# Web Services Stack Guide

## Configuration

Version 9.5 SP1

November 2013

WEB SERVICES STACK

This document applies to Web Services Stack Version 9.5 SP1.

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



**Document ID: WSS-CONCEPTS-95SP1-20130930**

## Table of Contents

# Preface

This document describes the configuration tasks you need to complete before you can use Web Services Stack (WSS) to manage, monitor, and secure services.

The information is organized under the following headings:

| | |
|---|---|
| **Web Services Stack Runtime** | Details on the configurations of *axis2.xml*, client, server, and MTOM. |
| **Security** | Configurations for securing the message content and communication channel. |
| **Transports** | Configuration of Web Services Stack for sending and receiving messages over different transports. |
| **Monitoring and Logging** | Configuration of Web Services Stack facilities for monitoring and logging. |

# 1 Web Services Stack Runtime

The information is organized under the following headings:

## Understanding axis2.xml Configuration

The *axis2.xml* is a configuration file provided by the Apache Software Foundation. For further information about the Axis2 parameters available in this file and their values, see the **Axis2 Configuration Guide**.

In the *axis2.xml* file, Web Services Stack also uses some custom parameters. They are set on the server side of the Axis2 configuration. You can find more information about these parameters in the following table:

| Parameter | Default Value | Description |
|---|---|---|
| `includeWrappedTypesDeclaration` | true | Defines whether to include message-wrapper types/elements in the WSDL XSD schema. Axis2 processes an RPC-style WSDL definition and automatically creates a wrapper element and type definition for each message. Then, Axis2 processes internally any request/response as if it is in a document style with an element declaration for each message.<br><br>Possible values for this parameter are:<br><br>■ true (by default) - Axis2 creates the web service instance and automatically adds the autogenerated types to the XSD schema in the original WSDL definition.<br>■ false - Axis2 creates a copy of the WSDL definition when processing the message types and modifies the copy instead of the original WSDL document. |
| `enableWSDLValidation` | false | Defines whether to validate WSDL documents.<br><br>Possible values are:<br><br>■ true - Axis2 validates the provided WSDL document against the external resources.<br>■ false (by default) - No validation against external resources is performed. |
| `enableSoapValidation` | false | Defines whether to validate SOAP messages.<br><br>Possible values are:<br><br>■ false (default value) - When the Axis2 client side and server side exchange SOAP messages, the messages are not validated automatically if they are compliant with the SOAP specification.<br>■ true - The SOAP validation can be enabled both on the server side and on the client side. |

| Parameter | Default Value | Description |
|---|---|---|
| | | On the server side, the SOAP validation can be enabled on the follow levels:<br>■ globally - by setting the parameter in the axis2.xml file, you enable validation globally.<br>■ on service group level - by setting the parameter inside a ServiceGr tag in the services.xml file, you enable the validation for a specific group.<br>■ on service level - by setting the parameter inside a Service tag in th services.xml file, you enable the validation for a specific service.<br>■ on operation level - by setting the parameter inside an Operation t in the services.xml file, you enable the validation for a specific operati<br>■ on message context level - by setting the parameter programatically MessageContext, you enable the validation for a specific request.<br>On the client-side, the SOAP validation can be enabled on the follow levels:<br>■ globally - by setting the parameter in the axis2.xml file, you enable validation globally.<br>■ for a specific client execution - programatically by calling `Options.setProperty("disableSoapValidation",Boolean.TRU` you enable the validation for operations that expect to deal with hu SOAP messages. |

Since messages that Software AG Web Services Stack processes are not always in SOAP format, the message builders and formatters provided by Axis2 are extended to ensure all messages are correctly converted. Below you will find some Web Services Stack specific information about the proprietory message formatters and message builders available in the *axis2.xml* configuration file.

- **Message Formatters**
  The Web Services Stack *axis2.xml* file has defined proprietary 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.RawXMLApplicationXMLFormatter" />
  <messageFormatter contentType="application/soap+xml"
      class="com.softwareag.formatters.RawXMLFormatter"/>
  <messageFormatter contentType="application/x-www-form-urlencoded"
      class="org.apache.axis2.transport.http.XFormURLEncodedFormatter"/>
  <messageFormatter contentType="multipart/form-data"
      class="org.apache.axis2.transport.http.MultipartFormDataFormatter"/>
```

- **Message Builders**
  The Web Services Stack *axis2.xml* file has defined proprietary 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"/>
  <messageBuilder contentType="application/x-www-form-urlencoded"
     class="org.apache.axis2.builder.XFormURLEncodedBuilder"/>
  <messageBuilder contentType="multipart/form-data"
     class="org.apache.axis2.builder.MultipartFormDataBuilder"/>
```

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

## Runtime Configuration

This section provides information about the global runtime configuration. There are three kinds of configuration files - *axis2.xml, services.xml,* and *module.xml*:

- *axis2.xml* is used for configuring the client side and the server side of all deployed Web services. The *axis2.xml* file is in the following directory: *<Software AG_directory>*/*profiles*/*CTP*/*workspace*/*wsstack*/*repository*/*conf*/.
- *services.xml* is used for configuring Web services. The *services.xml* file is in the META-INF directory that is in the service archive (a file with an .aar extension) or in the unpacked service directory.

**Note:** The service archive is in the *<Software AG_directory>/profiles/CTP/workspace/wsstack/repository/services/* directory.

- *module.xml* is used for configuring modules. The *module.xml* file is in the META-INF directory that is in the module archive (a file with a .mar extension) or in the unpacked module directory.

  **Note:** The module archive is in the *<Software AG_directory>/profiles/CTP/workspace/wsstack/repository/modules/* directory.

**Caution:** The *axis2.xml* file contains important information such as the user name and password for the administration console logon. System administrators must change these default credentials to protect the access to the *axis2.xml* configuration file.

There are some additional Web Services Stack specific configuration files:

1. **SMH Agents Configuration Files:**
   - The *deployclient.properties* file is used by Software AG System Management Hub (SMH) for deploying Web services.
   - The *argusagent.properties* file contains information about the host name and server port of the deployed Web Services Stack.

   For more information about the *deployclient.properties* file and the *argusagent.properties* file, see the overview of *The Administration Tool*
2. **Client-side Configuration Files:**
   - *wsclientsec.properties* - this file contains security-related information.

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

The installation of Web Services Stack provides the following runtime configuration:

- Web Services Stack Client Runtime Configuration
- Web Services Stack Runtime Configuration in Standalone Server
- Web Services Stack Runtime Configuration in Web Application

- Web Services Stack Runtime Configuration in Platform Tomcat Server

### Web Services Stack Client Runtime Configuration

The installation of Web Services Stack provides a client runtime configuration. It is primarily used by the Web Services Stack SMH agents but it can be used by any user-implemented Web services client as well. Essentially, the Web Services Stack is administered by an administration service deployed on it and is referred by the SMH agents (the SMH agents are in the role of Web services clients).

**Note:** To administer Web Services Stack on Platform Tomcat Server using the SMH web administration user interface, you must install Web Services Stack Administration tools and Core Files.

The client runtime configuration uses the following files:

1. Runtime configuration file:

   *<Software AG_directory>/WS-Stack/repository/conf/axis2.xml*

2. Repository directory:

   *<Software AG_directory>/WS-Stack/repository*

   a. Web services directory:

      *<Software AG_directory>/WS-Stack/repository/services*

   b. Modules directory:

      *<Software AG_directory>/WS-Stack/repository/modules*

### Web Services Stack Runtime Configuration in Standalone Server

Web Services Stack provides a Standalone Server that can be started without using a servlet container or application server. The Standalone Server uses the following runtime configuration files by default (when started through *<Software AG_directory>WS-Stack/bin/axis2server.bat|sh*):

1. Runtime configuration file:

   *<Software AG_directory>/WS-Stack/conf/axis2.xml*

2. Repository directory:

   *<Software AG_directory>/WS-Stack/repository*

   a. Web services directory:

      *<Software AG_directory>/WS-Stack/repository/services*

   b. Modules directory:

*<Software AG_directory>/WS-Stack/repository/modules*

### Web Services Stack Runtime Configuration in Web Application

Web Services Stack is also distributed as a web application (*wsstack.war* file). In this case the repository is in the web application directory. You can find the *wsstack.war* file in the Web Services Stack installation directory *<Software AG_directory>/WS-Stack/webapp/wsstack*. You can deploy this web application on any servlet container. The following runtime configuration is used:

1. Runtime configuration file:

   *<WSS_WebApp_Directory>/WEB-INF/conf/axis2.xml*

2. Repository directory:

   *<WSS_WebApp_Directory>/WEB-INF/*

   a. Web services directory:

      *<WSS_WebApp_Directory>/WEB-INF/services*

   b. Modules directory:

      *<WSS_WebApp_Directory>/WEB-INF/modules*

### Web Services Stack Runtime Configuration in Platform Tomcat Server

Web Services Stack is also integrated and distributed with Platform Tomcat Server. Web Services Stack uses the following runtime configuration when the Platform Tomcat Server is started:

1. Runtime configuration file:

   *<Software AG_directory>/profiles/CTP/workspace/wsstack/repository/conf/axis2.xml*

2. Repository directory:

   *<Software AG_directory>/profiles/CTP/workspace/wsstack/repository*

   a. Web services directory:

      *<Software AG_directory>/profiles/CTP/workspace/wsstack/repository/services*

   b. Modules directory:

      *<Software AG_directory>/profiles/CTP/workspace/wsstack/repository/modules*

## Client-side Configuration

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

You can use the Web Services Security (WS-Security) specification by giving a value to the `securityConfigFile` parameter in the following way:

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

The value of the parameter in the preceding example must be an absolute or relative path either to the current working directory, or to the *<repository path>/conf* directory and should point to *wsclientsec.properties* (the file that contains security-related information).

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 Axis2 supports Base64 encoding, SOAP with Attachments and MTOM (SOAP Message Transmission Optimization Mechanism).

MTOM is a specification of the World Wide Web Consortium (W3C) that focuses on solving the "Attachments" problem.

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

For more information about MTOM, see the documentation of W3C.

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

You can set the `enableMTOM` parameter at all permitted levels (the global level in *axis2.xml*, the service level in *services.xml*, and the operation level in *services.xml*).

The `enableMTOM` parameter can have three different values:

- **true**
  The engine always responds with MTOM-ized messages in case of included binary data of schema type "xmime:base64Binary". For example:

  ```
  <xsd:element minOccurs="0" name="binaryData" type="xmime:base64Binary"/>
  ```

- **false**
  The response is always non-MTOM-ized, no matter whether the request is MTOM-ized or not.
- **optional**
  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.

# 2 Security

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.

You can configure message-level security in many different ways, 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 Software AG Designer 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 ***Software AG Designer 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.

#### Enabling Keystore Caching

If you want to enable keystore caching, follow this step:

**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.

You can also enable caching of initialized password callback handler classes to additionally improve performance (depending on the cost of instantiation and initialization of the password callback handler). The option to enable caching may be set either globally in the *axis2.xml* configuration, or per service - as a parameter of the service in *services.xml* descriptor. The callback handler instance is always cached on the service instance and will be lost once the service is undeployed.

### Enabling Password Callback Handler Caching

If you want to enable password callback handler caching, follow this step:

**To enable password callback handler caching**

- Set the parameter `cachePasswordCallbackHandler` to true.

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

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="UserDefined" ↩
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:encryptionCypto>
            <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:encryptionCypto>
          </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).

#### Configuring sp:RequiredElements and sp:RequiredParts Assertions Validation

The `sp:RequiredElements` and `sp:RequiredParts` assertion may not be resolved and validated properly if they are available in the security policy. By default, when XPath expressions are handled in `sp:RequiredElements` assertion, the expressions are validated against the soap:Envelope element (instead the soap:Header element). If you want to configure sp:RequiredElements and sp:Required-Parts assertions validation, follow these steps:

**To configure `sp:RequiredElements` and `sp:RequiredParts` assertions validation**

1 Open one of the following configuration files for editing:

- *axis2.xml*

  The configuration of *axis2.xml* file enables the change on the entire runtime.

- *services.xml*

  The configuration of *services.xml* file enables the change on the particular Web service only.

2 Add the following parameter to the XML file content:

```
...
<parameter name="enableRequiredElementsXPathCompatibility">true</parameter>
<parameter name="enableRequiredPartsValidation">true</parameter>
...
```

3 Save your changes.

Alternatively, you can enable `sp:RequiredElements` and `sp:RequiredParts` assertions in the business logic of a Web service client using the following code snippet:

```
IWSStaxClient client = SampleService;
client.getWSOptions().setProperty("enableRequiredElementsXPathCompatibility", "true");
```

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

For more information, see ***Software AG Designer Plug-in***, **Web Services Security: SOAP Message Security 1.1**, and **WS-Security Policy Language**.

#### Modifying the WS-I Basic Security Profile Compliance Mode

The `wsiBSPCompliant` boolean parameter in the *services.xml* file for your service enables you to activate and deactivate the WS-I Basic Security Profile compliance mode. By default, this parameter is set to true.

**To deactivate the WS-I Security Profile compliance mode**

- Set the `wsiBSPCompliant` to false.

```
<ramp:wsiBspCompliant>false</ramp:wsiBspCompliant>
```

**Note:** Any value other than false is interpreted as true.

For more information about the usage of the WS-I Basic Security Profile compliance mode, see its specification at **http://ws-i.org/profiles/BasicProfile-2.0-2010-11-09.html**.

### 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 followi<br>■ The UsernameToken function sets this name in the UsernameToken.<br>■ The signing function uses this name as the alias name in the *keystore* to get t perform signing.<br>■ The encryption function uses this parameter as fallback if `ENCRYPTION_USE`<br>. |
| ENCRYPTION_USER | The user's name for encryption. The encryption function uses the public key o generated symmetric key. If this parameter is not set, then the encryption funct to get the certificate. |
| USER_CERTIFICATE_ALIAS | The alias of the key pair in the keystore, to get the private key used for the sig 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 hea callback class is the org.apache.rampart.PolicyBasedResultsValidator. |
| TIMESTAMP_PRECISION_IN_MS | Defines whether timestamp precision is in milliseconds.<br>The default value is TRUE. |

| Key | Description |
|---|---|
| | The expected values are TRUE or FALSE.<br><br>This parameter is passed to wss4j WSSConfig. The setting concerns the T<br>in the security header. If the precision is set to be in milliseconds, the t<br>the time stamp is written in the following simple date format: yyyy-M |
| TIMESTAMP_TTL | Timestamp time-to-live in seconds. An integer value is expected. Defa |
| TIMESTAMP_MAX_SKEW | Used in timestamp validation where the timestamp creation timestamp<br>skew. The max time skew is an integer and is expected in seconds. De |
| PASSWORD_CALLBACK_HANDLER_CLASS | A class that implements the `javax.security.auth.callback.Cal`<br>module loads the class and calls the callback method to get the passwor<br>with no parameters. |
| OPTIMIZE_PARTS_EXPRESSIONS | A list of Xpath expressions that refer to nodes that must be MTOM-op<br>delimited list of Xpath expressions. Note that if this property is set, it<br>expressions and does not add them to the list. |
| OPTIMIZE_PARTS_NAMESPACES | A list of namespaces that is taken into consideration when searching f<br>essential for the correct retrieval of the nodes from the document, that<br>OPTIMIZE_PARTS_EXPRESSIONS list are recognized by the optimiz<br>namespaces:<br><br>`xmlns:ds=http://www.w3.org/2000/09/xmldsig#`<br>`xmlns:xenc=http://www.w3.org/2001/04/xmlenc#`<br>`xmlns:wsse=http://docs.oasis-open.org/wss/2004/01/oa`<br>`xmlns:wsu=http://docs.oasis-open.org/wss/2004/01/oas`<br><br>plus all the declared namespaces here. That property is expected as a<br>declarations (for example, `OPTIMIZE_PARTS_NAMESPACES=xmlns:n`<br>).<br><br>**Note:** If this property is set, it overwrites any previously configured li<br>list. |
| CRYPTO_PROVIDER_SIGN | The WSS4J-specific Crypto implementation that is to be used for gene<br>following:<br><br>■ `org.apache.ws.security.components.crypto.Merlin` is the<br>■ `org.apache.ws.security.components.crypto.BouncyCastl` |
| KEYSTORE_PROVIDER_SIGN | The signature keystore provider.<br><br>If not set the JVM uses the default (normally Oracle) keystore provide<br>`java.security.Provider` javadocs. |
| KEYSTORE_TYPE_SIGN | The signature keystore type. If not set, the JVM uses the default keystore<br>refer to the `java.security.KeyStore#getDefaultType() metho` |
| KEYSTORE_FILE_SIGN | The signature keystore file. |
| KEYSTORE_PASSWORD_SIGN | The signature keystore password. |

| Key | Description |
|---|---|
| CRYPTO_PROVIDER_ENCRYPT | The WSS4J-specific Crypto implementation to use for encryption. It can be set<br>■ org.apache.ws.security.components.crypto.Merlin<br>This is the default one if the property is not set.<br>■ org.apache.ws.security.components.crypto.BouncyCastle |
| KEYSTORE_PROVIDER_ENCRYPT | The encryption keystore provider. If not set the JVM uses the default (normall<br>For additional information, refer to `java.security.Provider` javadocs. |
| KEYSTORE_TYPE_ENCRYPT | The encryption keystore type. If not set, the JVM uses the default keystore typ<br>For additional information, refer to `java.security.Provider` 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 S<br>following:<br>■ `org.apache.ws.security.components.crypto.Merlin` is the default<br>■ `org.apache.ws.security.components.crypto.BouncyCastle` |
| KEYSTORE_PROVIDER_STS | The keystore provider used in case of a STS. If not set the JVM uses the defaul<br>For additional information, refer to `java.security.Provider` javadocs. |
| KEYSTORE_TYPE_STS | The keystore type used in case of a STS. If not set, the JVM uses the default ke<br>For additional information, refer to the`java.security.KeyStore#getDefa` |
| 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 *keystore* specified under `KEYSTORE_SSL_LOCATION`. |
| SSL_KEYSTORE_PASSWORD | The password for the *keystore* specified under KEYSTORE_SSL_LOCATION. T<br>`javax.net.ssl.keyStorePassword` system property. |
| KEYSTORE_SSL_LOCATION | The *keystore* file for SSL authentication. This property corresponds to the JSSE<br>property.<br>For more information, refer to the **JSSE Reference Guide**.<br>**Note:** Specifying the *keystore* is required only if the remote SSL server requires |
| TRUSTSTORE_SSL_LOCATION | The truststore file for SSL authentication. The client requires that the server's c<br>it is trusted. This property corresponds to the JSSE `javax.net.ssl.trustSt`<br>not set the client falls back to *<java-home>lib/security/jssecacerts* and *<java-home>*<br>**Note:** For more information, refer to the **JSSE Reference Guide**. |
| TRUSTSTORE_SSL_PASSWORD | The password for the truststore specified under `TRUSTSTORE_SSL_LOCATION`<br>`javax.net.ssl.trustStorePassword` system property. |

| Key | Description |
|---|---|
| | **Note:** For more information, refer to the **JSSE Reference Guide**. |

**Note:** The last five entries 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(WSClientConstants.KEYSTORE_PASSWORD_SIGN, "changeit");
options.setProperty(WSClientConstants.KEYSTORE_FILE_SIGN, "C:\\client.jks");

//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.

#### Configuring Tomcat to Use SSL at the Server Side

If you want to configure Tomcat to use SSL at server side, follow these steps:

**To configure Tomcat to use SSL at the server side**

1 Navigate to *<Software AG_directory>/profiles/CTP/configuration/tomcat/conf* and open the *server.xml* file to configure an SSL Connector.

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="10011" 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 default value of the `connector port` is 10011. The port number is calculated during the installation of the product.

If a server declares explicitly the use of the HTTPS transport in its *services.xml*, you have to make sure 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.

```
<transportReceiver name="https"
class="com.softwareag.wsstack.transport.http.HTTPSListener">
        <parameter name="port">10011</parameter>
    </transportReceiver>
```

The 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.

#### Configuring SSL at the Client Side

If you want to configure SSL at the client side, follow these steps:

**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 (in the previous example "10011").

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:
IWSStaxClient client = ...

...

IWSOptions options = client.getWSOptions();

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

//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 javax.net.ssl.keyStorePassword` system property. |
| KEYSTORE_SSL_LOCATION | The keystore file for SSL authentication. This property corresponds to the `JSSE javax.net.ssl.keyStore` system property. Note that specifying the keystore is required only if the remote SSL server requires client authentication.<br><br>For more information, refer to the **JSSE Reference Guide**. |
| 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 `javax.net.ssl.trustStore` system property. If the property is not set, the client falls back to *<java-home>lib/security/jssecacerts* and *<java-home>/lib/security/cacerts* in that order.<br><br>For more information, refer to the **JSSE Reference Guide**. |
| TRUSTSTORE_SSL_PASSWORD | The password for the truststore specified under TRUSTSTORE_SSL_LOCATION. This property corresponds to the `javax.net.ssl.trustStorePassword` system property. |

| Property name | Description |
|---|---|
| | For more information, refer to the **JSSE Reference Guide**. |

## SSL with Client Authentication

The information in this topic is organized under the following headings:

- Server-side Configuration
- Client-side Configuration

### Server-side Configuration

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

#### Using Client Authentication with Tomcat

If you want to use client authentication with Tomcat, do the following:

**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.

#### Configuring the Truststore Location of Tomcat by Using the Respective Java System Property

If you want to configure the truststore location of Tomcat by starting it with the respective Java system property, follow this step:

**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="10011" 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 HTTPS, see how to configure Tomcat to use SSL at server side in *Prerequisites for the Setup and Use of Transport-Level Security*

### 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-contib) that is part of the commons-httpclient project, namely `AuthSSLProtocolSocketFactory`. This can be set in the following way:

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

### Setup and Use of HTTP Basic Authentication

The basic HTTP authentication is a common transport security mechanism. The server sends requests to the client to provide its credentials in an HTTP authorization header. Thus the mechanism provides both authentication and authorization means. The enforcement of the basic HTTP authentication request can be delegated to the servlet container or can be left to the Web Services Stack security module (rampart). More information and examples of how to set up your basic HTTP authentication scenario is given below.

#### Configuring a Specific Endpoint to Use the HTTP Basic Authentication Scheme

If you want to configure a specific endpoint to use the HTTP basic authentication scheme, follow these steps:

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

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.

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.

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.

#### Using the HTTP Basic Authentication with Tomcat

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

**Note:** By default, the installed Tomcat does not contain a *tomcat-users.xml* file. If *tomcat-users.xml* is missing in the configuration directory of Tomcat Server, you must create it manually.

#### 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 about 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.

### Validating Basic HTTP Authentication in Rampart

If you want to validate basic HTTP authentication in Rampart, do the following:

#### To validate basic HTTP authentication in Rampart

- The Rampart security module validates the usage of basic HTTP authentication. Rampart does not authenticate the user credentials sent in the HTTP header and only asserts whether the credentials are available. To authenticate successfully, you can use JAAS integration in Web Services Stack. For more information about that, see ***Client Authentication***

  To avoid malfunction of the functionality, the *wsstack* must be running inside a servlet container or a server such as Software AG Integration Server. This is required because rampart must be able to interact with the actual transport layer by accessing the transport level credentials and sending authorization request in case the basic HTTP authentication header is missing.

  To validate basic HTTP authentication, rampart must be informed that the service is secured by Web service Security Policy. The following security policy snippet denotes the basic HTTP authentication requirement:

```
<service name="ExampleService" ...>
...
<wsp:Policy xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy" ↩
xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702" ↩
xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd" ↩
wsu:Id="user">
  <wsp:ExactlyOne>
   <wsp:All>
    <sp:TransportBinding ↩
xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702">
     <wsp:Policy>
      <sp:TransportToken>
       <wsp:Policy>
        <sp:HttpsToken>
         <wsp:Policy>
          <sp:HttpBasicAuthentication />
         </wsp:Policy>
        </sp:HttpsToken>
       </wsp:Policy>
      </sp:TransportToken>
      <sp:AlgorithmSuite>
       <wsp:Policy>
        <sp:Basic256 />
       </wsp:Policy>
      </sp:AlgorithmSuite>
      <sp:Layout>
       <wsp:Policy>
        <sp:Lax />
```

```
       </wsp:Policy>
      </sp:Layout>
      <sp:IncludeTimestamp />
     </wsp:Policy>
    </sp:TransportBinding>
    ...
   </wsp:All>
  </wsp:ExactlyOne>
 </wsp:policy>
</service>
```

The `sp:HttpBasicAuthentication` assertion can appear only inside of an `sp:HttpsToken` assertion which means that the server also requires the usage of HTTPS transport. To use this feature you must engage rampart for your Web service. To engage rampart for the Web Service, add the following lines in the service descriptor (*services.xml*):

```
<service name="ExampleService" ...>
 ...
 <module ref="rampart"/>
</service>
```

Add a policy that contains the `sp:HttpBasicAuthentication` to your Web service (such as in the example snippet above).

```
<service name="ExampleService" ...>
 ...
 <sp:HttpsToken>
  <wsp:Policy>
   <sp:HttpBasicAuthentication />
  </wsp:Policy>
 </sp:HttpsToken>
...
</service>
```

#### Configuring your Web Service Client to Use HTTP Basic Authentication

If you want to configure your Web service client to use HTTP basic authentication, follow these steps:

**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)WSClientFactory.newClient(
  WSClientConstants.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);
```

**Important:** In the example above, you must supply the `HttpTransportProperties` Authenticator object first, and then set up a user name and a password. 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 JAAS-based `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

### 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* directory 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

User implemented password callback handlers are used to:

- Retrieve passwords to be placed inside a UsernameToken (corresponding to a given user name).
- Retrieve passwords to access user private keys from a keystore (the keystore password itself is directly set in the ramart configuration)
- Verify password in received UsernameToken (corresponding to a given user name).

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 `PWCBLDAPPropFile` 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="PWCBLDAPPropFileLocation"> common_properties_file_location ↩
</parameter>
...
 </service>
</serviceGroup>
```

- 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.
- The file may be also placed in a Java archive (.jar file) which resides in the *WEB-INF/lib* (for example, *pwcb-server.jar*) or directly in *WEB-INF/classes* directory. 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 in 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.

### 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 respecivtely 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 (in the Platfrom Tomcat Server, SIN artifacts are already available for Web services deployed as "aar" archives). 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*)
- To detect any changes in the configuration, the built-in policy validators provided by Web Services Stack automatically refresh the JAAS configuration prior to each login attempt. Since the configuration is shared for the entire Java virtual machine instance, this detection results in increased synchronization wait time on the server side. To improve the performance, you can disable the automatic refresh feature by setting the `autoRefreshJaasConfig` parameter to false.

The parameter can be set globally in the *axis2.xml* configuration file or locally in the *services.xml* service descriptor. The following excerpt outlines the configuration of the parameter:

```
<parameter name="autoRefreshJaasConfig">false</parameter>
```

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

For information about the authentication steps, see SIN documentation in the *webMethods Product Suite* directory on the **Software AG Documentation Web site**.

# 3 Transports

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

- HTTP or HTTPS
- TCP
- JMS
- Mail

By default, the HTTP transport is activated in all three Web Services Stack distributions (Web Services Stack in standalone server, Web Services Stack in web application, and Web Services Stack in Software AG Runtime). The HTTPS transport is activated by default only in the Web Services Stack in web application distribution, and needs to be explicitly enabled in the other installations. When you deploy a Web service archive file on the Web Services Stack runtime that runs on the Software AG Runtime server, service endpoints for all enabled transports are created automatically.

The following instructions show how to configure and activate or deactivate the other transports supported by Web Services Stack.

The information is organized under the following headings:

## HTTP/HTTPS Transport

By default, the HTTP transport is activated in all three Web Services Stack distributions (Web Services Stack in standalone server, Web Services Stack in web application, and Web Services Stack in Software AG Runtime). The HTTPS transport is activated only in the Web Services Stack in web application distribution, and needs to be explicitly enabled in the other installations. The following procedure outlines the steps you must perform to enable or disable the HTTP and HTTPS transports.

**Note:** If you disable a connector in Software AG Tomcat Server, make sure that the respective transport sender and receiver are disabled on the Web Services Stack side as well (*axis2.xml*). If a transport is enabled in the *axis2.xml* file and the respective connector in the *server.xml* file is disabled, an error occurs on the Web Services Stack side.

**Note:** Make sure at least one of your HTTPS connectors has the `description` property set to Default HTTPS Connector. This value exists by default in the predefined HTTPS connector definition. The Default HTTPS Connector value is used by the Common Platform to distinguish default connectors from other existing connectors. If this value is not present in at least one HTTPS connector definition, this results in an invalid or corrupted Software AG Runtime configuration the next time you install or upgrade a product in the profile.

## Activating or Deactivating HTTP or HTTPS Transport

The information in this topic is organized under the following headings:



### Activating or Deactivating HTTP or HTTPS in Web Services Stack

If you want to activate or deactivate HTTP or HTTPS in Web Services Stack, follow these steps:

#### To activate or deactivate HTTP or HTTPS 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 in the *webapps/wsstack/WEB-INF/conf* directory for the Web Service Stack web application.

**Note:** For Web Services Stack that uses Software AG Runtime Server the *axis2.xml* file is in the following directory: `<Software AG_directory>`/*profiles/CTP/workspace/wsstack/repository/conf/*.

3 Comment out the sections that define the transport receiver and transport sender with name="http" or name="https":

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

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

4 Restart the Web Services Stack runtime for the modifications to take effect.

For Web Services Stack that uses Software AG Runtime Server, restart the "Software AG Runtime" Windows Service (using the Control Panel > Administrative Tools > Services).

### Activating or Deactivating HTTP or HTTPS in Software AG Runtime Server

If you want to activate or deactivate HTTP or HTTPS in Software AG Runtime server, follow these steps:

#### To activate or deactivate HTTP or HTTPS in Software AG Runtime server

1 Open the *server.xml* configuration file under the *<Software AG_directory>/profiles/CTP/configuration/tomcat/conf/* directory.

2 Comment out the sections that define the HTTP and HTTPS connectors, for example:

```
<clientAuth="false"
sslProtocol="TLS"
SSLEnabled="true"
keystoreFile="C:/SoftwareAG/profiles/CTP/configuration/tomcat/conf/localhost_dont_use_in_production.jks"
enabled="true"
port="10011"
keystorePass="change_this_password"
scheme="https"
enableLookups="false"
secure="true"
alias="defaultHttps"
maxSpareThreads="75"
maxThreads="150"
server="SoftwareAG-Runtime"
keystoreType="JKS"
disableUploadTimeout="true"
description="Default HTTPS Connector"
algorithm="SunX509"
minSpareThreads="25"
acceptCount="100"
maxHttpHeaderSize="8192"/>
```

3 Restart Software AG Runtime Server by restarting the "Software AG Runtime" Windows Service (using the Control Panel > Administrative Tools > Services).

## 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 for the Web Service Stack web application.

**Note:** For Web Services Stack that uses Platform Tomcat Server the *axis2.xml* file is in the following directory: `<Software AG_directory>`*/profiles/CTP/workspace/wsstack/repository/conf/*.

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 may not be 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 Designer 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 Designer plug-in, see *Software AG Designer 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 Designer 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 in Web Services Stack Web Application



### Prerequisites

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

### Installing and Starting a Message Broker

If you want to install and start a message broker, do the following:

#### 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 comand prompt in which it is running.

### Running the ActiveMQ Message Broker

If you want to run the Active MQ Message Broker, follow these steps:

#### To 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***.

### Providing Additional Libraries for the WSS Runtime

If you want to provide additional libraries for the Web Services Stack runtime, follow these steps:

#### 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 for the Web Service Stack web application.

**Note:** For Web Services Stack that uses Platform Tomcat Server the *axis2.xml* file is in the following directory: `<Software AG_directory>`*/profiles/CTP/workspace/wsstack/repository/conf/*.

3 Uncommment 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.

**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.

#### Deploying over JMS Transport Only

If you want to deploy over JMS transport only, follow this step:

**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 Designer plug-in at deployment time by selecting **JMS Transport** from the list of transports in the **Services** tab.

### Specifying the Connection Factory Name

You can specify a name for the connection factory that the Web service will use. This can be done by modifying directly the *services.xml* file, or by using the WSS Designer plug-in. The parameters that define the connection factory name 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.

You can specify the connection factory name through the *services.xml* file by adding the elements in bold. The connection factory can be any of the connection factories defined in *axis2.xml* and the destination name can be anything. "transport.jms.ConnectionFactory" and "myQueueConnection-Factory" are samples for values of parameters.

```
<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>
```

You can also use the WSS Designer plug-in to specify the connection factory name.

#### To specify the connection factory name using the WSS Designer plug-in

1 In the **Project Explorer** view, select the Web service archive that will use the connection factory.

2 Click the **Services** tab.

3 To specify the connection factory, in the **Properties** section:

   1. Click **Add**.
   2. Type "transport.jms.ConnectionFactory" in the **Name** field.
   3. Type "myQueueConnectionFactory" (or another connection factory defined in *axis2.xml*) in the **Value** field.
   4. Click **OK**.

4 To add the destinations:

   1. Click **Add** .
   2. Type "transport.jms.Destination" in the **Name** field.
   3. Type "dynamicQueues/TestQueue" (or other value of your choice) in the **Value** field.

4. Click **OK**.

The connection factory name is now set and visible in the **Services.xml** tab.

For more information about working with the Web Services Stack Designer plug-in, see *Software AG Designer 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 repositor.

## Mail Transport in Web Services Stack Web Application

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
- Creating 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.

### Installing Apache James Server

If you want to install the Apache James Server, follow these steps:

#### 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.

### Opening the Configuration Files for Editing

If you want to open the configuration files for editing, follow these steps:

#### 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).

### Configuring the DNS Servers in the Mail Server

If you want to configure the DNS servers in the mail server, follow these steps:

#### 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***.

### Creating 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 instructions and the description of the required parameters for the transport receiver and the transport sender.

### To activate mail transport in Web Services Stack

1 Open the *axis2.xml* configuration file under the *webapps/wsstack/WEB-INF/conf* directory for the Web Service Stack web application.

**Note:** For Web Services Stack that uses Platform Tomcat Server the *axis2.xml* file is in the following directory: `<Software AG_directory>`*/profiles/CTP/workspace/wsstack/repository/conf/*.

2 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>
```

3 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.<br><br>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.<br><br>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". |

| Parameter | Description |
|---|---|
| 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 followng values:<br>■ Supplies the endpoint reference for the response and represents the server email address.<br>■ Contains the user name specified in the `mail.pop3.user` parameter and the server name of James mail server, separated by the @ sign.<br>**Note:** The server name is configured in the *JAMES_HOME/apps/james/SAR-INF/config.xml* configuration file. If you have not specified a different one, the initial value is "localhost". |
| transport.listener.interval | Controls the time interval (in milliseconds) for checking the mail server for new messages.<br>**Note:** 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 its description:

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

| Parameter | Description |
|---|---|
| mail.smtp.from | Corresponds to the value of the `mail.transport.replyToAddress` parameter of the transport receiver. |

Following is a sample code listing of the usage of the required parameter 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>
```

# 4 Monitoring and Logging

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

The information is organized under the following headings.

## SOAP Monitor in Web Services Stack

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

The information is organized under the following headings:



### 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 (general purpose tcp 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 SOAP Monitor in Web Services Stack Web Application

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 Copy the following `SOAPMonitor` classes from *soapmonitor* folder and paste 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.

5 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.

6 Restart Tomcat or the *wsstack webapp*.

7 Go to *http://<host>:<port>/wsstack/SOAPMonitor* to start using the SOAP monitor.

For more details on the SOAP monitor configuration, see ***http://axis.apache.org/axis2/java/core/docs/soapmonitor-module.html***.

### Using SOAP Monitor in Web Services Stack on Tomcat Platform Server

SOAP monitor is disabled by default.

#### To enable SOAP monitor

1 Open the *axis2.xml* file that is located in the `<Software AG_directory>\profiles\CTP\workspace\wsstack\repository\conf` directory.

2 Engage the *soapmonitor* Axis2 module globally in the *axis2.xml* by adding the following line:

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

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

3 Add a `soapMonitorPort` parameter which defines the port to use for communication with the SOAP Monitor Applet

```
<parameter name="soapMonitorPort">5001</parameter>
```

**Note:** If the port parameter is missing or empty, the SOAP Monitor servlet will not be available.

4 Restart the Platform Tomcat Server.

5 Go to *http://<host>:<port>/wsstack/SOAPMonitor* to start using the SOAP monitor.

For more details on the SOAP monitor configuration, see ***http://axis.apache.org/axis2/java/core/docs/soapmonitor-module.html***.

## Logging in Web Services Stack Web Application

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

The information is organized under the following headings:

- Overview

### Overview

Web Services Stack uses Apache Commons 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_directory>/webapp/wsstack/WEB-INF/classes*.

**Note:** Those files are also included in the *wsstack.war* web archive in *<Web Services Stack_Install_directory>/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 |
|---|---|
| fatal | Severe errors that cause premature termination. Expect these to be immediately visible on a status console. |
| error | Other runtime errors or unexpected conditions. Expect these to be immediately visible on a status console. |
| warn | Use of deprecated APIs, poor use of API, error-like situations, other runtime situations that are undesirable or unexpected, but not necessarily incorrect. Expect these to be immediately visible on a status console. |
| info | Interesting runtime events (start /shut down). Expect these to be immediately visible on a console, so be conservative and keep to a minimum. |
| debug | Detailed information on the flow through the system. Expect these to be written to logs only. |
| trace | More detailed information. Expect these to be written to logs only. |

## Logging in Web Services Stack on Platform Tomcat Server

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

The information is organized under the following headings:

- Overview
- Log4J Integration and Logging Levels

### Overview

Web Services Stack running on the Platform Tomcat Server uses Journal Logging as a logging mechanism. The Journal Logging is delivered with the following shared component bundle: `com.softwareag.sc.core` and its configuration file is located in the following directory on the file system: *<Software AG_directory>/profiles/<profile>/configuration/logging/log_config.xml*

Basically, the format of the *log_config.xml* file is the same as the format of the Log4J XML configuration. Apart from that, the Journal Logger contains several additional appenders than the standard Log4J appenders.

To enable logging and configure the corresponding severity, edit the *log_config.xml* file and edit the following excerpt as shown in the sample below:

```
<root>
        <level value="info" />
        <appender-ref ref="Platform.Console" />
        <appender-ref ref="Platform.RollingLogFile" />
</root>
```

### Log4J Integration and Logging Levels

The Journal Logger is a wrapper around Log4J and every Journal Logging (JL) logger wraps a standard Log4J logger. For this reason the JL component delivers Log4J as part of its implementation. The JL configuration is a standard Log4J configuration that sets up the underlying Log4J library. If you need you can use Log4J directly. You should add your Log4J settings to the JL configuration file.

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

- `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, as well.

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

## System Management Hub 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 System Management Hub agents. They manipulate the Web Services Stack environment under the System Management Hub web interface. For more information, see Administration Tool for details.

If you experience problems when using the administration tool, you must enable the logging for the System Management Hub 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 may decrease.

### To enable logging on System Management Hub agents

1 Start the web interface of System Management Hub in a web browser.

2 To open the registry editor, navigate to:

**Managed Hosts** > *<host_name>* > **System Management Hub** > **Registry** > **HKEY_LOCAL_MACHINE\SOFTWARE\Software AG** > **System Management Hub** > **Products** > **Web Services Stack 9.0** > **Versions** > **9.0** > **Parameters**.

3 Right click the **Parameters** node.

4 On the menu that opens, click **Modify Value**.

5 Switch the value of the registry parameter `enableLog` to "1".

6 Click **OK**.

You can find the output log file in *<Software AG_directory>/WS-Stack/argus/wsstack.log*.