Web Services Security White Paper

VERSION 2018

jade

Copyright©2018 Jade Software Corporation Limited. All rights reserved.

Jade Software Corporation Limited cannot accept any financial or other responsibilities that may be the result of your use of this information or software material, including direct, indirect, special or consequential damages, or loss of profits. There are no warranties extended or granted by this document or software material.

You should be very careful to ensure that the use of this software material and/or information complies with the laws, rules, and regulations of the jurisdictions with respect to which it is used. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of Jade Software Corporation Limited.

The information contained herein is subject to change without notice. Revisions may be issued to advise of such changes and/or additions.

Copyright © 2018 Jade Software Corporation Limited.

All rights reserved.

JADE is a trademark of Jade Software Corporation Limited. All trade names referenced are the service mark, trademark, or registered trademark of the respective manufacturer.

For details about other licensing agreements for third-party products, you must read the JADE **ReadMe.txt** file.

Contents

Contents iii

Web Services Security 4

UsernameToken Profile 4
Namespaces 4
User Names and Passwords 5
XML Syntax 5
Examples 6
Security Class Library 7
Security Classes 7
Jadwssec Class 7
JadeSecurityToken Class 8
JadeUsernameToken Class 10
JadeWSAddressingHeader Class 13
JadeWSTimestampHeader Class 14
JadeWebServicesSecurity Class 15
Example of Use 16
Import the Library 16
Create a Package 19
Import the Package 20
Create a Web Service 21
Consuming the Web Service 21
Generating Security Headers (Client) 23
Processing Security Headers (Service) 24
Sample SOAP Message 29

Web Services Security

This white paper discusses the use of a .NET class library to implement Web services security in JADE. The initial implementation of this class library supports only the use of **UsernameToken** profile.

The implementation follows the specifications as set out by the OASIS Standard Specification (1 February 2006), Web Services Security UsernameToken Profile 1.1. This specification describes how a Web service consumer can supply a **UsernameToken** as a means of identifying the requestor by "username" and optionally using a password (or shared secret, or password equivalent) to authenticate that identity to the Web service provider.

For more details, see the following subsections.

UsernameToken Profile

This section contains the following topics.

- Namespaces
- User Names and Passwords
- XML Syntax
- Examples

Namespaces

Namespace URIs (of the general form "some-URI") represents some application-dependent or context-dependent URI as defined in RFC 3986 [URI]. This specification is designed to work with the general SOAP [SOAP11, SOAP12] message structure and message processing model, and should be applicable to any version of SOAP. The current SOAP 1.1 namespace URI is used here to provide detailed examples.

The namespaces used in this document are shown in the following table.

Prefix	Namespace
S11	http://schemas.xmlsoap.org/soap/envelope/
S12	http://www.w3.org/2003/05/soap-envelope
wsse	http://docs.oasis-open.org/wss/2004/01/oasis-200401-wsswssecurity- secext-1.0.xsd
wsse11	http://docs.oasis-open.org/wss/oasis-wss-wssecurity-secext-1.1.xsd
wsu	http://docs.oasis-open.org/wss/2004/01/oasis-200401-wsswssecurity- utility-1.0.xsd

The *Oasis Web Services Security UsernameToken Profile 1.0* document that provides information about the **#PasswordDigest**, **#PasswordText**, **#UsernameToken** URI fragments referred to in this document is available at:

http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0.pdf

User Names and Passwords

The **<wsse:UsernameToken>** element is introduced in SOAP Message Security documents as a way of providing a user name.

Within a **<wsse:UsernameToken>** element, a **<wsse:Password>** element can be specified. Passwords of type **PasswordText** and **PasswordDigest** are not limited to actual passwords, although this is a common case. Any password equivalent such as a derived password or S/KEY (one-time password) can be used. Having a type of **PasswordText** merely implies that the information held in the password is "in the clear", as opposed to holding a "digest" of the information. For example, if a server does not have access to the clear text of a password but it does have the hash, the hash is considered a *password equivalent* and can be used anywhere where a password is indicated in this specification.

Passwords of type **PasswordDigest** are defined as being the Base64-encoded, SHA-1 hash value, of the UTF8-encoded password (or equivalent). However, unless this digested password is sent on a secured channel or the token is encrypted, the digest offers no real additional security over use of **wsse:PasswordText**.

Two optional elements are introduced in the **<wsse:UsernameToken>** element to provide a counter-measure for replay attacks: **<wsse:Nonce>** and **<wsu:Created>**. A nonce is a random value that the sender creates to include in each **UsernameToken** that it sends. Although using a nonce is an effective counter-measure against replay attacks, it requires a server to maintain a cache of used nonces, consuming server resources. Combining a nonce with a creation timestamp has the advantage of allowing a server to limit the cache of nonces to a "freshness" time period, establishing an upper bound on resource requirements. If either or both of **<wsse:Nonce>** and **<wsu:Created>** are present, they *must* be included in the digest value, as follows.

```
Password_Digest = Base64 ( SHA-1 ( nonce + created + password ) )
```

This concatenates the nonce, creation timestamp, and the password (or shared secret or password equivalent), digests the combination using the SHA-1 hash algorithm, then includes the Base64 encoding of that result as the password (digest). This helps to obscure the password and offers a basis for preventing replay attacks.

For Web service providers to effectively thwart replay attacks, three counter measures are recommended. It is recommended that:

- Web service providers reject any **UsernameToken** *not* using both nonce and creation timestamps.
- Web service providers provide a timestamp "freshness" limitation, and that any **UsernameToken** with "stale" timestamps be rejected.

  As a guideline, a value of five minutes can be used as a minimum to detect, and thus reject, replays.
- Used nonces be cached for a period at least as long as the above timestamp freshness limitation period, and that **UsernameTokens** with nonces that have already been used (and are thus in the cache) be rejected.

Note that **PasswordDigest** can be used only if the plain text password (or password equivalent) is available to both the requestor and the recipient.

XML Syntax

The following illustrates the XML syntax of this element.

```
<wsse:UsernameToken wsu:Id="Example-1">
    <wsse:Username> ... </wsse:Username>
    <wsse:Password Type="..."> ... </wsse:Password>
    <wsse:Nonce EncodingType="..."> ... </wsse:Nonce>
    <wsu:Created> ... </wsu:Created>
</wsse:UsernameToken>
```

The following describes the attributes and elements listed in the above example.

- **wsse:Username**

  Required, specifies a user name.

- **wsse:Password**

  Optional, provides password information. This element should be passed only when using a secure transport (such as https) or if the token itself is encrypted.

- **Password Type**

  This optional attribute specifies the type of password and can take one of two values.

- **PasswordText** (default)

  The actual password for the user name, a password hash, or derived password. This type should be used when hashed password equivalents do not rely on nonce or creation timestamps, or a digest algorithm other than SHA1 is used.

- **PasswordDigest**

  The digest of the password using the above password algorithm.

- **wsse:Nonce**

  Optional, specifies a random nonce. Each message that includes a nonce must provide a unique nonce value.

- **Encoding Type**

  This optional attribute specifies the encoding type of the nonce. If not specified, Base64 encoding is used.

- **wsu:Created**

  This optional element specifies a timestamp used to indicate the creation time.

Examples

The following example illustrates the use of this element. In this example, the password is sent as clear text and therefore this message should be sent over a confidential channel.

```
<S11:Envelope xmlns:S11="..." xmlns:wsse="...">
    <S11:Header>
        ...
        <wsse:Security>
            <wsse:UsernameToken>
                <wsse:Username>wilbur</wsse:Username>
                <wsse:Password>cheese</wsse:Password>
            </wsse:UsernameToken>
        </wsse:Security>
        ...
    </S11:Header>
    ...
</S11:Envelope>
```

The following example illustrates using a digest of the password along with a nonce and a creation timestamp.

```
<S11:Envelope xmlns:S11="..." xmlns:wsse="...">
    <S11:Header>
        ...
        <wsse:Security>
            <wsse:UsernameToken>
                <wsse:Username>wilbur</wsse:Username>
                <wsse:Password Type="#PasswordDigest">
                    weYI3nXd8LjMNVksCKFV8t3rgHh3Rw==
                </wsse:Password>
                <wsse:Nonce>WScqanjCEAC4mQoBE07sAQ==</wsse:Nonce>
                <wsu:Created>2003-07-16T01:24:32Z</wsu:Created>
            </wsse:UsernameToken>
        </wsse:Security>
        ...
    </S11:Header>
    ...
</S11:Envelope>
```

Security Class Library

The class library is called **jadwssec** and is supplied as a .NET assembly. In order to use it, you need to import this library into your system.

As the library is likely to be required by more than one schema, you may want to import it into one schema, create a package with the required classes, and then import this package into the schemas that need this feature.

For more details, see the following subsections.

Security Classes

The import generates six classes. Note that these are the default names, which you can change on import. The discussion in the following subsections is based on the default names.

For more details, see the following subsections.

Jadwssec Class

The **Jadwssec** class is an abstract class that groups together all of the generated .NET classes corresponding to the assembly to which they belong. This class also holds the public constants and enums that are defined in the library. Note that enums are generated as class constants. The following constants are defined in this class.

Name	Type	Value	Description
AssemblyName	String	jadwssec, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null	Assembly Details
EncodingType_Base64Binary	Integer	0	Nonce Encoding Base64 (default)
EncodingType_HexBinary	Integer	1	Nonce Encoding Binary
PasswordOption_SendHashed	Integer	0	Password Type Digest (default)

Name	Type	Value	Description
PasswordOption_SendNone	Integer	1	Password Type None
PasswordOption_SendPlainText	Integer	2	Password Type Plain Text
ProtectionType_Encrypt	Integer	2	Encrypt Body
ProtectionType_EncryptAndSign	Integer	4	Encrypt and Sign Body
ProtectionType_None	Integer	0	No Encryption or Signing (default)
ProtectionType_Sign	Integer	1	Sign Body
ProtectionType_SignAndEncrypt	Integer	3	Sign and Encrypt Body

JadeSecurityToken Class

The **JadeSecurityToken** class is the abstract superclass for all security token classes. The following properties are defined in this class.

Name	Type	Description
clearPassword	String	The clear text password to be used for signature and encrypting messages
protectionOrder	Integer	Message protection type, which must be one of the **ProtectionType** constants defined in the constants table above

The following methods are defined for this class.

decryptXml

This method is called to decrypt an encrypted SOAP message.

Parameters

Name	Type	Description
Xml	StringUtf8	The SOAP message to decrypt

Returns

A **StringUtf8** string representing the decrypted SOAP message.

Remarks

Only the **<body>** of the SOAP message is decrypted.

Note that all strings passed to and returned from the class library are UTF-8 strings.

Example

```
incomingMessage := unt.decryptXml(incomingMessage.StringUtf8).String;
```

encryptXml

This method is called to encrypt a SOAP message.

Parameters

Name	Type	Description
Xml	StringUtf8	The SOAP message to encrypt

Returns

A **StringUtf8** string representing the decrypted SOAP message.

Remarks

Only the **<body>** of the SOAP message is encrypted. This routine does not generate **<EncryptedKey>** tags nor does it handle multiple **<EncryptedData>** tags in the **<body>**.

Note that all strings passed to and returned from the class library are UTF-8 strings.

Example

```
outString := unt.encryptXml(inString);
```

getXml

This method is called to serialize the security token into XML. This is an abstract method. Its implementation is token-dependent.

Parameters

Name	Type	Description
Xml	StringUtf8	The SOAP message to which to add the XML

Returns

A **StringUtf8** string representing the SOAP message with the serialized security token.

Remarks

Only the **<body>** of the SOAP message is encrypted. This routine does not generate **<EncryptedKey>** tags nor does it handle multiple **<EncryptedData>** tags in the **<body>**.

Note that all strings passed to and returned from the class library are UTF-8 strings.

Example

```
outString := unt.getXml(inString);
```

signXml

This method is called to sign a SOAP message. The routine will sign the **<body>** tag. In addition, it will also sign the addressing and timestamp tags, if present.

Parameters

Name	Type	Description
Xml	StringUtf8	The SOAP message to sign

Returns

A **StringUtf8** string representing the signed SOAP message.

Remarks

Note that all strings passed to and returned from the class library are UTF-8 strings.

Example

```
outString := unt.signXml(inString);
```

verifySignature

This method is called to verify the signature of a SOAP message. An exception is raised if the signature verification fails.

Parameters

Name	Type	Description
Xml	StringUtf8	The SOAP message to verify

Returns

Nothing. An exception is raised if signature verification fails.

Remarks

Note that all strings passed to and returned from the class library are UTF-8 strings.

Example

```
unt.verifySignature(inString);
```

JadeUsernameToken Class

The **JadeUsernameToken** class represents the UserNameToken profile. The following methods are defined for this class.

createDotNetObject_1

This method is used to create a **JadeUsernameToken** instance.

Parameters

Name	Type	Description
username	StringUtf8	The SOAP message to verify.
password	StringUtf8	The clear password to use for the verification.

<table>
<tr><th>Name</th><th>Type</th><th>Description</th></tr>
<tr><td>passType</td><td>StringUtf8</td><td>Specify the password type, which can be one of the following.
■ PasswordOption_SendHashed
■ PasswordOption_SendPlain
■ PasswordOption_SendNone
Default is PasswordOption_SendHashed.</td></tr>
<tr><td>encType</td><td>StringUtf8</td><td>Specify the encoding type, which can be one of the following.
■ EncodingType_Base64Binary
■ EncodingType_HexBinary
Default is EncodingType_Base64Binary.</td></tr>
</table>

Returns

Nothing.

Remarks

Note that all strings passed to and returned from the class library are UTF-8 strings.

Example

```
vars
    unt: JadeUsernameToken;
begin
    create unt;
    unt.createDotNetObject_1('wilbur',
                             'password',
                             unt.PasswordOption_SendHashed,
                             0);
```

getPassword

This method will return the password on a **JadeUsernameToken** instance.

Parameters

None.

Returns

The password associated with the **JadeUsernameToken** instance.

Remarks

Password can be null, plain text, or hashed.

Example

```
vars
    pword: StringUtf8;
begin
```

```
        // unt is an existing JadeUserNameToken instance
        pword := unt.getPassword();
```

getUsername

This method will return the user name on a **JadeUsernameToken** instance.

Parameters

None.

Returns

The user name associated with the **JadeUsernameToken** instance.

Example

```
    vars
        user: StringUtf8;
    begin
        // unt is an existing JadeUserNameToken instance
        user := unt.getUsername();
```

getXml

This method is called to serialize a **JadeUserNameToken** instance into XML.

Parameters

Name	Type	Description
xml	StringUtf8	The SOAP message to which to add the XML

Returns

A **StringUtf8** string representing the SOAP message with the serialized user name token embedded in the supplied string.

Remarks

The **<Header>** and **<Security>** tags are also generated if they are not present in the input string.

Example

```
    vars
        user: StringUtf8;
    begin
        // unt is an existing JadeUserNameToken instance
        utString := unt.getXml(inString);
```

validatePassword

This method is used to validate the password that was in the SOAP message or the XML string.

Parameters

None.

Returns

A Boolean. A value of **true** indicates that the supplied clear password matches the incoming password and **false** if they do not match.

Remarks

If the incoming password is hashed, the supplied password is hashed with the nonce and creation timestamp from the incoming message before the values are compared.

Example

```
vars
    success: Boolean;
begin
    // unt is an existing JadeUserNameToken instance
    unt.clearPassword := "password";
    success := unt.validatePassword();
```

JadeWSAddressingHeader Class

The **JadeWSAddressingHeader** class is used to define the addressing information based on the WS-Addressing specification. This class has the following properties, which are all of type **StringUtf8**.

Name	Description
action	Represents the **<Action>** tag. An identifier that uniquely (and opaquely) identifies the semantics implied by this message. Required. The general form of an action URI is as follows. `[target namespace]/[port type name]/[input/output name]`
sendTo	Represents the **<To>** tag. This element provides the value of the destination URL. Required.
messageID	Read-only property that is a generated global unique id (GUID).
relatesTo	Required in the response message only, and should have the value of the **<MessageID>** tag from the request message.
replyTo	Read-only property. Value is a constant and it is always: `http://schemas.xmlsoap.org/ws/2004/08/addressing/role/anonymous`

The following method is defined for this class.

getXml

This method is called to serialize the addressing header into XML.

Parameters

Name	Type	Description
xml	StringUtf8	The SOAP message to which to add the XML

Returns

A **StringUtf8** string representing the SOAP message with the serialized addressing header.

Example

```
outString := addr.getXml(inString);
```

JadeWSTimestampHeader Class

The **JadeWSTimestampHeader** class is used to define the timestamp security information in the Security section of the SOAP message.

The following properties are defined for this class.

Name	Type	Description
created	TimeStamp	Read-only property that defines the creation time of the message.
expires	TimeStamp	Read-only property that defines the expiry time. This value is obtained by adding the seconds to timeout to the created time.
secondsToTimeout	Integer	Sets the expiry time based on this value. The number of seconds defined by this property is added to the creation time. Defaults to 300 seconds.

The following methods are defined for this class.

getXml

This method is called to serialize the timestamp security information into XML.

Parameters

Name	Type	Description
xml	StringUtf8	The SOAP message to which to add the XML

Returns

A **StringUtf8** string representing the SOAP message with the serialized timestamp information.

Example

```
outString := ts.getXml(inString);
```

validateTimestamp

This method is used to validate the timestamp that was in the SOAP message or the XML string. An exception is raised if the timestamp has expired.

Parameters

None.

Returns

Nothing. An exception is raised if validation fails.

Example

```
begin
    // ts is an existing WSTimeStampHeader instance
    ts.validateTimestamp();
end;
```

JadeWebServicesSecurity Class

The **JadeWebServicesSecurity** class is used to obtain the security tokens defined in an incoming SOAP message.

The following properties defined for this class are populated with values from the message.

Name	Type	Description
addressing	JadeWSAddressingHeader	Read-only property that contains the addressing information if present in the message.
creationTimeStamp	JadeWSTimestampHeader	Read-only property that contains the timestamp information if present in the message.
isEncrypted	Boolean	Read-only property is set to **true** if the **<EncryptedData>** tag is present in the message.
isSigned	Boolean	Read-only property is set to **true** if the **<Signature>** tag is present in the message.
usernameToken	JadeUsernameToken	Read-only property that contains the user name token if present in the message.

The following method is defined for this class.

getTokens

This method is called to deserialize the XML string parameter into user name token, addressing, and timestamp security information, and sets the signature and encryption status.

Parameters

Name	Type	Description
Xml	StringUtf8	The SOAP message to process for header information

Returns

A Boolean, set to **true** if there is header information; otherwise **false**. If header information is present, the properties are set to the appropriate value.

Example

```
success := ts.getTokens(inString);
```

Example of Use

In this example, we will import the class library into a schema called **WebServiceUtilitiesSchema** and from this schema, export the classes that make up this class library in a package called **UserNameTokenSecurityProfile**.

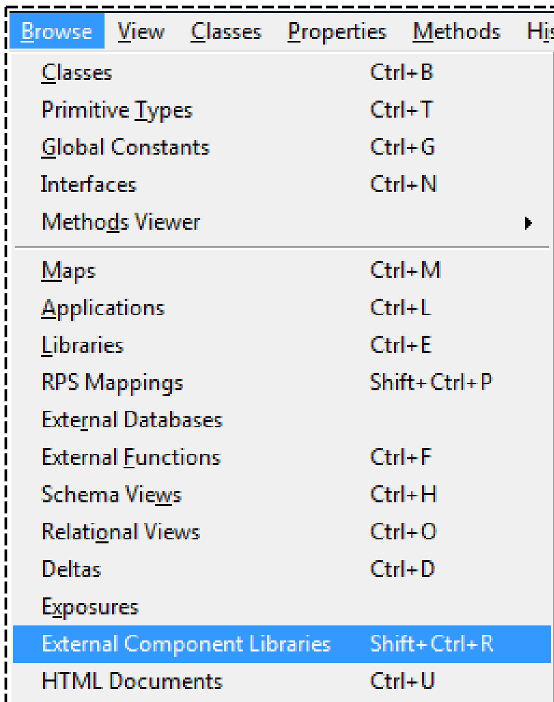
We will then import this package into a Web service provider schema called **CalculatorServices** and to a Web service consumer schema called **CalculatorServicesClient**. This consumer schema will import the WSDL generated by the **CalculatorServices** Web service.

For more details, see the following subsections.

Import the Library

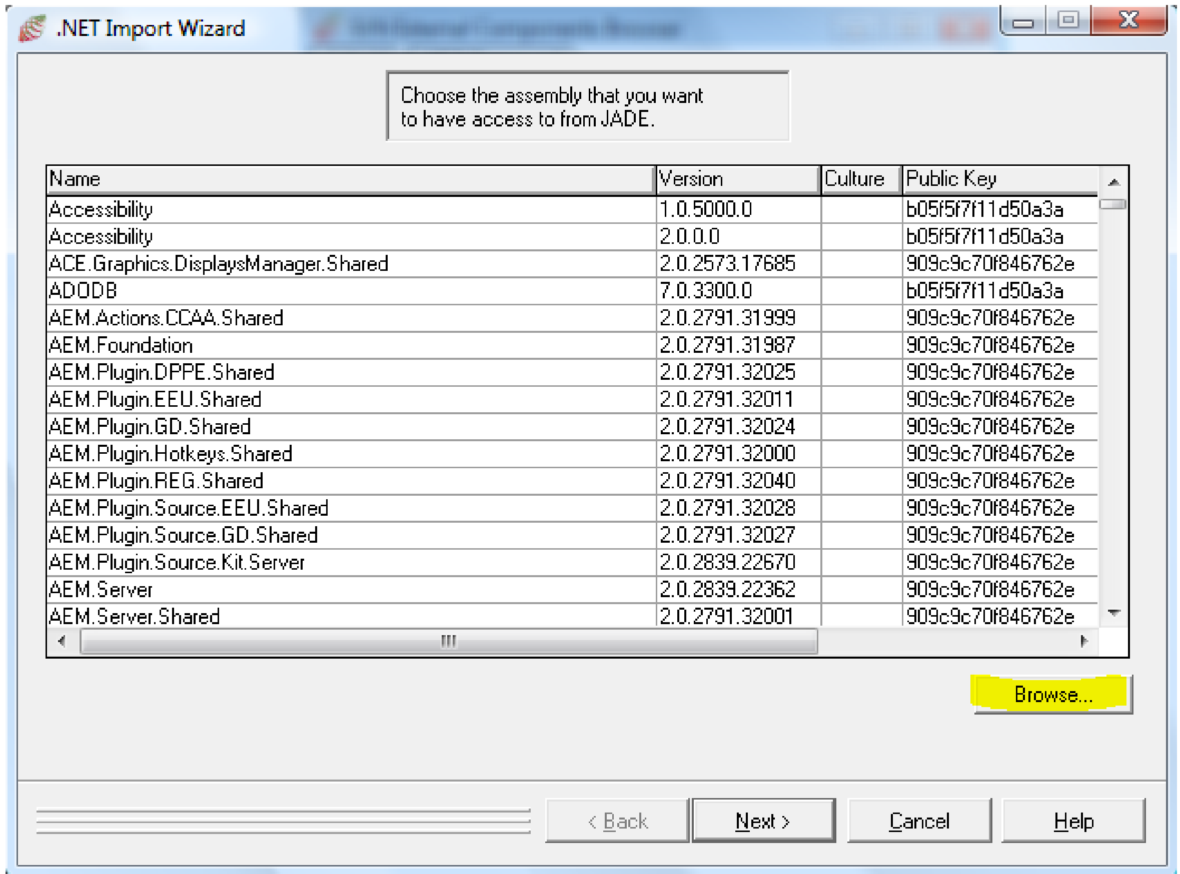
Create a schema called **WebServiceUtilitiesSchema** and then import the library into this schema. The name of the library is **jadwssec.dll**.

To import the library, select the **External Component Libraries** menu item from the Browse menu.



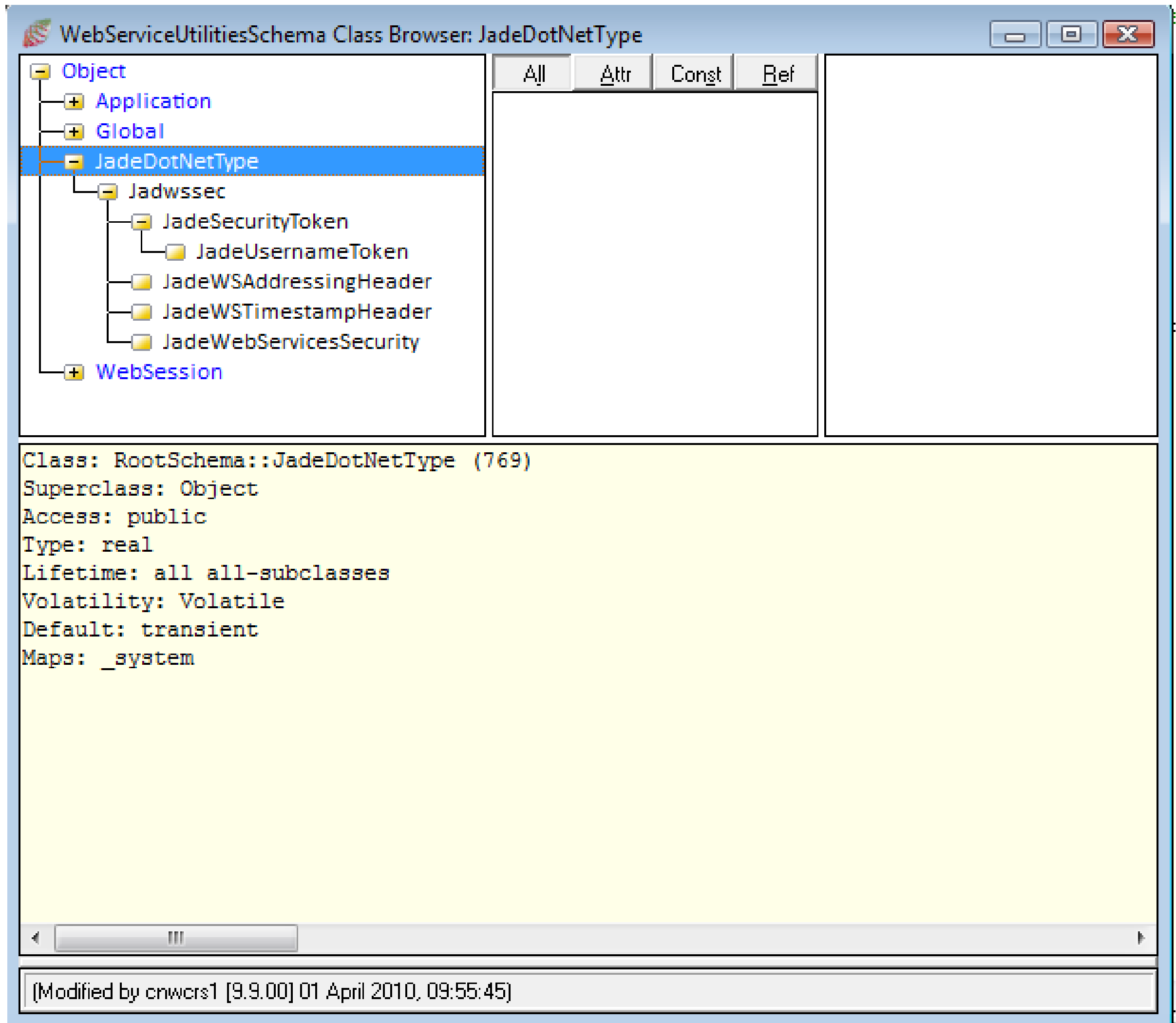
This will then display the External Components Browser. Make sure that the .NET framework tab is selected, right-click, and then select the **Import** option.

This will display the .NET Import Wizard, shown in the following image.



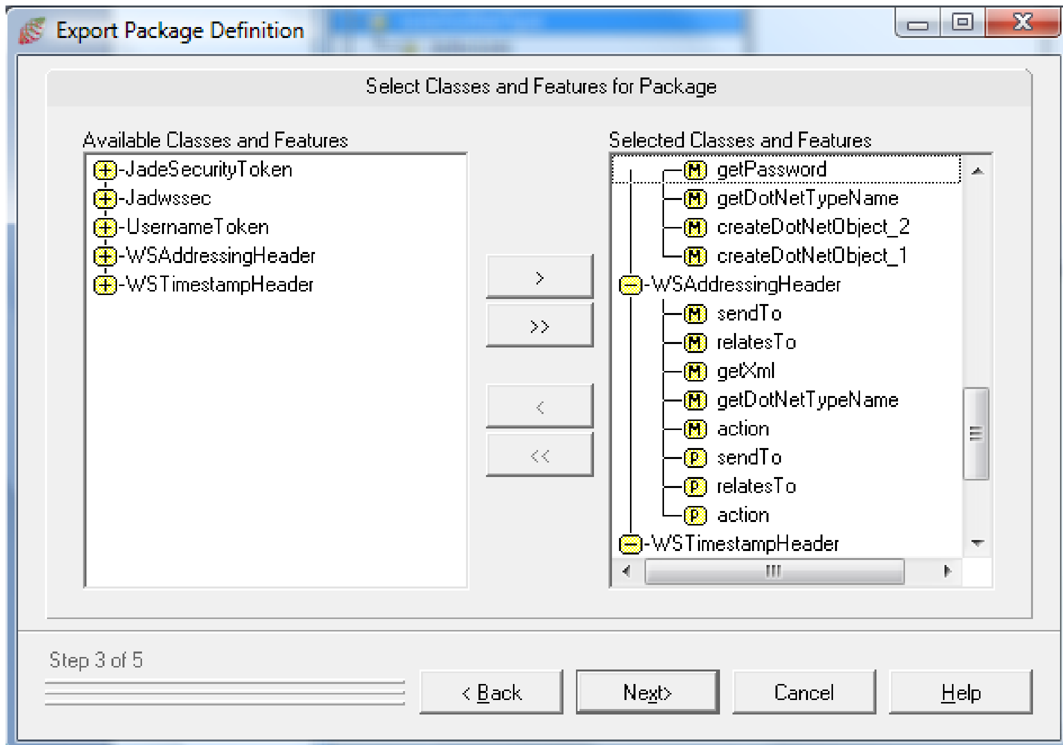
Select the **Browse** button, change the directory to your JADE installation directory, select the **jadwssec.dll** from the list, and then go through the import process. Once this is complete, open a Class Browser.

The following classes should now be displayed in the Class List.



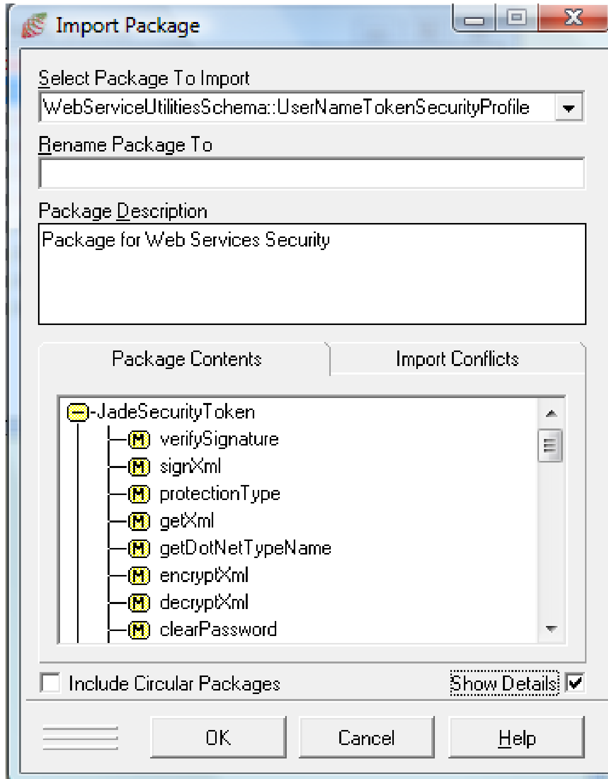
Create a Package

Create an export package called **UserNameTokenSecurityProfile** and then select all of the classes, constants, properties, and methods that were imported from the library, as shown in the following image.

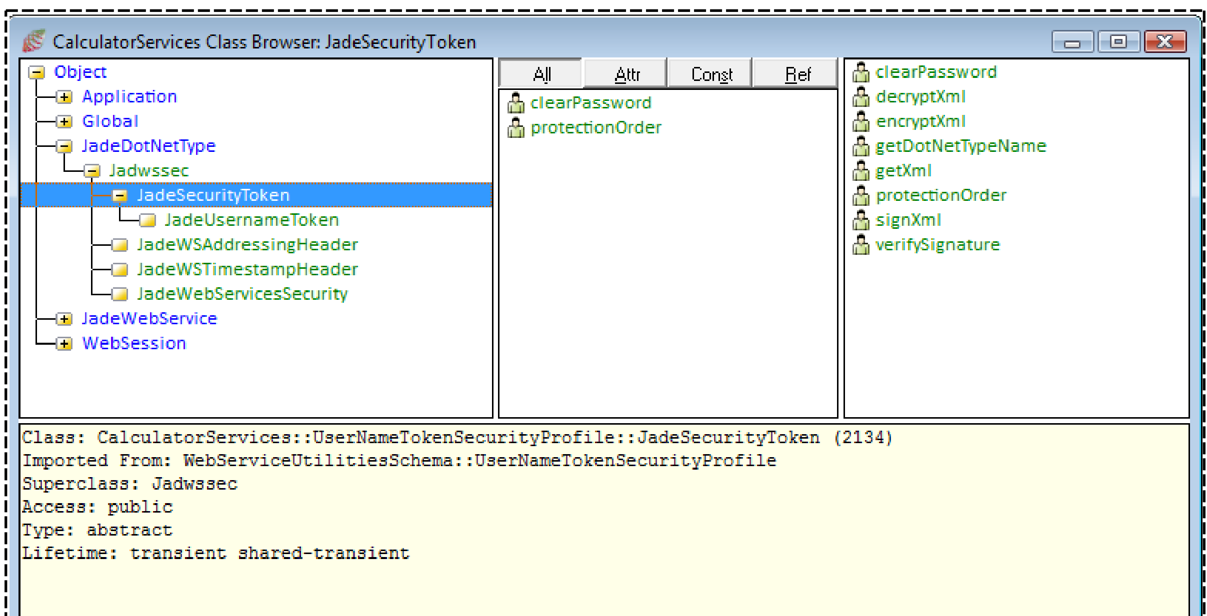


Import the Package

Create a new schema called **CalculatorServices** and then import the **UserNameTokenSecurityProfile** package into this schema.



The Class Browser will then look similar to the example in the following image.

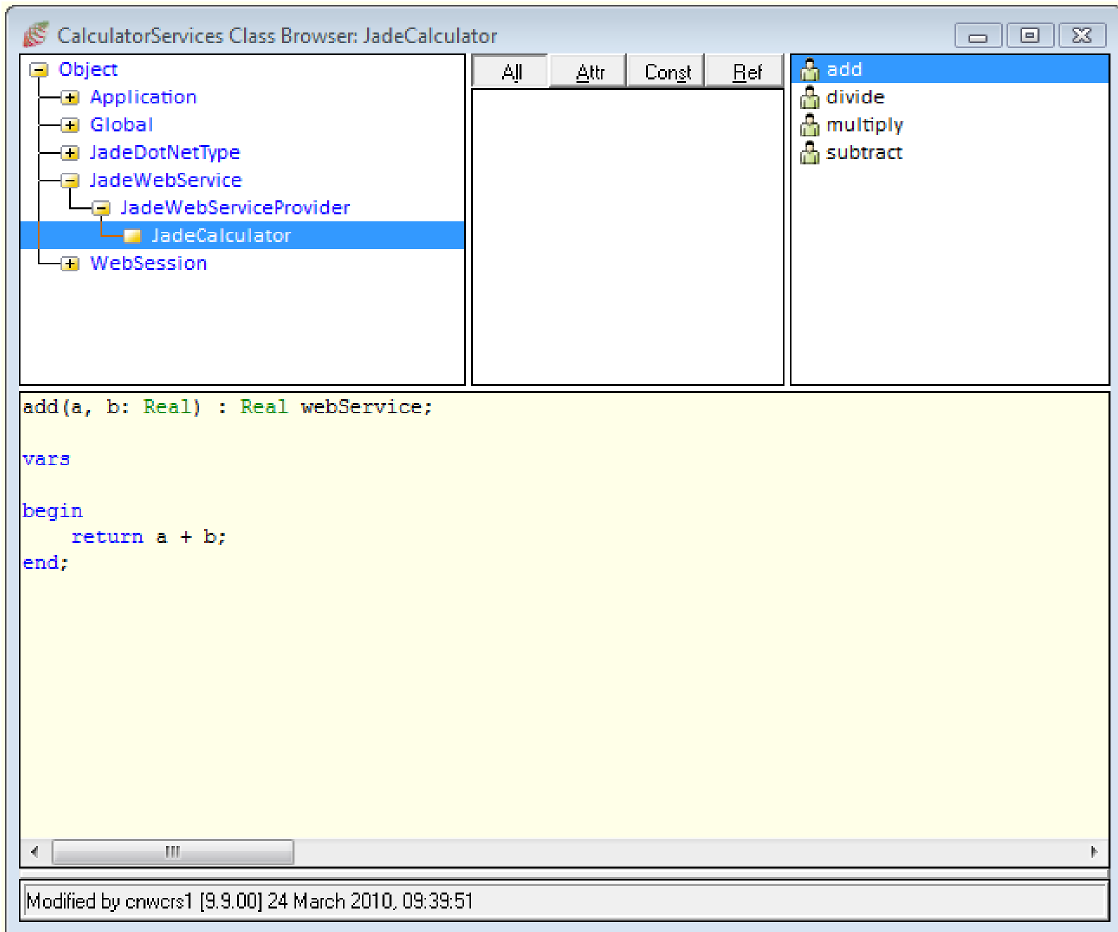


Create another schema called **CalculatorServicesClient** and then repeat the package import into this schema.

Note that normally the Web service and the Web service client will not be in the same system, so you have to repeat the exercise of importing the library for both systems. In this case, you may or may not want to use packages.

Create a Web Service

In this example, create a **Calculator** Web service class, as follows.



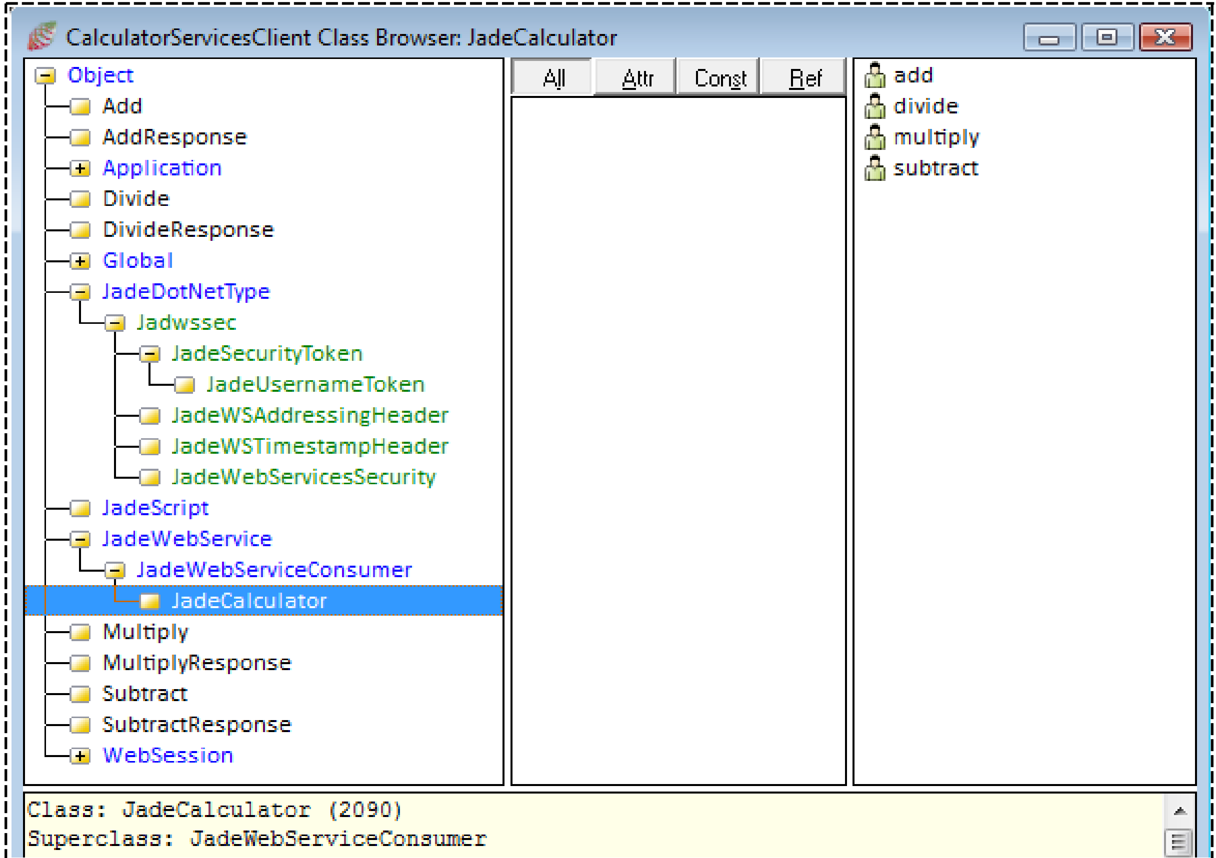
Set up the Web service by defining the exposure list, setting up the application, and then generating the WSDL. This WSDL will be imported into the **CalculatorWebServicesClient** schema. Set the required **jadehttp.ini** setting, virtual directory, and so on, as required. For details, see the *Web Services* and *Web Services Tips and Techniques* white papers.

We will need to set up the provider so that it can process the incoming message that has security headers, but before we do this, we will set up the consumer.

Consuming the Web Service

We now import the Web service into the **CalculatorWebServicesClient** using the Web Service Consumer Browser.

At the end of the import process, the Class Browser for this schema will have the following classes shown in the following image.



We will now define a **JadeScript** method that will set up the required security tokens and call the Web service. The method shown in the following image does this.

```
 1 testCalculator();
 2
 3 vars
 4     webService:          JadeCalculator;
 5     addRequest:          Add;
 6 begin
 7     // set up the web service and the parameters for the add method call
 8     create webService;
 9     create addRequest;
10     addRequest.a := 15;
11     addRequest.b := 20;
12
13     // now call the web service
14     write webService.add(addRequest).addResult;
15
16 epilog
17     delete webService;
18     delete addRequest;
19 end;
```

In this method:

- Lines 4 and 5 declare the variables required for this call.
- Lines 8 through 11 create the Web service consumer instance and set up the parameters for the Web service call.
- Line 14 makes the call to the Web service.
- Lines 16 and 17 delete the transient objects.

Generating Security Headers (Client)

The Web service client needs to know what headers are expected by the Web service. There is no WS-Security Policy information defined in the imported WSDL. In order to set up the security headers to be sent by the Web service client, we need to re-implement the **invoke** method on the **JadeCalculator** class. In the following example, we are going to set up addressing, timestamp, and hashed user name token, and we are also going to sign and encrypt the message.

The following method shows how to achieve this.

```
 1 invoke(inputMessage: String): String updating;
 2
 3 vars
 4     usernameToken:      JadeUsernameToken;
 5     wsAddress:          JadeWSAddressingHeader;
 6     wsTimestamp:        JadeWSTimestampHeader;
 7     str:                StringUtf8;
 8
 9 begin
10     // add addressing information
11     create wsAddress;
12     wsAddress.action := getSoapAction('add').StringUtf8;
13     wsAddress.sendTo := getEndpointURL.StringUtf8;
14     str := wsAddress.getXml(inputMessage.StringUtf8);
15
16     // add the timestamp header and set the expiry time to 1000
17     create wsTimestamp;
18     wsTimestamp.secondsToTimeout := 1000;
19     str := wsTimestamp.getXml(str);
20
21     // set up the user name token we will use the hased password
22     create usernameToken;
23     usernameToken.createDotNetObject_1("wilbur", "password", usernameToken.PasswordOption_SendHashed,
24                                        usernameToken.EncodingType_Base64Binary);
25
26     usernameToken.protectionOrder := Jadwssec.ProtectionType_SignAndEncrypt;
27     usernameToken.clearPassword := "password"; //password for signing end encryption
28
29     str := usernameToken.getXml(str);
30
31     // now send the message by calling the superclass method and wait for response
32
33     return inheritMethod(str.String);
34
35 epilog
36     delete usernameToken;
37     delete wsAddress;
38     delete wsTimestamp;
39 end;
```

The parameter to this method is the SOAP message that is to be sent to the Web service. What we need to do here is to insert the security headers into this message before it is sent.

- Lines 4 through 8 declare the required local variables.
- Lines 11 through 14 set up the WS-Addressing header and call the **getXml** method to insert the header into

the message.

- Lines 17 through 19 create the timestamp security header and call the **getXml** method to insert this header into the message. In this example, the expiry time is set to be 1000 seconds after the creation time.
- Lines 22 through 29 create the **JadeUserNameToken** and call the **createDotNetObejct_1** method to set up the user name, password, and password option and encoding type. We are using a hashed password and a base-64 binary encoding. At this point, we also set up the **protectionOrder** property to say that we want to sign and encrypt the message. We also set up the password to use for the signing and encryption. We then call the **getXml** method, which will set up the required headers and set up the information required for signing and encrypting the message.
- Line 33 calls the **invoke** method defined in the superclass, using the string returned by the last **getXml** method as a parameter.

This will send the message with the required information to the Web service.

Processing Security Headers (Service)

The provider of the service knows what information is required to be in the headers. In order to process the incoming message, we need to re-implement the **processRequest** method on the **JadeCalculator** class.

The following shows the method required for the example Web service.

```
 1 processRequest() updating, protected;
 2
 3 vars
 4     jwss:               JadeWebServicesSecurity;
 5     str:                StringUtf8;
 6     isPasswordValid:    Boolean;
 7 begin
 8     // get the tokens from the incoming message
 9     // we expect to get 3 tokens, addressing, timestamp and username
10     // raise exception if one of these is missing
11
12     create jwss;
13     jwss.getTokens(incomingMessage.StringUtf8);
14
15     if jwss.addressing = null then
16         raiseSecurityTokenException("Addressing header is missing");
17     endif;
18
19     if jwss.creationTimestamp = null then
20         raiseSecurityTokenException("Timestamp security header is missing");
21     endif;
22
23     if jwss.usernameToken = null then
24         raiseSecurityTokenException("UsernameToken security header is missing");
25     endif;
26
27     // now validate the incoming data.
28     // This will raise an exception if the timestamp is not valid
29     jwss.creationTimestamp.validateTimestamp();
30
31     jwss.usernameToken.clearPassword := "password";
32
33     // Validate password - check the return result
34     // and take appropriate action
35     isPasswordValid := jwss.usernameToken.validatePassword();
36
37     // save addressing information for sending with the reply
38     // in user defined properties
39     messageID := jwss.addressing.messageID;
40     soapAction := jwss.addressing.action;
41     endpoint := jwss.addressing.replyTo;
42
43     // if encryption is set, decrypt the message - exception raised if decryption fails
44     if jwss.isEncrypted then
45         str := jwss.usernameToken.decryptXml(incomingMessage.StringUtf8);
46         // save the descrypted message
47         incomingMessage := str.String;
48     endif;
49
50     // if the message is signed, verify the signature. assumes that the message is signed
51     // then encrypted - exception riased if verify fails
52      if jwss.isSigned then
53         jwss.usernameToken.verifySignature(str);
54     endif;
55
56     inheritMethod();
57 epilog
58     delete jwss;
59 end;
```

In this method:

- Lines 4 through 6 declare the local variables required for processing the message.
- Lines 12 and 13 create an instance of the **JadeWebServicesSecurity** class and call the method **getTokens** on it. This method will scan for addressing and security headers and populate the properties on the created instance.
- Lines 15 through 25 validate that the required headers are present. If they are not, an exception is raised by calling the **raiseSecurityTokenException** method (defined elsewhere).
- Line 29 validates the timestamp information. If the timestamp is not valid or if the current time is past the *expires* time, an exception is raised.
- Line 31 sets up the password for validation, signing, and encryption.
- Line 35 validates the password. If the password is not valid, a Boolean value of **false** is returned.
- Lines 39 through 41 save information in the addressing header that will be sent with the response message.
- Lines 44 through 48 decrypt the message if it is encrypted and save the decrypted message for processing by the Web services framework. An exception is raised if the decryption fails; for example, if the supplied password does not match the password used to encrypt the message, an exception is raised.
- Lines 52 through 54 validate the signature if the message has been signed. An exception is raised if the validation fails; for example, if the supplied password does not match the password used to sign the message, an exception is raised.
- Line 56 then calls the **processRequest** on the superclass, to continue the processing of the message.
- Line 58 deletes the transient instance.

If the Web service wants to send headers in the response message, the code will need to be placed in a re-implemented **reply** method, as shown in the following method.

```
 1 reply(): String updating, protected;
 2
 3 vars
 4     wsAddress:          JadeWSAddressingHeader;
 5     wsTimestamp:        JadeWSTimestampHeader;
 6     out:                String;
 7     str:                StringUtf8;
 8     unt:                JadeUsernameToken;
 9 begin
10
11     // get the generated message
12     out := inheritMethod();
13
14     // add addressing information
15     create wsAddress;
16     wsAddress.action := soapAction & "Response";
17     wsAddress.sendTo := endpoint;
18     wsAddress.relatesTo := messageID;
19     str := wsAddress.getXml(out.StringUtf8);
20
21     // add the timestamp header and set the expiry time to 1000
22     create wsTimestamp;
23     wsTimestamp.secondsToTimeout := 1000;
24     str := wsTimestamp.getXml(str);
25
26    // NOTE:
27    // if we want to encrypt and/or sign the message we will need to create
28    // a username token to send with the response. In this case, we do not
29    // want to encrypt or sign the message.
30
31     return str.String;
32
33 epilog
34     delete wsAddress;
35     delete wsTimestamp;
36 end;
```

Similarly, if the Web service client needs to process headers in the message that it receives, the code is implemented in the **invoke** method; that is, in the same method from which it generates the headers, as shown in the following method.

```
 1 invoke(inputMessage: String): String updating;
 2
 3 vars
 4     usernameToken:      JadeUsernameToken;
 5     wsAddress:          JadeWSAddressingHeader;
 6     wsTimestamp:        JadeWSTimestampHeader;
 7     out:                String;
 8     str:                StringUtf8;
 9     jwss:               JadeWebServicesSecurity;
10
11 begin
12     // add addressing information
13     create wsAddress;
14     wsAddress.action := getSoapAction('add').StringUtf8;
15     wsAddress.sendTo := getEndpointURL.StringUtf8;
16     str := wsAddress.getXml(inputMessage.StringUtf8);
17
18     // add the timestamp header and set the expiry time to 1000
19     create wsTimestamp;
20     wsTimestamp.secondsToTimeout := 1000;
21     str := wsTimestamp.getXml(str);
22
23     // set up the user name token we will use the hased password
24     create usernameToken;
25     usernameToken.createDotNetObject_1("wilbur", "password", usernameToken.PasswordOption_SendHashed,
26                                     usernameToken.EncodingType_Base64Binary);
27
28     usernameToken.protectionOrder := Jadwssec.ProtectionType_SignAndEncrypt;
29     usernameToken.clearPassword := "password"; //password for signing end encryption
30
31     str := usernameToken.getXml(str);
32
33     // now send the message by calling the superclass method and wait for response
34
35     out := inheritMethod(str.String);
36
37     // the out variable now contains the response message, get the tokens and validate them.
38
39     create jwss;
40     jwss.getTokens(out.StringUtf8);
41
42     if jwss.addressing = null then
43         raiseSecurityTokenException("Addressing header is missing");
44     endif;
45
46     if jwss.creationTimestamp = null then
47         raiseSecurityTokenException("Timestamp security header is missing");
48     endif;
49
50     jwss.creationTimestamp.validateTimestamp();
51
52     return out.String;
53
54 epilog
55     delete usernameToken;
56     delete wsAddress;
57     delete wsTimestamp;
58
59 end;
```

Sample SOAP Message

The following is an example SOAP message that is sent from the Web service client to the Web service. This sample contains WS-Addressing headers, Timestamp, UserNameToken, with a hashed password, signature, and encryption.

```
<?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/"
xmlns:tns="urn:JadeWebServices/CalculatorService/"
xmlns:s1="urn:JadeWebServices/CalculatorService/"
xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/03/addressing"
xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-
utility-1.0.xsd" xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
wss-wssecurity-secext-1.0.xsd">
  <soap:Header>
    <wsa:Action wsu:Id="Id-378ad0c7-777e-48d0-8d4e-789634b0e757"></wsa:Action>
    <wsa:MessageID wsu:Id="Id-89d7c1e0-b989-4cb1-8319-c2b3cc9259bc">uuid:301dc198-
5d2b-4f72-9bbb-b6de3785ec7f</wsa:MessageID>
    <wsa:ReplyTo wsu:Id="Id-953d69ec-9756-4367-b88f-e1c0d6859c13">

<wsa:Ad-
dress>http://schemas.xmlsoap.org/ws/2004/03/addressing/role/anonymous</wsa:Address>
    </wsa:ReplyTo>
    <wsa:To wsu:Id="Id-7034a4d8-4142-4b71-997e-d0c7a8a7e9ef"></wsa:To>
    <wsse:Security soap:mustUnderstand="1">
      <wsu:Timestamp wsu:Id="Timestamp-b098ebcf-14ca-472f-b643-1c85b68a0493">
        <wsu:Created>2010-04-13T21:22:27Z</wsu:Created>
        <wsu:Expires>2010-04-13T21:39:07Z</wsu:Expires>
      </wsu:Timestamp>
      <wsse:UsernameToken wsu:Id="SecurityToken-339cb9af-73ad-4405-9223-
3f1cfce02a1e">
        <wsse:Username>wilbur</wsse:Username>
        <wsse:Password Type="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
wss-username-token-profile-
1.0#PasswordDigest">y+RiI7GYQE4J8lX/e1yOS+mZfI4=</wsse:Password>
        <wsse:Nonce>5FiJYx352dYgamYU7CHDqOrfzrA=</wsse:Nonce>
        <wsu:Created>2010-04-13T21:22:27Z</wsu:Created>
      </wsse:UsernameToken>
      <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
        <ds:SignedInfo>
          <ds:CanonicalizationMethod Algorithm="http://www.w3.org/2001/10/xml-exc-
c14n#" />
          <ds:SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#hmac-
sha1" />
          <ds:Reference URI="#Id-378ad0c7-777e-48d0-8d4e-789634b0e757">
            <ds:Transforms>
              <ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#" />
            </ds:Transforms>
            <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
            <ds:DigestValue>+LMRkGFO6gtY91ey8OXKEAutohE=</ds:DigestValue>
          </ds:Reference>
          <ds:Reference URI="#Id-89d7c1e0-b989-4cb1-8319-c2b3cc9259bc">
            <ds:Transforms>
              <ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#" />
```

```
            </ds:Transforms>
            <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
            <ds:DigestValue>j3xYQQXDX+xBbET1qLbNFO2A63s=</ds:DigestValue>
          </ds:Reference>
          <ds:Reference URI="#Id-953d69ec-9756-4367-b88f-e1c0d6859c13">
            <ds:Transforms>
              <ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#" />
            </ds:Transforms>
            <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
            <ds:DigestValue>5W11ZeYp1Xrh+GsIQnbjOHVf2vg=</ds:DigestValue>
          </ds:Reference>
          <ds:Reference URI="#Id-7034a4d8-4142-4b71-997e-d0c7a8a7e9ef">
            <ds:Transforms>
              <ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#" />
            </ds:Transforms>
            <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
            <ds:DigestValue>YF/+6N++1bXgYGYEpWuEKDjFCwA=</ds:DigestValue>
          </ds:Reference>
          <ds:Reference URI="#Timestamp-b098ebcf-14ca-472f-b643-1c85b68a0493">
            <ds:Transforms>
              <ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#" />
            </ds:Transforms>
            <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
            <ds:DigestValue>NAToSMCQMcO+9jDWvTDe1hpgbfU=</ds:DigestValue>
          </ds:Reference>
          <ds:Reference URI="#Id-00299f17-588c-4f1f-987e-23b4534cfc21">
            <ds:Transforms>
              <ds:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#" />
            </ds:Transforms>
            <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
            <ds:DigestValue>So2/+F/h+EO1FOORwX2n1kkLbbs=</ds:DigestValue>
          </ds:Reference>
        </ds:SignedInfo>
        <ds:SignatureValue>sTeId/otJygDBxEx8sW3iGDjLxM=</ds:SignatureValue>
        <ds:KeyInfo>
          <wsse:SecurityTokenReference>
            <wsse:Reference URI="#SecurityToken-339cb9af-73ad-4405-9223-
3f1cfce02a1e" ValueType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-
username-token-profile-1.0#UsernameToken" />
          </wsse:SecurityTokenReference>
        </ds:KeyInfo>
      </ds:Signature>
    </wsse:Security>
  </soap:Header>
  <soap:Body wsu:Id="Id-00299f17-588c-4f1f-987e-23b4534cfc21">
    <xenc:EncryptedData Id="EncryptedContent-d028b5dd-bc55-4dd8-8cc6-0b4cfdd98f4b"
Type="http://www.w3.org/2001/04/xmlenc#Content"
xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
      <xenc:EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#tripledes-
cbc" />
      <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
        <wsse:SecurityTokenReference>
          <wsse:Reference URI="#SecurityToken-339cb9af-73ad-4405-9223-3f1cfce02a1e"
ValueType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-
profile-1.0#UsernameToken" />
        </wsse:SecurityTokenReference>
```

```
      </ds:KeyInfo>
      <xenc:CipherData>
        <xenc:CipherValue>v0SsdDFqxztnIsuF3loPvYwdNyChY3QeIHcFKPWTSXVjnrBe9VEY
PeEqbiFZSy7sqJoAwd2iGLs-
m+JWwYHC1jpAoYQhzhAC8+WLNmk5v8FfquRHGEr+5p9+/oY82cHjg-
p4ZV2k5C9qDxgYWhmEA9hA==</xenc:CipherValue>
      </xenc:CipherData>
    </xenc:EncryptedData>
  </soap:Body>
</soap:Envelope>
```