



J A D E™

Web Services Security

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 © 2010 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.

Contents

Web Services Security	3
Overview	3
UsernameToken Profile	3
Namespaces	3
User Names and Passwords	4
XML Syntax	5
Examples	6
The Security Class Library	6
Security Classes	7
Jadwssec Class	7
JadeSecurityToken Class	7
decryptXml Method	8
encryptXml Method	8
getXml Method	9
signXml Method	9
verifySignature Method	9
JadeUsernameToken Class	10
createDotNetObject_1 Method	10
getPassword Method	11
getUsername Method	11
getXml Method	11
validatePassword Method	12
JadeWSAddressingHeader Class	13
getXml Method	13
JadeWSTimestampHeader Class	14
getXml Method	14
validateTimestamp Method	14
JadeWebServicesSecurity Class	15
getTokens Method	15
Example of .NET Security Class Library Use	16
Importing the Library	16
Creating a Package	17
Importing the Package	18
Creating a Web Service	20
Consuming the Web Service	21
Generating Security Headers (Client)	22
Processing Security Headers (Service)	23
Sample SOAP Message	28

Web Services Security

Overview

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 user name and optionally using a password (or shared secret, or password equivalent) to authenticate that identity to the Web service provider.

UsernameToken Profile

Namespaces

Namespace Uniform Resource Identifiers (URIs), of the general form "some-URI", represent some application-dependent or context-dependent URIs 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 in this document to provide detailed examples.

The namespaces used in this document are listed 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 following table lists the full URI for each URI fragment referred to in this document.

URI Fragment	Full URI
#PasswordDigest	http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#PasswordDigest
#PasswordText	http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#PasswordText
#UsernameToken	http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#UsernameToken

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, you can specify a **<wsse>Password>** element. 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 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, which consumes 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.

$$\text{PasswordDigest} = \text{Base64} (\text{SHA-1} (\text{nonce} + \text{created} + \text{password}))$$

The above digest value 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 obscure the password and offers a basis for preventing replay attacks.

For Web service providers to effectively thwart replay attacks, the following three counter-measures are recommended.

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

As a guideline, a value of five minutes can be used as a minimum to detect, and thus reject, replays.

- Used nonces are cached for a period at *least* as long as the timestamp freshness limitation period in the previous list item, and that **UsernameTokens** with nonces that have already been used (and are thus in the cache) are rejected.

Note You can use **PasswordDigest** 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 attributes and elements listed in the above example are as follows.

- **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**
Optional attribute that specifies the type of password and can take one of the following 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 password algorithm described in the "User Names and Passwords" section, earlier in this document.
- **wsse:Nonce**
Optional, specifies a random nonce. Each message including a nonce must provide a unique nonce value.
- **Encoding Type**
Optional attribute that specifies the encoding type of the nonce. If not specified, Base64 encoding is used.
- **wsu:Created**
Optional element that specifies a timestamp used to indicate the creation time.

Examples

In the following example that illustrates the use of this element, the password is sent as clear text. This message should therefore 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 as well as 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>
```

The Security Class Library

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

As the library is likely to be required by more than one schema, it may be desirable 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.

Security Classes

The import generates six classes.

Note These classes are generated with default names, which you can change after you have imported the **jadwssec** class library.

The following discussion is based on the default names.

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 constants defined in this class are listed in the following table.

Name	Type	Value	Description
AssemblyName	String	jadwssec, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null"	Assembly Details
EncodingType_Base64Binary	Integer	0	Nonce Encoding Base 64 (default)
EncodingType_HexBinary	Integer	1	Nonce Encoding Binary
PasswordOption_SendHashed	Integer	0	Password Type Digest (default)
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 properties defined in this class are listed in the following table.

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

The following subsections describe the methods defined for the **JadeSecurityToken** class.

decryptXml Method

The **decryptXml** method is called to decrypt an encrypted SOAP message.

Parameter

- xml

This **StringUtf8** parameter contains the SOAP message to decrypt.

Returns

This method 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,  
                                'password').String;
```

encryptXml Method

The **encryptXml** method is called to encrypt a SOAP message.

Parameter

- xml

This **StringUtf8** parameter contains the SOAP message to encrypt.

Returns

This method returns a **StringUtf8** string representing the encrypted 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, 'password');
```

getXml Method

The **getxml** method, which is an abstract method, is called to serialize the security token into XML. The implementation of this method is token-dependent.

Parameter

- xml

This **StringUtf8** parameter contains the SOAP message to which the XML is to be added.

Returns

This method 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 Method

The **signXML** 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.

Parameter

- xml

This **StringUtf8** parameter contains the SOAP message to sign.

Returns

This method returns a **StringUtf8** string representing the signed SOAP message.

Remarks

All strings passed to and returned from the class library are UTF-8 strings.

Example

```
outString := unt.signXml(inString, 'password');
```

verifySignature Method

The **verifySignature** method is called to verify the signature of a SOAP message.

An exception is raised if the signature verification fails.

Parameter

- xml
This **StringUtf8** parameter contains the SOAP message to verify.

Returns

Nothing.

An exception is raised if signature verification fails.

Remarks

All strings passed to and returned from the class library are UTF-8 strings.

Example

```
unt.verifySignature(inString, 'password');
```

JadeUsernameToken Class

The **JadeUsernameToken** class represents the **UserNameToken** profile.

The following subsections describe the methods that are available in this class.

createDotNetObject_1 Method

The **createDotNetObject_1** method is used to create a **JadeUsernameToken** instance.

Parameters

- username
This **StringUtf8** parameter contains the SOAP message to verify.
- password
This **StringUtf8** parameter contains the clear password to use for the verification.
- passType
This **StringUtf8** parameter specifies the password type. It can be one of the following values.
 - PasswordOption_SendHashed (the default value)
 - PasswordOption_SendPlain
 - PasswordOption_SendNone
- encType
This **StringUtf8** parameter specifies the encoding type. It can be one of the following values.
 - EncodingType_Base64Binary (the default value)
 - EncodingType_HexBinary

Returns

Nothing.

Remarks

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 Method

The **getPassword** method returns the password on a **JadeUsernameToken** instance.

Parameter

None.

Returns

This method returns the password associated with the **JadeUsernameToken** instance.

Remarks

The password can be null, plain text, or hashed.

Example

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

getUsername Method

The **getUsername** method returns the user name on a **JadeUsernameToken** instance.

Returns

Nothing.

Remarks

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

Example

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

getXml Method

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

Parameter

- xml

This **StringUtf8** parameter contains the SOAP message to which to add the XML.

Returns

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

validatePassword Method

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

Parameters

None.

Returns

A Boolean value. A value of **true** indicates that the supplied clear password matches the incoming password and **false** indicates that 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.

The properties defined in this class are listed in the following table.

Name	Type	Description
action	StringUtf8	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	StringUtf8	Represents the <To> tag. This element provides the value of the destination URL. Required.
messageID	StringUtf8	Read-only property that is a generated Global Unique Identifier (GUID).
relatesTo	StringUtf8	Required only in the response message and should have the value of the <MessageID> tag from the request message.
replyTo	StringUtf8	Read-only property, whose value is a constant.

The value of the **replyTo** property constant will always be:

<http://schemas.xmlsoap.org/ws/2004/08/addressing/role/anonymous>

The following subsection describes the method defined for this clas.:

getXml Method

The **getXml** method is called to serialize the addressing header into XML.

Parameter

- xml
This **StringUtf8** parameter contains the SOAP message to which the XML is to be added.

Returns

This method returns a **StringUtf8** string representing the SOAP message with the serialized addressing header.

Remarks

None.

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 properties defined in this class are listed in the following table.

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 subsections describe the methods defined for the **JadeWSTimestampHeader** class.

getXml Method

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

Parameter

- xml
This **StringUtf8** parameter contains the SOAP message to which the XML is to be added.

Returns

This message returns a **StringUtf8** string representing the SOAP message with the serialized timestamp information.

Remarks

None.

Example

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

validateTimestamp Method

The **validateTimestamp** 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.

Remarks

None.

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 properties defined in this class, listed in the following table, 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 it is present in the message.

The following subsections describe the methods defined for the methods are defined for the **JadeWebServicesSecurity** class.

getTokens Method

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

Parameter

- xml

This **StringUtf8** parameter contains the SOAP message to process for header information.

Returns

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

Remarks

None.

Example

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

Example of .NET Security Class Library Use

In this example:

1. The class library is imported into a schema called **WebServiceUtilitiesSchema**.
2. From this schema, the classes that make up this class library in a package called **UserNameTokenSecurityProfile** are exported.
3. This package is then imported into a:
 - a. Web service provider schema called **CalculatorServices**.
 - b. Web service consumer schema called **CalculatorServicesClient**.

This Web service consumer schema **CalculatorServicesClient** imports the WSDL generated by the **CalculatorServices** Web service.

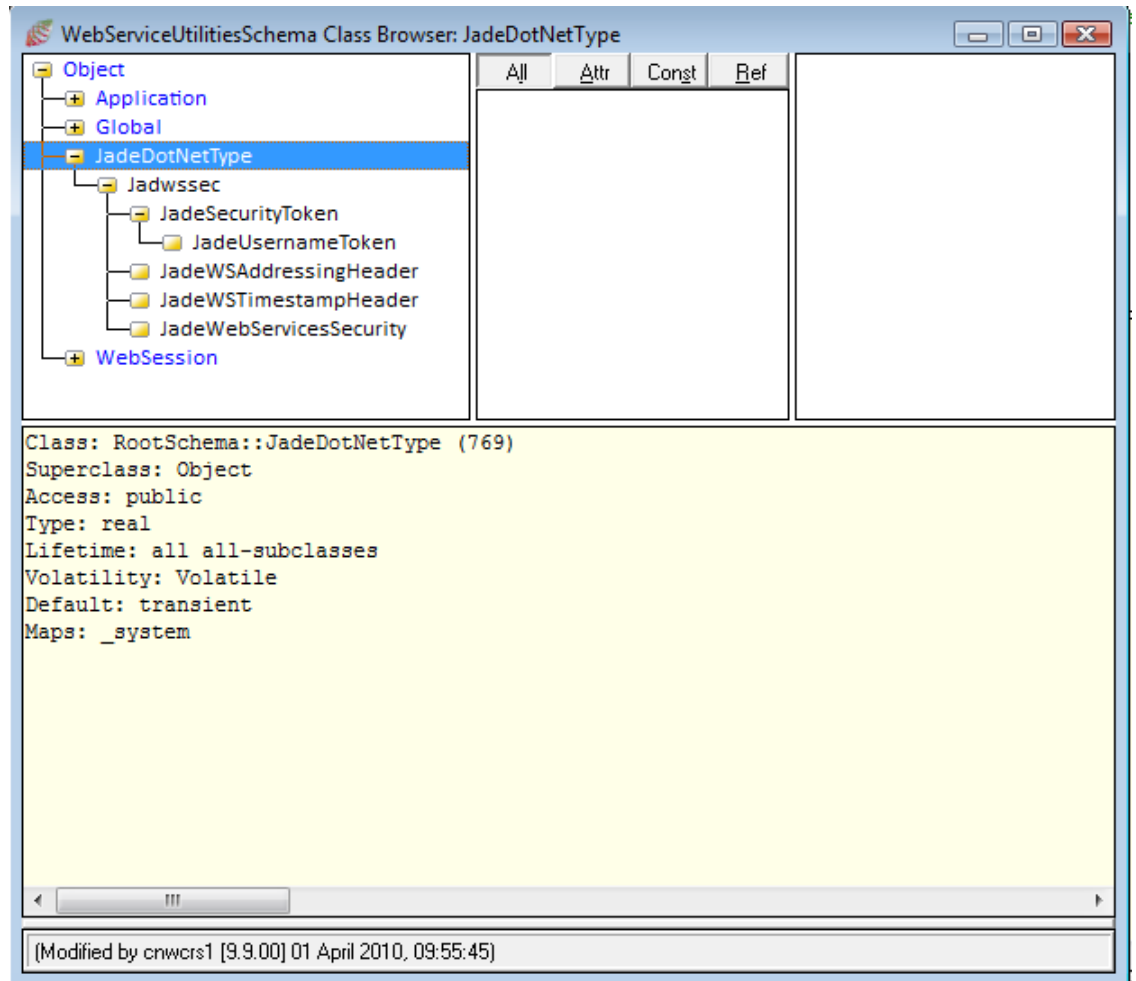
Importing the Library

The library is imported by creating a schema called **WebServiceUtilitiesSchema** and import the library called **jadwssec.dll** into this schema.

➤ To import the library

1. Select the **External Component Libraries** command from the Browse menu in the JADE development environment.
The External Components Browser is then displayed.
2. Make sure the **.NET Framework** tab is selected, right-click, and select the **Import** command from the popup menu that is then displayed.
The .NET Import Wizard is then displayed.
3. Click the **Browse** button and then change the directory to your JADE installation directory, select **jadwssec.dll** from the list of assemblies, and then go through the import process.
4. When you have completed all import process steps, open a Class Browser.

The classes shown in the following diagram are now displayed in the Class List.

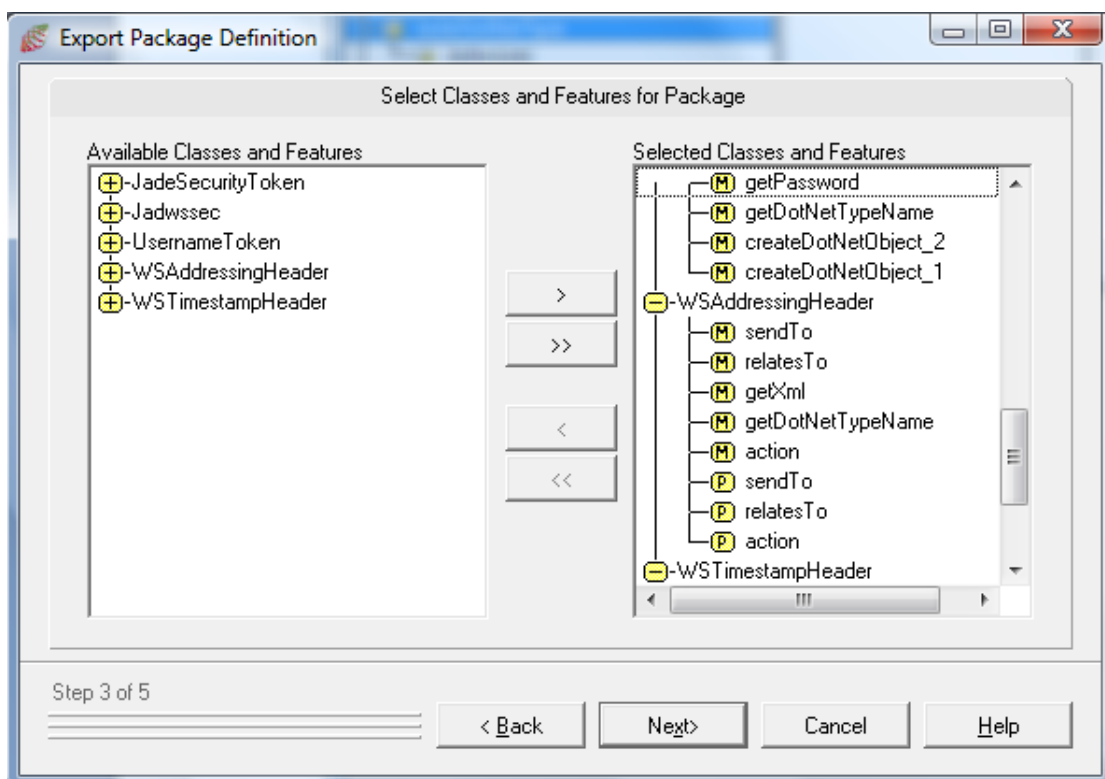


Creating a Package

➤ To create a package

1. Select the **Packages** command from the Browse menu.
2. Select the **Export Package** command from the submenu that is then displayed.
The Export Packages Browser window is then opened.
3. Right-click and select the **Add** command from the popup menu that is then displayed.
The Export Package Definition Wizard is then displayed.
4. On the first sheet of this wizard, create an export package called **UserNameTokenSecurityProfile**.

5. On the third sheet of the wizard, select all of the classes, constants, properties, and methods that were imported from the library, as shown in the following diagram.

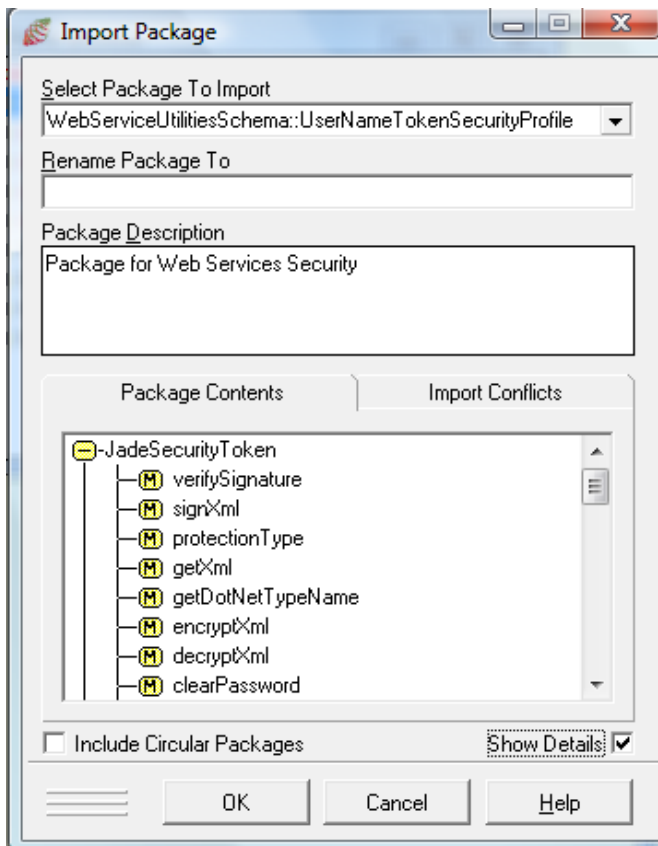


Importing the Package

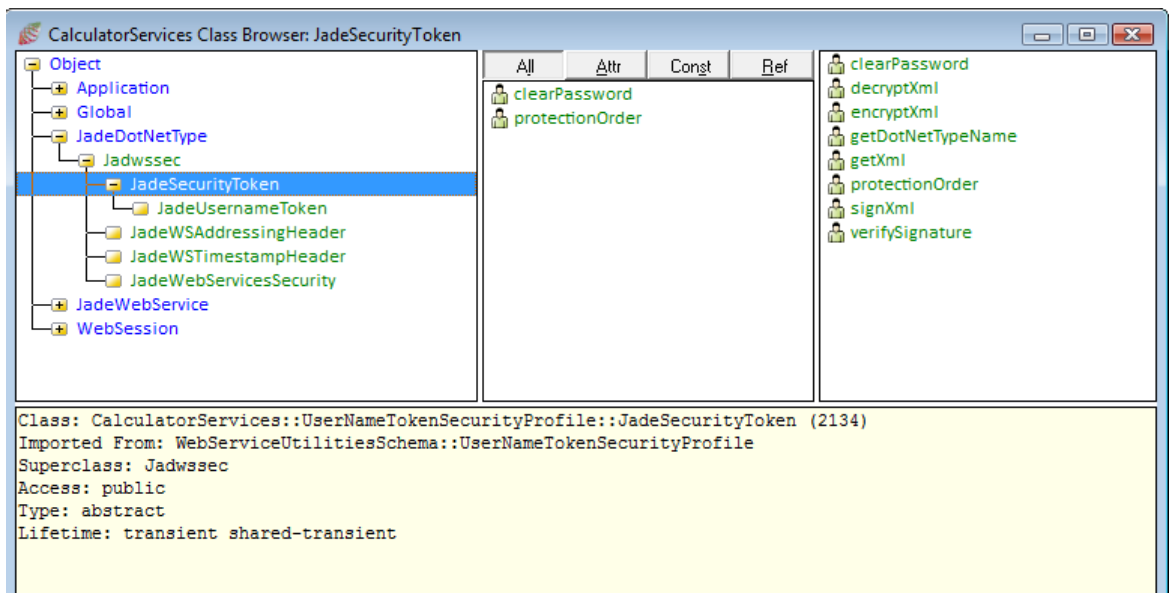
➤ To import the package

1. Create a new schema called **CalculatorServices**.

- Import the **UserNameTokenSecurityProfile** package into your new schema, as shown in the following diagram.



The imported classes, properties, and methods are then displayed in the applicable lists in the Class Browser, as shown in the following diagram.

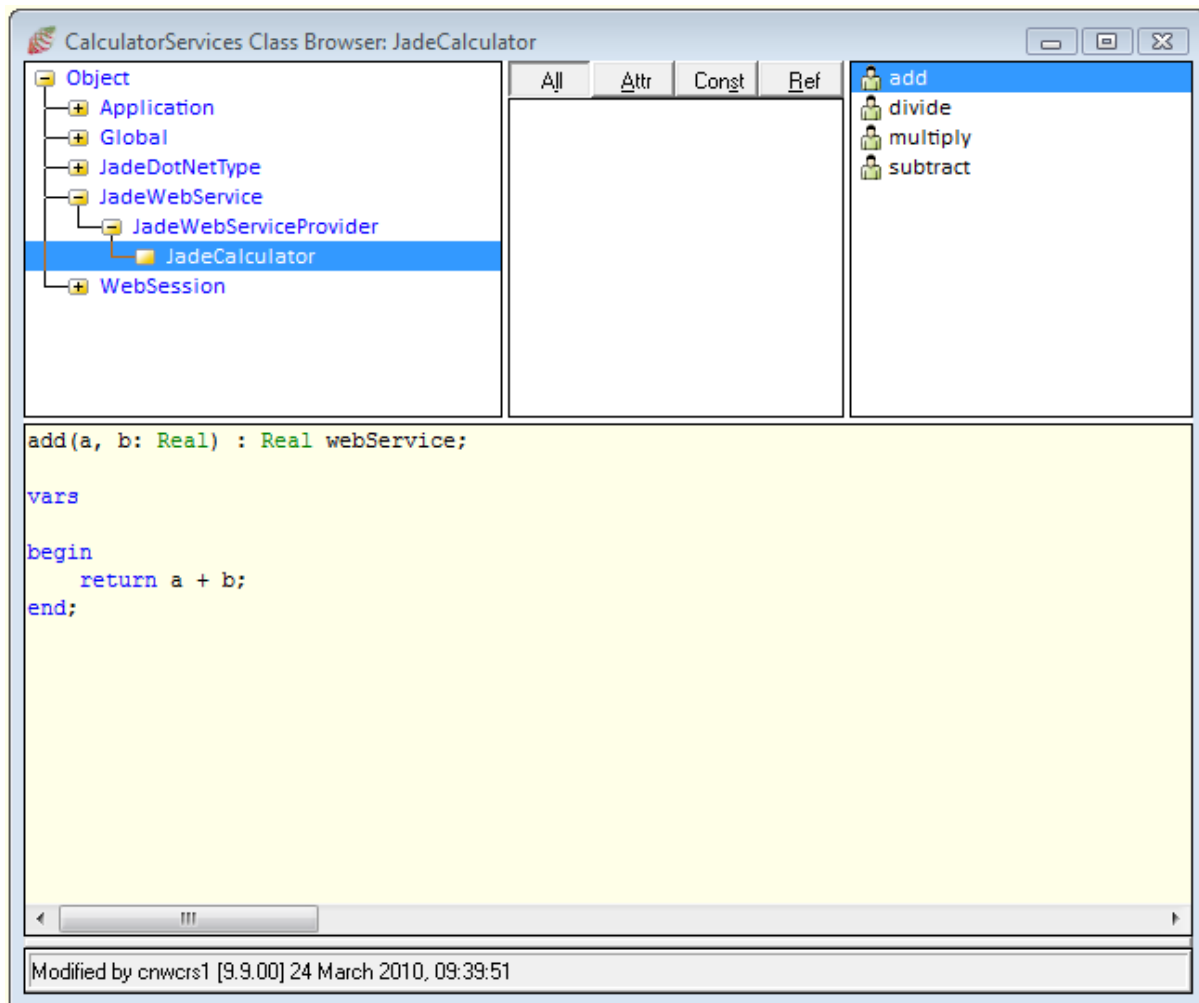


- Create another schema called **CalculatorServicesClient** and then import the package into it.

Note The Web service and the Web service client will not normally 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.

Creating a Web Service

The example in this section creates a **Calculator** Web service class, as shown in the following diagram.



➤ To set up the Web service

1. Define the exposure list.
2. Set up the application.
3. Generate the WSDL.

This WSDL will be imported into the **CalculatorWebServicesClient** schema.

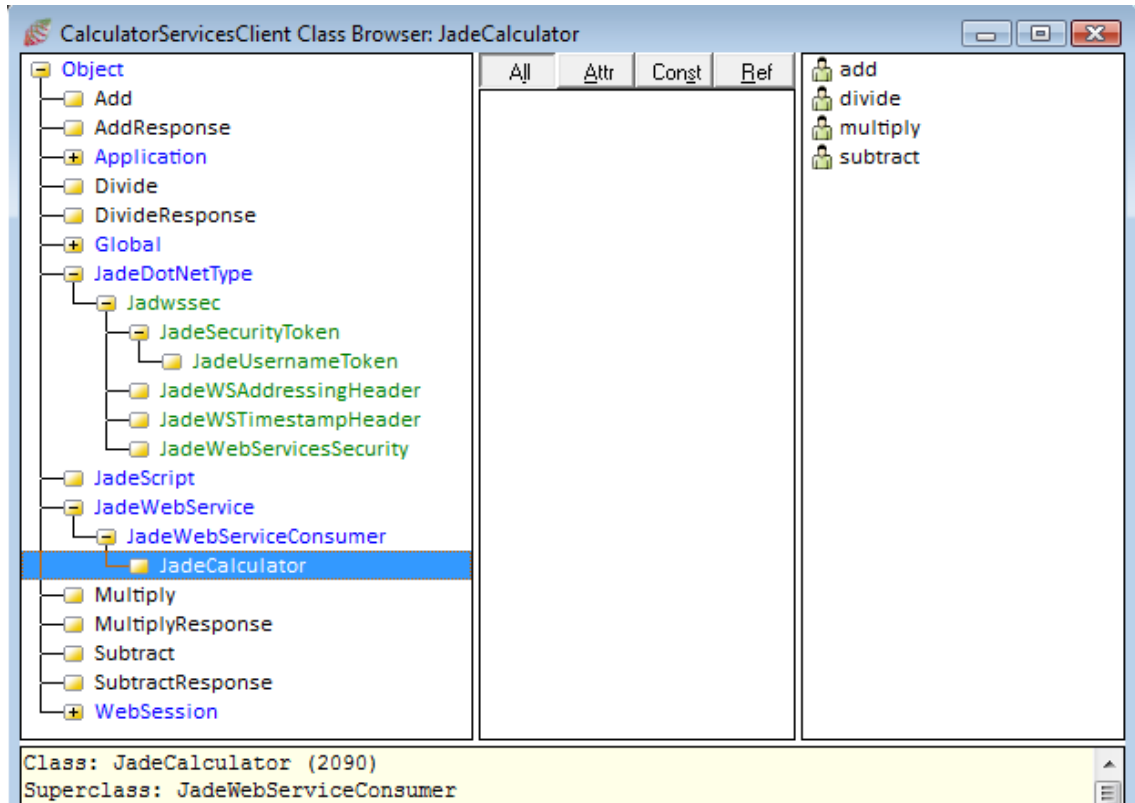
4. Set up the required virtual directory setting, and so on, in the **jadehttp.ini** file, to meet your requirements.

For more information, see the *Web Services in JADE* and *Web Services Tips* white papers on the JADE Web site (that is, <http://www.jade.co.nz/jade/whitepapers.htm>).

- Set up the Web service provider so that it can process the incoming message that has security headers. Before doing so, set up the Web service consumer. For details, see the following section.

Consuming the Web Service

Import the Web service into the **CalculatorWebServicesClient** schema, using the Web Service Consumer Browser. At the end of the import process, the Class Browser for this schema has the classes shown in the following diagram.



Write a JADE script that will set up the required security tokens and call the Web service. The following is an example of a JADE script that 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 example:

- 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.

To set up the security headers to be sent by the Web service client, re-implement the **invoke** method on the **JadeCalculator** class. The example in the following code fragment sets up addressing, timestamp, and hashed user name token, and signs and encrypts the message.

```
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 hashed 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. This method inserts the security headers into this SOAP message prior to it being sent.

In this example:

- 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 1,000 seconds after the creation time.

- Lines 22 through 29 create **JadeUserNameToken** and call the **createDotNetObject_1** method to set up the user name, password, password option, and encoding type. A hashed password and a base 64 binary encoding are being used.

At this point, the **protectionOrder** property is set up so that the message is signed and encrypted. In addition, the password is set up to use for the signing and encryption.

Following that, the **getXml** method is called, which sets up the required headers and 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 sends the message with the required information to the Web service.

Processing Security Headers (Service)

The Web service provider knows what information is required to be in the headers.

To process the incoming message, re-implement the **processRequest** method on the **JadeCalculator** class.

The code required for the example Web service is shown in the following diagram.

```
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 decrypted 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 raised 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 example:

- 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 **getTokens** method on it.

The **getTokens** method scans 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, which is defined elsewhere in this document.
- Line 29 validates the timestamp information.

If the timestamp is not valid or if the current time is later than 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 calls the **processRequest** method 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 needs to be placed in a re-implemented **reply** method, as shown in the following example.

```
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 as that from which it generates the headers, as shown in the following example.

```
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 hashed 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 example 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:Address>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/elYOS+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>+LMRkGF06gtY9ley8OXKEAutohE=</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>j3xYQQDX+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-4405-9223-3f1cfcce02a1e"
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>v0SsdDFqxztnIsuF3loPvYwdNyChY3QeIhcFKPWTsXVjnrBe9VEYPeEqbiFZSy7sqJoAwd2i
        GLsm+JWwYHC1jpAoYQhzhAC8+WLNmk5v8FfquRHGEr+5p9+/oY82cHjgp4ZV2k5C9qDxgYWhmEA9hA==</xenc:Cip
        herValue>
      </xenc:CipherData>
    </xenc:EncryptedData>
  </soap:Body>
</soap:Envelope>
```