalable Web application or Web services, Caching can play an important role in your design, and help you achieve your goals. The idea behind Caching is to "remember" data or page text so that it can be reused and hence saving resources and time, ultimately improving the performance and serving more requests. The data/page text can be cached either on the server (in-memory) or at some intermediaries (example: proxy server) or on the client-side (browser cache).

I would recommend the following links if you want to learn about ASP.NET Caching:

- [ASP.NET Caching - Article by Rob Howard](#)
- [Caching Architecture Guide for .NET Framework Applications](#)
- [ASP.NET Caching Features - .NET Framework Documentation](#)
- [INFO: ASP.NET Caching Overview \(KB Article 307225\)](#)
- [ASP.NET Caching \(O'Reilly OnDotNet Web site\)](#)

In this article, I'll introduce you to the concept by presenting a very simple example of using ASP.NET Caching (*Method: Page Output Caching*) with Web service methods. Please refer to above links and .NET Framework documentation for complete details on Caching.

The following sample Web service has just one method: `GetFilesCount` that returns the number of files in the current directory. The entire response of this `WebMethod` is cached for 60 seconds by using the `CacheDuration=60` property on the `WebMethod` attribute.

CachingDemo.asmx

```
<%@ WebService Language="C#" Class="CachingDemo"%>
using System;
using System.Web;
using System.Web.Services;
using System.Xml;
using System.IO;

[WebService (Namespace="uri:PerfectXML-TestWS", Description=".NET XML Web Services Caching Demo")]
public class CachingDemo : WebService
{
    [WebMethod (CacheDuration=60, Description="Returns number of files in the current folder.")]
    public string GetFilesCount()
    {
        string CurDir = Server.MapPath(".");

        string Message = "As of: " + System.DateTime.Now.ToString() + " " +
            CurDir + " contains " + Directory.GetFiles(CurDir).Length.ToString() +
            " files.";

        return Message;
    }
}
```

Save the above .asmx file under a virtual root, then browse to it and try the `GetFilesCount` method. You should get response similar to one shown in the following screenshot:

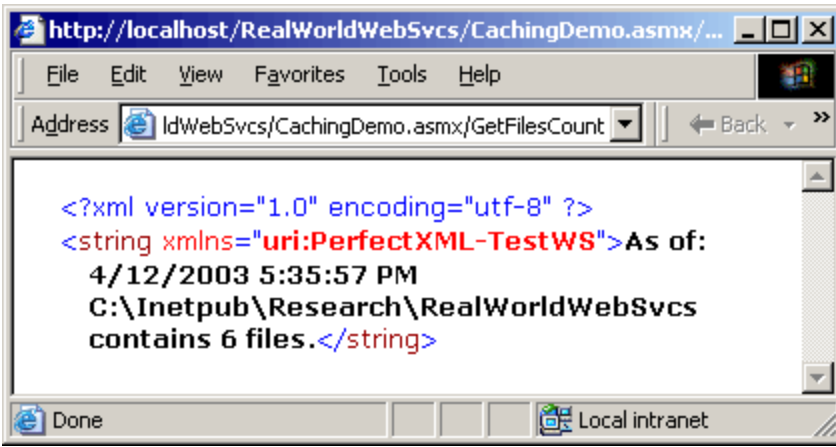


Figure 4 - WebMethod response is cached for 60 seconds by using the `CacheDuration=60` attribute.

Immediately after browsing to the above Web service page, try refreshing (F5) the page and you'll notice that neither the date time nor the "number of files" count change, even if you copy some files into the virtual directory folder or delete some files - the reason being we have set to cache the Web method response for 60 seconds. Keep refreshing the page and after 60 seconds Web method will return the updated date time along with the current number of files count.

The above Web service example illustrated how a single property value on `WebMethod` attribute enables the ASP.NET caching functionality. I encourage you to do some more [reading](#) on Caching and see how you can further use it for creating scalable ASP.NET Web services and Web sites.

Hooking into HTTP Pipeline

Before .NET was available, if you were to hook into the IIS request-response processing, the most common approach taken was to write the ISAPI extension and filter DLLs, mostly using C++. In .NET Framework, the equivalent of extension DLLs is classes that implement the `IHttpHandler` interface; and the equivalent of filter DLLs is classes that implement the `IHttpModule` interface.

HTTP handlers, like ISAPI extension DLLs, are capable of servicing HTTP requests. In ASP.NET Framework, the request eventually is serviced by a class that implements `IHttpHandler` interface. For instance, each `.aspx` Web page is a class that derives from `System.Web.UI.Page` which in turn implements the `IHttpHandler` interface.

The `IHttpHandler` interface defines one public property (`IsReusable`) and a single method (`ProcessRequest`). The `ProcessRequest` method has one parameter of type `HttpContext`, which encapsulates the HTTP request, response, and state management objects.

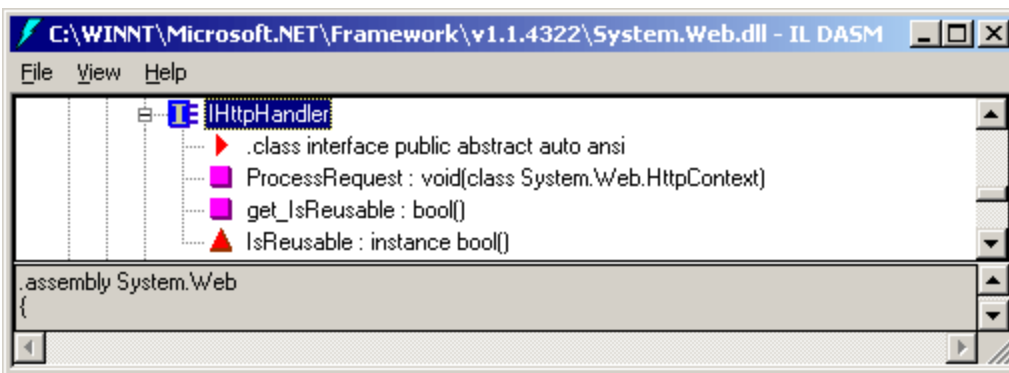


Figure 5 - `IHttpHandler` interface as defined in `System.Web.dll`

In summary, you can write classes that implement `IHttpHandler` in order to participate in low-level request/response processing (like ISAPI extension DLLs); and write classes that implement `IHttpModule` for pre/post processing of HTTP requests (like ISAPI filter DLLs). Instead of further elaborating on HTTP pipeline and how it works, I would point you to some good resources on MSDN and somewhere else. If you are not familiar with how HTTP pipeline (the flow of

HTTP request and response from Web server to ASP.NET classes and back) in ASP.NET works, read these articles and come back here for an example of how to write customer HttpHandler to implement a Web service.

- [HOW TO: Create an ASP.NET HTTP Handler by Using Visual C# .NET](#)
- [INFO: ASP.NET HTTP Modules and HTTP Handlers Overview](#)
- [HTTP Modules](#)
- [Serving Up ASP.NET Handlers the Easy Way](#)
- [HTTP Handlers and HTTP Modules in ASP.NET](#)
- [HTTP Runtime Support](#)
- [Deploying HTTP Handlers and HTTP Modules](#)

IHttpHandler Web Service Demo

Let's write a simple Web service that accepts a string and returns hex-formatted SHA1 encrypted hash for the input string.

The following .cs file contains a C# class that implements IHttpHandler interface. The ProcessRequest method:

- first calls a supporting private function that uses DOM to load the SOAP message from the HTTP request stream and extracts the string to be encrypted,
- it then calls another supporting function to do SHA1 encryption on the input string,
- finally, it calls yet another supporting private function, that loads a local XML file that contains SOAP response template, fills it with the SHA1 encrypted string, and writes that to the response stream.

SampleHttpHandler.cs

```
namespace PerfectXML.HttpHandlerTest
{
    using System;
    using System.Web;
    using System.Xml;
    using System.Security.Cryptography;

    public class SampleHttpHandler : IHttpHandler
    {
        public bool IsReusable {get {return true; } }

        public void ProcessRequest(HttpContext ctx)
        {
            string InputString = GetInputString(ctx);

            string SHA1EncryptedString = DoSHA1Encryption(InputString);

            SendResponse(ctx, SHA1EncryptedString);

            return;
        }

        private string GetInputString(HttpContext ctx)
        {
            try
            {
                XmlReader reqStreamReader = new XmlTextReader(ctx.Request.InputStream);
                XmlDocument soapRequestDoc = new XmlDocument();
                soapRequestDoc.Load(reqStreamReader);

                XmlNode inputStringNode;
                inputStringNode = soapRequestDoc.DocumentElement.SelectSingleNode("//InputString");
            }
        }
    }
}
```

```

        return inputStringNode.InnerText;
    }
    catch(Exception e1)
    {
        return e1.ToString();
    }
}

private string DoSHA1Encryption(string InputString)
{
    const int DATA_SIZE = 40;
    byte[] data = new byte[InputString.Length];

    char[] chardata = InputString.ToCharArray();
    char[] resultdata = new char[DATA_SIZE];

    int iIndex=0;
    foreach (char c in chardata)
        data[iIndex++] = Convert.ToByte(c);

    byte[] result;

    SHA1 sha = new SHA1CryptoServiceProvider();
    result = sha.ComputeHash(data);
    return System.BitConverter.ToString(result).Replace("-", "").ToLower();
}

private void SendResponse(HttpContext ctx, string SHA1EncryptedString)
{
    try
    {
        ctx.Response.BufferOutput = true;
        ctx.Response.ContentType = "text/xml";

        XmlDocument soapResponseDoc = new XmlDocument();
        soapResponseDoc.Load(ctx.Server.MapPath("SOAPResponse.xml"));

        XmlNode resultStringNode;
        resultStringNode = soapResponseDoc.DocumentElement.SelectSingleNode("//GetSHA1HashResult");
        resultStringNode.InnerText = SHA1EncryptedString;

        ctx.Response.Write(soapResponseDoc.DocumentElement.InnerXml);
    }
    catch(Exception e)
    {
        ctx.Response.Write(e.ToString());
    }
}
}
}

```

The above C# class when compiled can now acts as an endpoint for HTTP request. Let's deploy this HTTP handler: [\(Related link: Deploying HTTP Handlers and HTTP Modules\)](#)

- **Step 1:** Compile the C# class
csc /out:bin\SampleHttpHandler.dll /t:library SampleHttpHandler.cs

Make sure you have a sub-folder named bin under the current virtual root and run the above command on the

DOS command prompt. The above command executes the C# command line compiler asking it to build a assembly DLL and put it in the bin folder.

- **Step 2:** Update IIS Metabase

Start IIS Service Manager, right click on the virtual directory, select Properties, click on Configuration button on the "Virtual Directory" tab to bring the Application Configuration dialog box. Click "Add" button to add the Application Extension Mapping. For Executable text box, select **aspnet_isapi.dll** from the same location as it is for .aspx or .asmx handler (for example: C:\WINNT\Microsoft.NET\Framework\v1.1.4322\aspnet_isapi.dll), type extension as **.enc**. Leave Verbs to "All Verbs" and only "Script engine" checked.

This tells IIS that for any HTTP request for file with extension .enc, it should forward the request to ASP.NET (aspnet_isapi.dll). Only thing remaining now is to tell ASP.NET to use our handler for such requests. We do this by updating web.config file under the vroot.

- **Step 3:** Updating web.config

Add the following entry in the web.config file under the **system.web** section:

```
<httpHandlers>
  <add verb="*" path="*.enc"
        type="PerfectXML.HttpHandlerTest.SampleHttpHandler, SampleHttpHandler" />
</httpHandlers>
```

The above web.config entry defines a new HttpHandler for all types of requests for all files with extension .enc. The type attribute contains the fully qualified class name followed by the assembly name.

It's now time to write a client for the above Web service. Once again, we'll write a VB.NET Windows Forms application. [Download](#) the code accompanied with this article and open **HttpHandlerClient.sln**.

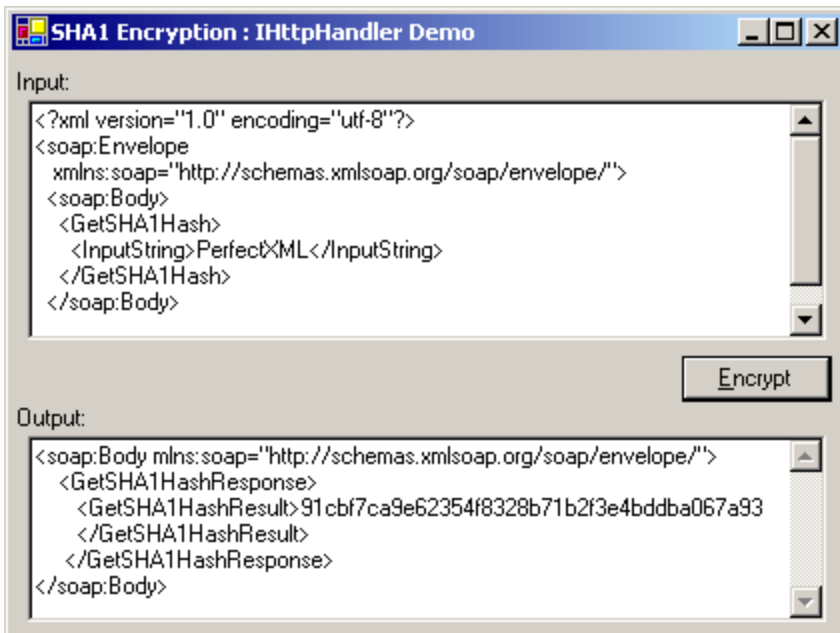


Figure 6 - Web service client

Form1.vb

Try

```
Cursor = Cursors.WaitCursor
```

```
'Send POST request to *.enc file - the request stream contains the SOAP message
```

```
Dim req As HttpWebRequest = _
```

```
    CType(WebRequest.Create("http://localhost/RealWorldWebSvcs/test.enc"), HttpWebRequest)
```

```
req.Method = "POST"
```

```

req.ContentLength = txtInput.Text.Length
Dim aStream As Stream = req.GetRequestStream()
Dim encoding As New ASCIIEncoding()
Dim byteData As Byte() = encoding.GetBytes(txtInput.Text)
aStream.Write(byteData, 0, byteData.Length)
aStream.Close()

'Read the Response
Dim res As HttpWebResponse = req.GetResponse()
aStream = res.GetResponseStream()
Dim byteResultData(res.ContentLength) As Byte
aStream.Read(byteResultData, 0, res.ContentLength)
txtOutput.Text = encoding.GetString(byteResultData)
aStream.Close()

Catch e1 As Exception
    txtOutput.Text = e1.ToString()
Finally
    Cursor = Cursors.Default
End Try

```

The above client application uses `HttpRequest` and `HttpResponse` classes from the `System.Net` to POST a SOAP request message and receive the response.

Tips

Here are some ASP.NET Web services tips:

- **Disabling GET and POST**

By default, ASP.NET Web services support GET, POST, and SOAP methods to invoke Web services. The `remove` element can be used to turn off GET and POST:

```

<system.web>
  <webServices>

    <protocols>
      <remove name="HttpPost"/>
      <remove name="HttpGet"/>
    </protocols>

  </webServices>
</system.web>

```

- **Web Service URL from the web.config file**

When you run `wsdl.exe` (or when Visual Studio .NET runs it for you to add Web Reference), the generated Web service proxy class contains the hard-coded reference to the Web service URL. However if you wanted the flexibility to configure the Web service URL, you can pass `/appsettingurlkey:` (or `/urlkey:` for short) parameter with `wsdl.exe` and along with the key in `web.config` file from where the generated proxy class will read the Web service URL.

- **Generating code using wsdl.exe**

By default the `wsdl.exe` tool generates the Web service client (proxy) code. However, if you pass the `/server` parameter, it will generate an abstract class for an XML Web service implementation. By default C# code is generated, use `/language` (or `/l` for short) for other languages such as VB.NET.

- **web.config editor**

If you hate text editing `.config` files, then \$24.95 can get you an excellent editor. See [HunterStone Inc. Web](#)

[site](#) for more details.

Summary

The goal of this article was to present some of the ASP.NET Web services development techniques that you can use while building SOAP applications. This article illustrated using SOAP headers, writing asynchronous clients, raising and handling SOAP faults, working with binary data, state management, caching, and hooking into HTTP pipeline.