

Microsoft .NET Web Service Enhancements 3.0

Contents	Page No
1. Introduction.....	1
2. The problems.....	1
3. Creating a Web reference.....	1
4. Dealing with X509 certificates.....	1
5. Configuring the project.....	2
6. Adding a custom PolicyAssertion.....	11
6.1 Fixing the <Timestamp> problem.....	12
6.2 Fixing the courseCatalogResponse problem.....	14
7. Troubleshooting.....	15

1.Introduction

As part of the SLC HEI Database project, it was necessary to connect to a Web Service exposed by the SLC. This Web Service required user credentials to be included in the SOAP Header, but did not require any X509 certification of the client, any encryption over and above that provided by the HTTPS transport layer, any password hashing or the provision of additional security parameters such as a timestamp, creation date or nonce.

2.The problems

Using the WSE 3.0 GUI configuration tools provided with the WSE 3.0 plugin for Visual Studio did not allow for the generation of a header formatted in the way required by the SLC. Furthermore, because the Web Service WSDL file was distributed out-of-band, creating a Web reference in the normal way in Visual Studio was not possible, as no access to the WSDL file on the remote server is granted.

Problems were also experienced with respect to the `courseCatalogResponse` message sent by the Web Service in response to a `courseCatalogRequest` message. Apparently redundant `<xsi:type>` attributes appear in `<code>` elements of the response with a value of "ns1:code". Such elements fail validation unless they also contain a namespace definition for ns1:

```
xmlns:ns1="http://www.slc.co.uk/course/types/1.0"
```

3.Creating a Web reference

Web references are created in Visual Studio by calling the `wsdl.exe` application behind-the-scenes. It results in the creation of the C# source code for a proxy, that can be added to a .NET solution and compiled and referenced in the normal way.

Unfortunately, the proxy created in this way inherits from the `SoapHttpClientProtocol` parent class. This class is not WSE 3.0 aware. The class declaration must therefore be altered manually from:

```
public partial class CourseServices :  
    System.Web.Services.Protocols.SoapHttpClientProtocol
```

To:

```
public partial class CourseServices :  
    Microsoft.Web.Services3.WebServicesClientProtocol
```

4.Dealing with X509 certificates

Secure Web Services have the ability for both client and server to authenticate each other using X509 certificates. At the very least, the client will want to authenticate the server. For this to work, it is necessary to define a method that will act as a callback for certificate verification. Such a method may look something like this:

```

private bool CertificateValidationCallback(object sender,
                                         X509Certificate certificate,
                                         X509Chain chain,
                                         SslPolicyErrors sslPolicyErrors)
{
    if (sslPolicyErrors == SslPolicyErrors.None)
    {
        return true;
    }
    else
    {
        if (sslPolicyErrors == SslPolicyErrors.RemoteCertificateChainErrors)
        {
            Console.WriteLine("The X509Chain.ChainStatus returned an array " +
                              "of X509ChainStatus objects containing error information.");
        }
        else if (sslPolicyErrors ==
                 SslPolicyErrors.RemoteCertificateNameMismatch)
        {
            Console.WriteLine("There was a mismatch of the name " +
                              "on a certificate.");
        }
        else if (sslPolicyErrors ==
                 SslPolicyErrors.RemoteCertificateNotAvailable)
        {
            Console.WriteLine("No certificate was available.");
        }
        else
        {
            Console.WriteLine("SSL Certificate Validation Error!");
        }
    }
}
}

```

For test purposes, the callback can be set to simply return true.

For the callback to work, it must be registered, usually in the method that calls the Web Service. This is done using the `ServicePointManager` class in the `System.Net` namespace:

```

ServicePointManager.ServerCertificateValidationCallback =
new RemoteCertificateValidationCallback(CertificateValidationCallback);

```

Note that the project will also need to reference the `System.Security` namespace.

5. Configuring the project

The project that needs configuring is the solution's executable assembly, or the Web project in the case of an ASP.NET project. Usually, this means the startup project for your solution. If the project is layered in an n-Tier arrangement, WSE 3.0 must be activated both for the executable assembly/Web project and for the project that actually calls the Web Service. For instance in a Windows Forms project with an interface layer and a business layer, where the business layer object actually interacts with the Web Service, WSE 3.0 must be activated for both projects:



(See below for details of configuring WSE 3.0)

Although both projects in this scenario must have WSE enabled, only the executable assembly/Web project needs to have the policy file and `app.config` defined (see below).

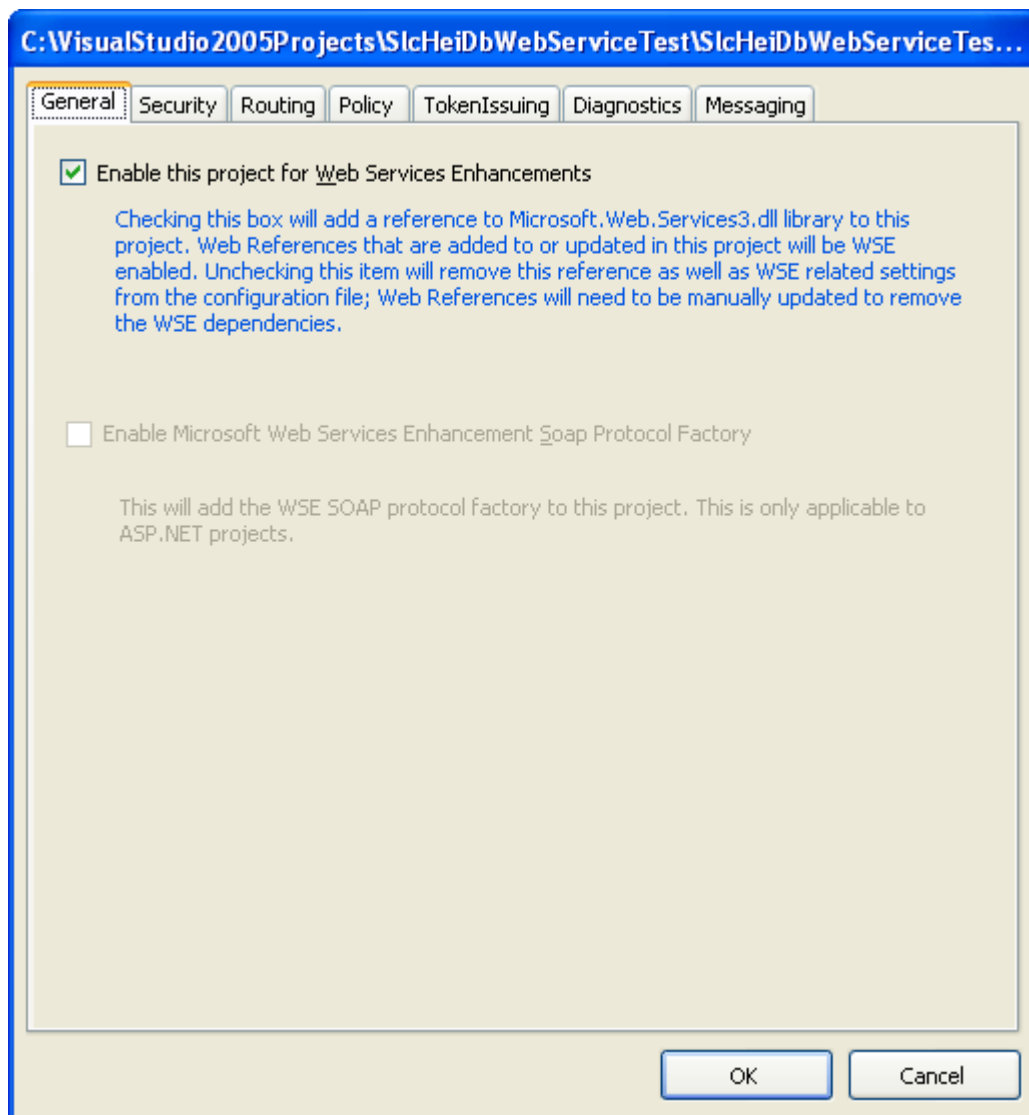
The SLC Web Service exposed by the SLC required a SOAP message in the following format (representing the `qualificationListRequest` in the SOAP body):

```
<soap:Envelope
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-
    secext-1.0.xsd" xmlns:wsu="http://docs.oasis-
    open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd">
  <soap:Header>
    <wsse:Security soap:mustUnderstand="1">
      <wsse:UsernameToken>
        <wsse:Username>*****</wsse:Username>
        <wsse:Password Type="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
          wss-username-token-profile-1.0 #PasswordText">*****</wsse:Password>
      </wsse:UsernameToken>
    </wsse:Security>
  </soap:Header>
  <soap:Body xmlns:ns1="http://www.slc.co.uk/course/schema/1.0">
    <ns1:qualificationListRequest>
      <schemaVersion>1.0</schemaVersion>
    </ns1:qualificationListRequest>
  </soap:Body>
</soap:Envelope>
```

To this end, the following WSE 3.0 configurations were made using the tool provided by Visual Studio, which must be installed before the solution is created. The installer can be found at:

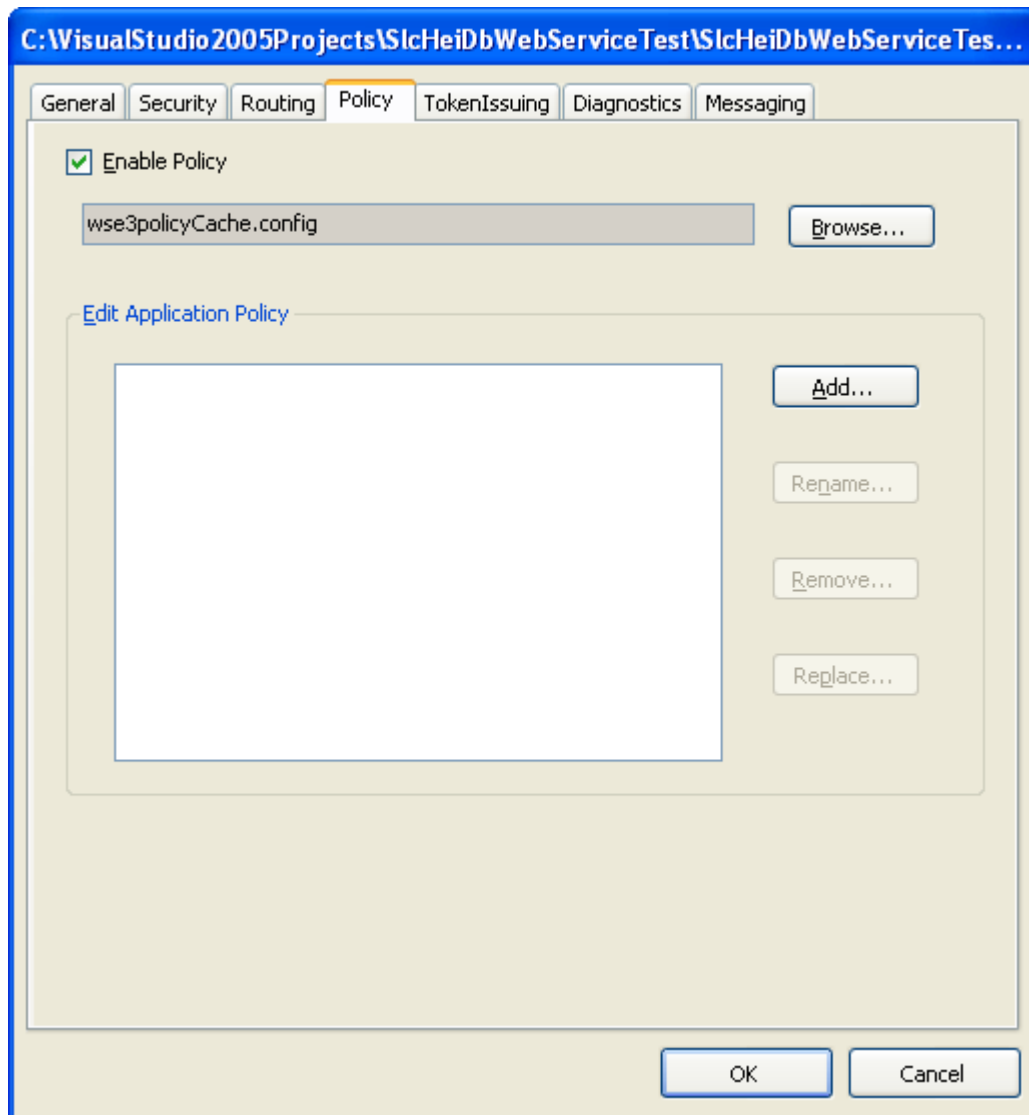
[Microsoft WSE 3.0.msi](#)

The tool is accessed by right-clicking on the project in the Visual Studio Solution Explorer, and selecting **WSE Settings 3.0 ...** from the context menu. The following tabbed dialog then appears:

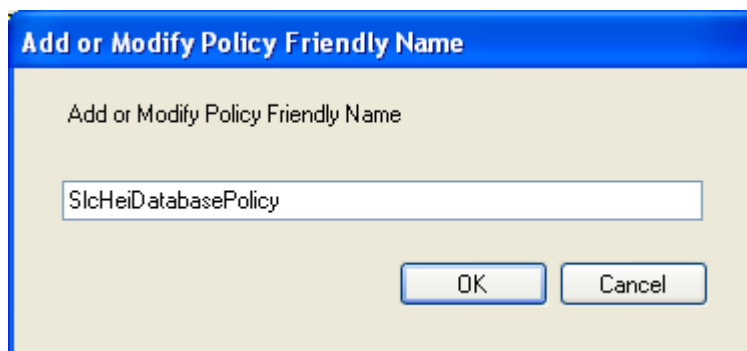


On the *General* tab, ensure that the *Enable this project for Web Services Enhancements* check box is checked. The second checkbox should be disabled because it only applies to server applications.

The *Security* and *Routing* tabs can be left at their default settings. On the *Policy* tab, check the Enable Policy check box:



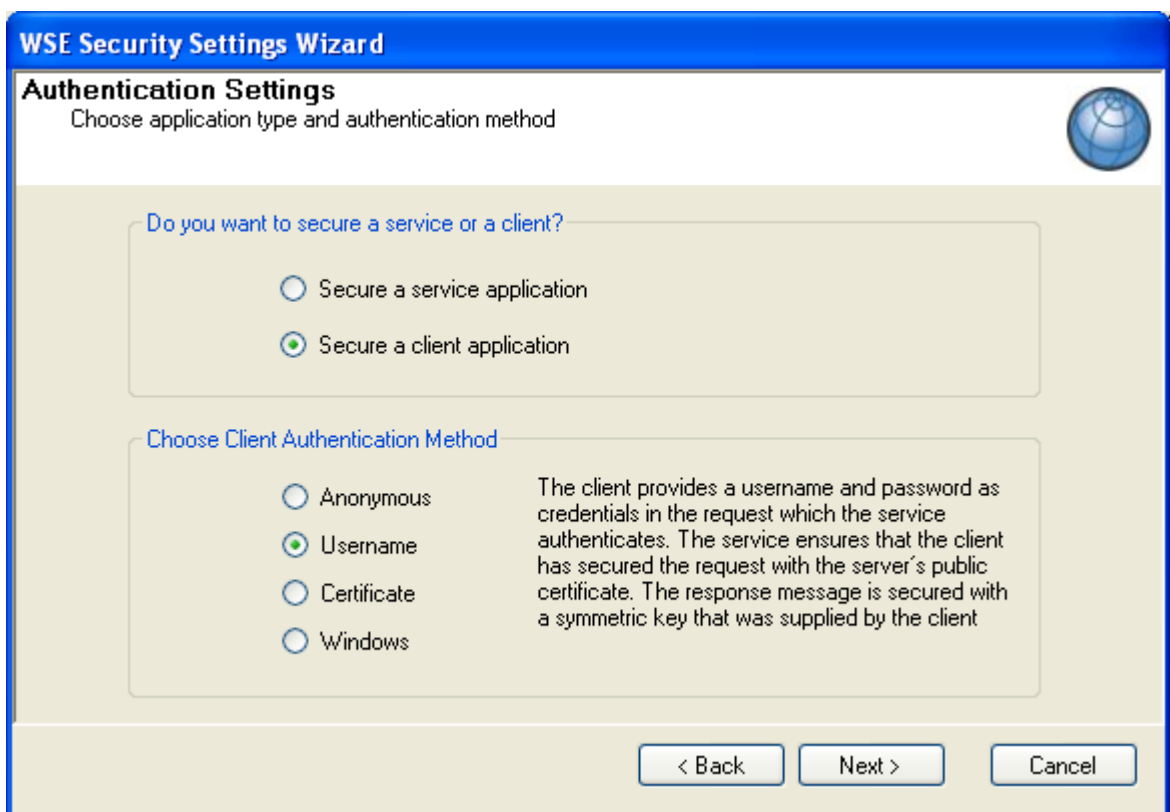
To create a policy file that will control the way the SOAP header is configured, click on the *Add...* button and enter the name for the policy:



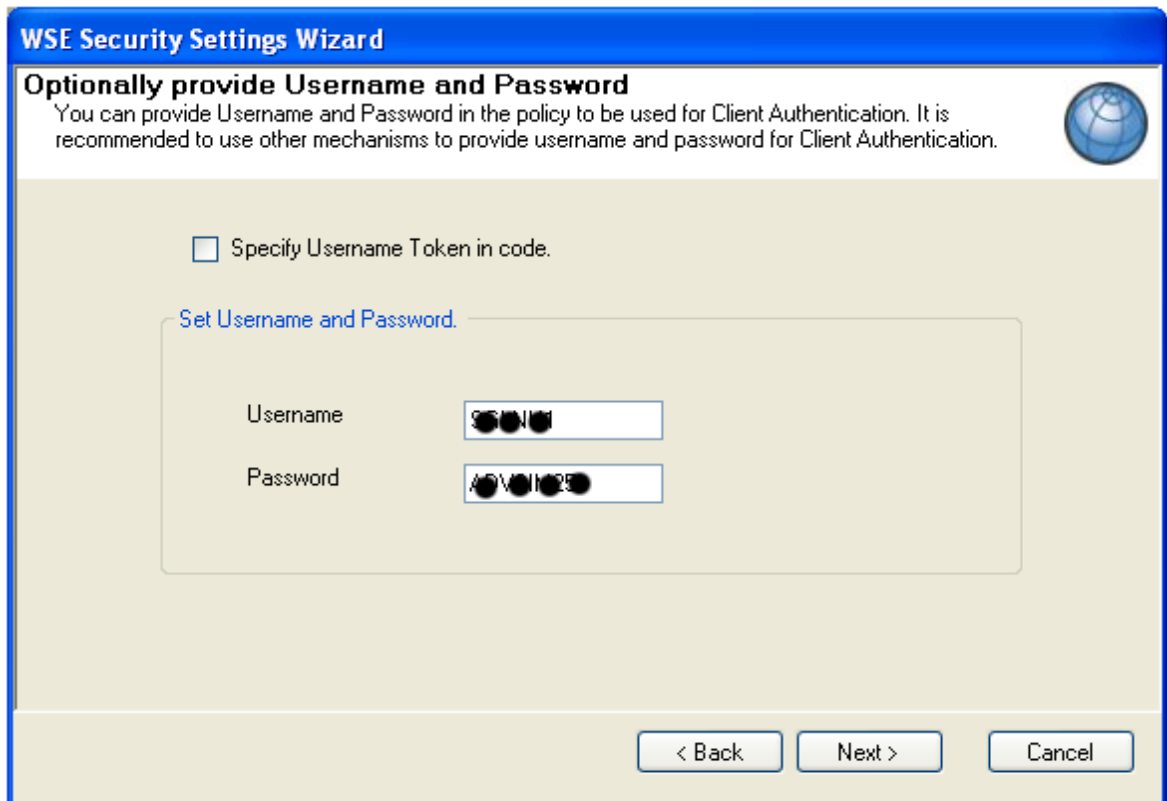
Click *Next* on the first page of the wizard:



Because the SLC only required a username and password, select the *Secure a client* and *Username* radio buttons:

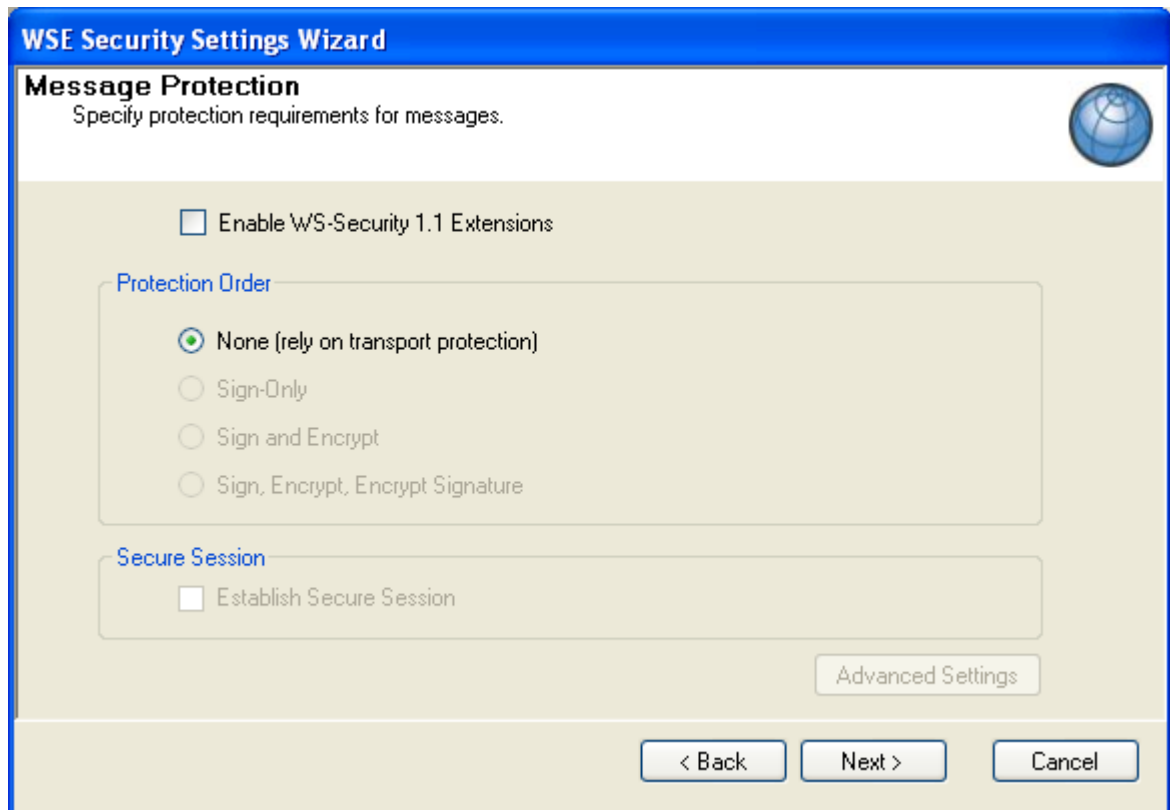


If you want the username and password to be in the configuration file, uncheck the *Specify Username Token in code* check box and enter the details in the text boxes provided. It is, of course, more secure to specify credentials in code.



The image shows a Windows-style dialog box titled "WSE Security Settings Wizard". The main heading is "Optionally provide Username and Password". Below this, a message states: "You can provide Username and Password in the policy to be used for Client Authentication. It is recommended to use other mechanisms to provide username and password for Client Authentication." There is a globe icon in the top right corner. A checkbox labeled "Specify Username Token in code." is present and is currently unchecked. Below the checkbox is a section titled "Set Username and Password." which contains two text input fields. The "Username" field contains the text "ADMIN" and the "Password" field contains the text "ADMIN123". At the bottom of the dialog are three buttons: "< Back", "Next >", and "Cancel".

Click *Next*, and on the Message Protection page uncheck the *Enable WS-Security 1.1 Extensions* check box. This will automatically select the *None (rely on transport protection)* radio button:



Click *Next* and then click *Finish* on the Create Security Settings page to create the policy cache file.

Back in the WSE 3.0 configuration tabbed dialog, confirm that the policy is listed in the list box.

The *TokenIssuing* page can be left at the default values.

On the *Diagnostics* tab, check the *Enable Message Trace* check box if you want to produce text files containing the output sent to the Web Service and the input that it returns. A full path is required in the *Input File* and *Output File* text boxes.

This can be useful in order to check that the messages being sent are correct (see *Trouble Shooting*, below).

The *Messaging* tab can be left at its default values.

Click *OK* to close the dialog. Confirm that your project now contains a policy cache configuration file (probably called *wse3policyCache.config*) and that your *app.config/web.config* file contains elements like the following:

```
<configuration>
  <configSections>
    <section name="microsoft.web.services3"
type="Microsoft.Web.Services3.Configuration.WebServicesConfiguration,
Microsoft.Web.Services3,
Version=3.0.0.0,
Culture=neutral,
```

```

    PublicKeyToken=31bf3856ad364e35" />
  </configSections>
  <microsoft.web.services3>
    <policy fileName="wse3policyCache.config" />
    <messaging>
      <mtom clientMode="Off" />
    </messaging>
    <security>
      <x509 verifyTrust="false" />
    </security>
    <diagnostics>
      <trace enabled="false"
        input="InputTrace.webinfo"
        output="OutputTrace.webinfo" />
    </diagnostics>
  </microsoft.web.services3>
</configuration>

```

In addition a policy file will also be added to your project, probably called `wse3policyCache.config` if you've left all the defaults, and looking something like this:

```

<policies xmlns="http://schemas.microsoft.com/wse/2005/06/policy">
  <extensions>
    <extension name="usernameOverTransportSecurity"
      type="Microsoft.Web.Services3.Design.UsernameOverTransportAssertion,
        Microsoft.Web.Services3,
        Version=3.0.0.0,
        Culture=neutral,
        PublicKeyToken=31bf3856ad364e35" />
    <extension name="username"
      type="Microsoft.Web.Services3.Design.UsernameTokenProvider,
        Microsoft.Web.Services3,
        Version=3.0.0.0,
        Culture=neutral,
        PublicKeyToken=31bf3856ad364e35" />
    <extension name="requireActionHeader"
      type="Microsoft.Web.Services3.Design.RequireActionHeaderAssertion,
        Microsoft.Web.Services3,
        Version=3.0.0.0,
        Culture=neutral,
        PublicKeyToken=31bf3856ad364e35" />
  </extensions>
  <policy name="credentials">
    <usernameOverTransportSecurity>
      <clientToken>
        <username username="*****" password="*****" />
      </clientToken>
    </usernameOverTransportSecurity>
    <requireActionHeader />
  </policy>
</policies>

```

Whilst these configurations are correct as far as they go, they do not produce a SOAP message acceptable to the SLC Web Service. The SOAP message produced is as follows:

```

<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:wsa="http://schemas.xmlsoap.org/ws/
2004/08/addressing" xmlns:wss="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
wss-wssecurity-secext-1.0.xsd" xmlns:wsu="http://docs.oasis-
open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd">
  <soap:Header>
    <wsa:Action>urn:getQualificationsList</wsa:Action>
    <wsa:MessageID>urn:uuid:dfedf7ad-8178-404e-9ec5-5ad1d6105b75</wsa:MessageID>
    <wsa:ReplyTo>
      <wsa:Address>http://schemas.xmlsoap.org/ws/2004/08/addressing/role/anon-
ymous</wsa:Address>
    </wsa:ReplyTo>
    <wsa:To>https://trainingsecure.heservices.slc.co.uk/heicoursesWS/services/Course
Services
    </wsa:To>
    <wsse:Security soap:mustUnderstand="1">
      <wsu:Timestamp wsu:Id="Timestamp-c03a31a7-a832-421f-b056-3ac2775c19e6">
        <wsu:Created>2008-07-08T13:54:14Z</wsu:Created>
        <wsu:Expires>2008-07-08T13:59:14Z</wsu:Expires>
      </wsu:Timestamp>
      <wsse:UsernameToken wsu:Id="SecurityToken-d7e57dc4-b9fb-4e14-
adf2-85a3f7c7a6a3">
        <wsse:Username>*****</wsse:Username>
        <wsse:Password Type="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
wss-username-token-profile-1.0#PasswordText">*****</wsse:Password>
        <wsse:Nonce>mlxh2CI+Mj8AfKcYcFdy1A==</wsse:Nonce>
        <wsu:Created>2008-07-08T13:54:14Z</wsu:Created>
      </wsse:UsernameToken>
    </wsse:Security>
  </soap:Header>
  <soap:Body>
    <qualificationsListRequest xmlns="http://www.slc.co.uk/course/schema/1.0">
      <schemaVersion xmlns="">1.0</schemaVersion>
    </qualificationsListRequest>
  </soap:Body>
</soap:Envelope>

```

There are several things to notice about this message:

- More namespaces are declared and they are declared on different elements
- The following elements are generated, but not required by the SLC:
 - <wsa:Action>
 - <wsa:MessageID>
 - <wsa:ReplyTo> and children
 - <wsa:To>
 - <wsu:Timestamp> and children
 - The <wsse:Username> children:
 - <wsse:Nonce>
 - <wsu:Created>

It turns out that there are a number of ways of modifying this output to make it acceptable to the SLC Web Service: however, the easiest is simply to delete the <Timestamp> element. If the <Timestamp> element is deleted, nothing else needs to change.

The final format required is therefore:

```

<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:wsa="http://schemas.xmlsoap.org/ws/
2004/08/addressing" xmlns:wss="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
wss-wssecurity-secext-1.0.xsd" xmlns:wsu="http://docs.oasis-
open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd">
  <soap:Header>
    <wsa:Action>urn:getQualificationsList</wsa:Action>

```

```

    <wsa:MessageID>urn:uuid:dfedf7ad-8178-404e-9ec5-5ad1d6105b75</wsa:MessageID>
    <wsa:ReplyTo>
<wsa:Address>http://schemas.xmlsoap.org/ws/2004/08/addressing/role/anonymous</wsa:Address>
    </wsa:ReplyTo>
<wsa:To>https://trainingsecure.heservices.slc.co.uk/heicoursesWS/services/CourseServices</wsa:To>
    <wsse:Security soap:mustUnderstand="1">
        <wsse:UsernameToken wsu:Id="SecurityToken-d7e57dc4-b9fb-4e14-
adf2-85a3f7c7a6a3">
            <wsse:Username>*****</wsse:Username>
            <wsse:Password Type="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
wss-username-token-profile-1.0#PasswordText">*****</wsse:Password>
            <wsse:Nonce>mlxh2CI+Mj8AfKcYcFdylA==</wsse:Nonce>
            <wsu:Created>2008-07-08T13:54:14Z</wsu:Created>
        </wsse:UsernameToken>
    </wsse:Security>
</soap:Header>
<soap:Body>
    <qualificationsListRequest xmlns="http://www.slc.co.uk/course/schema/1.0">
        <schemaVersion xmlns="">1.0</schemaVersion>
    </qualificationsListRequest>
</soap:Body>
</soap:Envelope>

```

6. Adding a custom PolicyAssertion

Note that there are some naming confusions to be overcome:

- The `CreateClientOutputFilter()` method of a custom policy assertion filters output from the client. It therefore creates an input filter (i.e. a filter on input to the server) and returns it.
- The `CreateClientInputFilter()` method of a custom policy assertion filters input to the client (i.e. output from the server). It therefore creates an output filter.

These appear to be only conventions, and are the ones used by Microsoft – so it's probably best to stick with them even though they are confusing.

6.1 Fixing the <Timestamp> problem

To get rid of the `<Timestamp>` element, a custom `PolicyAssertion` with an accompanying SOAP filter must be supplied. In this case a client output filter is what is needed – to filter the output from the client. The custom `PolicyAssertion` and the SOAP filter can both be defined in the same file, for convenience.

The `PolicyAssertion` creates a new SOAP Filter. The SOAP Filter contains an overridden `ProcessMessage()` method that looks for the `<Timestamp>` element and deletes it. The code for the `PolicyAssertion` is:

```

public class SlcHeiPolicyAssertion : PolicyAssertion
{
    public SlcHeiPolicyAssertion() : base()
    {
    }

    public override SoapFilter CreateClientOutputFilter(
        FilterCreationContext context)
    {
        return new SlcHeiInputSoapFilter();
    }

    public override SoapFilter CreateClientInputFilter(
        FilterCreationContext context)

```

```

    {
        return null;
    }

    public override SoapFilter CreateServiceInputFilter(
        FilterCreationContext context)
    {
        return null;
    }

    public override SoapFilter CreateServiceOutputFilter(
        FilterCreationContext context)
    {
        return null;
    }

    public override IEnumerable<KeyValuePair<string, Type>> GetExtensions()
    {
        return new KeyValuePair<string, Type>[] {
            new KeyValuePair<string, Type>(
                "SlcHeiPolicyAssertion",
                typeof(SlcHeiPolicyAssertion)) };
    }

    public override void WriteXml(XmlWriter writer)
    {
    }

    public override void ReadXml(XmlReader reader,
        IDictionary<string,
        Type> extensions)
    {
        if (reader == null) throw new ArgumentNullException("reader");

        bool isEmpty = reader.IsEmptyElement;
        reader.ReadStartElement("SlcHeiPolicyAssertion");
        if (!isEmpty) reader.ReadEndElement();
    }
}

```

Notice that the `ReadXml()` method looks for an element called `SlcHeiPolicyAssertion`. This is an arbitrary element name, but it must correspond to an (empty) element in the policy file. In this case, therefore, the element is added to the `<policy>` element in the policy file:

```
<SlcHeiPolicyAssertion />
```

The `<policy>` element, therefore, ends up looking like this:

```

<policy name="credentials">
  <usernameOverTransportSecurity>
    <clientToken>
      <username username="*****" password="*****" />
    </clientToken>
  </usernameOverTransportSecurity>
  <SlcHeiPolicyAssertion />
  <requireActionHeader />
</policy>

```

In addition, a new `<extension>` element must be added to the policy file, as follows:

```

<extension name="SlcHeiPolicyAssertion"
  type="SlcHeiDbWebServiceTest.SlcHeiPolicyAssertion,
  SlcHeiDbWebServiceTest" />

```

This element consists of the following parts:

- The name is the name of the element that will be found in the `<policy>` section – "SlcHeiPolicyAssertion" in this example.
- The type attribute consists of the fully qualified type name (namespace + class) of the policy assertion class, followed by a comma followed by the name of the assembly that contains the class (i.e. the name of the DLL file in which the class is defined without its extension). This allows the policy assertion class to be found at runtime.

Note that Visual Studio will complain and say that the new element is invalid according to the schema. You could fix this by adding it to the `##other` namespace for VS, but it's hardly worth the bother.

The code for the custom SOAP input filter is as follows:

```
class SlcHeiInputSoapFilter : SoapFilter
{
    public SlcHeiSoapFilter()
    {
    }

    public override SoapFilterResult ProcessMessage(SoapEnvelope envelope)
    {
        XmlElement eSecurity = null;
        if (envelope.Header != null)
        {
            foreach (XmlNode n in envelope.Header.ChildNodes)
            {
                if (n.LocalName == "Security")
                {
                    eSecurity = (XmlElement)n;
                    break;
                }
            }

            XmlElement eTimeStamp = null;
            foreach (XmlNode n in eSecurity.ChildNodes)
            {
                if (n.LocalName == "Timestamp")
                {
                    eTimeStamp = (XmlElement)n;
                    break;
                }
            }

            if (eTimeStamp != null)
            {
                eSecurity.RemoveChild(eTimeStamp);
            }
        }

        return SoapFilterResult.Continue;
    }
}
```

Obviously the class name is arbitrary.

The system works like this. The presence of the `<SlcHeiPolicyAssertion />` element causes the .NET Framework to look for a `PolicyAssertion` class of that name, as defined in the `<extension>` section. If one exists, it creates an instance and calls its methods in order to filter input and output. This works on both the client and the server side, but this document is concerned only with client-side aspects.

6.2 Fixing the courseCatalogResponse problem

The easiest way to solve this problem is to filter the output from the Web Service containing the defective elements. The same `PolicyAssertion` class is used, but this time an output filter is created and its `ProcessMessage()` method overridden.

The `PolicyAssertion` class is altered to instantiate an instance of the the output filter, as follows:

```
public override SoapFilter
    CreateClientInputFilter(FilterCreationContext context)
{
    return new SlcHeiOutputSoapFilter();
}
```

The output SOAP filter looks like this:

```
class SlcHeiOutputSoapFilter : SoapFilter
{
    public SlcHeiOutputSoapFilter() {}

    public override SoapFilterResult
        ProcessMessage(SoapEnvelope envelope)
    {
        XmlNameTable nameTable = envelope.NameTable;
        XmlNamespaceManager namespaceManager =
            new XmlNamespaceManager(nameTable);

        namespaceManager.AddNamespace(
            "soapenv",
            "http://schemas.xmlsoap.org/soap/envelope/");

        namespaceManager.AddNamespace(
            "ns3",
            "http://www.slc.co.uk/course/schema/1.0");

        XmlNodeList codes = envelope.SelectNodes(
            "soapenv:Envelope/soapenv:Body/ns3:courseCatalogResponse/catalog/institution/course
            /code", namespaceManager);

        if (codes.Count != 0)
        {
            foreach (XmlNode code in codes)
            {
                XmlAttribute attribute =
                    envelope.CreateAttribute(
                        "xmlns:ns1",
                        "http://www.w3.org/2000/xmlns/");

                attribute.Value = "http://www.slc.co.uk/course/types/1.0";

                code.Attributes.Append(attribute);
            }
        }

        return SoapFilterResult.Continue;
    } // ProcessMessage()
} // SlcHeiOutputSoapFilter class
```

7.Troubleshooting

As mentioned previously, the WSE 3.0 configuration can be set to dump the output and input messages to text files. However Visual Studio provides no mechanism by which such files could be altered manually and resubmitted to test the Web Server's reaction to various changes in the submission.

Such facilities are available from the Test-Driven Programming tool, *soapUI*. This tool allows developers to copy and paste messages into editors, and then submit the edited messages. This was the mechanism used to discover the SOAP message formats acceptable to the SLC HEI Course Database Web Service: the failed message was pasted into a *soapUI* editor, modified in some way (e.g. deleting the <Timespan> element) and the message re-submitted.

The *soapUI* installation executable is available from:

[soapUI-2.0.2-installer.exe](#)

It is free software, and requires no license. An enhanced, paid-for, version is available if required.