

# Web Services Stack Guide

## Configuration

Version 8.0 SP4

March 2010

This document applies to WSS Guide Version 8.0 SP4.

Specifications contained herein are subject to change and these changes will be reported in subsequent release notes or new editions.

Copyright © 2007-2010 Software AG, Darmstadt, Germany and/or Software AG USA, Inc., Reston, VA, United States of America, and/or their licensors.

The name Software AG, webMethods and all Software AG product names are either trademarks or registered trademarks of Software AG and/or Software AG USA, Inc. and/or their licensors. Other company and product names mentioned herein may be trademarks of their respective owners.

Use of this software is subject to adherence to Software AG's licensing conditions and terms. These terms are part of the product documentation, located at <http://documentation.softwareag.com/legal/> and/or in the root installation directory of the licensed product(s).

This software may include portions of third-party products. For third-party copyright notices and license terms, please refer to "License Texts, Copyright Notices and Disclaimers of Third-Party Products". This document is part of the product documentation, located at <http://documentation.softwareag.com/legal/> and/or in the root installation directory of the licensed product(s).

## Table of Contents

1 Configuration .....	1
2 Web Services Stack Runtime .....	3
Understanding axis2.xml Configuration .....	4
Run-Time Configuration .....	9
Server-side Configurations .....	12
Client-side Configurations .....	13
MTOM in Web Services Stack .....	13
3 Security .....	15
Message-Level Security .....	16
Transport-Level Security .....	24
Client Authentication .....	31
4 Transports .....	37
TCP Transport .....	38
JMS Transport .....	41
Mail Transport .....	45
5 Monitoring and Logging .....	53
SOAP Monitor .....	54
Logging .....	56
6 Eclipse Plug-in .....	59
Introduction .....	60
Creating and Removing a Web Service Package .....	61
Configuring a Web Service Package .....	62
Enabling Advanced Policy Configurations .....	64
Deploying and Undeploying a Web Service Package .....	65
Registering a Web Service Package in CentraSite .....	66



# 1 Configuration

---

This document introduces the configuration tasks in Web Services Stack (WSS).

*Configuration* provides users with sets of instructions on the features of the product and shows how to configure and use Web Services Stack for managing, monitoring, and securing services.

The information is organized under the following headings:

• <b>Web Services Stack Runtime</b>	Details on the configurations of <i>axis2.xml</i> , client, server, and MTOM.
• <b>Security</b>	Configurations for securing the message content and the communication channel.
• <b>Transports</b>	Configuration of Web Services Stack for sending and receiving of messages over different transports.
• <b>Monitoring and Logging</b>	Configuration of Web Services Stack facilities for monitoring and logging.
• <b>Eclipse Plug-in</b>	Describes Web Services Stack basic tool for packaging, configuring, and deploying web service archives.



# 2 Web Services Stack Runtime

---

- Understanding axis2.xml Configuration ..... 4
- Run-Time Configuration ..... 9
- Server-side Configurations ..... 12
- Client-side Configurations ..... 13
- MTOM in Web Services Stack ..... 13

The information is organized under the following headings:

## Understanding axis2.xml Configuration

---

Following are the top-level elements that can be seen in the *axis2.xml* configuration file:

### 1. Parameters

In Axis 2, a parameter is a name value pair. Each top level parameter available in the *axis2.xml* file (direct sub-elements of a root element) is transformed into properties in *AxisConfiguration*. Therefore, if you want to access the top level parameters in the configuration document, you must use the *AxisConfiguration* in the running system.

Following is the correct way of defining a parameter:

```
<parameter name="name of the parameter" >parameter value </parameter>
```

### 2. Transport Receivers

You must use a corresponding transport receiver for every other transport that Axis 2 runs on. Following is an example of how you can add and define transport receivers in the *axis2.xml* file:

```
<transportReceiver name="http"  
class="org.apache.axis2.transport.http.SimpleHTTPServer">  
    <parameter name="port" >6060</parameter>  
</transportReceiver>
```

The attribute name of the `transportReceiver` property is the name of the transport itself. It can be HTTP, TCP, SMTP, etc. As soon as the system starts up, or when you set the transport at the client side, you actually load the transport that you have specified. The class attribute defines the actual Java class that implements the interfaces required for the transport. Transports can have zero parameters, but they can also have one or more than one. Nevertheless, the corresponding transport receiver gives access to the parameters.

### 3. Transport Senders

You can register transport senders in the system in the same way as transport receivers. Later, at run time, those senders can be used to send messages. As an example, consider Axis 2 running under Tomcat. In this case, Axis can use TCP rather HTTP transport senders to send messages. Following is the way to specify transport senders:



```
<transportSender name="http"
class="org.apache.axis2.transport.http.CommonsHTTPTransportSender">
    <parameter name="PROTOCOL" locked="xsd:false">HTTP/1.0</parameter>
</transportSender>
```

In the preceding example, the attribute `name` is the name of the transport. The attribute `class` is the implementation class of the corresponding transport. Transport senders can have zero or more than one parameters in the same way as transport receivers. If there are any parameters specified, then you can access them at run time through the corresponding transport.

#### 4. Phase Orders

The specifying order of phases in the execution chain has to be done using a phase order element. Following is an example of such an element:

```
<phaseOrder type="inflow">
    <phase name="TransportIn"/>
    .
    .
</phaseOrder>
```

If you want to add a handler that must go into that phase, you can do that by adding directly a handler element into it. In addition, configurations for handler chains in Axis 2 are also done in the phase order element.

Following is an example of the complete configuration of the order of phases in the execution chain with specified corresponding handlers:

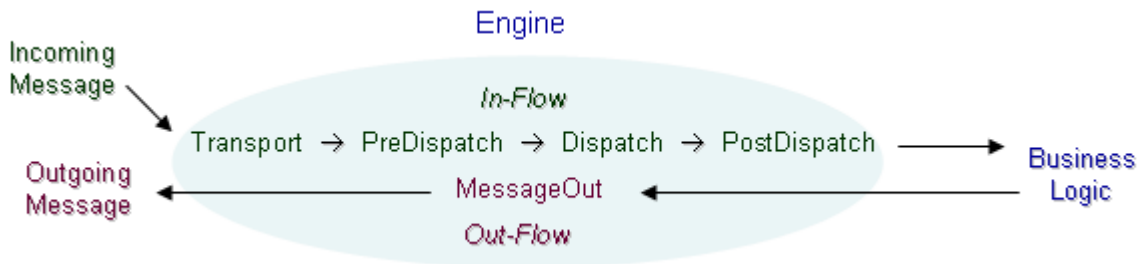
```
<phaseOrder type="InFlow">
<!-- System pre-defined phases -->
<phase name="Transport">
<handler name="RequestURIBasedDispatcher"
class="org.apache.axis2.engine.RequestURIBasedDispatcher">
<order phase="Transport"/>
</handler>
<handler name="SOAPActionBasedDispatcher"
class="org.apache.axis2.engine.SOAPActionBasedDispatcher">
<order phase="Transport"/>
</handler>
</phase>
<phase name="Security"/>
<phase name="PreDispatch"/>
<phase name="Dispatch" class="org.apache.axis2.engine.DispatchPhase">
<handler name="RequestURIBasedDispatcher"
class="org.apache.axis2.engine.RequestURIBasedDispatcher"/>
<handler name="SOAPActionBasedDispatcher"
class="org.apache.axis2.engine.SOAPActionBasedDispatcher"/>
<handler name="AddressingBasedDispatcher"
class="org.apache.axis2.engine.AddressingBasedDispatcher"/>
<handler name="RequestURIOperationDispatcher"
class="org.apache.axis2.engine.RequestURIOperationDispatcher"/>
```

```
<handler name="SOAPMessageBodyBasedDispatcher"
class="org.apache.axis2.engine.SOAPOutputMessageBodyBasedDispatcher"/>
<handler name="HTTPLocationBasedDispatcher"
class="org.apache.axis2.engine.HTTPLocationBasedDispatcher"/>
</phase>
<!-- System pre-defined phases -->
<!-- After Postdispatch phase module author or service
author can add any phase he or she wants -->
<phase name="MyOwnPhase"/>
<phase name="soapmonitorPhase"/>
</phaseOrder>
```

The attribute `type` represents the type of message flow and can be only one of the following kinds:

- In-Flow
- Out-Flow
- In-FaultFlow
- Out-FaultFlow

The following diagram shows a sample message flow with default phases:



In addition to the `<handler>` child element allowed inside the `phaseOrder`, the other possible element is the `<phase>` element. It represents available phases in the execution chain. Following is the right way to specify phases inside `phaseOrder`:

```
<phase name="Transport"/>
```

In this example, `name` is the name of the phase.

There are two types of phases: pre-defined ones (required by Axis 2) and user-defined ones. In addition to the pre-defined phases, a user can add user-defined phases to the flows in the configuration file. For the in-flow, user-defined phases can be added only after the `PostDispatch` pre-defined phase. This is because the Axis 2 engine keeps the flow related information attached to the operations, and it is only after the `Dispatch` phase that the operation is found. There are no rules restricting the placement of user-defined phases in the out-flow. If any of the user-defined phases has the same name as that of a pre-defined phase, the engine fails to operate.

Web Services Stack 8.0 uses Axis 2 1.4 with the following definition of the default pre-defined system phases for the In-Flows:

- `<phase name="Transport"/>`
- `<phase name="Addressing"/>`
- `<phase name="PreSecurity"/>`
- `<phase name="Security"/>`
- `<phase name="PreDispatch"/>`
- `<phase name="Dispatch"/>`
- `<phase name="RMPhase"/>`
- `<phase name="OperationInPhase"/>`
- `<phase name="soapmonitorPhase"/>`

Add your own phases after the system ones in In-Flows.

Web Services Stack 8.0 uses Axis 2 1.4 with the following definition of default pre-defined system phases for the Out-Flows:

- `<phase name="soapmonitorPhase"/>`
- `<phase name="OperationOutPhase"/>`
- `<phase name="RMPhase"/>`
- `<phase name="PolicyDetermination"/>`
- `<phase name="MessageOut"/>`
- `<phase name="PreSecurity"/>`
- `<phase name="Security"/>`

Add your own phases before the system ones in Out-Flows.



**Important:** Do not change the order of the system pre-defined phases in all four kinds of flows. If you want to add a new phase, you can do that after the system pre-defined phase, as is done in the default *axis2.xml* file.

## 5. Module References

If you want to engage a module system-wide, you can do so by adding a top-level module element in the *axis2.xml* file. Following is an example of how to do it:

```
<module ref="addressing"/>
```

The parameter `ref` in the preceding example is the module name that is going to be engaged system-wide.

## 6. Listeners (Observers)

In Axis 2, *AxisConfiguration* is observable, so that one can register observers into that, and they are automatically informed whenever a change occurs in *AxisConfiguration*. The observers are informed of the following events:

- Deploying a service
- Removing a service
- Activating/Deactivating a service
- Deploying a module
- Removing a module

Registering observers is very useful for additional features (such as RSS feed generation). Those observers provide service information to subscribers. The correct way of registering observers is the following:

```
<listener class="org.apache.axis2.ObserverIMPL">  
  <parameter name="RSS_URL" >http://127.0.0.1/rss</parameter>  
</listener>
```

The parameter `class` represents an observer implementation class. The implementation class must implement *AxisObserver* interface, and the class has to be available in the classpath.

## 7. Message Formatters

The Web Services Stack *axis2.xml* file has defined new message formatters for the following content types to extend the default functionality provided by Axis2:

- ="text/xml"
- ="application/xml"
- "application/soap+xml"

The new definitions are as follows:

```
<messageFormatter contentType="text/xml"
  class="com.softwareag.formatters.RawXMLFormatter" />
<messageFormatter contentType="application/xml"
  class="com.softwareag.formatters.RawXMLFormatter" />
<messageFormatter contentType="application/soap+xml"
  class="com.softwareag.formatters.RawXMLApplicationXMLFormatter"/>
```

## 8. Message builders

The Web Services Stack *axis2.xml* file has defined new message builders for the following content types to extend the default functionality provided by Axis2:

- "text/xml"
- "application/xml"
- "application/soap+xml"

The new definitions are as follows:

```
<messageBuilder contentType="text/xml"
  class="com.softwareag.builders.RawXMLMessageBuilder"/>
<messageBuilder contentType="application/soap+xml"
  class="com.softwareag.builders.RawXMLMessageBuilder"/>
<messageBuilder contentType="application/xml"
  class="com.softwareag.builders.RawXMLMessageBuilder"/>
```

The preceding message builders extend the default functionality provided by Axis 2 and handle some specific scenarios.



**Note:** See the *axis2.xml* file for additional information about the possible settings and their values. The file also contains additional comments and samples.

## Run-Time Configuration

This section provides information on the configuration of global runtime through the *axis2.xml* file.

In Axis 2, there are three kinds of configuration files for configuring the system - the *axis2.xml*, the *services.xml*, and the *module.xml*:

- *axis2.xml* for the global configuration

The file is used for configuration of the client side and the server side of all the deployed web services.

- *services.xml* for the service configuration

The file is used for the configuration of web services and is only contained in service archives (AAR).



**Note:** The service archive (AAR) is in the *WEB-INF\services\* folder.

■ *module.xml* for the module configuration

The file is used for the configuration of modules and is only contained in module archives (MAR).



**Note:** The module archive (MAR) is in the *WEB-INF\modules\* folder.

There are some additional Web Services Stack-specific configuration files:

**1. SMH Agents Configuration Files:**

- The *deployclient.properties* file that is used by SMH for the deployment functionality provided by it
- The *argusagent.properties* file that is used by SMH for the Agents

For information on the *deployclient.properties* file and the *argusagent.properties* file, see the overview of *The Administration Tool*

**2. Client-side Configuration Files:**

- The *wsclientsec.properties* file.

For information on the *wsclientsec.properties* file, see the heading "Client-side Configuration" of *Message-Level Security*.

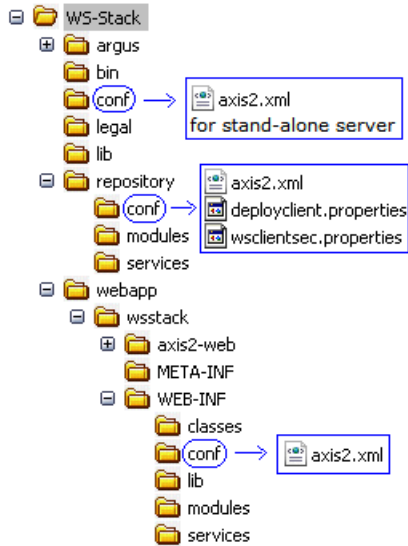
Following are details on the *axis2.xml* configuration and on the proper way of adding and registering elements in *axis2.xml*:

In Axis 2, there are two kinds of *axis2.xml* configuration files - one for the configuration of the web service (server-side config file) and another one for configuration of the client of the web service (client-side config file).

Following is a screenshot of the locations of the occurrences of the *axis2.xml* file.



**Note:** The installation folders of Web Services Stack are under *\SoftwareAG\<Web Services Stack> folder*.



The first location is in the *conf* folder of the Web Services Stack tree. This is the file that is used for configuring a stand-alone server (server-side configuration).

The second location is the *conf* folder of the `<Web Services Stack> folder\repository` folder. This is the file for configurations of the client side of web services (client-side configuration).

The third location is the *conf* folder of the `\<Web Services Stack> folder\webapp\META-INF` folder. This is the file for configurations of the server side of web services (server-side configuration).

- *axis2.xml* files for the server-side configuration;

Two different versions of the *axis2.xml* file are used for configuration of the web service (server-side config file)

The files for the web service (server-side config file) are placed into two different locations:

in `\<Web Services Stack> folder\webapp\WEB-INF\conf` after the Web Services Stack installation (see the preceding screen capture);

in `\<Web Services Stack> folder\conf\` for the configuration of a stand-alone server.

The file for the client of the web service (client-side config file) is usually stored in the so-called client repository - `\<Web Services Stack> folder\repository\conf`.

**Caution:** The *axis2.xml* file contains important information such as the user name and password for the administration console logon:

Systems administrators must change these default credentials and protect access to the *axis2.xml* configuration file because the administration console is the key to many functional operations including activating and deactivating of web services.

## Server-side Configurations

---

Following are details on the HTTPS Deployment configurations performed at server side. These are the configurations for secured access to the Admin and Deploy Servlets.

If you run the Web Services Stack runtime under both HTTP and HTTPS you may also want to disable unsecured HTTP access to the Admin and Deploy Servlets. This can be done by uncommenting the two security constraints in the *WEB-INF/web.xml*.

The functionality of the Deploy Servlet is used in the Eclipse plug-in. For details, see [Deploying and Undeploying a Web Service Package](#).

For details on the Admin Servlet, see *Admin Servlet*.

### ▶ To allow access over HTTPS only

- Modify the *WEB-INF/web.xml* file by uncommenting the following two security constraints:

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>DeployServlet</web-resource-name>
    <url-pattern>/sagdeployer/*</url-pattern>
  </web-resource-collection>
  <user-data-constraint>
    <transport-guarantee>CONFIDENTIAL</transport-guarantee>
  </user-data-constraint>
</security-constraint>
<security-constraint>
  <web-resource-collection>
    <web-resource-name>AxisAdminServlet</web-resource-name>
    <url-pattern>/axis2-admin/*</url-pattern>
  </web-resource-collection>
  <user-data-constraint>
    <transport-guarantee>CONFIDENTIAL</transport-guarantee>
  </user-data-constraint>
</security-constraint>
```



## Client-side Configurations

---

There is only one Web Services Stack-specific configuration at client side.

Following are details on how to take advantage of WS-Security by giving a value to the parameter `securityConfigFile`:

```
<parameter name="securityConfigFile">wsclientsec.properties</parameter>
```

The value of the parameter in the preceding example is the relative (or absolute) path to the *properties* file that contains security-related information.

This file must be part of the client's CLASSPATH.

For more information about the configuration of *wsclientsec.properties* file, see the heading "Client-side Configuration" from [Message-Level Security](#).

## MTOM in Web Services Stack

---

Apache Axis 2 supports Base64 encoding, SOAP with Attachments & MTOM (SOAP Message Transmission Optimization Mechanism).

MTOM is a W3C specification that focuses on solving the "Attachments" problem.

Binary content often has to be re-encoded to be sent as text data with SOAP messages. MTOM allows you to selectively encode portions of the message. In that case, you can send base64 encoded data as well as externally attached raw binary data.

For details of MTOM, see the specification at <http://www.w3.org/TR/2004/PR-soap12-mtom-20041116/>.

Following are some useful configurations that you need to get use of MTOM in Web Services Stack.

The parameter `<parameter name="enableMTOM">...</parameter>` can be set at all permitted levels (global level in *axis2.xml*, service level in *services.xml*, operation level in *services.xml*, etc.). It is up to the user to decide where to apply MTOM and to set that parameter in *axis2.xml* or *service.xml*.

`<parameter name="enableMTOM">...</parameter>` can have three different values:

■ **"true"**

If this value is set, then the engine always responds with MTOM-ized messages in cases of included binary data of schema type "xmime:base64Binary". For example, `<xsd:element minOccurs="0" name="binaryData" type="xmime:base64Binary"/>`.

■ **"false"**

If this value is set, then the response is always non-MTOM-ized, no matter whether the request is MOTM-ized or not.

■ **"optional"**

If this value is set, then the response is MTOM-ized only if the request is MTOM-ized. Otherwise, it is non-MTOM-ized.



**Note:** Setting the value to "optional" prevents you from failures.

# 3 Security

---

- Message-Level Security ..... 16
- Transport-Level Security ..... 24
- Client Authentication ..... 31

Web Services Stack has security facilities for securing the message content, support for HTTP basic authentication, SSL support for securing the communication channel, and user authentication based on Software AG SIN `LoginModules`.

This chapter covers the following topics:

## Message-Level Security

---

This section covers the following topics:

- [Overview](#)
- [Server-side Configuration](#)
- [Client-side Configuration](#)

### Overview

The symmetric message security and the asymmetric message security are both part of the WS-Security specification. Message-level security is applied between the web service client and the web service itself in both directions.

Message-level security secures the message content itself, but it does not secure the communication channel. This is in contrast to transport-level security, where the communication channel is secured. To apply message security, you have to make several configurations on both the client side and the server side.

There are many different ways you can configure message-level security, based on your security needs. For examples, see [Web Services Security: SOAP Message Security 1.1](#) and [WS-Security Policy Language](#).

Web Services Stack provides an Eclipse plug-in graphical user interface that can be used to create the needed security configuration. You can install the plug-in in Eclipse and use it to create web service archives (that is, AAR archives). For more information, see [Eclipse Plug-in](#).

Security configurations in Web Services Stack are based on [WS-Security Policy specification](#).

## Server-side Configuration

You can configure the server side by changing the properties in the *services.xml* file

You need a *keystore* file that contains the X.509 certificate of the server. It may also contain client public keys.

You can re-use initialized (loaded) keystore instances by caching them the first time they are loaded. Any other configuration (such as key aliases and password callback handlers) will be retrieved from the Rampart configuration separately for every invocation.

You have the option to set caching globally in the *axis2.xml* for all services that are deployed in Web Services Stack runtime, or for a service, service group, specific operation or message in the *services.xml* descriptor of your service. However, keep in mind that keystore caching is per message because the keystore configuration itself may be different for each message.

### ▶ To enable keystore caching

- Set the parameter `cacheCryptoInstances` to “true”.

```
<parameter name="cacheCryptoInstances">true</parameter>
```



**Note:** When a service is undeployed or simply stopped any cached keystores will be removed. You can deactivate (stop) services using the *Administration module* or deactivate and undeploy (delete completely the service and all its files) using the *Administration Tool* of the System Management Hub.

Depending on the security policy, the client may be required to send the token used for encryption signature within the message itself. Thus, there is no need to have all client certificates at the server side. Rampart, however, still verifies whether the certificates are trustworthy and therefore requires that at least the certificate of the issuer is present in the truststore. In this case, instruct Rampart/WSS4J (used to sign the request) to use the client’s certificate.

Following is the value assigned to the `<encryptionUser>` field:

```
<encryptionUser>useReqSigCert</encryptionUser>
```

“useReqSigCert” is a special fictional encryption user that is recognized by the security module. In this case, your certificate (that is used to verify your signature) is used for the encryption of the response. Thus, it is possible to have only one configured encryption user for all clients that access the service.

Look at the following example of symmetric binding security configuration in the services.xml file:

```
<ramp:encryptionUser>client</ramp:encryptionUser>
```

The original value "client" is in fact an example of an alias for a client's certificate that has to be stored into the keystore used at server side.

If you want to authenticate a client who uses a user name token, you have to provide a password callback handler class to validate the user name and the password received from the client.

When you provide a password using the callback handler class, you make a check towards a given authentication module. The module may be a JAAS module, or some other one.



**Note:** This authentication mechanism applies to the user name security token and is used in a similar way with other security tokens, too.

The keystore properties can be configured by adding a Rampart custom policy assertion to the services.xml file. Following is an example of symmetric binding security configuration in the services.xml file:

```
<wsp:Policy wsu:Id="User defined"
xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd">
  <wsp:ExactlyOne>
    <wsp>All>
      <sp:SymmetricBinding
xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
        <wsp:Policy>
          <sp:ProtectionToken>
            <wsp:Policy xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy">
              <sp:X509Token
sp:IncludeToken="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy/IncludeToken/Never">
                <wsp:Policy>
                  <sp:WssX509V3Token10/>
                  <sp:RequireDerivedKeys/>
                </wsp:Policy>
              </sp:X509Token>
            </wsp:Policy>
          </sp:ProtectionToken>
          <sp:AlgorithmSuite
xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
            <wsp:Policy>
              <sp:Basic128/>
            </wsp:Policy>
          </sp:AlgorithmSuite>
          <sp:Layout>
            <wsp:Policy>
```

```

        <sp:Strict/>
        </wsp:Policy>
        </sp:Layout>
    <sp:IncludeTimestamp/>
        </wsp:Policy>
        </sp:SymmetricBinding>
    <sp:Wss10 xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
        <sp:Policy>
            <sp:MustSupportRefKeyIdentifier/>
            <sp:MustSupportRefIssuerSerial/>
        </sp:Policy>
    </sp:Wss10>
    <sp:SignedSupportingTokens
xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
        <wsp:Policy/>
    </sp:SignedSupportingTokens>
    <ramp:RampartConfig xmlns:ramp="http://ws.apache.org/rampart/policy">
        <ramp:user>service</ramp:user>
        <ramp:encryptionUser>client</ramp:encryptionUser>

    <ramp:passwordCallbackClass>com.softwareag.wsstack.pwcb.PasswordCallbackHandler</ramp:passwordCallbackClass>
        <ramp:signatureCrypto>
            <ramp:crypto provider="org.apache.ws.security.components.crypto.Merlin">
                <ramp:property
name="org.apache.ws.security.crypto.merlin.keystore.type">JKS</ramp:property>
                <ramp:property
name="org.apache.ws.security.crypto.merlin.file">service.jks</ramp:property>
                <ramp:property
name="org.apache.ws.security.crypto.merlin.keystore.password">openssl</ramp:property>
            </ramp:crypto>
        </ramp:signatureCrypto>
        <ramp:encryptionCrypto>
            <ramp:crypto provider="org.apache.ws.security.components.crypto.Merlin">
                <ramp:property
name="org.apache.ws.security.crypto.merlin.keystore.type">JKS</ramp:property>
                <ramp:property
name="org.apache.ws.security.crypto.merlin.file">service.jks</ramp:property>
                <ramp:property
name="org.apache.ws.security.crypto.merlin.keystore.password">openssl</ramp:property>
            </ramp:crypto>
        </ramp:encryptionCrypto>
    </ramp:RampartConfig>
    </wsp:All>
</wsp:ExactlyOne>
</wsp:Policy>

```

You can configure the *keystores* for signing and encrypting using the custom Rampart configuration (the code listing in bold in the preceding example).

Configurations on the server side can also be done using the Web Services Stack Eclipse plug-ins. With the Eclipse plug-in, you can complete the preceding configuration using graphical user interface.

For more information, see [Eclipse Plug-in, Web Services Security: SOAP Message Security 1.1](#), and [WS-Security Policy Language](#).

## Client-side Configuration

When you use the client API to invoke web services that require security, you can specify security configuration settings through a properties file.

Specify in the client *axis2.xml* configuration file the file name and the path to it. This file must be a part of the client's CLASSPATH file. This file holds the required parameters for the client configuration, that is, the `securityConfigFile` parameter:

```
<parameter
name="securityConfigFile">D:/wsdev/SampleWSClient/wsclientsec.properties
</parameter>
```

If you do not define such a parameter, the client implementation looks for a *wsclientsec.properties* file in the current working directory.

If a `securityConfigFile` parameter exists but the file specified cannot be found, you get an exception. If the parameter is not defined or a *wsclientsec.properties* file is not present in the current working directory, then the configuration loading routine does not throw any exceptions.



**Note:** The loading of the security configuration settings takes place only if the web service policy contains security assertions (that is only if the security module is engaged).

Following is the list of the supported configuration keys:

Key	Description
USERNAME	This is the user's name. It is used by the WS-Security functions for the following: <ul style="list-style-type: none"> <li>■ The UsernameToken function sets this name in the UsernameToken.</li> <li>■ The signing function uses this name as the alias name in the <i>keystore</i> to get the certificate to perform signing.</li> <li>■ The encryption function uses this parameter as fallback if ENCRYPTION_USER is not set.</li> </ul>
ENCRYPTION_USER	The user's name for encryption. The encryption function uses the public key of the user to generate a symmetric key. If this parameter is not set, then the encryption function uses the user's name to get the certificate.



Key	Description
USER_CERTIFICATE_ALIAS	The alias of the key pair in the keystore, to get the private key used for signing. If the property is not set, the function falls back to USERNAME property.
STS_ALIAS	The STS alias is used as an encryption user in case of a STS authentication.
POLICY_VALIDATOR_CLASS	The policy validator callback class is responsible for validating the security policy. The default callback class is the org.apache.rampart.PolicyBasedResultsValidator.
TIMESTAMP_PRECISION_IN_MS	Defines whether timestamp precision is in milliseconds.  The default value is TRUE.  The expected values are TRUE or FALSE.  This parameter is passed to wss4j WSSConfig. The setting concerns the timestamp precision in the security header. If the precision is set to be in milliseconds, the timestamp in the security header is written in the following simple date format: yyyy-MM-dd'T'HH:mm:ss.SSSZ.
TIMESTAMP_TTL	Timestamp time-to-live in seconds. An integer value is expected. Default value is 300.
TIMESTAMP_MAX_SKEW	Used in timestamp validation where the timestamp creation timestamp is not available. The max time skew is an integer and is expected in seconds. Default value is 300.
PASSWORD_CALLBACK_HANDLER_CLASS	A class that implements the javax.security.auth.callback.CallbackHandler interface. The default module loads the class and calls the callback method to get the password. The class is expected to have no parameters.
OPTIMIZE_PARTS_EXPRESSIONS	A list of Xpath expressions that refer to nodes that must be MTOM-optimized. The property is a delimited list of Xpath expressions. Note that if this property is set, it overrides the default list of expressions and does not add them to the list.
OPTIMIZE_PARTS_NAMESPACES	A list of namespaces that is taken into consideration when searching for nodes. The property is a delimited list of namespaces. Essential for the correct retrieval of the nodes from the document, that are not in the default list. The OPTIMIZE_PARTS_EXPRESSIONS list are recognized by the optimizer. The default list of namespaces is:  <pre> xmlns:ds=http://www.w3.org/2000/09/xmldsig# xmlns:xenc=http://www.w3.org/2001/04/xmenc# xmlns:wsse=http://docs.oasis-open.org/wss/2004/01/oas-200401-wss-wssecurity-secext-200401- xmlns:wsu=http://docs.oasis-open.org/wss/2004/01/oas-200401-wss-wssecurity-secext-200401- </pre> plus all the declared namespaces here. That property is expected as a delimited list of namespace declarations (for example, OPTIMIZE_PARTS_NAMESPACES=xmlns:ds=http://www.w3.org/2000/09/xmldsig# xmlns:xenc=http://www.w3.org/2001/04/xmenc# ).  <b>Note:</b> If this property is set, it overwrites any previously configured list of namespaces.
CRYPTO_PROVIDER_SIGN	The WSS4J-specific Crypto implementation that is to be used for generating signatures. The following implementations are supported: <ul style="list-style-type: none"><li>■ org.apache.ws.security.components.crypto.Merlin is the default implementation.</li><li>■ org.apache.ws.security.components.crypto.BouncyCastle is the BouncyCastle implementation.</li></ul>

Key	Description
KEYSTORE_PROVIDER_SIGN	The signature keystore provider.  If not set the JVM uses the default (normally Sun's) keystore provider. For additional information, refer to <code>java.security.Provider</code> javadocs.
KEYSTORE_TYPE_SIGN	The signature keystore type. If not set, the JVM uses the default keystore type (normally <code>JKS</code> ). For additional information, refer to the <code>java.security.KeyStore#getDefaultType()</code> method javadocs.
KEYSTORE_FILE_SIGN	The signature keystore file.
KEYSTORE_PASSWORD_SIGN	The signature keystore password.
CRYPTO_PROVIDER_ENCRYPT	The WSS4J-specific Crypto implementation to use for encryption. It can be set to one of the following:  <ul style="list-style-type: none"> <li>■ <code>org.apache.ws.security.components.crypto.Merlin</code> This is the default one if the property is not set.</li> <li>■ <code>org.apache.ws.security.components.crypto.BouncyCastle</code></li> </ul>
KEYSTORE_PROVIDER_ENCRYPT	The encryption keystore provider. If not set the JVM uses the default (normally Sun's) keystore provider. For additional information, refer to <code>java.security.Provider</code> javadocs.
KEYSTORE_TYPE_ENCRYPT	The encryption keystore type. If not set, the JVM uses the default keystore type (normally <code>JKS</code> ). For additional information, refer to <code>java.security.Provider</code> javadocs.
KEYSTORE_FILE_ENCRYPT	The encryption keystore file.
KEYSTORE_PASSWORD_ENCRYPT	The encryption keystore password.
CRYPTO_PROVIDER_STS	The WSS4J-specific Crypto implementation to use for protection in case of a STS. It can be set to one of the following:  <ul style="list-style-type: none"> <li>■ <code>org.apache.ws.security.components.crypto.Merlin</code> is the default one if the property is not set.</li> <li>■ <code>org.apache.ws.security.components.crypto.BouncyCastle</code></li> </ul>
KEYSTORE_PROVIDER_STS	The keystore provider used in case of a STS. If not set the JVM uses the default (normally Sun's) keystore provider. For additional information, refer to <code>java.security.Provider</code> javadocs.
KEYSTORE_TYPE_STS	The keystore type used in case of a STS. If not set, the JVM uses the default keystore type (normally <code>JKS</code> ). For additional information, refer to the <code>java.security.KeyStore#getDefaultType()</code> method javadocs.
KEYSTORE_FILE_STS	The keystore file used in case of a STS.
KEYSTORE_PASSWORD_STS	The keystore password used in case of a STS.
SSL_KEYSTORE_TYPE	The type of the <i>keystore</i> specified under <code>KEYSTORE_SSL_LOCATION</code> .
SSL_KEYSTORE_PASSWORD	The password for the <i>keystore</i> specified under <code>KEYSTORE_SSL_LOCATION</code> . This property corresponds to the <code>javax.net.ssl.keystorePassword</code> system property.
KEYSTORE_SSL_LOCATION	The <i>keystore</i> file for SSL authentication. This property corresponds to the <code>javax.net.ssl.keystore</code> property.  For more information, refer to the <a href="#">JSSE Reference Guide</a> .

Key	Description
	<b>Note:</b> Specifying the <i>keystore</i> is required only if the remote SSL server
TRUSTSTORE_SSL_LOCATION	The truststore file for SSL authentication. The client requires that the s it is trusted. This property corresponds to the JSSE <code>javax.net.ssl</code> not set the client falls back to <code>&lt;java-home&gt;lib/security/jssecacerts</code> and <code>&lt;ja</code>  <b>Note:</b> For more information, refer to the <a href="#">JSSE Reference Guide</a> .
TRUSTSTORE_SSL_PASSWORD	The password for the truststore specified under <code>TRUSTSTORE_SSL_LO</code> <code>javax.net.ssl.trustStorePassword</code> system property.  <b>Note:</b> For more information, refer to the <a href="#">JSSE Reference Guide</a> .



**Note:** The last five entries actually refer to transport-level security configuration (SSL settings).

The configuration loading routine puts all those entries in the client options. Thus, you can overwrite any particular option every other time Rampart is to be executed. For example, all security keys can be specified programmatically using the Web Services Stack client options:

```
//create the WS Stack client:
IWSStaxClient client = ...

...

IWSOptions options = client.getWSOptions();

options.setProperty(WSCliientConstants.KEYSTORE_PASSWORD_SIGN, "changeit");
options.setProperty(WSCliientConstants.KEYSTORE_FILE_SIGN, "C:\\client.jks");

//set the options, overwriting the ones loaded from the wsclientsec.properties //file:
client.setOptions(options);

//execute the client
client.sendReceive(...);
```

The Rampart is afterwards configured through a Rampart assertion that is generated by the `RampartConfigLoader` handler. The Web Services Stack client takes care of engaging that handler if Rampart itself is engaged. The function of the `RampartConfigHandler` is basically to gather all the security configuration keys, build up the Rampart configuration assertion, and put it as a property in the message context options where Rampart can find it.

## Transport-Level Security

---

This section covers the following topics:

- [Prerequisites for the Setup and Use of Transport-Level Security](#)
- [SSL with Client Authentication](#)
- [Setup and Use of HTTP Basic Authentication](#)

### Prerequisites for the Setup and Use of Transport-Level Security

Transport-level security addresses the problem of securing web service conversation by securing the communication channel instead of the message data itself. Although the web service security policy specification does not state that the transport-level security requires the use of HTTP transport over SSL, it is the most typical use case.

This section provides details on the configuration and usage of web service communication over HTTPS.

To enable transport-level security, configure your application server to use SSL:

#### ▶ To configure Tomcat to use SSL at server side

- 1 Navigate to `<TOMCAT_INSTALL_DIR>\conf` and open the `server.xml` file to configure an SSL Connector.



**Note:** `<TOMCAT_INSTALL_DIR>` is the path where your copy of Tomcat is installed.

- 2 The configured scheme needed for the SSL communication is `https`. The required parameters are listed in the following table:

Property name	Description
KEYSTORE_FILE_PATH	The path to the keystore file that is used by Tomcat to decrypt the requests and encrypt the responses.
KEYSTORE_PASSWORD	The password that protects the keystore.
ENCRYPTION_KEY_ALIAS	The alias that identifies the key pair in the keystore (in case there is more than one public-private key pair in the keystore).

Following is a sample code listing for an SSL connector configuration:

```
<Connector port="8443" maxHttpHeaderSize="8192" maxThreads="150"
  minSpareThreads="25" maxSpareThreads="75" enableLookups="false"
  disableUploadTimeout="true" acceptCount="100" scheme="https" secure="true"
  clientAuth="false" sslProtocol="TLS"
  keystoreFile="<KEYSTORE_FILE_PATH>"
  keystorePass="<KEYSTORE_PASSWORD>"
  keyAlias="<ENCRYPTION_KEY_ALIAS>"/>
```



**Note:** The value of the connector `port` is "8443" because this is the standard Tomcat HTTPS port.

If a server declares explicitly the use of the HTTPS transport in its *services.xml*, you have to define that such a transport listener is defined in the *axis2.xml* configuration file. Otherwise, the runtime throws an exception while trying to deploy the respective service.

As of Web Services Stack ver. 8.x, the `com.softwareag.wsstack.transport.http.HTTPListener` and the `com.softwareag.wsstack.transport.http.HTTPSListener` are introduced to solve the problem when Web Services Stack is served by a servlet container.

In that case, the actual transport receiver is the `SAGAdminServlet` that registers itself as an HTTP listener only. However, you still need to have an HTTPS listener configured in the *axis2.xml* file. The `com.softwareag.wsstack.transport.http.HTTPSListener` is a full functional replacement of the previously available `org.apache.axis2.transport.http.HTTPSListener` in Web Services Stack 1.2.

To use the new HTTPS listener, uncomment the respective configuration in the *axis2.xml* file shipped with Web Services Stack. Following is a sample configuration:

```
<transportReceiver name="https"
class="com.softwareag.wsstack.transport.http.HTTPSListener">
  <parameter name="port">8443</parameter>
</transportReceiver>
```

In this code listing, the configured port number must be set to the port configured for the HTTPS connector of the servlet container. The newly configured `HTTPSListener` is responsible for generating correct endpoint addresses in the WSDL and supplying a `SessionContext`, although the actual requests are served by the `SAGAdminServlet`.

In this case, the SSL configuration uses only server authentication (see `clientAuth="false"` in the preceding configuration) and the client encrypts automatically the requests with the server public key.

**▶ To configure SSL at the client side**

- 1 The client must send a request against HTTPS endpoint with a port that is equal to the one specified at server side (that is "8443").
- 2 Set the properties in your security configuration file. You can configure this file as a parameter in the *axis2.xml* configuration file:

```
<parameter
name="securityConfigFile">your client security config file path
</parameter>
```

For information on the *axis2.xml* configuration file, see [Web Services Stack Runtime](#).

If you do not define a security configuration file, the client uses information in the *wsclient-sec.properties* file in the current working directory.

Or:

Use the Web Services Stack client API to set the required properties:

```
//create the WS Stack client:
IWStaxClient client = ...

...

IWSOptions options = client.getWSOptions();

options.setProperty(WSClientConstants.KEYSTORE_PASSWORD_SIGN, "changeit");
options.setProperty(WSClientConstants.KEYSTORE_FILE_SIGN, "C:\\client.jks");

//set the options, overwriting the ones loaded from the wsclientsec.properties
//file:
client.setOptions(options);

//execute the client
client.sendReceive(...);
```

The following security properties at the client side relate to the SSL configuration:

Property name	Description
SSL_KEYSTORE_TYPE	The type of the keystore specified under the KEYSTORE_SSL_LOCATION.
SSL_KEYSTORE_PASSWORD	The password for the keystore specified under KEYSTORE_SSL_LOCATION. This property corresponds to the JSSE <code>javax.net.ssl.keyStorePassword</code> system property.
KEYSTORE_SSL_LOCATION	The keystore file for SSL authentication. This property corresponds to the JSSE <code>javax.net.ssl.keyStore</code> system property. Note that specifying the keystore is required only if the remote SSL server requires client authentication.  For more information, refer to the <a href="#">JSSE Reference Guide</a> .
TRUSTSTORE_SSL_LOCATION	The truststore file for SSL authentication. The client requires that the server's certificate is installed in this truststore and it is trusted. This property corresponds to the JSSE <code>javax.net.ssl.trustStore</code> system property. If the property is not set, the client falls back to <code>&lt;java-home&gt;lib/security/jssecacerts</code> and <code>&lt;java-home&gt;/lib/security/cacerts</code> in that order.  For more information, refer to the <a href="#">JSSE Reference Guide</a> .
TRUSTSTORE_SSL_PASSWORD	The password for the truststore specified under TRUSTSTORE_SSL_LOCATION. This property corresponds to the <code>javax.net.ssl.trustStorePassword</code> system property.  For more information, refer to the <a href="#">JSSE Reference Guide</a> .

## SSL with Client Authentication

### 1. Server-side Configuration

The Tomcat web server may also be configured to use a client certificate to encrypt the transferred data.

#### ► To use client authentication with Tomcat

Set the following parameters in the HTTPS connector settings in the Tomcat `server.xml` configuration file.

- 1 Set `clientAuth` to "true".
- 2 Set the keystore properties.
- 3 Set the truststore properties.



**Important:** You can also configure the truststore location of Tomcat by starting it with the respective Java system property, because if the truststore properties are not set in your configuration, Tomcat uses the default Java trusted authority keystore.

▶ **To configure the truststore location of Tomcat by starting it with the respective Java system property**

- Add the following options when starting Tomcat:

```
-Djavax.net.ssl.trustStore=your_path_to/truststore.jks
-Djavax.net.ssl.trustStorePassword=your_password
)
```

Use the following settings to configure the truststore properties in the HTTPS connector:

Property name	Description
truststoreFile	The TrustStore file to use to validate client certificates.
truststorePass	The password to access the truststore. This defaults to the value of keystorePass.
truststoreType	Add this element if your are using a different format for the truststore than you are using for the keystore. The keystoreType defaults to "JKS".

Look at the following example to see a sample of the connector configuration:

```
<Connector port="8443" maxHttpHeaderSize="8192"
maxThreads="150" minSpareThreads="25" maxSpareThreads="75"
enableLookups="false" disableUploadTimeout="true"
acceptCount="100" scheme="https" secure="true"
clientAuth="true" sslProtocol="TLS"
keystoreFile="<KEYSTORE_FILE_PATH>"
keystorePass="<KEYSTORE_PASSWORD>"
keyAlias="<KEY_ALIAS>"
truststoreFile="<TRUSTSTORE_FILE_PATH>"
truststorePass="<TRUSTSTORE_PASSWORD>"
truststoreType="<TRUSTSTORE_TYPE>" />
```



**Note:** If you encounter a problem with a service that declares the usage of the HTTPS transport in its *services.xml* descriptor, uncomment the respective HTTPS listener configuration in the *axis2.xml* file shipped with Web Services Stack. For details, see how to configure Tomcat to use SSL at server side in [Prerequisites for the Setup and Use of Transport-Level Security](#)



## 2. Client-side Configuration

It is also possible to use client certificate with the Web Services Stack client, although additional work is needed to use the Java 1.4 compatible HTTP sender (utilizing the Jakarta Commons `HttpClient` component). In order to make the Commons `HttpClient` use client certificate for the encryption one needs to register a new HTTPS socket factory since the default one does not handle the case with the client certificate. The Commons `HttpClient` library does not provide the appropriate socket factory implementation but there is one in the `contrib` package (`commons-httpclient-contrib`) that is part of the `commons-httpclient` project, namely `AuthSSLProtocolSocketFactory`. This can be set in the following way:

```
IWSStaxClient client = ...
...
ProtocolSocketFactory socketfactory = new AuthSSLProtocolSocketFactory(
    new File("keystore.jks").toURL(),
    "keystorePassword",
    new File("truststore.jks").toURL(),
    "truststorePassword");
Protocol authhttps = new Protocol("https", socketfactory, 8443);
client.getWSOptions().setProperty(HTTPConstants.CUSTOM_PROTOCOL_HANDLE,
    authhttps);
```

### Setup and Use of HTTP Basic Authentication

To use the HTTP basic authentication, you must configure a specific endpoint that is to use the HTTP basic authentication scheme. The needed configuration is not server-specific and is part of the J2EE specification.

#### ► To configure a specific endpoint to use the HTTP basic authentication scheme

- 1 Open your web application descriptor of Web Services Stack (the `web.xml` file that is located in the `\webapps\wsstack\WEB-INF` directory).
- 2 Add a security constraint for a particular URL.


In the following sample code listing, the constraint is the web service endpoint. The relative URL that you are securing is `<url-pattern>/services/ut_asym_xpath</url-pattern>`:

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>Protected Pages</web-resource-name>
    <url-pattern>/services/ut_asym_xpath</url-pattern>
    <http-method>POST</http-method>
    <http-method>GET</http-method>
  </web-resource-collection>
  <auth-constraint>
    <role-name>manager</role-name>
  </auth-constraint>
```

```

</security-constraint>
<security-role>
  <role-name>tomcat</role-name>
  <description>Web Services Stack user realm</description>
</security-role>
<login-config>
  <auth-method>BASIC</auth-method>
  <realm-name>Basic Authentication</realm-name>
</login-config>

```


 **Note:** Set `<role-name>tomcat</role-name>` to configure a role that ensures client authentication. This configuration is server-specific.

The `http` methods from the preceding code listing can be used for different purposes.

 **Note:** For more information, see [Java™ Servlet Specification version 2.4](#).

With the `<http-method>` tag, you can list the "http" methods that require authentication. By removing `<http-method>GET</http-method>`, you can access the ?wsdl formed URL without authentication (for example, `http://myhost:port/MyWebContext/MyService?wsdl`). If no `http` methods are listed explicitly in the configuration, all `http` methods require authentication by default.

When using Tomcat, you can use the HTTP basic authentication with `tomcat-users.xml`.

 **Note:** Software AG Tomcat Kit is not installed in one location as with any other arbitrary installation of Apache Tomcat and does not contain a default `tomcat-users.xml`. If `tomcat-users.xml` is missing in your Tomcat configuration directory (`\Documents and Settings\All Users\Application Data\Software AG\Tomcat\v5.5\conf`), you must create it.

#### ▶ To use the HTTP basic authentication with `tomcat-users.xml`

- 1 Define the role name in the `tomcat-users.xml`.

Following is a sample of a `tomcat-users.xml` file:

```

<?xml version='1.0' encoding='utf-8'?>
<tomcat-users>
</tomcat-users>

```

 **Note:** For information on how roles are defined in Tomcat, see Tomcat documentation.

- 2 Restart Tomcat for the changes to take effect.

Once you have configured your endpoint for HTTP basic authentication, you can configure your web service client, so that it is aware of the changes.

► **To configure your web service client to use HTTP basic authentication**

- 1 Supply the `HttpTransportProperties.Authenticator` object
- 2 Set the user name to “tomcat”.
- 3 Set the password to “tomcat”.
- 4 Set this configuration as an option of the web service client.

Following is a sample code listing of a web service client implementation when you want to use HTTP basic authentication:

```
IWSStaxClient client = (IWSStaxClient)WSCClientFactory.newClient(
    WSCClientConstants.STAX_WSCLIENT,
    "C:/ut_asym_xpath.wsdl",
    null,
    null,
    "C:\\Software AG\\WS-Stack\\repository");

HttpTransportProperties.Authenticator auth = new
    HttpTransportProperties.Authenticator();
auth.setUsername ("tomcat");
auth.setPassword ("tomcat");
auth.setPreemptiveAuthentication (true);

IWSOptions options = client.getWSOptions();
options.setProperty(
    org.apache.axis2.transport.http.HTTPConstants.AUTHENTICATE,auth);
client.setOptions(options);
```



**Important:** In the above example, you must supply the `HttpTransportProperties.Authenticator` object first, and then set up a user name and a password according to the configured Tomcat role. Finally, you need to set this configuration as an option of the web service's client.

## Client Authentication

Web Services Stack provides a mechanism for authenticating clients in Web Services Stack runtime layer using the common Java Authentication and Authorization Service (JAAS) security framework.

Software AG Security Infrastructure (SIN) provides you with an implementation of JAAS `LoginModules` for client authentication.

When you log on using JAAS `LoginContext`, a `javax.security.auth.Subject` is produced. That subject contains user principals and credentials and is available to anyone on the execution chain through the message context.

Web Services Stack collects all available security credentials from the client request and populates them in `SIN SagCredentials`. After that, the logon process is performed in the policy validator implementation of Rampart.

The information is organized under the following headings:

- [JAAS Configuration](#)
- [Security Credentials](#)
- [Implementation of Password Callback Handlers](#)
- [Implementations of Policy Validation Callbacks](#)
- [Authentication Steps](#)

### JAAS Configuration

Before you can log on, you must configure JAAS. For information about the JAAS configuration file, see SIN documentation in the *webMethods Product Suite 8.0* folder on the [Software AG Documentation Web site](#).

### Security Credentials

There are two types of user credentials that are used for authentication in Web Services Stack:

#### ■ Transport-level credentials

Transport-level credentials refer to the communication channel used for the message exchange and are specific for the respective transport that is used. Web Services Stack extracts those credentials from the HTTP(S) transport only:

- User name and password, in the case of a basic HTTP authentication
- A client certificate chain in the case of a client certificate used for encryption of the transferred data

#### ■ Message-level credentials

In the case of message-level credentials, Web Services Stack can extract those from the SOAP security header:

- A user name and a password if you use `UsernameToken` with plain text password
- `X509Certificate` used for the signatures if there are signed parts or elements in the message

## Implementation of Password Callback Handlers

You use password callback handlers in particular identifiers to provide passwords, which are used in the following use cases:

- Passwords that are needed by the Web Services Stack security module to build user name tokens and to create signatures to send messages.
- Passwords that are required for validation of incoming username tokens and decryption of the content of incoming messages.

The callback handlers can retrieve passwords from configuration files, data bases, LDAP servers, or other application components which are used for user management (for example Security Infrastructure).

Web Services Stack has a predefined set of password callback handlers, which facilitate different scenarios for retrieving passwords. You can use these handlers directly or you can develop your own password callback handlers out of them. The following password callback handlers are available:

- `com.softwareag.wsstack.pwcb.ConfigFilePasswordCallbackHandler`

The password callback handler retrieves identifier-password pairs from a configuration file and then loads the pairs which can be used to find the needed password for a particular identifier. The configuration file must be in XML format and similar to the configuration file of the Web Services Stack (*axis2.xml*). You can provide a configuration file to the callback handler as follows:

- You can specify the configuration file in the Web service archive. In the *services.xml* file, you add a `PWCBConfigFile` parameter, which is set to point to the configuration file resource on the service class path. The class path includes the service archive, the libraries which are in the service archive, the web application class path (all jar files in *WEB-INF/lib* and the *WEB-INF/classes* class folder) and so on.

```
<serviceGroup>
  <service name="Sample_Web_Service">
    <parameter name="PWCBConfigFileLocation"> configuration_file_location
  </parameter>
  ...
</service>
</serviceGroup>
```

- If you do not specify the configuration file resource, by default the callback handler searches for a resource with name *users.xml* in the service class path. If it is not available, a `FileNotFoundException` is thrown.

The same password callback handler is also available at the client side if there is no service archive. Then, presumably, the configuration file is *users.xml* and is searched on the class path of the client. Then it is loaded as a resource.

- `com.softwareag.wsstack.pwcb.LdapPasswordCallbackHandler`

The password callback handler retrieves identifier-password pairs from an LDAP server and then loads the pairs which can be used to find the needed password for a particular identifier. To retrieve data from the server, you set the URL of the LDAP server as well as some more properties in the handler. These properties are passed to the handler in a common properties file. You can provide a common properties file to the callback handler as follows:

- You can specify the location of the common properties file in the Web service archive. In the *services.xml* file, you add a `PWCBLDAPPPropFile` parameter, which is set to point to the location of the properties file. The location of the file can be any valid path from which the handler can load the file (for example, *conf/my-ldap.properties*).

```
<serviceGroup>
  <service name="Sample_Web_Service">
    <parameter name="PWCBLDAPPPropFileLocation"> common_properties_file_location
  </parameter>
  ...
</service>
</serviceGroup>
```

- You can set the name of the common properties file to *ldap.properties* and place the file into the root directory of the Web service archive. If you do not provide an explicit properties file in the *services.xml* file, the password callback handler is configured to use a default properties file (*ldap.properties*) from the root directory.
- You can set the name of the common properties file to *ldap.properties* and place the file into a Java archive (.jar file) which resides in the */WEB-INF/lib* or *WEB-INF/classes* directories (for example, *pwcb-server.jar*). If the password callback handler does not discover the properties file in a pre-set directory, or in the root directory of the Web service archive, it searches for the file into a central location on the class path of the handler and loads the properties file as a resource. If this process is unsuccessful, a `FileNotFoundException` is thrown.

The same password callback handler is also available at the client side if there is no service archive. Then, presumably, the configuration file is *users.xml* and is searched on the class path of the client. Then it is loaded as a resource.

The Web Services Stack installation contains samples of custom password callback handlers (on provider and consumer sides) which retrieve identifier-password pairs from configuration files or LDAP servers. You can find the samples in the following location: */samples/PWCBHandlersExample*. The folder contains the following projects:

- `PWCBHandlersExampleServices`

This project enables you to create custom password callback handlers for the sample Web service providers. The sample Web service archives are ready to be deployed on the application server.

#### ■ `PWCBHandlersExampleClient`

This project enables you to create custom client password callback handlers for the consumption of the sample Web service providers. The consumer samples are ready to be executed standalone and do not require an application server which hosts the Web service providers. However, you can modify the clients to discover and consume Web services that reside and are configured on a remote application server. For more information, see the *readme.txt* file which is in the consumer project.

Note that these sample projects enable you to create custom password callback handlers. They do not enable you to reuse handlers which retrieve identifier-password pairs from a common central location.

## Implementations of Policy Validation Callbacks

In the *wsstack-jaas.jar* module, there are ready-to-use policy validator implementations that may be configured and used easily to log on.

Following are examples of those implementations:

- `com.softwareag.wsstack.jaas.callback.SimpleSINPolicyValidatorCallback`  
Attempts to log on with all available credentials (message-level credentials are with higher priority over transport-level credentials) against the JAAS logon context. Specify the login context name as a parameter under the key `sin.jaas.login.context`. The resulting JAAS login subject is available as a property of the message context under the key `sin.jaas.subject`.
- `com.softwareag.wsstack.jaas.callback.ServletRequestLoginPolicyValidatorCallback`  
Attempts to log on using the servlet request resource populated in the SIN credentials list. Specify the login context name as a parameter under the key `sin.jaas.login.context`. The resulting JAAS logon subject is available as a property of the message context under the key `sin.jaas.subject`.
- `com.softwareag.wsstack.jaas.callback.MultiLoginPolicyValidatorCallback`  
Attempts to log on first with transport-level credentials and then again with message-level credentials. Specify the login context name as a parameter under the key `sin.jaas.login.context`. The name of the transport login context is available as a parameter under the key `sin.jaas.transport.login.context` (the default value is `WSS_Transport_IS`) and for message-level credentials logging on under `sin.jaas.msg.login.context` (the default value is `WSS_Message_IS`). The resulting subjects are respectively populated as properties of the message context under the keys `sin.jaas.transport.subject` and `sin.jaas.msg.subject`.

These policy validator callbacks extend the standard callback that is provided by Rampart. This means that all basic functionality for validating security policy conformation is still present.



**Note:** To use one of the preceding callbacks, specify the `policyValidatorCbClass` in the Rampart policy assertion.

## Authentication Steps

This section provides you with guidelines on the authentication steps when you use SIN in Web Services Stack.

### ▶ To authenticate using SIN

You must include the path to SIN JAR in the classpath. All classes that are used in the JAAS configuration file must also be set in the classpath.

- 1 Configure the JAAS configuration file.
- 2 Configure a web service to do the following:
  - Specify the `policyValidatorCbClass` in the Rampart configuration policy assertion.

Following is a sample code listing of the Rampart policy assertion with specified `policyValidatorCbClass`:

```
<ramp:RampartConfig xmlns:ramp="http://ws.apache.org/rampart/policy">
  <ramp:user>service</ramp:user>
  <ramp:encryptionUser>client</ramp:encryptionUser>
<ramp:policyValidatorCbClass>
com.softwareag.wsstack.jaas.callback.MultiLoginPolicyValidatorCallback
</ramp:policyValidatorCbClass>
```

- Specify the `LoginContext` name as a parameter on one of the web service levels (global level in *axis2.xml*; service group level in the *services.xml*; service level in *services.xml*; operation level in *services.xml*; message level in *services.xml*)

With those settings, you are authenticated when logging on by SIN.

For information about the authentication steps, see SIN documentation in the *webMethods Product Suite 8.0* folder on the [Software AG Documentation Web site](#).



# 4 Transports

---

- TCP Transport ..... 38
- JMS Transport ..... 41
- Mail Transport ..... 45

Web Services Stack supports sending and receiving of messages over the following transports:

- HTTP
- TCP
- JMS
- Mail

HTTP is the single transport activated in the default Web Services Stack installation. The following instructions show how to configure and activate the other transports supported by Web Services Stack.

The information is organized under the following headings:

## TCP Transport

---

There are no prerequisites for the activation of TCP transport.

- [Activating TCP Transport \(Server-side Configuration\)](#)
- [Enabling WS-Addressing](#)
- [Forcing Deployment Over TCP Transport Only](#)
- [Invoking a Web Service Over TCP Transport \(Client-side Configuration\)](#)

### Activating TCP Transport (Server-side Configuration)

#### ▶ To activate TCP transport in Web Services Stack

- 1 Go to the web application server where the Web Services Stack runtime is installed.
- 2 Open the *axis2.xml* configuration file under the */webapps/wsstack/WEB-INF/conf* directory.
- 3 Uncomment the sections that define the transport receiver and transport sender with `name="tcp"`:

```
<transportReceiver name="tcp" ... />  
<transportSender name="tcp" ... />
```

The only parameter required for the transport receiver is its port number. The suggested default value is 6060.



**Note:** Restart the Web Services Stack runtime for the modifications to take effect.

## Enabling WS-Addressing

Since the TCP transport has no application level headers (and no target endpoint URI), you need WS-Addressing to dispatch the service.



**Note:** WS-Addressing is not enabled in the default Web Services Stack installation.

### ▶ To enable WS-Addressing

- Engage the WS-Addressing module globally by adding in the *axis2.xml* configuration file the following line:

```
<module ref="addressing"/>
```

Or:

Engage the WS-Addressing module on a `<service>` level. Engagement is for the service that is deployed on TCP transport.

You can enable WS-Addressing in the *services.xml* configuration file by adding the following line:

```
<service ...>
  <transports>
    <transport>tcp</transport>
  </transports>
  <module ref="addressing"/>
  ...
</service>
```

Or:

Enable WS-Addressing by using the Web Services Stack Eclipse plug-in. To do so, select **Enable WS-Addressing** from the **Modules** list in the **Services** tab.

For more information about working with the Web Services Stack Eclipse plug-in, see [Eclipse Plug-in](#).

## Forcing Deployment Over TCP Transport Only

If not explicitly configured, a web service is deployed over all activated transports in the Web Services Stack runtime. In this case, the web service is accessible at all enabled endpoints.

You may, however, want to restrict a web service to be accessible only over TCP transport.

### ▶ To deploy over TCP transport only

- Configure the web service's *services.xml* file by adding the following on the <service> level:

```
<service ...>
  <transports>
    <transport>tcp</transport>
  </transports>
  ...
</service>
```

Or:

Use Web Services Stack Eclipse plug-in at deployment time.

To do this, select **TCP Transport** from the list of transports in the **Services** tab.



**Note:** Since TCP transport has no application level headers, and thus no target endpoint URI, you need WS-Addressing to dispatch the service. If WS-Addressing is not globally enabled, you have to enable it for the service.

## Invoking a Web Service Over TCP Transport (Client-side Configuration)

To make a call to a web service over TCP transport, configure the client's repository.

### ▶ To invoke a web service over TCP

- 1 Uncomment the sections that define the transport receiver and transport sender with name="tcp" in the client's *axis2.xml* configuration file:

```
<transportReceiver name="tcp" ... />
<transportSender name="tcp" ... />
```

- 2 Engage globally the WS-Addressing module (*addressing.mar*) in the client's *axis2.xml* file:

```
<module ref="addressing"/>
```

- 3 Ensure the WS-Addressing module (*addressing.mar*) is present in the */modules* directory in the client's repository.

## JMS Transport

- Prerequisites
- Activating JMS Transport (Server-Side Configuration)
- Forcing Deployment Over JMS Transport Only
- Invoking a Web Service Over JMS Transport (Client-side Configuration)

### Prerequisites

Following are guidelines to the prerequisites for the activation of JMS transport.

#### ▶ To install and start a message broker

In order to achieve JMS communication, you need a message broker that handles the distribution of messages between communicating parties. Web Services Stack does not include a built-in message broker. This requires the use of an external one. Apache ActiveMQ is an open source message broker that you can download from <http://activemq.apache.org/activemq-411-release.html>.

- Extract the files from the downloaded archive into a directory of your choice. For example, *ACTIVEMQ\_HOME*.

After the installation, ActiveMQ is running with a basic configuration that is sufficient for its integration with the Web Services Stack.



**Note:** If you want to terminate the broker, type the CTRL-C command in the command prompt in which it is running.

#### ▶ Run the ActiveMQ message broker

- 1 Open the command prompt
- 2 Navigate to the *ACTIVEMQ\_HOME/bin* directory
- 3 Run the *activemq.bat* file.

You can find more about installing and using Apache ActiveMQ open source message broker at <http://activemq.apache.org/getting-started.html>.

### ▶ To provide additional libraries for the Web Services Stack runtime

Configuring Web Services Stack to work with Apache ActiveMQ message broker requires the provision of additional libraries for the Web Services Stack runtime.

- 1 Go to the *ACTIVEMQ\_HOME/lib* directory.
- 2 Copy the following libraries to the *<web\_app\_server>/webapps/wsstack/WEB-INF/lib* directory of the Web Services Stack runtime in the web application server:
  - *activemq-core-4.1.1.jar*
  - *activeio-core-3.0.0-incubator.jar*
  - *geronimo-jms\_1.1\_spec-1.0.jar*
  - *geronimo-j2ee-management\_1.0\_spec-1.0.jar*



**Note:** You need those libraries for any client that invokes a service over JMS transport.

## Activating JMS Transport (Server-Side Configuration)

### ▶ To activate JMS transport in Web Services Stack

- 1 Go to the web application server, where the Web Services Stack runtime is installed
- 2 Open the *axis2.xml* configuration file under the */webapps/wsstack/WEB-INF/conf* directory.
- 3 Uncomment the sections that define the transport receiver and transport sender with `name="jms"`:

```
<transportReceiver name="jms" ... />  
<transportSender name="jms" ... />
```

- 4 Define the custom connection factories

You can define custom connection factories as parameters under JMS transport receiver. They can be used by the services deployed over JMS transport. Refer to the *axis2.xml* configuration file to see the sample connection factories that the JMS transport receiver configuration includes.



**Note:** One of the connection factories is named as `default` for use by services that do not explicitly specify in their *services.xml* configuration file the connection factory they want to use.

Those connection factories are associated with Apache ActiveMQ implementation whose libraries are required for the Web Services Stack runtime. Each connection factory specifies the following parameters:

- An initial naming factory class
- Naming provider URL
- The JNDI name of an actual JMS connection factory.

Web Services Stack can run with the default configuration of Apache ActiveMQ. In this case, you only have to uncomment the JMS transport receiver and JMS transport sender configuration in the *axis2.xml* file.



**Note:** You must always run the message broker before you start Web Services Stack.

### Forcing Deployment Over JMS Transport Only

If not explicitly configured, a web service is deployed over all activated transports in the Web Services Stack runtime. However, you can restrict a web service to be deployed over JMS transport only.

#### ▶ To deploy over JMS transport only

- Configure the web service's *services.xml* file by adding the element in bold:

```
<service ...>
  <transports>
    <transport>jms</transport>
  </transports>
  ...
</service>
```

Or:

Use Web Services Stack Eclipse plug-in at deployment time by selecting **JMS Transport** from the list of transports in the **Services** tab.



**Note:** You can also specify the destination where the service listens for messages, as well as the name of the connection factory to be used. The service can use one of the connection factories defined within the JMS transport receiver in the *axis2.xml* configuration file.

▶ **To specify the name of the connection factory**

- Configure the web service's *services.xml* file by adding the element in bold:

```
<service ...>
  <transports>
    <transport>jms</transport>
  </transports>
  <parameter name="transport.jms.ConnectionFactory"
locked="true">myQueueConnectionFactory</parameter>
  <parameter name="transport.jms.Destination"
locked="true">dynamicQueues/TestQueue</parameter>
  ...
</service>
```



**Note:** Those values are samples. The connection factory can be any of the connection factories defined in *axis2.xml* and the destination name can be anything.

Or:

Use Web Services Stack Eclipse plug-in to add the two parameters in the table in the section **Properties** in the **Services** tab.

▶ **To use the Web Services Stack Eclipse plug-in to add the first of the preceding parameters to the table in the section Properties in the Services tab**

- 1 Choose the **Add** button.
- 2 Type "transport.jms.ConnectionFactory" for name.
- 3 "myQueueConnectionFactory" (or another connection factory defined in *axis2.xml*) for value.
- 4 Select **OK**.

▶ **To use the Web Services Stack Eclipse plug-in to add the second of the preceding parameters to the table in the section Properties in the Services tab**

- 1 Choose the **Add** button.
- 2 Type "transport.jms.Destination" for name.
- 3 "dynamicQueues/TestQueue" (or other value of your choice) for value.
- 4 Select **OK**.



**Note:** These parameters are optional. If they are not specified, the service uses the default connection factory (named as `default` in the configuration of the JMS transport receiver in the *axis2.xml* file) and listens for messages on a JMS queue by the same name as the name of the service.



For more information about working with the Web Services Stack Eclipse plug-in, see [Eclipse Plug-in](#).

## Invoking a Web Service Over JMS Transport (Client-side Configuration)

To make a call to a web service over JMS transport, you have to configure the client's repository.

### ▶ To invoke a web service over JMS

- 1 Uncomment the sections that define the transport receiver and transport sender with name="jms" in the client's *axis2.xml* configuration file:

```
<transportReceiver name="jms" ... />  
<transportSender name="jms" ... />
```

- 2 Engage globally the WS-Addressing module (*addressing.mar*) in the client's *axis2.xml* file.

```
<module ref="addressing"/>
```

- 3 Ensure the WS-Addressing module (*addressing.mar*) is present in the */modules* directory in the client's repository.

## Mail Transport

---

The information is organized under the following headings:

- [Prerequisites](#)
- [Activating Mail Transport](#)
- [Forcing Deployment Over Mail Transport Only](#)
- [Invoking a Web Service Over Mail Transport](#)
- [Sample Client Configuration](#)

### Prerequisites

To activate mail transport in Web Services Stack, you need the following prerequisites:

- [Install, Configure and Start a Mail Server](#)

- [Create Accounts in the Mail Server](#)

### Install, Configure and Start a Mail Server

The activation of mail transport in Web Services Stack requires a mail server that transfers e-mail messages. The Apache Java Enterprise Mail Server (James) is an open source SMTP and POP3 mail server that is used by Web Services Stack.

#### ▶ To install Apache James server

- 1 Download the archive with the binary distribution of the Apache James mail server from <http://james.apache.org/download.cgi>.
- 2 Extract the files from the downloaded archive to a *JAMES\_HOME* directory of your choice.
- 3 Start and stop the mail server once so that it unpacks its configuration files.

#### ▶ To open the configuration files for editing

- 1 Open the command prompt.
- 2 Navigate to *JAMES\_HOME/bin* directory.
- 3 Run *run.bat* to start the server.
- 4 Use the CTRL+C command to stop the mail server.
- 5 Type the *ipconfig/all* command to check your network configuration.



**Note:** You need this information for the next instruction (configuring the DNS servers).

#### ▶ To configure the DNS servers in the mail server

- 1 Open the *config.xml* file under the *JAMES\_HOME/apps/james/SAR-INF* directory
- 2 Find the tag `<dnsserver>` and enter the IP address of each DNS server from your network configuration as shown in the following example:

```
<dnsserver>
  <servers>
    <server>[DNS.Server.IP.address]</server>
    <server>...</server>
  </servers>
  ...
</dnsserver>
```



**Note:** Apache James mail server requires the valid IP addresses of the DNS servers in your network configuration.

- 3 Start the mail server again.

You can read more about the configuration of Apache James mail server in the "Configuring James" section of the James server documentation at <http://james.apache.org/server/2.3.1/index.html>.

### Create Accounts in the Mail Server

After you have installed and configured your mail server, you have to create accounts. You need to create a mail account that represents the e-mail address of the Web Services Stack runtime. Additional accounts can be created to correspond to different clients.

#### ▶ To create an account

- 1 Start the Apache James mail server if it is not started.

To start Apache James Server, run the console command prompt, navigate to `JAMES_HOME/bin` directory and run `run.bat`.

- 2 Start James Remote Manager Service (this tool is used for administration purposes).

Run the console command prompt and type the following `telnet` command:

```
telnet localhost 4555
```

Port number 4555 is the default port, where the Remote Manager Service starts. It is configured in the James configuration file (`JAMES_HOME/apps/james/SAR-INF/config.xml`). If you have changed the default port number in a previous step, use the new value in the preceding command

- 3 Log on the Remote Manager. You are prompted for the logon ID and password. They are configured in the James configuration file (`JAMES_HOME/apps/james/SAR-INF/config.xml`). The initial values are "root" for both, the ID and the password, unless you have changed them.

Type "root" for the logon ID and for the password.

- 4 Create the account. The command for adding a new user is `adduser username password`. After executing the command, you get a confirmation.

Type the following command:

```
adduser server wsstack
```

- 5 Exit the Remote Manager Service using the `quit` command.

After you have executed the commands in the command prompt, you get a result similar to the following one:

```
>telnet localhost 4555

JAMES Remote Administration Tool 2.3.1
Please enter your login and password
Login id:
root
Password:
root
Welcome root. HELP for a list of command
adduser server wsstack
User server added
quit
Bye
```

### Activating Mail Transport

There are prerequisites for the activation of mail transport. Refer to the following instruction and the description of the required parameters for the transport receiver and the transport sender.

#### ▶ To activate mail transport in Web Services Stack

- 1 Go to the `/webapps/wsstack/WEB-INF/conf` directory.
- 2 Open the `axis2.xml` configuration file.
- 3 Configure the context root of Web Services Stack runtime.

In the `axis2.xml` file, find the parameter with the name `contextRoot`. Uncomment it (if it is commented) and ensure that its value is "wsstack":

```
<parameter name="contextRoot" locked="false">wsstack</parameter>
```

- 4 Activate the mail transport receiver and the mail transport sender.

In the *axis2.xml* file find and uncomment the sections that define the transport receiver and the transport sender with name="mailto":

```
<transportReceiver name="mailto" ... />
<transportSender name="mailto" ... />
```

The parameters under the transport receiver and the transport sender have fake default values. They need to be verified.

### Required Parameters for the Transport Receiver

The following table lists the required parameters and their description:

Parameter	Description
mail.pop3.host	The host name (or IP address) where the James mail server is running.  If the server is running on the same machine as the Web Services Stack runtime, then the value can be "localhost" or "127.0.0.1".
mail.pop3.user	The user name of a user registered in the James mail server.  The user name in the following sample code is the user registration from the example in the preceding topic "Creating accounts in the mail server".
transport.mail.pop3.password	The user's corresponding password for his account.
mail.store.protocol	The value "pop3" is expected for that parameter.
transport.mail.replyToAddress	This parameter is responsible for the following values: <ul style="list-style-type: none"> <li>■ Supplies the endpoint reference for the response and represents the server email address.</li> <li>■ Contains the user name specified in the <code>mail.pop3.user</code> parameter and the server name of James mail server, separated by the @ sign.</li> </ul> <p><b>Note:</b> The server name is configured in the <i>JAMES_HOME/apps/james/SAR-INF/config.xml</i> configuration file. If you have not specified a different one, the initial value is "localhost".</p>
transport.listener.interval	Controls the time interval (in milliseconds) for checking the mail server for new messages.  <b>Note:</b> This parameter is optional. If omitted, its default value is = "3000 " milliseconds (which equals to 3 seconds).

Following is a sample code listing of the usage of the required parameters for the transport receiver:

```
<transportReceiver name="mailto"
class="org.apache.axis2.transport.mail.SimpleMailListener">
  <parameter name="mail.pop3.host">localhost</parameter>
  <parameter name="mail.pop3.user">server</parameter>
  <parameter name="transport.mail.pop3.password">wsstack</parameter>
  <parameter name="mail.store.protocol">pop3</parameter>
  <parameter name="transport.mail.replyToAddress">server@localhost</parameter>
  <parameter name="transport.listener.interval">3000</parameter>
</transportReceiver>
```

### Required Parameters for the Transport Sender

The following table lists the required parameters and their description:

Parameter	Description
mail.smtp.host	The host name (or IP address), where James mail server is running.  It corresponds to the mail.pop3.host parameter under the Mail transport receiver.
mail.smtp.user	Corresponds to the value of mail.pop3.user parameter of the transport receiver.
transport.mail.smtp.password	Corresponds to the value of transport.mail.pop3.password parameter of the transport receiver.
mail.smtp.from	Corresponds to the value of mail.transport.replyToAddress parameter of the transport receiver.

Following is a sample code listing of the usage of the required parameters for the transport sender:

```
<transportSender name="mailto"
class="org.apache.axis2.transport.mail.MailTransportSender">
  <parameter name="mail.smtp.host" locked="false">localhost</parameter>
  <parameter name="mail.smtp.user">server</parameter>
  <parameter name="transport.mail.smtp.password">wsstack</parameter>
  <parameter name="mail.smtp.from">server@localhost</parameter>
</transportSender>
```

## Forcing Deployment Over Mail Transport Only

If you want to restrict a web service to be deployed only over Mail transport, you must add the following element in the web service's *services.xml* file:

```
<service ...>
  <transports>
    <transport>mailto</transport>
  </transports>
  ...
</service>
```



**Note:** If not configured explicitly, a web service is deployed over all activated transports in the Web Services Stack runtime.

## Invoking a Web Service Over Mail Transport

To call a web service over mail transport, configure the client's repository.

### ▶ To configure the client's repository

- 1 In the client's *axis2.xml* configuration file, find and uncomment the sections that define the transport receiver and transport sender with name="mailto":

```
<transportReceiver name="mailto" ... />
<transportSender name="mailto" ... />
```

- 2 Check the parameters under the mail transport receiver and the mail transport sender. You must configure the user name, the password, and the e-mail address of a user registered in the James mail server. That user must be different from the one configured in the Web Services Stack runtime.

For details, see [Activating Mail Transport](#).

## Sample Client Configuration

Following is a sample code listing of client configuration with a user that is registered in the James mail server. The user name is "client" and the password is "pass":

```
<transportReceiver name="mailto"  
class="org.apache.axis2.transport.mail.SimpleMailListener">  
  <parameter name="mail.pop3.host">localhost</parameter>  
  <parameter name="mail.pop3.user">client</parameter>  
  <parameter name="mail.store.protocol">pop3</parameter>  
  <parameter name="transport.mail.pop3.password">pass</parameter>  
  <parameter name="transport.mail.replyToAddress">client@localhost</parameter>  
  <parameter name="transport.listener.interval">3000</parameter>  
</transportReceiver>  
  
<transportSender name="mailto"  
class="org.apache.axis2.transport.mail.MailTransportSender">  
  <parameter name="mail.smtp.host">localhost</parameter>  
  <parameter name="mail.smtp.user">client</parameter>  
  <parameter name="transport.mail.smtp.password">pass</parameter>  
  <parameter name="mail.smtp.from">client@localhost</parameter>  
</transportSender>
```



# 5 Monitoring and Logging

---

- SOAP Monitor ..... 54
- Logging ..... 56

This chapter covers the logging facility and the utility for monitoring of SOAP messages.

The information is organized under the following headings.

## SOAP Monitor

---

This section provides details on the SOAP monitoring utility in Web Services Stack.

The information is organized under the following headings:

- [Overview](#)
- [Using the SOAP Monitor](#)

### Overview

The distribution of Web Services Stack comes with a SOAP monitor that allows users to monitor SOAP messages exchanged between web service clients and web services running in Web Services Stack.

SOAP messages are shown with the structure that they have after they have passed all system phases in the Axis 2 engine. This means that the original SOAP messages, sent by a user, can be visually different, but semantically equal to the ones shown into the SOAP monitor. Examples of such a case are MTOM SOAP messages. SOAP monitor shows the binary data exchanged “by value” (included into the SOAP message itself). On the other hand, the original SOAP message has MIME parts in it.

For example, take a binary data shown into a TCPMon (another monitor). To make easy to understand, only part of the message related to the MTOM-ized binary data is shown:

```
<<ns1:binaryData><xop:Include
href="cid:1.urn:uuid:EFF202258F699D83131220514272228@apache.org"
xmlns:xop="http://www.w3.org/2004/08/xop/include" /></ns1:binaryData>
...
--MIMEBoundaryurn_uuid_EFF202258F699D83131220514272117
Content-Type: text/plain
Content-Transfer-Encoding: binary
Content-ID: <1.urn:uuid:EFF202258F699D83131220514272228@apache.org>

text
--MIMEBoundaryurn_uuid_EFF202258F699D83131220514272117--
```

The binary data that a SOAP monitor shows is the following:

```
<ns1:binaryData>dGV4dA==</ns1:binaryData>
```

As you can see, the binary data is shown “by value”. This is because it was already processed by the system phases of the Axis 2 engine.

## Using the SOAP Monitor

SOAP monitor is disabled by default.

### ▶ To enable SOAP monitor

- 1 Open the *web.xml* file that is located in the *WEB-INF* directory of the *wsstack webapp*.
- 2 Uncomment the `<servlet-name>SOAPMonitorService</servlet-name>` part.
- 3 Uncomment the `<servlet-mapping>` part.
- 4 Extract the following `SOAPMonitor` classes from *soapmonitor.jar* and copy them directly under the expanded *wsstack* context root:

```
org\apache\axis2\soapmonitor\applet\SOAPMonitorApplet$ServiceFilterPanel.class
org\apache\axis2\soapmonitor\applet\SOAPMonitorApplet$SOAPMonitorData.class
org\apache\axis2\soapmonitor\applet\SOAPMonitorApplet$SOAPMonitorFilter.class
org\apache\axis2\soapmonitor\applet\SOAPMonitorApplet$SOAPMonitorPage.class

org\apache\axis2\soapmonitor\applet\SOAPMonitorApplet$SOAPMonitorTableModel.class

org\apache\axis2\soapmonitor\applet\SOAPMonitorApplet$SOAPMonitorTextArea.class
org\apache\axis2\soapmonitor\applet\SOAPMonitorApplet.class
```



**Important:** Ensure you keep the classes packaging structure.

There is no effect if you try to replace this step with any of the following:

- Copy the *soapmonitor.jar* into the *WEB-INF \lib*
- Copy the classes from *soapmonitor.jar* into *WEB-INF \classes*

- Engage the *soapmonitor* Axis 2 module globally in the *axis2.xml* by adding the following line:

```
<module ref="soapmonitor"/>
```

You can engage it in the same way for a service in the *services.xml* file.

- Restart Tomcat or the *wsstack webapp*.
- Go to <http://<host>:<port>/wsstack/SOAPMonitor> to start using the SOAP monitor.

For more details on the SOAP monitor configuration, see [http://ws.apache.org/axis2/1\\_4\\_1/soap-monitor-module.html](http://ws.apache.org/axis2/1_4_1/soap-monitor-module.html).

## Logging

---

This section provides details on the logging facility in Web Services Stack.

The information is organized under the following headings:

- [Overview](#)
- [log4J Logging Levels](#)
- ["Argus Agents" Logging](#)

### Overview

Web Services Stack uses Apache Common Logging (JCL) and its *log4J* facility. The JCL provides thin-wrapper log implementations for other logging tools, including the default *log4J*.

For details on *log4J*, refer to Apache logging services at <http://logging.apache.org/log4j/1.2/index.html>.



#### Note:

The distribution of Web Services Stack comes with a *log4j.properties* file and a *commons-logging.properties* file by default. You can find them in `<Web Services Stack_Install_Folder>/webapp\wsstack\WEB-INF\classes`.



**Note:** Those files are also included in the *wsstack.war* web archive in `<Web Services Stack_Install_Folder>/webapp`, in case you deploy Web Services Stack another servlet container or application server.

### ▶ To enable *log4J*

- Place the *commons-logging.properties* file into the given module classpath.

## log4J Logging Levels

The *log4j.properties* files come with a default value of the logging level. You can change those values according to the requirements of your system.

The default logging level is `info`. Following are the standard levels in descending (in terms of priority) order:

- `fatal`
- `error`
- `warn`
- `info`
- `debug`
- `trace`



**Note:** A lower level covers all levels above it. For example, if `warn` is set, then all logs of level `error` and `fatal` are logged, too.

It is important to ensure that the log messages are appropriate in content and severity. See the following table for guidelines on the usage of logging levels:

Logging Level	Usage
<code>fatal</code>	Severe errors that cause premature termination. Expect these to be immediately visible on a status console.
<code>error</code>	Other run-time errors or unexpected conditions. Expect these to be immediately visible on a status console.
<code>warn</code>	Use of deprecated APIs, poor use of API, error-like situations, other run-time situations that are undesirable or unexpected, but not necessarily "wrong". Expect these to be immediately visible on a status console.
<code>info</code>	Interesting run-time events (start /shut down). Expect these to be immediately visible on a console, so be conservative and keep to a minimum.
<code>debug</code>	Detailed information on the flow through the system. Expect these to be written to logs only.
<code>trace</code>	More detailed information. Expect these to be written to logs only.

## "Argus Agents" Logging

Web Services Stack provides a logging mechanism for its agent programs that use the System Management Hub administration functionality. These agent programs are called "argus agents". They manipulate the Web Services Stack environment under the System Management Hub web interface. See Administration Tool for details.

If you experience problems when using the administration tool, you must enable the logging for the "argus agents" to see a detailed message log. It is recommended to use this logging mechanism only when you want to search for faults in the operation of the system. Otherwise, the performance of your interface might slow down.

### ▶ To enable logging for the "argus agents"

- 1 Open the registry editor.
- 2 Go to HKEY\_LOCAL\_MACHINE\SOFTWARE\Software AG\System Management Hub\Products\Web Services Stack\Versions\Parameters\enableLog
- 3 Switch the registry parameter enableLog to "1".

You can find the output log file in *<Web Services Stack\_Install\_Folder>/argus/wsstack.log*.

# 6 Eclipse Plug-in

---

- Introduction ..... 60
- Creating and Removing a Web Service Package ..... 61
- Configuring a Web Service Package ..... 62
- Enabling Advanced Policy Configurations ..... 64
- Deploying and Undeploying a Web Service Package ..... 65
- Registering a Web Service Package in CentraSite ..... 66

This chapter describes the Web Services Stack basic tool for packaging, configuring, and deploying web service archives.

## Introduction

---

The Packaging and Configuration Eclipse plug-in provides a graphical user interface that you can use to do the following:

- Create web service archives
- Configure web service archives in various ways (including advanced configurations like addressing, security, transactional behavior, and others)
- Deploy web service archives to the Web Services Stack runtime
- Register web service archive in CentraSite

The Web Services Stack installation installs the Eclipse plug-in by default with webMethods Designer. If you want to use your own installation of Eclipse 3.4 SP2, you must first install the plug-in with the Installer.

These archives are located under *SoftwareAG\webMethods\eclipse\updates*.


### ► To install the Eclipse plug-in in your own Eclipse installation

- 1 Open Eclipse.
- 2 Go to **Help -> Software Updates -> Available Software**
- 3 Select **Add site...**
- 4 Select **Archive**.

Browse to the *<eclipse\updates>* folder of your Web Services Stack installation and select the *com.softwareag.common.zip* archive inside.

- 5 Select **Install...** and follow the instructions.
- 6 Repeat steps 2 through 6 for *eclipse.wss.<version\_number>.UpdatePackage.0000.zip*.

The version number of the particular update package is a three-digit number which depends on the version of the current installation. You can construct the version number using the version of the release (omitting any periods in the number) and the SP version (using the number of the SP release). For example, if you want to update a Web Services Stack 8.0 SP4 installation, the version number of the update package is *804*, and the file name is: *eclipse.wss.804.UpdatePackage.0000.zip*.

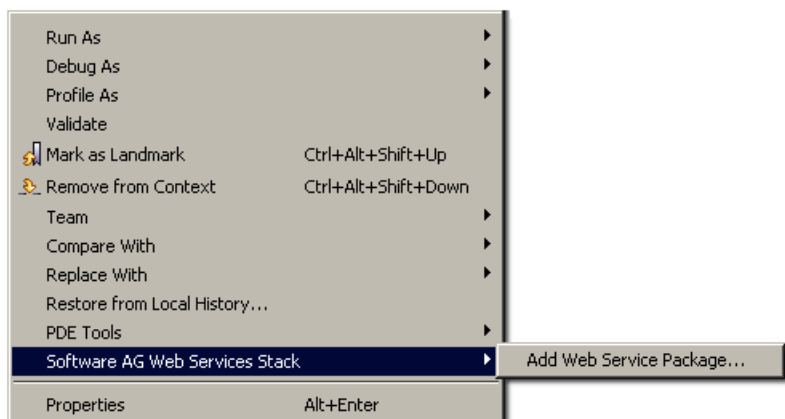
 **Important:** CentraSite Eclipse plug-in is a prerequisite for the proper functioning of the CentraSite registration plug-in is.




## Creating and Removing a Web Service Package

Following are guidelines on creating (or removing) web service packages with the Eclipse plug-in.

After you have created a new project, **Software AG Web Services Stack/ Add Web Service Package** appears in the context menu of the Java projects in the Package Explorer view. You use this option to create a new deployment archive and add it to the project.



 **Note:** One project might have several packages associated with it.

### ▶ To create a web service package using the Eclipse plug-in wizard

- 1 Select **Software AG Web Services Stack ->Add Web Service Package...**
- 2 Specify name and container of the package. By default, the container is the project itself.
- 3 Select a service source file. This may be either a WSDL file or Java class.

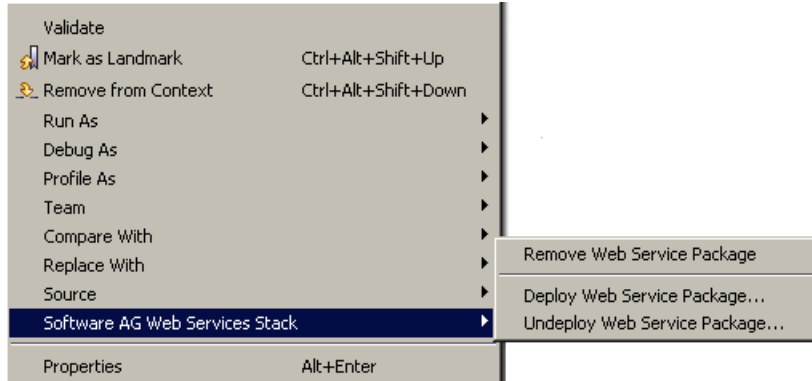


**Important:** The files must be part of the project. External files can not be added.

- 4 Add additional meta-information files. These are files to be included in the package under the *meta-inf* directory.
- 5 Add additional files to be included into the service package. These are files to be included in the package under the root directory.
- 6 The archive is created.

► To remove a web service package using the wizard

1 Select **Software AG Web Services Stack ->Remove Web Service Package**



2



**Note:** This command is valid only for previously added web service package.

## Configuring a Web Service Package

After creating the archive, the editor provides an interface to configure the settings of the package itself, the services, contained in it, and the operations of the services. This imposes three different layouts: archive view, services view, and operation view. The different views appear as different tabs in Web Services Stack. There is one more additional tab that contains a textual view of the *services.xml* file, but it displays a read-only version of the file that cannot be modified directly.

In the outline view of Eclipse, the package structure is represented in a tree view. The package itself is the root element of the tree; the second level contains the services; the third level contains the operations. Selecting an item changes the view in the editor to the view corresponding to the selected item.

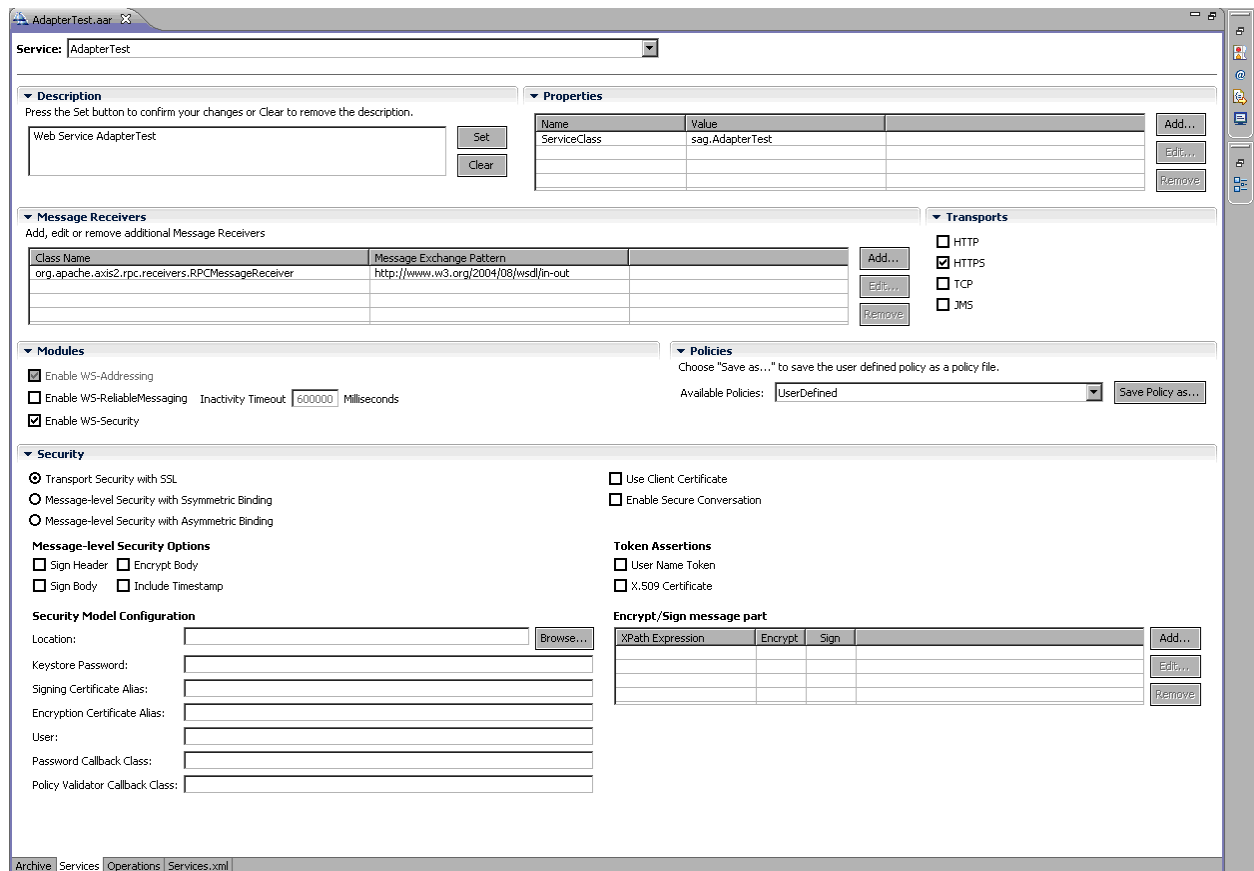
In the archive view, you can add additional files to the root of the package or to the *meta-inf* directory. You can also add additional Java files (POJOs) that adds additional services to the archive.

In the services view, you can select any of the available services in the archive. After that you can perform various configurations on it:

- Add custom description
- Add message receivers
- Configure the transports on which the service is accessible

- Enable and configure the available modules like WS-Addressing, WS-Reliable Messaging, and WS-Security

The available transports are: HTTP, HTTPS, TCP, and JMS transport. If no transport is selected, then all transports that are enabled on the server are available to the service.



In the operations view, you can select any of the operations in the package by selecting a service from the drop down list and one of its operations. The available configurations on the operation level are:

- Setting the message receiver class
- Adding parameters
- Enabling and configuring service modules

## Enabling Advanced Policy Configurations

---

The available policy configurations on a <service> level and on an <operation> level are WS-Addressing, WS-Reliable Messaging, and WS-Security.

By selecting **Enable WS-Addressing** check box, you enable WS-Addressing module.

When WS-Reliable Messaging is selected, you must provide a value for the inactivity time period that is by default 600 000 milliseconds by default.

By selecting the **Enable WS-Security** check box, a new configuration view is opened in the editor. You have three types of security to choose from: Transport Security (SSL), Message-level Security with symmetric binding, and Message-level Security with asymmetric binding. When you select **Enable WS-Security** check box, Transport Security with SSL and HTTPS transport are chosen by default. The Addressing module is engaged always when the Security module is engaged. You must provide a *keystore* configuration in the cases of message-level security and a `Password callback` class if **Authentication** with user name token is selected.

For details on message-level security and transport, see *Security*.

The keystore configuration includes the location of the server *keystore* file, the *keystore* password, aliases of the certificates to be used for signing and encryption, user (default user name for username token and alias of the certificate for signing if the last is not specified in the corresponding field) and the `password callback` class. You can specify a policy validator callback class to ensure that the security header corresponds to the policy in the *services.xml* file. If the class is not explicitly specified, there is a default policy validator callback class that is used.

You also have the option to enable standard message security to sign header, sign body, encrypt body, or include timestamp. Else, you can write an XPath expression to encrypt and sign arbitrary parts of the message. For the authentication, you can use either X.509 certificate, or a Username token. You can also choose to engage WS-Secure Conversation.

The other service modules available are WS-Addressing and WS-Reliable Messaging. When WS-Reliable Messaging is selected, you must provide a value for the inactivity time period that is 600 000 milliseconds by default.

With the editor, you can save policy configurations in a file that you can later reuse by choosing the button **Save Policy as...** and load it when needed as saved policy.

## Deploying and Undeploying a Web Service Package

---

When the package is set and configured, it can be deployed to Web Services Stack by choosing **Deploy Web Services Package** in the project context menu.



**Important:** The default deploy location of Web Services Stack is "localhost" (that is, default Web Services Stack running in default Tomcat) at port 49981. It is recommended that users configure a deploy target with the real hostname instead of "localhost".

Type in the following information:

- The URL of the Web Services Stack deploy servlet

The default servlet is */wsstack/sagdeployer*.

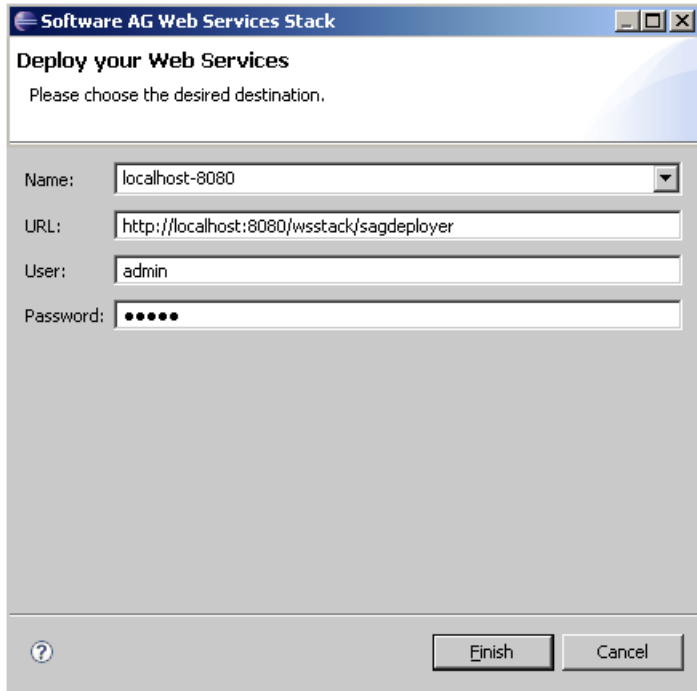
- The user name and password

These authentication credentials are the same as the credentials for administration that are configured in the *axis2.xml* file. The default user name is "admin" and the default password is "axis2".



**Note:** See *Changing Logon Credentials* for details on changing the default user name and password at first logon.

The default servlet name is */wsstack/sagdeployer*.



The context menu of the package has also the `Undeploy Web Services Package` command. When it is selected, a window is opened containing the same required information as for the deploy command.

The Eclipse console provides feedback on the execution of the operations.

## Registering a Web Service Package in CentraSite

---

Another option from the context menu of the package is to register it in CentraSite. Web Service Stack Eclipse UI provides a context menu on AAR service files to register the service using the JAXR interface to CentraSite. The registration process requires authentication to CentraSite with user name and password credentials. These credentials give access to exactly those Organizations in CentraSite the user is authorized to register service into.



**Note:** Create and configure organization with CentraSite tooling. However, there is always a "Default Organization".

Following is a sample screen capture to illustrate the use of the required parameters for the registration:

The screenshot shows a dialog box titled "Generate Web Service" with a sub-header "Connect to CentraSite - server information". Below the sub-header is the instruction "Choose your CentraSite connection." and a navigation icon. The dialog contains several input fields: "Host" with the value "localhost", "Port" with "53305", "Product ID" with "CentraSite", and "Repository Path" with "WebServicesStack". There are also empty fields for "User", "Password", and "Organisation". Below these is a text box labeled "Select Organisation for Completion." At the bottom, there are buttons for "< Back", "Next >", "Finish", and "Cancel".

The required parameters are as follows:

- Host - the host of the CentraSite instance
- Product ID – CentraSite product ID
- Port - the port of the CentraSite instance
- Repository Path - the path under which the package is registered
- Authentication data like user name, password and organization

**⚠ Important:** To have the registration functionality available in Eclipse environment, you must have CentraSite Eclipse plug-ins as a prerequisite.

Prior to registration, deploy the service in a Web Service Stack runtime. For details, see [Deploying and Undeploying a Web Service Package](#).

### Description of the Registration Process

1. The service is registered in the CentraSite Registry via the WSDL.

For details on importing web services, see section *User Interface -> Using the Asset Catalog -> Publishing a New Asset into the Catalog -> Adding an Asset to the Catalog Using an Importer -> Importing Web Services* in CentraSite documentation at [Software AG documentation Web site](#).

2. The WSDL and the service archive is stored in the CS Repository.

For details, see *User Interface -> Using the Asset Catalog -> Publishing a New Asset into the Catalog -> Adding an Asset to the Catalog Using an Importer -> Importing Web Services* in CentraSite documentation at [Software AG documentation Web site](#).

3. The service object gets an external link to the archive.

For details on storing objects (e.g. service archive, or any other object) in the CentraSite Supporting Document Library, see section *User Interface -> Using the Asset Catalog -> Attaching a Supporting Document to an Asset* in CentraSite documentation at [Software AG documentation Web site](#).