Oracle® Fusion Middleware

Oracle API Gateway Explorer User Guide
11g Release 2 (11.1.2.4.0)

March 2015

Oracle API Gateway Explorer User Guide, 11g Release 2 (11.1.2.4.0)

13 March 2015

Contents

Oracle API Gateway Explorer Overview

Overview

Oracle API Gateway Explorer enables developers to test the performance, scalability, and security of enterprise services. Using API Gateway Explorer, developers can test how services perform under load, how they deal with unexpected input, and what their traffic ceiling is.

API Gateway Explorer can automatically generate request messages from the service interfaces defined in WSDL files. Requests can then be signed and encrypted using XML Signature and XML Encryption before being sent to a target system. Traditional security technologies such as SSL and HTTP authentication are also supported, as well as next-generation security paradigms such as WS-Security and SAML.

Using WSDL files in this manner, you can create test suites for each service defined in the WSDL, where each suite comprises several test cases. Test cases can then be run in parallel to stress test a target service.

The following sections describe some of the high-level functionality available in Oracle API Gateway Explorer.

Stress Test Services

How do your services perform under stress? What are your Web Service's traffic ceilings? What happens when they receive more traffic than they can cope with? API Gateway Explorer answers these questions with comprehensive services stress testing.

Traffic Simulation

Use API Gateway Explorer to create and run your own battery of tests against internal application servers and Enterprise Service Bus (ESB) platforms.

Sample SOAP Messages

Get started quickly with pre-built SOAP messages provided as standard.

Application-level Networking

API Gateway Explorer can create signed and encrypted XML messages without any requirement for coding. It supports SSL, WS-Security, XML Signature, XML Encryption, and SAML.

Test Federated Identity Deployments

API Gateway Explorer creates SAML authentication and authorization assertions to test them against federated identity infrastructures. Creating SAML assertions using API Gateway Explorer is significantly more simple than using a programming toolkit for the same purpose.

Test XML, REST, and SOAP

Non-SOAP services can also be tested using API Gateway Explorer. In addition, services that are only called directly by browsers (for example, REST services) can also be tested using API Gateway Explorer.

SOAP Attachments

You can use API Gateway Explorer to attach SOAP attachments to SOAP messages, simply and easily.

Simple Graphical Keystore

Encryption key management is difficult using programming toolkits. API Gateway Explorer turns key management into a point-and-click task using a graphical keystore. Users can load multiple keys and certificates without using command-line tools. With the keystore, you can encrypt XML for multiple recipients in one simple step.

Add or Remove Security Tokens

Automatically remove WS-Security tokens from messages so that you can quickly and simply move on to your next test.

Transfer Encoding

API Gateway Explorer includes all the varieties of transfer encoding used by Web services platforms.

Testing Tool for Design-time Governance

In a SOA Governance environment, API Gateway Explorer enables policies to be tested and refined at design time, before they are deployed. This enables a virtual circle of service definition, policy definition, policy testing, and policy refinement.

System Requirements

Prerequisites

WS-I Toolkit:
Before running the WS-I compliance test on imported WSDL files, first ensure the WS-I testing toolkit is installed on the same machine on which you are running API Gateway Explorer. You must specify the location of the toolkit on the **WS-I Settings** screen on the global **Preferences** screen, available in the **Window > Preferences** menu option. You can download the toolkit from www.ws-i.org [http://www.ws-i.org/].

Requirements

This section describes the system requirements for API Gateway Explorer:

Platform	Supported Versions
Windows	• Windows 7 • Windows Server 2008 • Windows XP Professional
Solaris	• Solaris 10
Linux	• Centos 5.2 • Debian GNU/Linux 5.0 (Lenny) • Oracle Linux 5.5 • Red Hat Enterprise Linux 5, 6.2 • SUSE Linux Enterprise Server 11 • Ubuntu 11.04, 12.04 Both i386 and x86_64 architectures are supported.

Important

API Gateway Explorer may not run on systems that do not meet these requirements. When new Linux and distributions are released, Oracle modifies and tests its products for stability and reliability on these platforms. Oracle makes every effort to add support for new kernels and distributions in a timely manner. However, until a kernel or distribution is added to this list, its use with Oracle products is not supported. If you are running any popular Linux distribution with a 2.6 kernel and libc version 6, Oracle endeavors to support you if issues arise.

Installation Instructions

For details on how to install API Gateway Explorer, see the *API Gateway Installation and Configuration Guide*.

API Gateway Explorer Release Notes

Overview

The following release notes describe installation, running, and other known issues for API Gateway Explorer.

In this Version

The following features are available in this version of API Gateway Explorer:

- Auto-generation of SOAP messages from WSDL files.
- Support for generation of SOAP 1.2 messages from WSDL files.
- Check WSDL files for WS-I compliance.
- Support for different WSU, WSSE, and SOAP namespaces.
- Advanced stress testing functionality.
- Improved interface.
- Ability to create, save, and load test suites.
- Ability to create and send multiple requests to a Web service as part of a test suite.
- Extra general configuration options, including JVM arguments and trusted certificates for connecting to SSL servers.

Installation

Do not install into a directory that contains the following characters: `';?@=&$-_+! *'()'`

Documentation

The documentation cannot be viewed from the CD when the CD-ROM driver adheres strictly to the ISO 9600 standard. This is the international standard for CD-ROM file format, and does not allow filenames longer than 8 characters, or filenames containing capital letters. As a result several of the document links are broken. This problem should only manifest itself when reading the documentation from the CD on Solaris machines. The work around here is to copy the documentation on to the local drive when you wish to view the documentation.

Acknowledgements

This product includes software developed by the Apache Software Foundation [http://www.apache.org/]

This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit (http://www.openssl.org/) [http://www.openssl.org/]. It also includes software written by Tim Hudson (tjh@cryptsoft.com). For more information, please refer to the full license terms in *OpenSSL License Issues*.

This product includes software developed by James Cooper.

OpenSSL License Issues

Overview

The OpenSSL toolkit stays under a dual license, i.e. both the conditions of the OpenSSL License and the original SSLeay license apply to the toolkit.

See below for the actual license texts. Actually both licenses are BSD-style Open Source licenses. In case of any license issues related to OpenSSL please contact openssl-core@openssl.org.

OpenSSL License

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- All advertising materials mentioning features or use of this software must display the following acknowledgment:
  "This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit. (http://www.openssl.org/)"
- The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact openssl-core@openssl.org.
- Products derived from this software may not be called "OpenSSL" nor may "OpenSSL" appear in their names without prior written permission of the OpenSSL Project.
- Redistributions of any form whatsoever must retain the following acknowledgement:
  "This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit (http://www.openssl.org/)"

THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT ``AS IS'' AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE OpenSSL PROJECT OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
==========

This product includes cryptographic software written by Eric Young (eay@cryptsoft.com). This product includes software written by Tim Hudson (tjh@cryptsoft.com).

Original SSLeay License

This package is an SSL implementation written by Eric Young (eay@cryptsoft.com).

The implementation was written so as to conform with Netscape's SSL.

This library is free for commercial and non-commercial use as long as the following conditions are adhered to. The following conditions apply to all code found in this distribution, be it the RC4, RSA, hash, DES, etc, code; not just the SSL code. The SSL documentation included with this distribution is covered by the same copyright terms except that the holder is Tim Hudson (tjh@cryptsoft.com).

Copyright remains Eric Young's, and as such any Copyright notices in the code are not to be removed. If this package is used in a product, Eric Young should be given attribution as the author of the parts of the library used.

This can be in the form of a textual message at program startup or in documentation (online or textual) provided with the package.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- All advertising materials mentioning features or use of this software must display the following acknowledgement:
  This product includes cryptographic software written by Eric Young (eay@cryptsoft.com)"

  The word 'cryptographic' can be left out if the routines from the library being used are not cryptographic related.
- If you include any Windows specific code (or a derivative thereof) from the apps directory (application code) you must include an acknowledgement:
  "This product includes software written by Tim Hudson (tjh@cryptsoft.com)."

THIS SOFTWARE IS PROVIDED BY ERIC YOUNG ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

The license and distribution terms for any publicly available version or derivative of this code cannot be changed. i.e. this code cannot simply be copied and put under another distribution license (including the GNU Public License).

Introducing Oracle API Gateway Explorer

Overview

You can use the Oracle API Gateway Explorer testing utility to test the security and performance of enterprise services. API Gateway Explorer is a vital tool for administrators who want to test the policies acting on a service before moving the service from a testing environment to a production environment.

API Gateway Explorer can generate SOAP messages automatically from WSDL files and display them. Application-level security can then be applied to messages by adding XML Signatures, XML Encryption segments, SAML assertions, and WS-Security tokens. Requests can then be sent to the target service to make sure it is processing the security tokens appropriately. Traditional security technologies, such as SSL and HTTP authentication, are also supported.

Likewise, API Gateway Explorer can check the operational performance of the service through its comprehensive stress testing capabilities. By sending multiple request messages to the service in parallel, API Gateway Explorer can simulate the type of traffic the service will receive in the production environment.

Subsequent topics describe in detail exactly how to use API Gateway Explorer to perform security and performance testing on services. However, before doing this, this topic introduces some concepts to help define how API Gateway Explorer performs various types of testing.

API Gateway Explorer Classic View

Users of previous versions of API Gateway Explorer will recognize the **Classic** view because it has very similar features to previous incarnations of API Gateway Explorer.

You can use the **Classic** view to load SOAP messages, add security tokens into the messages, and send them to target services to see how they handle them. Whereas older API Gateway Explorer versions required you to copy and paste SOAP messages into the Request panel, API Gateway Explorer 11.1.2.4.0 enables you to auto-generate SOAP messages from the WSDL files that contain the service definitions.

For more details, see the topic on *Using the API Gateway Explorer Classic Mode*.

API Gateway Explorer Design View

The **Design** view enables you to auto-generate a SOAP message, or a Test Case, for each SOAP operation defined in the WSDL file. You can then configure a series of *message filters* to act on each of these messages before they are sent in batch mode to the service. For example, you can add security tokens, HTTP headers, and attachments, before sending them to the target service.

Similarly, you can configure a number of message filters to act on the SOAP response message returned from the service to ensure that the request was handled appropriately. For example, you may want to run an XML Schema against the response, or use an XPath expression to check the value of a particular element in the response, or you may even just want to check the HTTP status on the response.

You can also create a series of *Attack Vector Messages* that contain common attacks, such as SQL injection and XPath injection attacks, and send them to the service to make sure that the security in place can handle such threats.

For more information on generating Test Cases and Attack Vector Messages in the **Design** view, see the following topics:

- *Generating and Running Test Cases*
- *Running Attack Vectors*

Checking WSDL for WS-I Compliance

When using a WSDL file to auto-generate Test Cases, API Gateway Explorer can check the WSDL for compliance with the WS-I Basic Profile. The Basic Profile was designed to smooth out interoperabilty issues between different SOAP vendors. The assertions listed in the Profile are designed to ensure maximum interoperability between different implementations of SOAP and different services.

For more details, see the topic on *Testing WSDL Files for WS-I Compliance*.

Using the Send Request Command

You can use the Send Request utility to stress test services by sending large numbers of requests in parallel from the command line. The Send Request (`sr`) tool ships with a large number of configuration options that enable you to perform complicated tests on Web Services.

For more details, see the topic on *Stress test with send request (sr)*.

Using the API Gateway Explorer Classic Mode

Overview

API Gateway Explorer **Classic Mode** shows a simplified view of a SOAP request and its corresponding response, together with any other relevant information, including HTTP headers, SOAP Action, certificates, and attachments. **Classic Mode** is most suitable for sending single requests to a Web service or target system to see how it behaves. You can perform more complicated testing scenarios using the **Design Mode**.

The following sections describe the functionality available from the **Classic** tab on the right in the toolbar.

Auto-Generating SOAP Messages from WSDL Files

A WSDL file contains definitions of SOAP operations and describes the wire format of SOAP messages for those operations. API Gateway Explorer can use these definitions to auto-generate sample SOAP messages, which can then be sent to the service URL specified in the WSDL file. Complete the following steps to do this:

1. Click the **Load** button next to the **WSDL** field at the top of the API Gateway Explorer interface. The **Load WSDL** dialog is displayed.
2. The WSDL file can be loaded from a file, URL, or UDDI registry. Select the appropriate option, and enter or browse to the location of the URL. For more information on using the UDDI option, see the topic on *Retrieve WSDL files from a UDDI registry*.
3. On the **WSDL Operations** screen, select the SOAP operation for which you wish to auto-generate a sample SOAP message.

Note

Only one operation can be selected when in **Classic Mode**.

4. Click the **Finish** button to generate the SOAP message for the selected operation.
5. The auto-generated SOAP message is displayed in the **SOAP Request** panel of the API Gateway Explorer.

The auto-generated SOAP message contains comments above elements that have certain cardinality rules associated with them to help you create more formally correct SOAP messages. Similarly, where an element's content model comprises a sequence or choice of elements or values, these options are listed in the comment.

SOAP Request and Response

The **SOAP Request** panel contains the currently loaded SOAP message. When a request has been loaded into the panel, you can insert security tokens (for example, XML Signature and XML Encryption) into the message before sending it to the Web service specified in the **URL** field at the top of the API Gateway Explorer screen. The options to modify the message in this way are available to the left of the **SOAP Request** panel.

As discussed in the previous section, you can auto-generate a request for a SOAP operation exposed by a Web service using the WSDL for that service. In this case, the **SOAP Request** panel is automatically populated with the SOAP message. However, you can also load a SOAP message from a file using the **File** > **Load Request** menu option.

A number of sample SOAP messages for live sample Web services ship with the API Gateway Explorer. These samples are available from the **File** > **Samples** menu. When a sample service is selected from this menu, the message for that service is loaded into the **SOAP Request** panel. Furthermore, the **URL**, **SOAP Action**, and **WSDL** fields are all populated accordingly.

When a message has been loaded into the **SOAP Request** panel (or **SOAP Response** panel), there are several differ-

ent views of the message available, which can be selected by clicking one of the following tabs at the bottom of the panel:

- **Design**:
  Displays a tree-view of the SOAP message, which highlights the hierarchical nature of the message. Different node types, including element, attributes, comments, and the XML declaration itself, and their corresponding values are displayed in the Design Mode.
- **Source**:
  This view displays a textual representation of the SOAP message.
- **Headers**:
  Lists all HTTP headers associated with the message in a table. You can **Add**, **Remove**, and **Remove All** headers using the links at the top of the table.
- **Attachments**:
  Lists all attachments associated with the message. You can add and remove attachments using the links at the top of the table. You can also save an attachment to disk by selecting the attachment in the table and clicking the **Save attachment to disk** link at the bottom.

No matter how the message is loaded into the **SOAP Request** panel, it can always be sent using the **Send Request** button on the left of the screen.

When you press the **Send Request** button, the message in the **SOAP Request** panel is sent to the Web service running at the address specified in the **URL** field.

The response from the Web service is displayed in the **SOAP Response** panel. You can view any HTTP headers or attachments that were returned with the SOAP response by clicking the **Headers** and **Attachments** tabs, respectively. Similarly the HTTP response status (for example, `HTTP/1.1 200 OK` is displayed at the top of the panel.

Connection Settings

When you click the **Send Request** button, and if this is the first message you have sent to the Web service running at the address specified in the **URL** field, the **Connection Settings** dialog is displayed. You can configure the following settings:

- Proxy settings for the target Web service.
- CA and server certificates that are considered trusted for SSL purposes.
- Client SSL certificate to use to authenticate to the target Web service.
- A username and password to use to authenticate to the Web service using HTTP basic or digest authentication.

For more details, see the topic on *Configuring Connection Settings*.

Sign Request

As discussed earlier, when a SOAP message has been loaded into the **SOAP Request** panel, you can insert one or more different types of security token into the message. One such token is an XML Signature, which contains a digital signature of (a part of) the SOAP message.

By *signing* the message, the integrity of the message is guaranteed. In other words, any changes to the message after it has been signed can be detected by someone validating the XML Signature on the message.

You can use API Gateway Explorer to check the validating process on the server. Typically, you would achieve this by loading a message into API Gateway Explorer, signing it, and sending it to the server where the signature is being validated. The next step would be to change some of the content covered by the Signature to make sure that the changes are detected by the server.

To sign a message that has already been loaded into the **SOAP Request** panel, select the **Security** > **Decrypt Request**

menu option. You can use the **Sign Message** screen to configure what part of the message is signed, where to place the Signature, and what algorithms to use, along with other details about the signing process. For more details, see the topic on *XML signature generation*.

Encrypt Request

You can also insert an XML Encryption block into the message. The encrypted block (usually) replaces the original XML chunk that was encrypted. By encrypting the message, the sender can make sure that only the intended recipient of the message can read it.

You can use the **Encrypt Request** wizard, which is available by selecting the **Security** > **Decrypt Request** menu option, to encrypt a message that has been loaded into the **SOAP Request** panel.

The first step in configuring the **XML Encryption Wizard** is to select the certificate that contains the public key to use to encrypt the data. When the data has been encrypted with this public key, it can only be decrypted using the corresponding private key. Select the relevant certificate from the list of **Certificates in the Trusted Certificate Store**.

After clicking the **Next** button on the first screen of the wizard, the configuration options for the **XML Encryption Settings** screen are displayed. For more details, see the topic on *XML encryption settings*.

Decrypt Request

When a SOAP message containing an XML Encryption block is loaded into the **SOAP Request** panel, you can decrypt the encrypted block using the **XML Decryption Settings** screen. This is available by selecting the **Security > Decrypt Request** menu option.

For more information on configuring the fields on this screen, see the topic on *XML decryption settings*.

Insert SAML Token

You can insert a SAML authentication or authorization assertion into a SOAP message by selecting the **Security** > **Insert SAML Token** menu option from the **Classic Mode**. There are further menu options to add a SAML authentication and/or authorization assertion.

For more information on inserting SAML tokens into the message, see the topics on *Insert SAML authentication assertion* and *Insert SAML authentication assertion*.

Insert WS-Security UsernameToken

Finally, you can use API Gateway Explorer to insert a WS-Security UsernameToken into a SOAP message. This option is available by selecting the **Security** > **Insert WS-Security Username** menu option.

For more information on inserting a UsernameToken into a message, see the topic on *Insert WS-Security UsernameToken*.

Generating and Running Test Cases

Overview

You can use the Oracle API Gateway Explorer test tool to auto-generate SOAP messages, insert security tokens into them, send them to their target Web services, and validate the responses from them. You can view the results for each step of the overall transaction.

A *Test Case* is used to perform a single test run as described above. The Test Case sets the input message, and then a Test Case *policy* modifies the message in any way necessary (for example, inserting a SAML authentication assertion) before sending them on to the Web service. The response from the Web Service can then be validated using a series of validation filters.

This topic describes the stages involved in creating and running Test Cases, from auto-generating SOAP messages to validating the responses from the Web service.

Using WSDL to Generate Test Cases

A Web services Description Language (WSDL) file contains the interface definitions for a Web service. It lists the physical address of the service, the operations exposed by the service, together with the rules that determine the formats of the SOAP messages that should be sent to the service.

API Gateway Explorer can use this information to automatically generate a Test Case that can then be used to test every operation exposed by the Web Service. A Test Case is created for each operation defined in the WSDL. A single Test Case Input is created for each Test Case, which contains a sample request for the SOAP operation. The sample SOAP message is automatically populated from the operation definition contained in the WSDL file.

The auto-generated Test Case hierarchy can be summarized as follows:

- A Test Suite is generated to group together the Test Cases for the operations defined in the WSDL. binding.
- A Test Case is generated for each operation that has a SOAP binding defined in the WSDL, and that was selected in the **WSDL Operations** screen of the **Load WSDL** wizard.
- Each Test Case contains a single Test Case Input, which consists of an automatically generated sample SOAP message for the operation and an HTTP header containing the corresponding `SOAPAction`.

Complete the following steps to auto-generate a Test Suite, together with its Test Cases:

Step 1: Load the WSDL
The first step in creating a Test Suite is to load a WSDL file. To do this, select the **Load** button next to the **WSDL** field at the top of the API Gateway Explorer interface. The WSDL can be loaded from a file, URL, or a UDDI registry. For more details, see the topic on *Retrieve WSDL files from a UDDI registry*. Click the **Next** button when you have specified the WSDL.

Step 2: Select the Operations
On the **WSDL Operations** screen, select the SOAP operations that you wish to generate Test Cases for by selecting the boxes next to the required operations. Click the **Finish** button when you have selected the operations. You can see your newly generated Test Suite and its Test Cases in the **Navigator** panel.

Note

You can only generate Test Cases for those operations that have a SOAP binding.

Running Test Cases

Having generated some Test Cases using a WSDL file, you can now run them against the Web service.

Important

You may wish to modify the default parameters that were inserted automatically into the message parts in the SOAP message. You can modify message parts by clicking **Test Case Input** in the **Navigator**, and then editing the parts in the **SOAP Message** panel. You should modify any message parts, if necessary, before running the corresponding Test Cases.

Before running Test Cases you must be in **Design Mode**. To switch to this mode, click the **Design** tab on the left in the toolbar.

You can run Test Cases in the following ways:

- **Run a Single Test Case**:
  You can run a single Test Case by right-clicking the Test Case in the **Navigator** and selecting the **Run Test Case(s)** menu option.
- **Run all Test Cases in a Test Suite**:
  You can run all Test Cases belonging to a Test Suite in batch mode by right-clicking the Suite in the **Navigator**, and selecting the **Run Test Case(s)** menu. option.
- **Run all Test Cases in the Workspace**:
  If you wish to run all Test Suites (and their Test Cases), you can right-click the top-level **Workspace** node in the Navigator, and select the **Run Test Case(s)** menu option.

Viewing the Results

When any Test Cases are run (individually or in batch mode), the results are displayed in the **Results Mode** of the API Gateway Explorer interface. You can review results for individual Test Cases by clicking the **Tests** tab at the bottom of the **Results** tab.

The results are listed according to the name of their Test Suite. The Test Suite node can be expanded to display the results of each of the Test Cases. You can click the Test Case node to display the response from the Web service for the corresponding SOAP operation.

Note

You can format the SOAP response into a more user-friendly format by selecting the **Auto Format XML Response** box in the **Auto Format Response** section of the global **Preferences** dialog. This is available from the **Window** > **Preferences** top-level menu.

When you expand each of the Test Cases you can see that a number of steps were performed. For a default auto-generated Test Suite, these steps are called **Rewrite URL**, **Static Router**, and **Connection**. These steps are called *filters*, and can be configured for each Test Case by clicking the Test Case node in the **Navigator** view.

You can export all results currently on display in the table in the **Results** tab by selecting the **Export Results** option in the drop-down menu beside the triangular green Run button in the toolbar. When this option is selected, the current results set is written out to an XML file. You can import this results set again by selecting the **Import Results** option, and browsing to the XML file that you exported the results to.

You can also select **Clear Results** to clear the table completely. Make sure you have saved any results that you may want to reuse before clearing the **Stress Test Results** table.

Configuring Individual Test Cases

To see the filters that comprise an individual Test Case, click the Test Case node in the **Navigator** tree. The *policy* of filters that make up the Test Case is displayed in the **Design Mode** in the main panel of the API Gateway Explorer interface.

By default, any Test Case that was auto-generated from a WSDL file contains instances of the following filters:

- Rewrite URL
- Static Router
- Connection

These filters are automatically populated with the address of the Web service defined in the WSDL file. The auto-generated message from the Test Case Input is passed into this policy, which is responsible for routing the message on to the configured destination.

You can add several other filters to the policy to enable you to create powerful and complex tests. Several categories of filters are located on the right-hand side of the API Gateway Explorer interface.

To add a filter to a policy, drag and drop the filter from the palette on to the policy. You can drag a filter on to a green arrow on a previously configured policy to insert the filter at that point in the policy.

Filters can be joined together using the **Success Path** (green) and **Failure Path** (red) arrows, which are located just above the filter palette. If a filter fails (for example, schema validation fails), the failure path is followed. However, if a filter runs successfully (for example, schema validates successfully), the success path is followed.

If you want to validate the response message from the Web service, you must place the validating filters after the **Connection** filter in the policy. This filter is responsible for making the connection out to the Web service, sending the request, and receiving the response. Therefore, any filters that are placed after it in a policy are effectively acting on the response message.

You can mark a filter as the starting point of the policy by right-clicking the filter in the policy, and selecting the **Set as Start** menu option.

For more information on the filters available in API Gateway Explorer, see the **Message Filters** section in the product documentation. You can find a links to the main categories of filters from the main index page of the documentation.

Running Attack Vectors

Overview

The Oracle API Gateway Explorer testing utility enables you to insert common security *attack vectors* into Test Case messages. Attack vectors include SQL injection attacks where a SQL command is inserted as the value of a message parameter in a SOAP message. If the code processing the parameter is carelessly written, it can unknowingly execute the SQL command. For example, this SQL command could return all user data from a Users database, drop a table from a database, or delete an entire database altogether. Because of this, the consequences of not checking for SQL syntax (and other attack vectors) on the server-side can be very serious.

As discussed in the *Generating and Running Test Cases* tutorial, API Gateway Explorer can auto-generate a Test Case for each SOAP operation defined in a WSDL file. A sample input message is associated with each Test Case. You can then create an *Attack Vector Message* from the sample by inserting a pre-configured attack vector into the message.

This topic describes the various stages involved in creating and running Attack Vectors.

Configuring an Attack Vector

You can configure Attack Vectors in the **Design Mode** of the API Gateway Explorer interface. When a Test Case Input has been loaded (usually by loading a WSDL), you can insert Attack Vectors into the message by clicking the **Security Vectors** tab on the bottom of the screen.

All pre-configured attack vectors are listed by name in the table on the left of the screen. You can add a new attack vector by clicking the **Edit List** link at the bottom. You can use the **Security Attack Vectors** dialog to add a new Attack Vector to the library.

Click the **Add Attack Vector** button in the toolbar, and enter a name for the new vector in the field provided. On clicking the **OK** button, the new Attack Vector name is displayed in the table. You can then enter the content of the Vector in the text area. You can edit any of the existing Attack Vectors by clicking the Vector's name in the table, and entering the new content in the text area. Similarly, you can remove an existing attack vector by clicking it in the table, and clicking the **Remove Attack** button in the toolbar.

Click the **Close** button when you have finished adding your attack vectors. In the next section, you can find out how to insert these attack vectors into sample messages.

Inserting Attack Vectors into Sample Messages

Complete the following steps to insert Attack Vectors into sample SOAP messages and send them to a Web service. This feature is only available in the **Design Mode** of API Gateway Explorer, so you must select the **Design** on the right in the toolbar.

Step 1: Load the WSDL File
You must first auto-generate your Test Cases using a WSDL file. To do this, select the **Import WSDL** button at the top of the API Gateway Explorer interface. The WSDL can be loaded from a file, URL, or a UDDI registry. For more details, see topic on *Retrieve WSDL files from a UDDI registry*. Click the **Next** button when you have specified the WSDL.

Step 2: Select the Operations
On the **WSDL Operations** screen, select the SOAP operations that you wish to generate Test Cases for by selecting the boxes next to the required operations. Click the **Finish** button when you have selected the operations. You can see your newly generated Test Suite and its Test Cases in the Navigator panel.

Note

You can only generate Test Cases for those operations that have a SOAP binding.

Step 3: Open the Security Vectors Screen
Select the Test Case Input (node) in the **Test Navigator**, and click the **Security Vectors** tab at the bottom of the **Design Mode** screen. You can see the list of Security Vectors in the table on the left and the Test Case Input message in the texta area on the right. You can add any new Attack Vector at this time by clicking the **Edit List** button as described in the previous section.

Step 4: Insert the Attack Vector into the Message
To insert an Attack Vector into the message, select the Attack Vector in the table on the left of the message. Then select the node in the SOAP request where you want to insert the Attack Vector in the source view on the right of the screen. Finally, to insert the Attack Vector, click the **Insert** button in the middle of the screen. The selected Attack Vector is inserted as the content of the selected node in the message.

Step 6: Repeat for Multiple SOAP Operations
If you have generated Test Case Inputs for multiple SOAP operations from the WSDL, you can repeat the steps above to insert Attack Vectors into each of the generated requests.

Step 7: Run the Attack Vector Messages
You can run attack vectors at the Workspace, Test Suite, or Test Case level. By right-clicking the Test Case node in the Navigator tree, and selecting the **Run** menu option, you can just run the Attack Vector Messages associated with that Test Case. Similarly, you can run all Attack Vector Messages associated with a Test Suite by selecting the Test Suite node, and clicking the **Run** button at the top of the Navigator. (The right-click **Run** option is also available here.) Finally, you can run all Attack Vector Messages for the Workspace by selecting the **Workspace** node in the tree and clicking the **Run** button. The green Run button at the top of the API Gateway Explorer interface will run all tests in the **Workspace**.

Viewing the Results

When any Attack Vector Messages are run (individually or in batch mode), the results are displayed in the **Results Mode** of the API Gateway Explorer interface. By default, when tests are run you are automatically presented with the results in the **Results Mode** screen.

The results are listed according to the name of their Test Suite. You can expand the Test Suite node to display the results of each of the Test Cases. You can then click the Test Case node to display the response from the Web service for the corresponding SOAP operation.

Note

You can format the SOAP response into a more user-friendly format by selecting the **Auto Format XML Response** box in the **Auto Format Response** section of the global **Preferences** dialog. The global **Preferences** dialog is available from the **Window > Preferences** top-level menu.

When you expand each of the Test Cases you can see that a number of steps are performed. For a default auto-generated Test Suite there are just a single **Connect to URL** step. Each of these steps corresponds to a *message filter*, which can be configured by double clicking it in the **Test Navigator** tree.

The **Clear Results** option is available by right-clicking on any node in the Test Case Results tree and can be used to clear the Results table completely.

Testing WSDL Files for WS-I Compliance

Overview

Before loading a WSDL file to create sample SOAP messages or Test Cases, you can test the WSDL file for compliance with the WS-Interoperability (WS-I) Basic Profile (BP). The Basic Profile consists of a set of assertions or guidelines on how to ensure maximum interoperability between different implementations of Web services. For example, there are recommendations on the SOAP style to use (`document/literal`), how schema information is included in WSDL files, and how message parts are defined to avoid ambiguity for consumers of WSDL files.

API Gateway Explorer can test WSDL files for conformance against the recommendations described in the Basic Profile. A report is generated showing which recommendations have passed and which have failed. While you can still load a WSDL file that does not comply with the Basic Profile, there is no certainty that consumers of the Web Service can use it without encountering problems.

Important

Before you run the WS-I compliance test, you must ensure that the Java version of the Interoperability Testing Tools is installed on the machine on which the API Gateway Explorer is running. You can download these tools from www.ws-i.org [http://www.ws-i.org].

To configure the location of the WS-I testing tools, select **Window** > **Preferences** from the API Gateway Explorer main menu. In the **Preferences** dialog, select the **WS-I Settings**, and browse to the location of the WS-I testing tools. You must specify the *full path* to these tools. For example:

```
C:\Program Files\WSI_Test_Java_Final_1.1\wsi-test-tools
```

For more details on configuring WS-I settings, see the *General Preferences* topic.

Running the WS-I Compliance Test

To run the WS-I compliance test on a WSDL file, perform the following steps:

1. Select **Tools** > **Run WS-I Compliance Test** from the API Gateway Explorer main menu.
2. In the **Run WS-I Compliance Test** dialog, browse to the **WSDL File** or specify the **WSDL URL**.
3. Click **OK**. The WS-I Analysis tools run in the background in API Gateway Explorer.

The results of the compliance test are displayed in your browser in a **WS-I Profile Conformance Report**. The overall result of the compliance test is displayed in the **Summary** section. The results of the WS-I compliance tests are grouped by type in the **Artifact: description** section. For example, you can access details for a specific port type, operation, or message by clicking the appropriate link in the **Entry List** table. Each **Entry** displays the results for the relevant WS-I Test Assertions.

Manage certificates and keys

Overview

For API Gateway Explorer to trust X.509 certificates issued by a specific Certificate Authority (CA), you must import that CA's certificate into the API Gateway Explorer's trusted certificate store. For example, if API Gateway Explorer is to trust secure communications (SSL connections or XML Signature) from an external SAML Policy Decision Point (PDP), you must import the PDP's certificate, or the issuing CA's certificate into the API Gateway Explorer certificate store.

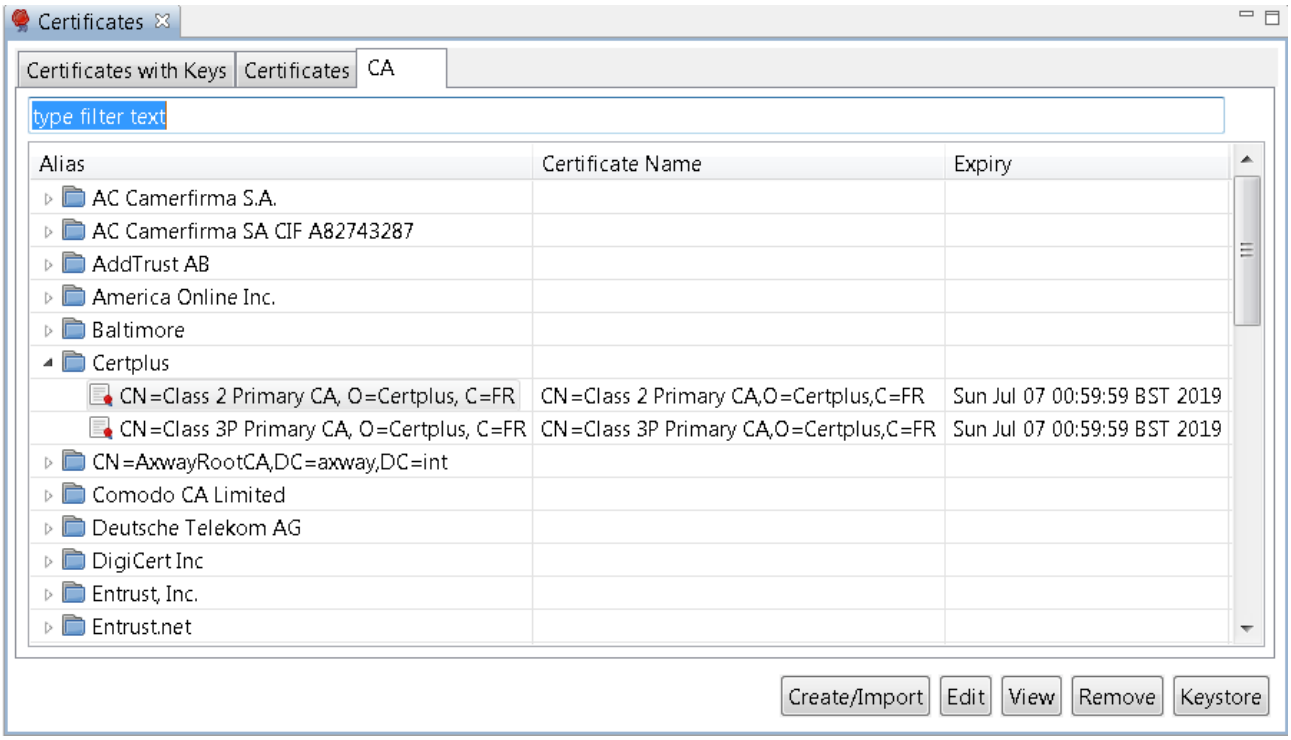
In addition to importing CA certificates, you can import and create server certificates and private keys in the certificate store. These can be stored locally or on an external Hardware Security Module (HSM). You can also import and create public-private key pairs. For example, these can be used with the Secure Shell (SSH) File Transfer Protocol (SFTP) or with Pretty Good Privacy (PGP).

View certificates and keys

To view the certificates and keys stored in the certificate store, select **Certificates and Keys** > **Certificates** in the tree. Certificates and keys are listed on the following tabs in the **Certificates** window:

- **Certificates with Keys**: Server certificates with associated private keys
- **Certificates**: Server certificates without any associated private keys
- **CA**: Certificate Authority certificates with associated public keys

You can search for a specific certificate or key by entering a search string in the text box at the top of each tab, which automatically filters the tree.



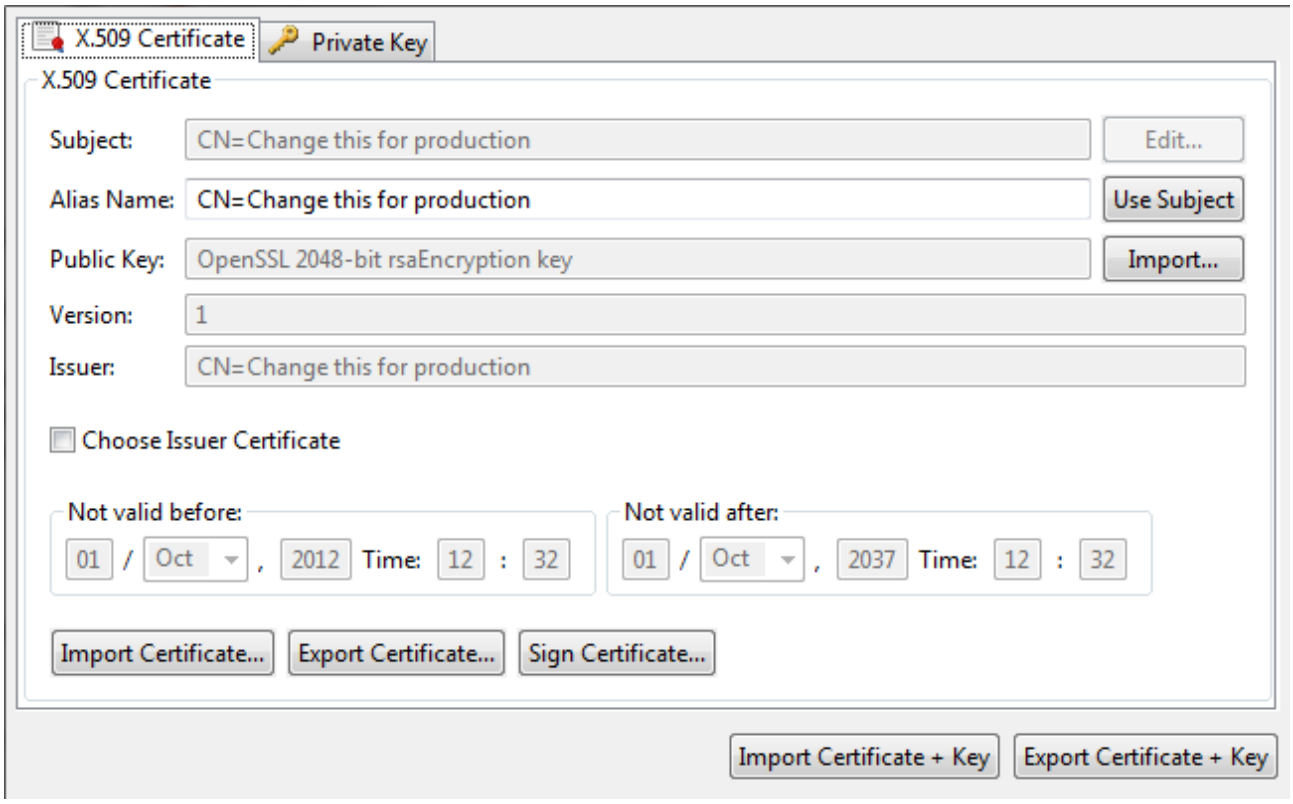
Certificate management options

The following options are available at the bottom right of the window:

- **Create/Import**: Click to create or import a new certificate and private key. For details, see the section called "Configure an X.509 certificate".
- **Edit**: Select a certificate, and click to edit its existing settings.
- **View**: Select a certificate, and click to view more detailed information.
- **Remove**: Select a certificate, and click to remove the certificate from the certificate store.
- **Keystore**: Click this to export or import certificates to or from a Java keystore. For details, see the section called "Manage certificates in Java keystores".

Configure an X.509 certificate

To create a certificate and private key, click **Create/Import**. The **Configure Certificate and Private Key** dialog is displayed. This section explains how to use the **X.509 Certificate** tab on this dialog.



Create a certificate

Configure the following settings to create a certificate:

- **Subject**:
  Click **Edit** to configure the *Distinguished Name* (DName) of the subject.
- **Alias Name**:
  This mandatory field enables you to specify a friendly name (or alias) for the certificate. Alternatively, you can click

Use Subject to add the DName of the certificate in the text box instead of a certificate alias.

- **Public Key**:
  Click **Import** to import the subject's public key (usually from a PEM or DER-encoded file).
- **Version**:
  This read-only field displays the X.509 version of the certificate.
- **Issuer**:
  This read-only field displays the distinguished name of the CA that issued the certificate.
- **Choose Issuer Certificate**:
  Select to explicitly specify an issuer certificate for this certificate (for example, to avoid a potential clash or expiry issue with another certificate using the same intermediary certificate). You can then click the browse button on the right to select an issuer certificate. This setting is not selected by default.
- **Not valid before**:
  Select a date to define the start of the validity period of the certificate.
- **Not valid after**:
  Select a date to define the end of the validity period of the certificate.
- **Sign Certificate**:
  You must click this button to sign the certificate. The certificate can be self-signed, or signed by the private key belonging to a trusted CA whose key pair is stored in the certificate store.

Import certificates

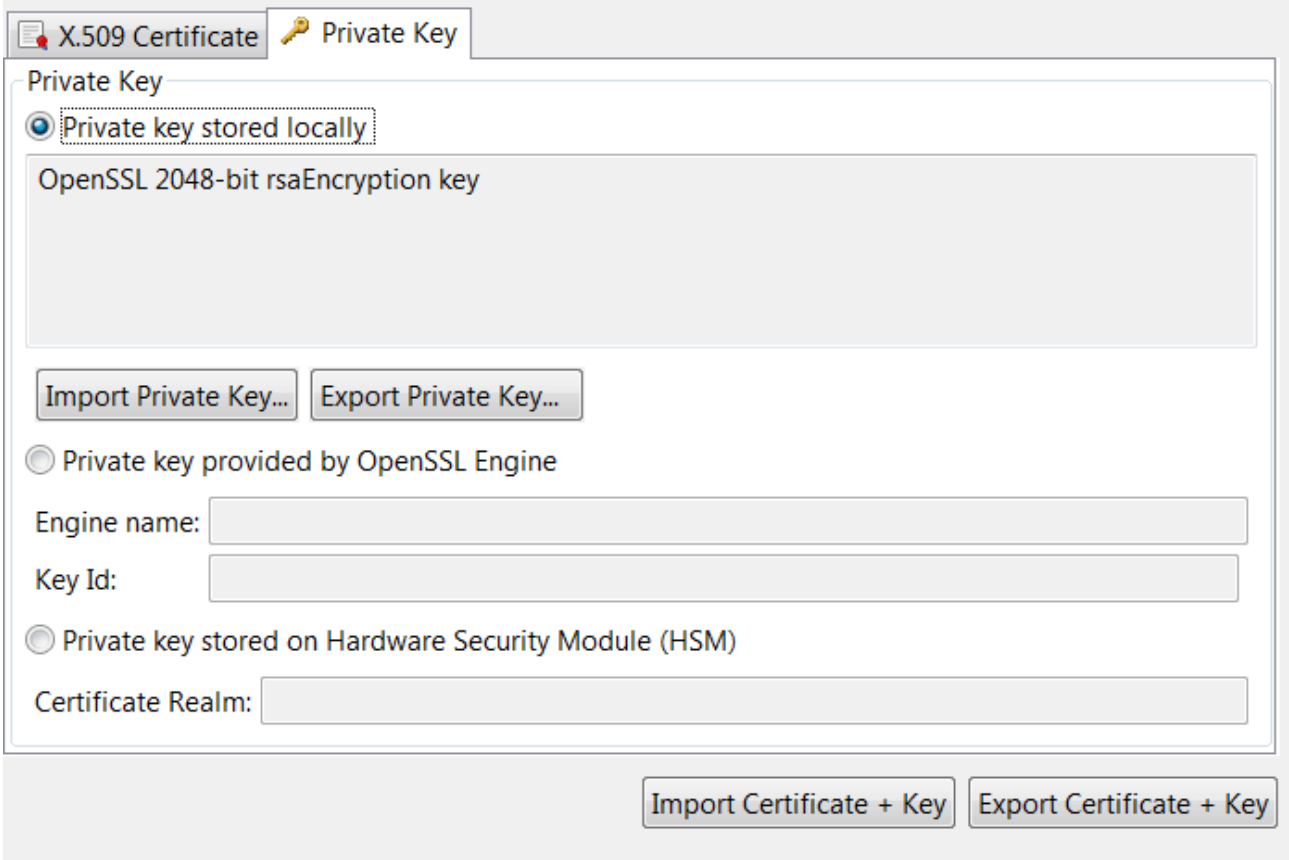
You can use the following buttons to import or export certificates into the certificate store:

- **Import Certificate**:
  Click to import a certificate (for example, from a `.pem` or `.der` file).
- **Export Certificate**:
  Click to export the certificate (for example, to a `.pem` or `.der` file).

Configure a private key

Use the **Private Key** tab to configure details of the private key. By default, private keys are stored locally (for example, in the API Gateway Explorer certificate store). They can also be provided by an OpenSSL engine, or stored on a Hardware Security Module (HSM) if required.

API Gateway Explorer supports PKCS#11-compatible HSM devices. For example, this includes Thales nShield Solo , SafeNet Luna SA, and so on.



Private key stored locally

If the private key is stored in the API Gateway Explorer certificate store, , select **Private key stored locally**. The following options are available for keys stored locally:

- **Private key stored locally**:
  This read-only field displays details of the private key.
- **Import Private Key**:
  Click to import the subject's private key (usually from a PEM or DER-encoded file).
- **Export Private Key**:
  Click to export the subject's private key to a PEM or DER-encoded file.

Private key provided by OpenSSL engine

If the private key that corresponds to the public key in the certificate is provided by an OpenSSL engine, select **Private key provided by OpenSSL Engine**.

Configure the following fields to associate a key provided by the OpenSSL engine with the current certificate:

- **Engine name**:
  Enter the name of the OpenSSL engine to use to interface to an HSM. All vendor implementations of the OpenSSL Engine API are identified by a unique name. See your vendor's OpenSSL engine implementation or HSM documentation to find out the name of the engine.

- **Key Id**:
  Enter the key ID used to uniquely identify a specific private key from all others stored on an HSM. When you complete this dialog, the private key is associated with the certificate that you are currently editing. Private keys are identified by their key ID by default.

Private key stored on external HSM

If the private key that corresponds to the public key stored in the certificate resides on an external HSM, select **Private key stored on Hardware Security Module (HSM)**, and enter the name of the **Certificate Realm**.

Note

To use the API Gateway's PKCS#11 engine to access objects in an external HSM, the corresponding HSM provider and certificate realms must also be configured. For more details, see the section called “Configure HSMs and certificate realms”.

Configure HSMs and certificate realms

Certificate realms are abstractions of private keys and public key certificates, which mean that policy developers do not need to enter HSM-specific configuration such as slots and key labels. Instead, if a private key exists on an HSM, the developer can configure the certificate to show that its private key uses a specific certificate realm, which is simply an alias for a private key (for example, `JMS Keys`).

For example, on the host machine, an administrator could configure the `JMS Keys` certificate realm, and create a *keystore* for the realm, which requires specific knowledge about the HSM (for example, PIN, slot, and private key label). The certificate realm is the alias name, while the keystore is the actual private keystore for the realm.

Manage HSMs with keystoreadmin

The `keystoreadmin` script enables you to perform the following tasks:

- Register an HSM provider
- List registered HSM providers
- Create a certificate realm
- List certificate realms

For example, if a policy developer is using JMS, and wants to indicate that private keys exist on an HSM, they could indicate that the certificate is using the `JMS Keys` certificate realm. On each instance using the configuration, it is the responsibility of the administrator to create the `JMS Keys` certificate realm.

For more details, enter `keystoreadmin` in the following directory, and perform the instructions at the command prompt:

Windows	`INSTALL_DIR\apigateway\Win32\bin`
UNIX/Linux	`INSTALL_DIR/apigateway/posix/bin`

Use keystoreadmin in interactive mode

When you enter `keystoreadmin` without arguments, this displays an interactive menu with the following options:

Option	Description	When to use
`1`	`Change group or instance`	When registering HSMs or configuring certificate realms,

Option	Description	When to use
		you must choose the local group and instance to configure.
2	`List registered HSM providers`	Display the HSMs that are currently registered.
3	`Register an HSM provider`	Before creating certificate realms, you must first register the HSM. This option guides you through the steps. The HSM must be installed, configured, and active, and you must know the full path to the HSM device driver (PKCS#11). You give the HSM an alias (for example, LunaSA), which you use later when registering certificate realms.
4	`List Certificate Realms`	List configured certificate realms and associated keystores.
5	`Create a Certificate Realm`	Create a keystore and assign it to a certificate realm.

Step 1—Register an HSM provider

You must first register an HSM provider as follows:

1. Open a command prompt in the API Gateway Explorer `bin` directory (for example, `apigateway\Win32\bin`).
2. Enter the `keystoreadmin` command.
3. Select option `3) Register an HSM provider`.
4. If prompted, select the appropriate API Gateway Explorer group or instance.
5. You are prompted for a provider alias name. The alias is local only. For example, if registering a LunaSA HSM, you might enter the `LunaSA` alias.
6. For convenience, `keystoreadmin` searches for supported HSM drivers. If found, it shows the list of supported drivers. If none are found, this does not mean the driver does not exist. You must see your HSM documentation for the location of the drivers. For example:

```
Choose from one of the following:

   1) C:\LunaSA\cryptoki.dll

   o) Other
   q) Quit
```

7. If successful, `keystoreadmin` loads the driver and displays its details. For example:

```
Registering HSM provider...
 Initializing HSM...
 Crypto Version: 2.20
 Manufacter Id: SafeNet, Inc.
 Library Description: Chrystoki
 Library Version: 5.1
 Device registered.
```

Step 2—Create a certificate realm and associated keystore

To create a certificate realm and associated keystore, perform the following steps:

1. Open a command prompt in the API Gateway Explorer `bin` directory (for example, `apigateway\Win32\bin`).

2. Enter the `keystoreadmin` command.
3. Select option `5) Create a Certificate Realm`.
4. You are prompted to enter a certificate realm name. This certificate realm name will be used in Policy Studio when configuring the private key of the corresponding X.509 certificate. The realm name is case sensitive (for example, `JMS Keys`).
5. The registered HSMs are listed. For example, select option `1) HSM`.
6. The command connects to the selected HSM, and a list of available slots is displayed. Select the slot containing the private key to use for the certificate realm (for example, select slot `1`).
7. You are prompted to input the PIN passphrase for the slot. The passphrase will not echo any output.
8. When you enter the correct PIN passphrase for the slot, this displays a list of private keys. Choose the key to use for the certificate realm. For example:

```
Choose from one of the following:

   1) server1_priv
   2) jms_priv
   q) Quit

Select option: 2
```

9. You are prompted for a file name for the keystore. For example:

```
Certificate realm filename [jms keys.ks]:
Successfully created the certificate realm: JMS Keys
Press any key to continue...
```

10. The keystore is output to the API Gateway Explorer instance directory. For example:

```
apigateway/groups/group-2/instance-1/conf/certrealms/jms keys.ks
```

Note

Each API Gateway instance must have its certificate realm configured. When finished creating certificate realms, you must restart the API Gateway instance for the changes to take effect.

Step 3—Start the API Gateway Explorer when using an HSM

When the API Gateway is configured to use certificate realms, these realms are initialized on startup, and a connection to the corresponding HSM is established. This requires the PIN passphrase for the specific HSM slots. At startup, you can manually enter the required HSM slot PIN passphrase, or you can automate this instead.

Start API Gateway with manually entered PIN passphrase

When the API Gateway is configured to use an HSM, the API Gateway Explorer stops all processing, prompts for the HSM slot PIN passphrase, and waits indefinitely for input. For example:

```
INFO    07/Jan/2015:16:31:54 Initializing certificate realm 'JMS Keys'...
Enter passphrase for Certificate Realm, "JMS Keys":
```

The API Gateway does not reprompt if the PIN passphrase is incorrect. It logs the error and continues, while any services that use the certificate realm cannot use the HSM.

Start API Gateway with automatic PIN passphrase

You can configure the API Gateway to start and initialize the HSM by invoking a command script on the operating system to obtain the HSM slot PIN passphrase. This enables the API Gateway for automatic startup without manually entering

the PIN passphrase.

To configure an automatic PIN passphrase, perform the following steps:

1. Edit the API Gateway instance's `vpkcs11.xml` configuration file. For example:

```
apigateway/groups/group-2/instance-1/conf/vpkcs11.xml
```

2. Add a `PASSPHRASE_EXEC` command that contains the full path to the script that executes and obtains the passphrase. The script should write the passphrase to stdout, and should have the necessary operating system file and execute protection settings to prevent unauthorized access to the PIN passphrase. The following example shows a `vpkcs11.xml` file that invokes the `hsmpin.sh` to echo the passphrase:

```
<?xml version="1.0" encoding="utf-8"?>
<ConfigurationFragment provider="cryptov">
    <Engine name="vpkcs11" defaultFor="">
        <EngineCommand when="preInit" name="REALMS_DIR"
            value="$VINSTDIR/conf/certrealms" />
        <EngineCommand when="preInit" name="PASSPHRASE_EXEC"
            value=""$VDISTDIR/hsmpin.sh"" />
    </Engine>
</ConfigurationFragment>
```

3. The API Gateway provides the certificate realm as an argument to the script, so you can use the same script to initialize multiple realms. The following examples show scripts that write a PIN of `1234` to stdout when initializing the `JMS Keys` certificate realm:

   Example hsmpin.bat file on Windows

```
@echo off
IF [%1]==[] GOTO _END

:: Strip out the double quotes around arg
SET REALM=%1
SET REALM=%REALM:"=%

IF "%REALM%"=="JMS Keys" ECHO 1234
```

   Example hsmpin.sh file on Linux/UNIX

```
#!/bin/sh
case $1 in
"JMS Keys")
    echo 1234
    ;;
esac
```

Configure SSH key pairs

To configure public-private key pairs in the certificate store, select **Certificates and Keys** > **Key Pairs**. The **Key Pairs** window enables you to add, edit, or delete OpenSSH public-private key pairs, which are required for the Secure Shell (SSH) File Transfer Protocol (SFTP).

Add a key pair

To add a public-private key pair, click **Add** on the right, and configure the following settings in the dialog:

- **Alias**:

Enter a unique name for the key pair.

- **Algorithm**:
  Enter the algorithm used to generate the key pair. Defaults to RSA.
- **Load**:
  Click to select the public key or private key files to use. The **Fingerprint** field is auto-populated when you load a public key.

Note

The keys must be OpenSSH keys. RSA keys are supported, but DSA keys are not supported. The keys must not be passphrase protected.

Manage OpenSSH keys

You can use the `ssh-keygen` command provided on UNIX to manage OpenSSH keys. For example:

- The following command creates an OpenSSH key:
  `ssh-keygen -t rsa`
- The following command converts an `ssh.com` key to an OpenSSH key:
  `ssh-keygen -i -f ssh.com.key > open.ssh.key`
- The following command removes a passphrase (enter the old passphrase, and enter nothing for the new passphrase):
  `ssh-keygen -p`
- The following command outputs the key fingerprint:
  `ssh-keygen -lf ssh_host_rsa_key.pub`

Edit a key pair
To edit a public-private key pair, select a key pair alias in the table, and click **Edit** on the right. For example, you can load a different public key and private key. Alternatively, double-click a key pair alias in the table to edit it.

Delete key pairs
You can delete a selected key pair from the certificate store by clicking **Remove** on the right. Alternatively, click **Remove All**.

Configure PGP key pairs

To configure Pretty Good Privacy (PGP) key pairs in the certificate store, select **Certificates and Keys** > **PGP Key Pairs**. The **PGP Key Pairs** window enables you to add, edit, or delete PGP public-private key pairs.

Add a PGP key pair

To add a PGP public-private key pair, click the **Add** on the right, and configure the following settings in the dialog:

- **Alias**:
  Enter a unique name for the PGP key pair.
- **Load**:
  Click **Load** to select the public key and private key files to use.

Note

The PGP keys added must not be passphrase protected.

Manage PGP keys

You can use the freely available GNU Privacy Guard (GnuPG) tool to manage PGP key files (available from http://www.gnupg.org/). For example:

- The following command creates a PGP key:
  `gpg --gen-key`
  For more details, see http://www.seas.upenn.edu/cets/answers/pgp_keys.html [http://www.seas.upenn.edu/cets/answers/pgp_keys.html]
- The following command enables you to view the PGP key:
  `gpg -a --export`
- The following command exports a public key to a file:
  `gpg --export -u 'UserName' -a -o public.key`
- The following command exports a private key to a file:
  `gpg --export-secret-keys -u 'UserName' -a -o private.key`
- The following command lists the private keys:
  `gpg --list-secret-keys`

Edit a PGP key pair
To edit a PGP key pair, select a key pair alias in the table, and click **Edit** on the right. For example, you can load a different public key and private key. Alternatively, double-click a key pair alias in the table to edit it.

Delete PGP key pairs
You can delete a selected PGP key pair from the certificate store by clicking **Remove** on the right. Alternatively, click **Remove All**.

Global import and export options

This section desribes global import and export options available when managing certificates and keys.

Import and export certificates and keys

The following global configuration options apply to both the **X.509 Certificate** and **Private Key** tabs:

- **Import Certificate + Key**:
  Use this option to import a certificate and a key (for example, from a `.p12` file).
- **Export Certificate + Key**:
  Use this option to export a certificate and a key (for example, to a `.p12` file).

Click **OK** when you have finished configuring the certificate and private key.

Manage certificates in Java keystores

You can also export a certificate to a Java keystore. You can do this by clicking **Keystore** on the main **Certificates** window. Click the browse button at beside the **Keystore** field at the top right to open an existing keystore, or click **New Keystore** to create a new keystore. Choose the name and location of the keystore file, and enter a passphrase for this keystore when prompted. Click **Export to Keystore**, and select a certificate to export.

Similarly, you can import certificates and keys from a Java keystore into the certificate store. To do this, click **Keystore** on the main **Certificates** window. On the **Keystore** window, browse to the location of the keystore by clicking the browse button beside the **Keystore** field. The certificates/keys in the keystore are listed in the table. To import any of these keys to the certificate store, select the box next to the certificate or key to import, and click **Import to Trusted certificate store**. If the key is protected by a password, you are prompted for this password.

You can also use the **Keystore** window to view and remove existing entries in the keystore. You can also add keys to

the keystore and to create a new keystore. Use the appropriate button to perform any of these tasks.

Further information

For more details on supported security features, see the *API Gateway Explorer Security Guide*.

Configuring Connection Settings

Overview

The **Connection Settings** dialog is available by selecting the **Settings > Connection Settings** option from the menu to the left of the **SOAP Request** panel. The fields on this dialog can be used to configure the following aspects of the request:

- Proxy settings for the target Web service.
- CA and server certificates that are considered trusted for SSL purposes.
- Client SSL certificate to use to authenticate to the target Web service.
- A username and password to use to authenticate to the Web service using HTTP basic or digest authentication.

The following sections describe each of the tabs that are available on the **Connection Settings** dialog.

URL

Enter the complete URL of the target Web service in the **URL** field.

Proxy Settings

The fields on this tab should be configured if API Gateway Explorer must connect to the target Web service through an HTTP proxy. In this case, API Gateway Explorer will include the full URL of the destination Web service in the request line of the HTTP request.

For example, if the destination Web service is running at `http://localhost:8080/services`, the request line would appear as follows if API Gateway Explorer is routing through a proxy:

```
POST http://localhost:8080/services HTTP/1.1
```

If API Gateway Explorer was not routing through a proxy, the request line would appear as follows:

```
POST /services HTTP/1.1
```

To configure API Gateway Explorer to send SOAP messages through a HTTP proxy, complete the following fields on the **Proxy Settings** tab:

Proxy Host:
Enter the hostname or IP address of the HTTP proxy.

Proxy Port:
Specify the port on which the proxy server is listening.

SSL Port:
If the proxy server has SSL enabled and you want to connect to the SSL port, enter the SSL port here.

Trusted Certificates

When API Gateway Explorer connects to a server over SSL it must decide whether or not to trust the server's SSL certificate. It is possible to select a list of CA or server certificates from the **Trusted Certificates** tab that will be considered trusted by API Gateway Explorer when connecting to the server specified in the **URL** field of this dialog.

The table displayed on the **Trusted Certificates** tab lists all certificates that have been imported into API Gateway Ex-

plorer's Certificate Store. To *trust* a certificate for this particular connection, simply check the box next to the certificate in the table.

Client SSL Authentication

In cases where the destination server requires clients to authenticate to it using an SSL certificate, you must select a client certificate on the **Client SSL Authentication** tab. Simply check the box next to the client certificate that you want to use to authenticate to the server specified in the **URL** field.

HTTP Authentication

If the destination server requires clients to authenticate to it using HTTP basic or digest authentication, the fields on this tab can be used.

None, HTTP Basic, or HTTP Digest:
Select the method that you want to use to authenticate to the server.

User Name:
Specify the user name you want to use to authenticate to the server.

Password:
Enter the password for this user.

Stress test with send request (sr)

Overview

Oracle API Gateway Explorer ships with the send request stress testing tool (`sr` command), which is available in the API Gateway Explorer root installation directory.

Important

On Linux, the `LD_LIBRARY_PATH` environment variable must be set to the directory from which you are running the sr tool. On Linux and Solaris, you must use the `vrun sr` command. For example:

```
vrun sr http://testhost:8080/stockquote
```

Basic sr examples

The following are some basic examples of using the sr command:

HTTP GET:

```
sr http://testhost:8080/stockquote
```

POST file contents (content-type inferred from file extension):

```
sr -f StockQuoteRequest.xml http://testhost:8080/stockquote
```

Send XML file with SOAP Action 10 times:

```
sr -c 10 -f StockQuoteRequest.xml http://testhost:8080/stockquote
```

Send XML file with SOAP Action 10 times in 3 parallel clients:

```
sr -c 10 -p 3 -f StockQuoteRequest.xml http://testhost:8080/stockquote
```

Send the same request quietly:

```
sr -c 10 -p 3 -qq -f StockQuoteRequest.xml http://testhost:8080/stockquote
```

Run test for 10 seconds:

```
sr -d 10 -qq -f StockQuoteRequest.xml http://testhost:8080/stockquote
```

POST file contents with SOAP Action:

```
sr -f StockQuoteRequest.xml -A SOAPAction:getPrice http://testhost:8080/stockquote
```

Advanced sr examples

The following are some advanced examples of using the sr command:

Send form.xml to http://192.168.0.49:8080/healthcheck split at 171 character size, and trickle 200 millisecond delay between each send with a 200 Content-Length header:

```
sr -h 192.168.0.49 -s 8080 -u /healthcheck -b 171 -t 200 -f form.xml
-a "Content-Type:application/x-www-form-urlenprogramlistingd" -a "Content-Length:200"
```

Send a multipart message to http://192.168.0.19:8080/test, 2 XML docs are attached to message:

```
sr -h 192.168.0.49 -s 8080 -u /test -{ -a Content-Type:text/xml -f soap.txt
-a Content-Type:text/xml -f attachment.xml -a Content-Type:text/xml -} -A c-timestamp:1234
```

Send only headers using a GET over one-way SSL running 10 parallel threads for 86400 seconds (1 day) using super quiet mode:

```
sr -h 192.168.0.54 -C -s 8443 -u /nextgen -f test_req.xml -a givenName:SHViZXJ0
-a sn:RmFuc3dvcnRo -v GET -p10 -d86400 -qq
```

Send query string over mutual SSL presenting client certificate and key doing a GET running 10 parallel threads for 86400 seconds (1 day) using super quiet mode:

```
sr -h 192.168.0.54 -C -s 8443 -X client.pem -K client.key
-u "https://localhost:8443/idp?TargetResource=http://oracle.test.com" -f test_req.xml
-v GET -p10 -d86400 -qq
```

Send zip file in users home directory to testhost on port 8080 with /zip URI, save the resulting response content into the result.zip file, and do this silently:

```
sr -f ~/test.zip -h testhost -s 8080 -u /zip -a Content-Type:application/zip
-J result.zip -qq
```

sr arguments

The main arguments to the sr command include the following:

Argument	Description
`--help`	List all arguments
`-a attribute:value`	Set the HTTP request header (for example, `-a Content-Type:text/xml`)
`-c [request-count]`	Number of requests to send per process
`-d [seconds]`	Duration to run test for
`-f [content-filename]`	File to send as the request
`-h [host]`	Name of destination host
`-i [filename]`	Destination of statistics data
`-l [file]`	Destination of diagnostic logging
`-m`	Recycle SSL sessions (use multiple times)
`-n`	Enable nagle algorithm for transmission
`-o [output]`	Output statistics information every [milliseconds] (only with `-d`)
`-p [connections]`	Number of parallel client connections (threads) to simulate
`-q , -qq, -qqq`	Quiet modes (quiet, very quiet, very very quiet)

Argument	Description
`-r`	Do not send HTTP Request line
`-s [service]`	Port or service name of destination (default is `8080`)
`-t [milliseconds]`	Trickle: delay between sending each character
`-u [uri]`	Target URI to place in request
`-v [verb]`	Set the HTTP verb to use in the request (default is `POST`)
`-w [milliseconds]`	Wait for [milliseconds] between each request
`-x [chunksize]`	Chunk-encode output
`-y [cipherlist]`	SSL ciphers to use (see OpenSSL manpage ciphers(1))
`-z`	Randomize chunk sizes up to limit set by `-x`
`-A attribute:value`	Set the HTTP request header (for example, `-a Content-Type:text/xml`) in the outermost attachment
`-B`	Buckets for response-time samples
`-C`	enCrypt (use SSL protocol)
`-I [filename]`	File for Input (received) data (`- = stdout`)
`-K`	RSA Private Key
`-L`	Line-buffer `stdout` and `stderr`
`-M`	Multiplier for response-time samples
`-N`	origiN for response-time samples
`-O [filename]`	File for Output (sent) data (`- = stdout`)
`-S [part-id]`	Start-part for multipart message
`-U [count]`	reUse each connection for `count` requests
`-V [version]`	Sets the HTTP version (`1.0`, `1.1`)
`-X`	X.509 client certificate
`-Y [cipherlist]`	Show expanded form of `[cipherlist]`
`[-{/-}`	Create multipart body (nestable: use `-f` for leaves)

Further information

For a listing of all arguments, enter `sr --help`. For more information, and details on advanced use, see the `srman-page.pdf` file in your `sr` installation directory.

Global Schema Cache

Overview

The **Schema Cache** contains XML Schemas that can then be used globally by **Schema Validation** filters. XML Schemas can be imported from either XML Schema files or WSDL files. WSDL files often contain XML Schemas that define the elements that in SOAP messages. To facilitate this, API Gateway Explorer can import WSDL files from the file system, from a URL, or from a UDDI registry.

When the XML Schema has been imported into the cache, and selected in a **Schema Validation** filter, API Gateway Explorer retrieves the schema from the cache instead of fetching it from its original location. This improves the runtime performance of the filter, but also ensures that an administrator has complete control over the Schemas that are used to validate messages.

From the main menu, select **File** > **View Schema Cache**. The list of schemas in the cache is listed in the **Imported Schemas** table. You can also view the contents of any of these schemas by clicking the schema in the table. The schema contents are displayed in the **Contents** text area.

At any stage, you can manually modify the contents of the schema in the text area. To save the modified contents to the cache, click the **Update** button.

Adding Schemas to the Cache

To add an XML Schema to the cache, click the **Add** button at the bottom of the **Schema Cache** screen. The **Load Schema** dialog offers two possible locations from which you can load a schema—from an XML Schema file directly, or from a WSDL file.

Select the **From XML Schema** radio button to load the schema directly from a schema file, and then click **Next**. On the next screen, enter or browse to the location of the schema file using the field provided. You can also enter a full URL here to pull the schema from a web location. Click the **Finish** button to import the schema into the cache.

Alternatively, if you want to load the schema file from a WSDL file, select the **From WSDL** radio button on the **Select Schema Source** screen, and then click the **Next** button.

The WSDL file can be located from the file system, from a URL, or from a UDDI registry. Select the appropriate option, and enter or browse to the location of the WSDL file in the fields provided. If you wish to retrieve the WSDL file from a UDDI registry, click the **WSDL from UDDI** radio button, and click the **Browse UDDI** button. The **Browse UDDI Server for WSDL** dialog enables you to connect to a UDDI, and search it for a particular WSDL file. For detail, see the topic on *Retrieve WSDL files from a UDDI registry*.

When importing the schema from the WSDL file, you can also check the WSDL file for compliance with the WS-I Basic Profile. The Basic Profile consists of a set of assertions or guidelines on how to ensure maximum interoperability between different implementations of Web services. For example, there are recommendations on what style of SOAP to use (for example, `document/literal`), how schema information should and should not be included in WSDL files, and how message parts should be defined to avoid ambiguity for consumers of WSDL files.

API Gateway Explorer can test WSDL files while extracting schemas from them for conformance against the recommendations laid out in the Profile. A report is generated showing exactly which recommendations have passed and which have failed. While a WSDL file that does not conform to the Profile can still be imported, there is no certainty that consumers of the Web service can use it without encountering problems.

Important

To run the WS-I compliance test, the WS-I Test Assertions Document (TAD) file must be installed on the machine on which the API Gateway Explorer is running. This file is available from www.ws-i.org [http://www.ws-i.org]. The *full path* to the location of this file (for example, `c:/WSI/SSBP10_BP11_TAD.xml`

must be specified in the API Gateway Explorer global preferences. To configure this setting, select the **Window** > **Preferences** option from the main menu. For more details, see *General Preferences*.

To run the WS-I compliance test on a WSDL file, select **Tools** > **Run WS-I Compliance Test** in the main menu, and specify the WSDL file to import the schema in this screen. Click **OK** to run the WS-I Analysis tools run in the background in API Gateway Explorer.

The results of the compliance test are displayed in your browser in a **WS-I Profile Conformance Report**. The overall result of the compliance test is displayed in the **Summary** section. The results of the WS-I compliance tests are grouped by type in the **Artifact: description** section. For example, you can access details for a specific port type, operation, or message by clicking the appropriate link in the **Entry List** table. Each **Entry** displays the results for the relevant WS-I Test Assertions.

Schema Validation

The **Schema Validation** filter is used to validate XML messages against schemas stored in the cache or in the **Web Services Repository**. The filter is in **Validate Message** category of filters, which is available in the **Design Mode** of the API Gateway Explorer GUI. For more details, see the topic on *Schema validation*.

General Preferences

Overview

The **Preferences** interface enables you to set several global configuration settings to optimize the behavior of the API Gateway Explorer. You can configure the **Preferences** interface by selecting the **Window** > **Preferences** main menu option.

You can view each of the configuration sections described below by clicking the corresponding menu item on the left of the **Preferences** dialog.

Auto Format Response

Select the **Auto Format XML Response** option if you want to pretty-print SOAP response messages when they are rendered in the API Gateway Explorer response panel.

Note

Enabling this setting affects response messages with digital signatures.

JMS

This section enables you to configure custom JMS Service Providers.

Kerberos

Use this section to configure system-wide Kerberos settings. See the *Kerberos configuration* topic for more information on configuring global Kerberos settings.

Proxy Settings

In cases where you have installed API Gateway Explorer on a machine that connects to other machines through a proxy, you can configure details of the proxy on this screen. Complete the following fields:

Host:
Enter the host name or IP address of the HTTP proxy.

Port:
Specify the TCP port of the proxy server.

Username:
If the proxy requires clients to authenticate to it using HTTP authentication, you must enter a valid username here.

Password:
Enter the password for this user.

Runtime Dependencies

The **Runtime Dependencies** setting enables you to add JAR files to the API Gateway Explorer classpath. For example, if you write a custom message filter, you must add its JAR file, and any third-party JAR files that it uses, to the **Runtime Dependencies** list.

Click **Add** to select a JAR file to add to the list of dependencies, and click **Apply** when finished. A copy of the JAR file is added to the `plugins` directory in your Policy Studio installation.

Important

You must restart API Gateway Explorer and the server for these changes to take effect.

SMTP

This section enables you to configure details of SMTP servers that the API Gateway Explorer can connect to and send messages to.

SSL Settings

The **SSL Settings** enable you to specify what action is taken when an unrecognized server certificate is presented to the client. This allows API Gateway Explorer to connect to SSL services without a requirement to add a certificate to its JVM certificate store.

Configure one of the following options:

Prompt User	When you try to connect to SSL services, you are prompted with a dialog. If you choose to trust this particular server certificate displayed in the dialog, it is stored locally, and you are not prompted again.
Trust All	All server certificates are trusted.
Keystore	Enter or browse to the location of the **Keystore** that contains the authentication credentials sent to a remote host for mutual SSL, and enter the appropriate **Keystore Password**.

TCP/IP Monitor

The **TCP/IP Monitor** settings enable you to configure TCP/IP Monitors on local and remote ports. A TCP/IP Monitor is a simple server that monitors all the requests and responses between the API Gateway Explorer client and the server. You can monitor TCP/IP activity using the **TCP/IP Monitor** view in API Gateway Explorer. This view contains a list of requests sent to the server. It displays information about each request when it is forwarded to the server, and each response when it is received from the server.

You can configure the following **TCP/IP Monitor** settings:

Show the TCP/IP Monitor View when there is activity
Specifies whether to display the TCP/IP Monitor view when there is activity through a TCP/IP monitoring server.

TCP/IP Monitors
Displays a list of TCP/IP monitoring servers. To add a TCP/IP monitoring server, click **Add**, and specify the following settings in the **New Monitor** dialog:

Local monitoring port	Specify a unique port number on your local machine. Defaults to `80`.
Host name	Specify the host name or IP address of the machine where the server is running.
Port	Specify the port number of the remote server. Defaults to `80`.

Type	Specify whether the request type from the client is sent by HTTP or TCP/IP. If the default `HTTP` option is selected, requests from the client are modified so that the HTTP header points to the remote machine, and separated if multiple HTTP requests are received in the same connection. If the `TCP/IP` option is selected, requests are sent byte for byte and the TCP/IP Monitor does not translate or modify any requests that it forwards.
Timeout (in milliseconds)	Specify the connection timeout in milliseconds. Defaults to `0`.
Start monitor automatically	Specifies whether the TCP/IP monitoring server starts automatically. This is selected by default.

Use the **Start** and **Stop** buttons to manage the TCP/IP monitoring servers. You can add, edit, remove, start or stop the available TCP/IP monitoring servers in the **TCP/IP Monitors** table. The **Status** column shows if the TCP/IP monitor is started or stopped.

Test Case Colors

Use this page to change the look-and-feel of the **Test Case** screen by modifying the colors used.

Trace Level

You can set the level at which API Gateway Explorer logs diagnostic output by selecting the appropriate level from the **Tracing Level** drop-down list. Diagnostic output is written to a file in the `/trace` directory of your API Gateway Explorer installation. You can also select **Window** > **Show View** > **Console** in the main menu to view the trace output in the **Console** window at the bottom of the screen. The default trace level is `INFO`.

VM Arguments

This page enables you to manually pass arguments to the JVM used by API Gateway Explorer.

Web and XML

Use these screens to alter the way that XML data is displayed in API Gateway Explorer.

Wildcards

This page enables you to set the values of various wildcards that can then be used at runtime by message filters in a **Test Suite**.

WS-I Settings

Before loading a WSDL file that contains the definition of a Web service into the Web Services Repository, you can test the WSDL file for compliance with the Web Service Interoperability (WS-I) Basic Profile. The WS-I Basic Profile contains a number of *Test Assertions* that describe rules for writing WSDL files for maximum interoperability with other WSDL authors, consumers, and other related tools.

The WS-I Settings are described as follows:

WS-I Setting	Description
WS-I Tool Location	Use the **Browse** button to specify the full path to the Java

WS-I Setting	Description
	version of the WS-Interoperability Testing tools (for example, `C:\Program Files\WSI_Test_Java_Final_1.1\wsi-test-tools`). The WS-I testing tools are used to check a WSDL file for WS-I compliance. You can download them from www.ws-i.org [http://www.ws-i.org].
Results Type	Select the type of WS-I test results that you wish to view in the generated report from the drop-down list. You can select from `all`, `onlyFailed`, `notPassed`, or `notInfo`.
Message Entry	Specify whether message entries should be included in the report using the checkbox (selected by default).
Failure Message	Specify whether the failure message defined for each test assertion should be included in the report using the checkbox (selected by default).
Assertion Description	Specify whether the description of each test assertion should be included in the report using the checkbox (unselected by default).
Verbose Output	Specify whether verbose output is displayed in the API Gateway Explorer console window using the checkbox (unselected by default). To view the console window, select **Window** > **Show Console** from the API Gateway Explorer main menu.

For details on running the WS-I Testing Tools, see the *Testing WSDL Files for WS-I Compliance* topic.

Retrieve attribute from HTTP header

Overview

The **Retrieve from HTTP Header** attribute retrieval filter can be used to retrieve the value of an HTTP header and set it to a message attribute. For example, this filter can retrieve an X.509 certificate from a USER_CERT HTTP header, and set it to the `authentication.cert` message attribute. This certificate can then be used by the filter's successors. The following HTTP request shows an example of such a header:

```
POST /services/getEmployee HTTP/1.1
Host: localhost:8095
Content-Length: 21
SOAPAction: HelloService
USER_CERT:  MIIEZDCCA0 ...9aKD1fEQgJ
```

You can also retrieve a value from a named query string parameter and set this to the specified message attribute. The following example shows a request URL that contains a query string:

```
http://hostname.com/services/getEmployee?first=john&last=smith
```

In the above example, the query string is `first=john&last=smith`. As is clear from the example, query strings consist of attribute name-value pairs. Each name-value pair is separated by the & character.

Configuration

The following fields are available on the **Retrieve from HTTP Header** filter configuration screen:

Name:
Enter an appropriate name for this filter.

HTTP Header Name:
Enter the name of the HTTP header contains the value that we want to set to the message attribute.

Base64 Decode:
Check this box if the extracted value should be Base64 decoded before it is set to the message attribute.

Use Query String Parameters:
Select this setting if the API Gateway Explorer should attempt to extract the **HTTP Header Name** from the query string parameters instead of from the HTTP headers.

Attribute ID:
Finally, select the attribute used to store the value extracted from the request.

Insert SAML attribute assertion

Overview

A Security Assertion Markup Language (SAML) attribute assertion contains information about a user in the form of a series of attributes. Having collated a certain amount of information about a user, the API Gateway Explorer can generate a SAML attribute assertion, and insert it into the downstream message.

A *SAML Attribute* (see example below) is generated for each entry in the `attribute.lookup.list` attribute. Other filters from the **Attributes** filter group can be used to insert user attributes into the `attribute.lookup.list` attribute.

It might be useful to refer to the following example of a SAML attribute assertion when configuring this filter:

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
               xmlns:xsi="http://www.w3.org/2000/10/XMLSchema-instance">
 <soap:Header>
  <wsse:Security>
   <saml:Assertion xmlns:saml="urn:oasis:names:tc:SAML:2.0:assertion"
                   ID="Id-0000010a3c4ff12c-0000000000000002"
                   IssueInstant="2006-03-27T15:26:12Z" Version="2.0">
    <saml:Issuer Format="urn:oasis ... WindowsDomainQualifiedName">
      TestCA
    </saml:Issuer>
    <saml:Subject>
     <saml:NameIdentifier Format="urn:oasis ... WindowsDomainQualifiedName">
      TestUser
     </saml:NameIdentifier>
    </saml:Subject>
    <saml:Conditions NotBefore="2005-03-27T15:20:40Z"
                     NotOnOrAfter="2028-03-27T17:20:40Z"/>
    <saml:AttributeStatement>
     <saml:Attribute Name="role" NameFormat="http://www.oracle.com">
      <saml:AttributeValue>admin</saml:AttributeValue>
     </saml:Attribute>
     <saml:Attribute Name="email" NameFormat="http://www.oracle.com">
      <saml:AttributeValue>joe@oracle.com</saml:AttributeValue>
     </saml:Attribute>
     <saml:Attribute Name="dept" NameFormat="">
      <saml:AttributeValue>engineering</saml:AttributeValue>
     </saml:Attribute>
    </saml:AttributeStatement>
   </saml:Assertion>
  </wsse:Security>
 </soap:Header>

 <soap:Body>
  <product>
   <name>API Gateway Explorer</name>
   <company>Oracle</company>
   <description>Web Services Security</description>
  </product>
 </soap:Body>
</soap:Envelope>
```

General settings

Configure the following field:

Name:

Enter an appropriate name for the filter.

Assertion Details

Configure the following fields on the **Assertion Details** tab:

Issuer Name:
Select the certificate containing the Distinguished Name (DName) to be used as the Issuer of the SAML assertion. This DName is included in the SAML assertion as the value of the `Issuer` attribute of the `<saml:Assertion>` element. For an example, see the sample SAML assertion above.

Expire In:
Specify the lifetime of the assertion in this field. The lifetime of the assertion lasts from the time of insertion until the specified amount of time has elapsed.

Drift Time:
The **Drift Time** is used to account for differences in the clock times of the machine hosting the API Gateway Explorer (that generate the assertion) and the machines that consume the assertion. The specified time is subtracted from the time at which the API Gateway Explorer generates the assertion.

SAML Version:
You can create SAML 1.0, 1.1, and 2.0 attribute assertions. Select the appropriate version from the drop-down list.

Important

SAML 1.0 recommends the use of the `http://www.w3.org/TR/2001/REC-xml-c14n-20010315` XML Signature Canonicalization algorithm. When inserting signed SAML 1.0 assertions into XML documents, it is quite likely that subsequent signature verification of these assertions will fail. This is due to the side effect of the algorithm including inherited namespaces into canonical XML calculations of the inserted SAML assertion that were not present when the assertion was generated.

For this reason, Oracle recommend that SAML 1.1 or 2.0 is used when signing assertions as they both uses the exclusive canonical algorithm `http://www.w3.org/2001/10/xml-exc-c14n#`, which safeguards inserted assertions from such changes of context in the XML document. Please see section 5.4.2 of the `oasis-sstc-saml-core-1.0.pdf` and section 5.4.2 of `sstc-saml-core-1.1.pdf` documents, both of which are available at `http://www.oasis-open.org`.

Assertion Location

The options on the **Assertion Location** tab specify where the SAML assertion is inserted in the message. By default, the SAML assertion is added to the WS-Security block with the current SOAP actor/role. The following options are available:

Append to Root or SOAP Header:
Appends the SAML assertion to the message root for a non-SOAP XML message, or to the SOAP Header for a SOAP message. For example, this option may be suitable for cases where this filter may process SOAP XML messages or non-SOAP XML messages.

Add to WS-Security Block with SOAP Actor/Role:
Adds the SAML assertion to the WS-Security block with the specified SOAP actor (SOAP 1.0) or role (SOAP 1.1). By default, the assertion is added with the current SOAP actor/role only, which means the WS-Security block with no actor. You can select a specific SOAP actor/role when available from the drop-down list.

XPath Location:
If you wish to insert the SAML assertion at an arbitrary location in the message, you can use an XPath expression to specify the exact location in the message. You can select XPath expressions from the drop-down list. The default is the `First WSSE Security Element`, which has an XPath expression of `//wsse:Security`. You can add, edit, or remove expressions by clicking the relevant button. For more details, see the *Configure XPath expressions* topic.

You can specify exactly how the SAML assertion is inserted using the following options:

- **Append to node returned by XPath expression** (the default)
- **Insert before node returned by XPath expression**
- **Replace node returned by XPath expression**

Insert into Message Attribute:
Specify a message attribute to store the SAML assertion from the drop-down list (for example, `saml.assertion`). Alternatively, you can also enter a custom message attribute in this field (for example, `my.test.assertion`). The SAML assertion can then be accessed downstream in the policy.

Subject Confirmation Method

The settings on the **Subject Confirmation Method** tab determine how the `<SubjectConfirmation>` block of the SAML assertion is generated. When the assertion is consumed by a downstream Web service, the information contained in the `<SubjectConfirmation>` block can be used to authenticate either the end-user that authenticated to the API Gateway Explorer, or the issuer of the assertion, depending on what is configured.

The following is a typical `<SubjectConfirmation>` block:

```
<saml:SubjectConfirmation>
  <saml:ConfirmationMethod>
    urn:oasis:names:tc:SAML:1.0:cm:holder-of-key
  </saml:ConfirmationMethod>
    <dsig:KeyInfo xmlns:dsig="http://www.w3.org/2000/09/xmldsig#">
      <dsig:X509Data>
        <dsig:X509SubjectName>CN=oracle</dsig:X509SubjectName>
        <dsig:X509Certificate>
          MIICmzCCAY ...... mB9CJEw4Q=
        </dsig:X509Certificate>
      </dsig:X509Data>
    </dsig:KeyInfo>
  </saml:SubjectConfirmation>
</saml:SubjectConfirmation>
```

The following configuration fields are available on the **Subject Confirmation Method** tab:

Method:
The value selected here determines the value of the `<ConfirmationMethod>` element. The following table shows the available methods, their meanings, and their respective values in the `<ConfirmationMethod>` element:

Method	Meaning	Value
Holder Of Key	The API Gateway Explorer includes the key used to prove that the API Gateway Explorer is the holder of the key, or includes a reference to the key.	`urn:oasis:names:tc:SAML:1.0:cm:holder-of-key`
Bearer	The subject of the assertion is the bearer of the assertion.	`urn:oasis:names:tc:SAML:1.0:cm:bearer`
SAML Artifact	The subject of the assertion is the user that presented a SAML Artifact to the API Gateway Explorer.	`urn:oasis:names:tc:SAML:1.0:cm:artifact`
Sender Vouches	Use this confirmation method to assert that the API Gateway Explorer is acting on behalf of the authenticated end-user. No other information relating to the context of the assertion is	`urn:oasis:names:tc:SAML:1.0:cm:bearer`

Method	Meaning	Value
	sent. It is recommended that both the assertion **and** the SOAP Body must be signed if this option is selected. These message parts can be signed by using the *XML signature generation* filter.	

Note

You can also leave the **Method** field blank, in which case no `<ConfirmationMethod>` block is inserted into the assertion.

Holder-of-Key Configuration:
When you select `Holder-of-Key` as the SAML subject confirmation in the **Method** field, you must configure how information about the key is to be included in the message. There are a number of configuration options available depending on whether the key is a symmetric or asymmetric key.

Asymmetric Key:
If you want to use an asymmetric key as proof that the API Gateway Explorer is the holder-of-key entity, you must select the **Asymmetric Key** radio button, and then configure the following fields on the **Asymmetric** tab:

- **Certificate from Store**:
  If you want to select a key that is stored in the Certificate Store, select this option and click the **Signing Key** button. On the **Select Certificate** screen, select the box next to the certificate that is associated with the key that you want to use.
- **Certificate from Message Attribute**:
  Alternatively, the key may have already been used by a previous filter in the policy (for example, to sign a part of the message). In this case, the key is stored in a message attribute. You can specify this message attribute in this field.

Symmetric Key:
If you want to use a symmetric key as proof that the API Gateway Explorer is the holder of key, select the **Symmetric Key** radio button, and configure the fields on the **Symmetric** tab:

- **Generate Symmetric Key, and Save in Message Attribute**:
  If you select this option, the API Gateway Explorer generates a symmetric key, which is included in the message before it is sent to the client. By default, the key is saved in the `symmetric.key` message attribute.
- **Symmetric Key in Message Attribute**:
  If a previous filter (for example, a **Sign Message** filter) has already used a symmetric key, you can to reuse this key as proof that the API Gateway Explorer is the holder-of-key entity. You must enter the name of the message attribute in the field provided, which defaults to `symmetric.key`.
- **Encrypt using Certificate from Certificate Store**:
  When a symmetric key is used, you must assume that the recipient has no prior knowledge of this key. It must, therefore, be included in the message so that the recipient can validate the key. To avoid meet-in-the-middle style attacks, where a hacker could eavesdrop on the communication channel between the API Gateway Explorer and the recipient and gain access to the symmetric key, the key must be encrypted so that only the recipient can decrypt the key. One way of doing this is to select the recipient's certificate from the Certificate Store. By encrypting the symmetric key with the public in the recipient's certificate, the key can only be decrypted by the recipient's private key, to which only the recipient has access. Select the **Signing Key** button and then select the recipient's certificate on the **Select Certificate** dialog.
- **Encrypt using Certificate from Message Attribute**:
  Alternatively, if the recipient's certificate has already been used (perhaps to encrypt part of the message) this certific-

ate is stored in a message attribute. You can enter the message attribute in this field.

- **Symmetric Key Length**:
  Enter the length (in bits) of the symmetric key to use.
- **Key Wrap Algorithm**:
  Select the algorithm to use to encrypt (*wrap*) the symmetric key.

Key Info:
The **Key Info** tab must be configured regardless of whether you have elected to use symmetric or asymmetric keys. It determines how the key is included in the message. The following options are available:

- **Do Not Include Key Info**:
  Select this option if you do not wish to include a `<KeyInfo>` section in the SAML assertion.
- **Embed Public Key Information**:
  If this option is selected, details about the key are included in a `<KeyInfo>` block in the message. You can include the full certificate, expand the public key, include the distinguished name, and include a key name in the `<KeyInfo>` block by selecting the appropriate boxes. When selecting the **Include Key Name** field, you must enter a name in the **Value** field, and select the **Text Value** or **Distinguished Name Attribute** radio button, depending on the source of the key name.
- **Put Certificate in Attachment**:
  Select this option to add the certificate as an attachment to the message. The certificate is then referenced from the `<KeyInfo>` block.
- **Security Token Reference**:
  The Security Token Reference (STR) provides a way to refer to a key contained in a SOAP message from another part of the message. It is often used in cases where different security blocks in a message use the same key material, and it is considered an overhead to include the key more than once in the message.
  When this option is selected, a `<wsse:SecurityTokenReference>` element is inserted into the `<KeyInfo>` block. It references the key material using a `URI` to point to the key material and a `ValueType` attribute to indicate the type of reference used. For example, if the STR refers to an encrypted key, you should select `EncryptedKey` from the drop-down list, whereas if it refers to a `BinarySecurityToken`, select `X509v3` from the dropdown. Other options are available to enable more specific security requirements.

Advanced settings

The settings on the **Advanced** tab include the following fields.

Select Required Layout Type:
WS-Policy and SOAP Message Security define a set of rules that determine the layout of security elements that appear in the WS-Security header in a SOAP message. The SAML assertion is inserted into the WS-Security header according to the layout option selected here. The available options correspond to the WS-Policy Layout assertions of `Strict`, `Lax`, `LaxTimestampFirst`, and `LaxTimestampLast`.

Indent:
Select this method to ensure that the generated signature is properly indented.

Security Token Reference:
The generated SAML attribute assertion can be encapsulated in a `<SecurityTokenReference>` block. The following example demonstrates this:

```
<soap:Header>
 <wsse:Security
     xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/12/secext"
     soap:actor="oracle">
  <wsse:SecurityTokenReference>
   <wsse:Embedded>
    <saml:Assertion xmlns:saml="urn:oasis:names:tc:SAML:2.0:assertion"
                    ID="Id-0000010a3c4ff12c-0000000000000002"
```

```
                    IssueInstant="2006-03-27T15:26:12Z" Version="2.0">
     <saml:Issuer Format="urn:oasis ... WindowsDomainQualifiedName">
       TestCA
     </saml:Issuer>
     <saml:Subject>
      <saml:NameID Format="urn:oasis ... WindowsDomainQualifiedName">
       TestUser
      </saml:NameID>
     </saml:Subject>
     <saml:Conditions NotBefore="2005-03-27T15:20:40Z"
                      NotOnOrAfter="2028-03-27T17:20:40Z"/>
     <saml:AttributeStatement>
      <saml:Attribute Name="role" NameFormat="http://www.oracle.com">
       <saml:AttributeValue>admin</saml:AttributeValue>
      </saml:Attribute>
      <saml:Attribute Name="email" NameFormat="http://www.oracle.com">
       <saml:AttributeValue>joe@oracle.com</saml:AttributeValue>
      </saml:Attribute>
      <saml:Attribute Name="attrib1" NameFormat="">
       <saml:AttributeValue xsi:nil="true"/>
       <saml:AttributeValue>value1</saml:AttributeValue>
      </saml:Attribute>
     </saml:AttributeStatement>
    </saml:Assertion>
   </wsse:Embedded>
  </wsse:SecurityTokenReference>
 </wsse:Security>
</soap:Header>
```

To add the SAML assertion to a `<SecurityTokenReference>` block like in this example, select the **Embed SAML assertion within Security Token Reference** option. Otherwise, select **No Security Token Reference**.

Retrieve attribute from message

Overview

The **Retrieve from Message** filter uses XPath expressions to extract the value of an XML element or attribute from the message and set it to an internal message attribute. The XPath expression can also return a `NodeList`, and the `NodeList` can be set to the specified message attribute.

Configuration

The following fields are available on the **Retrieve from Message** filter configuration screen:

Name:
Enter an appropriate name for this filter.

Use the following XPath:
Configure an XPath expression to retrieve the desired content.

Click the **Add** button to add an XPath expression. You can add and remove existing expressions by clicking the **Edit** and **Remove** buttons respectively.

Store the extracted content:
Select an option to specify how the extracted content is stored. The options are:

- **of the node as text (java.lang.String)**
  This option saves the content of the node retrieved from the XPath expression to the specified message attribute as a `String`.
- **for all nodes found as text (java.lang.String)**
  This option saves all nodes retrieved from the XPath expression to the specified message attribute as a `String` (for example, `<node1>test</node1>`). This option is useful for extracting `<Signature>`, `<Security>`, and `<UsernameToken>` blocks, as well as proprietary blocks of XML from messages.
- **for all nodes found as a list (java.util.List)**
  This option saves the nodes retrieved from the XPath expression to the specified message attribute as a Java `List`, where each item is of type `Node`. For example, if the XPath returns `<node1>test</node1>`, there is one node in the `List` (`<node1>`). The child text node (`test`) is accessible from that node, but is not saved as an entry in the `List` at the top-level.

Extracted content will be stored in attribute named:
The API Gateway Explorer sets the value of the message attribute selected here to the value extracted from the message. You can also enter a user-defined message attribute.

Optionally the message payload can be replaced by the extracted content:
Select this option to take the value being set into the attribute and add it to the content body of the response. This option is not selected by default.

Use the following content type for new payload:
This field is only available if the preceding option is selected. This allows you to specify the content type for the response, based on what will be added to the content body.

Insert SAML authentication assertion

Overview

After successfully authenticating a client, the API Gateway Explorer can insert a SAML (Security Assertion Markup Language) authentication assertion into the SOAP message. Assuming all other security filters in the policy are successful, the assertion is eventually consumed by a downstream web service.

You can refer to the following example of a signed SAML authentication assertion when configuring the **Insert SAML Authentication Assertion** filter:

```
<?xml version="1.0" encoding="UTF-8"?>
<soap-env:Envelope xmlns:soap-env="http://schemas.xmlsoap.org/soap/envelope/">
<soap-env:Header xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/04/secext">
<wsse:Security>
 <saml:Assertion xmlns:saml="urn:oasis:names:tc:SAML:1.0:assertion"
      AssertionID="oracle-1056477425082"
      Id="oracle-1056477425082"
      IssueInstant="2003-06-24T17:57:05Z"
      Issuer="CN=Sample User,....,C=IE"
      MajorVersion="1"
      MinorVersion="0">
    <saml:Conditions
      NotBefore="2003-06-20T16:20:10Z"
      NotOnOrAfter="2003-06-20T18:20:10Z"/>
    <saml:AuthenticationStatement
      AuthenticationInstant="2003-06-24T17:57:05Z"
      AuthenticationMethod="urn:oasis:names:tc:SAML:1.0:am:password">
      <saml:SubjectLocality IPAddress="192.168.0.32"/>
      <saml:Subject>
        <saml:NameIdentifier
      Format="urn:oasis:names:tc:SAML:1.0:assertion#X509SubjectName">
                  sample
        </saml:NameIdentifier>
      </saml:Subject>
    </saml:AuthenticationStatement>
  <dsig:Signature xmlns:dsig="http://www.w3.org/2000/09/xmldsig#"
                     id="Sample User">
    <dsig:SignedInfo>
      .....
    </dsig:SignedInfo>
    <dsig:SignatureValue>
      rpa/......0g==
    </dsig:SignatureValue>
    <dsig:KeyInfo>
      .....
    </dsig:KeyInfo>
  </dsig:Signature>
 </saml:Assertion>
</wsse:Security>
</soap-env:Header>
<soap-env:Body>
  <ns1:getTime xmlns:ns1="urn:timeservice">
</ns1:getTime>
</soap-env:Body>
</soap-env:Envelope>
```

General settings

Configure the following field:

Name:
Enter an appropriate name for the filter.

Assertion details settings

Configure the following fields on the **Assertion Details** tab:

Issuer Name:
Select the certificate containing the Distinguished Name (DName) that you want to use as the Issuer of the SAML assertion. This DName is included in the SAML assertion as the value of the `Issuer` attribute of the `<saml:Assertion>` element. For an example, see the sample SAML assertion above.

Expire In:
Specify the lifetime of the assertion in this field. The lifetime of the assertion lasts from the time of insertion until the specified amount of time has elapsed.

Drift Time:
The **Drift Time** is used to account for differences in the clock times of the machine hosting the API Gateway Explorer (that generate the assertion) and the machines that consume the assertion. The specified time is subtracted from the time at which the API Gateway Explorer generates the assertion.

SAML Version:
You can create SAML 1.0, 1.1, and 2.0 attribute assertions. Select the appropriate version from the drop-down list.

Important

SAML 1.0 recommends the use of the `http://www.w3.org/TR/2001/REC-xml-c14n-20010315` XML Signature Canonicalization algorithm. When inserting signed SAML 1.0 assertions into XML documents, it is quite likely that subsequent signature verification of these assertions will fail. This is due to the side effect of the algorithm including inherited namespaces into canonical XML calculations of the inserted SAML assertion that were not present when the assertion was generated.

For this reason, Oracle recommend that SAML 1.1 or 2.0 is used when signing assertions as they both uses the exclusive canonical algorithm `http://www.w3.org/2001/10/xml-exc-c14n#`, which safeguards inserted assertions from such changes of context in the XML document. Please see section 5.4.2 of the `oasis-sstc-saml-core-1.0.pdf` and section 5.4.2 of `sstc-saml-core-1.1.pdf` documents, both of which are available at `http://www.oasis-open.org`.

Assertion location settings

The options on the **Assertion Location** tab specify where the SAML assertion is inserted in the message. By default, the SAML assertion is added to the WS-Security block with the current SOAP actor/role. The following options are available:

Append to Root or SOAP Header:
Appends the SAML assertion to the message root for a non-SOAP XML message, or to the SOAP Header for a SOAP message. For example, this option may be suitable for cases where this filter may process SOAP XML messages or non-SOAP XML messages.

Add to WS-Security Block with SOAP Actor/Role:
Adds the SAML assertion to the WS-Security block with the specified SOAP actor (SOAP 1.0) or role (SOAP 1.1). By default, the assertion is added with the current SOAP actor/role only, which means the WS-Security block with no actor. You can select a specific SOAP actor/role when available from the drop-down list.

XPath Location:
If you wish to insert the SAML assertion at an arbitrary location in the message, you can use an XPath expression to specify the exact location in the message. You can select XPath expressions from the drop-down list. The default is the

`First WSSE Security Element`, which has an XPath expression of `//wsse:Security`. You can add, edit, or remove expressions by clicking the relevant button. For more details, see the *Configure XPath expressions* topic.

You can also specify how exactly the SAML assertion is inserted using the following options:

- **Append to node returned by XPath expression** (the default)
- **Insert before node returned by XPath expression**
- **Replace node returned by XPath expression**

Insert into Message Attribute:
Specify a message attribute to store the SAML assertion from the drop-down list (for example, `saml.assertion`). Alternatively, you can also enter a custom message attribute in this field (for example, `my.test.assertion`). The SAML assertion can then be accessed downstream in the policy.

Subject confirmation method settings

The settings on the **Subject Confirmation Method** tab determine how the `<SubjectConfirmation>` block of the SAML assertion is generated. When the assertion is consumed by a downstream Web service, the information contained in the `<SubjectConfirmation>` block can be used to authenticate the end-user that authenticated to the API Gateway Explorer, or the issuer of the assertion, depending on what is configured.

The following is a typical `<SubjectConfirmation>` block:

```
<saml:SubjectConfirmation>
  <saml:ConfirmationMethod>
    urn:oasis:names:tc:SAML:1.0:cm:holder-of-key
  </saml:ConfirmationMethod>
    <dsig:KeyInfo xmlns:dsig="http://www.w3.org/2000/09/xmldsig#">
      <dsig:X509Data>
        <dsig:X509SubjectName>CN=oracle</dsig:X509SubjectName>
        <dsig:X509Certificate>
          MIICmzCCAY ...... mB9CJEw4Q=
        </dsig:X509Certificate>
      </dsig:X509Data>
    </dsig:KeyInfo>
  </saml:SubjectConfirmation>
</saml:SubjectConfirmation>
```

The following configuration fields are available on the **Subject Confirmation Method** tab:

Method:
The selected value determines the value of the `<ConfirmationMethod>` element. The following table shows the available methods, their meanings, and their respective values in the `<ConfirmationMethod>` element:

Method	Meaning	Value
Holder Of Key	The API Gateway Explorer includes the key used to prove that the API Gateway Explorer is the holder of the key, or it includes a reference to the key.	`urn:oasis:names:tc:SAML:1.0:cm:holder-of-key`
Bearer	The subject of the assertion is the bearer of the assertion.	`urn:oasis:names:tc:SAML:1.0:cm:bearer`
SAML Artifact	The subject of the assertion is the user that presented a SAML Artifact to the API Gateway Explorer.	`urn:oasis:names:tc:SAML:1.0:cm:artifact`

Method	Meaning	Value
Sender Vouches	Use this confirmation method to assert that the API Gateway Explorer is acting on behalf of the authenticated end-user. No other information relating to the context of the assertion is sent. It is recommended that both the assertion **and** the SOAP Body must be signed if this option is selected. These message parts can be signed by using the *XML signature generation* filter.	`urn:oasis:names:tc:SAML:1.0:cm:bearer`

Note

You can also leave the **Method** field blank, in which case no `<ConfirmationMethod>` block is inserted into the assertion.

Holder-of-Key Configuration:
When you select `Holder-of-Key` as the SAML subject confirmation in the `Method` field, you must configure how information about the key is included in the message. There are a number of configuration options available depending on whether the key is a symmetric or asymmetric key.

Asymmetric Key:
If you want to use an asymmetric key as proof that the API Gateway Explorer is the holder-of-key entity, you must select the **Asymmetric Key** radio button and then configure the following fields on the **Asymmetric** tab:

- **Certificate from Store**:
  If you want to select a key that is stored in the Certificate Store, select this option and click the **Signing Key** button. On the **Select Certificate** screen, select the box next to the certificate that is associated with the key that you want to use.
- **Certificate from Selector Expression**:
  Alternatively, the key may have already been used by a previous filter in the policy (for example, to sign a part of the message). In this case, the key can be retrieved using the selector expression entered in this field. Using a selector enables settings to be evaluated and expanded at runtime based on metadata (for example, in a message attribute, Key Property Store (KPS), or environment variable). For more details, see *Select configuration values at runtime*.

Symmetric Key:
If you want to use a symmetric key as proof that the API Gateway Explorer is the holder of key, select the **Symmetric Key** radio button, and configure the fields on the **Symmetric** tab:

- **Generate Symmetric Key, and Save in Message Attribute**:
  If you select this option, the API Gateway Explorer generates a symmetric key, which is included in the message before it is sent to the client. By default, the key is saved in the `symmetric.key` message attribute.
- **Symmetric Key Selector Expression**:
  If a previous filter (for example, a **Sign Message** filter) has already used a symmetric key, you can reuse this key as proof that the API Gateway Explorer is the holder-of-key entity. Enter the name of the selector expresion (for example, message attribute) in the field provided, which defaults to `${symmetric.key}`. Using a selector enables settings to be evaluated and expanded at runtime based on metadata (for example, in a message attribute, Key Property Store (KPS), or environment variable). For more details, see *Select configuration values at runtime*.
- **Encrypt using Certificate from Certificate Store**:
  When a symmetric key is used, you must assume that the recipient has no prior knowledge of this key. It must, therefore, be included in the message so that the recipient can validate the key. To avoid meet-in-the-middle style

attacks, where a hacker could eavesdrop on the communication channel between the API Gateway Explorer and the recipient and gain access to the symmetric key, the key must be encrypted so that only the recipient can decrypt the key. One way of doing this is to select the recipient's certificate from the Certificate Store. By encrypting the symmetric key with the public in the recipient's certificate, the key can only be decrypted by the recipient's private key, to which only the recipient has access. Select the **Signing Key** button, and select the recipient's certificate on the **Select Certificate** dialog.

- **Encrypt using Certificate from Message Attribute**:
  Alternatively, if the recipient's certificate has already been used (perhaps to encrypt part of the message) this certificate is stored in a message attribute. You can enter this message attribute in this field.
- **Symmetric Key Length**:
  Enter the length (in bits) of the symmetric key to use.
- **Key Wrap Algorithm**:
  Select the algorithm to use to encrypt (*wrap*) the symmetric key.

Key Info:
The **Key Info** tab must be configured regardless of whether you have elected to use symmetric or asymmetric keys. It determines how the key is included in the message. The following options are available:

- **Do Not Include Key Info**:
  Select this option if you do not wish to include a <KeyInfo> section in the SAML assertion.
- **Embed Public Key Information**:
  If this option is selected, details about the key are included in a `<KeyInfo>` block in the message. You can include the full certificate, expand the public key, include the distinguished name, and include a key name in the `<KeyInfo>` block by selecting the appropriate boxes. When selecting the **Include Key Name** field, you must enter a name in the **Value** field, and then select the **Text Value** or **Distinguished Name Attribute** radio button, depending on the source of the key name.
- **Put Certificate in Attachment**:
  Select this option to add the certificate as an attachment to the message. The certificate is then referenced from the `<KeyInfo>` block.
- **Security Token Reference**:
  The Security Token Reference (STR) provides a way to refer to a key contained within a SOAP message from another part of the message. It is often used in cases where different security blocks in a message use the same key material and it is considered an overhead to include the key more than once in the message.
  When this option is selected, a `<wsse:SecurityTokenReference>` element is inserted into the `<KeyInfo>` block. It references the key material using a `URI` to point to the key material and a `ValueType` attribute to indicate the type of reference used. For example, if the STR refers to an encrypted key, you should select `EncryptedKey` from the dropdown, whereas if it refers to a `BinarySecurityToken`, you should select `X509v3` from the dropdown. Other options are available to enable more specific security requirements.

Advanced settings

Select Required Layout Type:
WS-Policy and SOAP Message Security define a set of rules that determine the layout of security elements that appear in the WS-Security header within a SOAP message. The SAML assertion will be inserted into the WS-Security header according to the layout option selected here. The available options correspond to the WS-Policy Layout assertions of `Strict`, `Lax`, `LaxTimestampFirst`, and `LaxTimestampLast`.

Insert SAML Attribute Statement:
You can insert a SAML attribute statement into the generated SAML authentication assertion. If you select this option, a SAML attribute assertion is generated using attributes stored in the `attribute.lookup.list` message attribute and subsequently inserted into the assertion. The `attribute.lookup.list` attribute must have been populated previously by an attribute lookup filter for the attribute statement to be generated successfully.

Indent:
Select this method to ensure that the generated signature is properly indented.

Security Token Reference:
The generated SAML authentication assertion can be encapsulated within a <SecurityTokenReference> block. The following example demonstrates this:

```
<soap:Header>
 <wsse:Security
     xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/12/secext"
     soap:actor="oracle">
  <wsse:SecurityTokenReference>
   <wsse:Embedded>
    <saml:Assertion xmlns:saml="urn:oasis:names:tc:SAML:1.0:assertion"
                    AssertionID="Id-00000109fee52b06-0000000000000012"
                    IssueInstant="2006-03-15T17:12:45Z"
                    Issuer="oracle" MajorVersion="1" MinorVersion="0">
     <saml:Conditions NotBefore="2006-03-15T17:12:39Z"
                      NotOnOrAfter="2006-03-25T17:12:39Z"/>
      <saml:AuthenticationStatement
          AuthenticationMethod="urn:oasis:names:tc:SAML:1.0:am:password"
          AuthenticationInstant="2006-03-15T17:12:45Z">
       <saml:Subject>
        <saml:NameIdentifier Format="Oracle-Username-Password">
            admin
        </saml:NameIdentifier>
       <saml:SubjectConfirmation>
        <saml:ConfirmationMethod>
            urn:oasis:names:tc:SAML:1.0:cm:artifact
        </saml:ConfirmationMethod>
       </saml:SubjectConfirmation>
      </saml:Subject>
     </saml:AuthenticationStatement>
    </saml:Assertion>
   </wsse:Embedded>
  </wsse:SecurityTokenReference>
 </wsse:Security>
</soap:Header>
```

To add the SAML assertion to a <SecurityTokenReference> block as in the example above, select the **Embed SAML assertion within Security Token Reference** option. Otherwise, select **No Security Token Reference**.

Insert WS-Security UsernameToken

Overview

When a client has been successfully authenticated, the API Gateway Explorer can insert a *WS-Security UsernameToken* into the downstream message as proof of the authentication event. The `<wsse:UsernameToken>` token enables a user's identity to be inserted into the XML message so that it can be propagated over a chain of web services.

A typical example would see a user authenticating to the API Gateway Explorer using HTTP digest authentication. After successfully authenticating the user, the API Gateway Explorer inserts a WS-Security UsernameToken into the message and digitally signs it to prevent anyone from tampering with the token.

The following example shows the format of the `<wsse:UsernameToken>` token:

```
<wsse:UsernameToken wsu:Id="oracle"
      xmlns:wsu="http://schemas.xmlsoap.org/ws/2003/06/utility">
  <wsu:Created>2006.01.13T-10:42:43Z</wsu:Created>
  <wsse:Username>oracle</wsse:Username>
  <wsse:Nonce EncodingType="UTF-8">
      KFIy9LgzhmDPNiqU/B9ZiWKXfEVNvFyn6KWYP+1zVt8=
  </wsse:Nonce>
  <wsse:Password Type="wsse:PasswordDigest">
      CxWj1OMnYj7dddMnU/DrOhyY3j4=
  </wsse:Password>
</wsse:UsernameToken>
```

This topic explains how to configure the API Gateway Explorer to insert a WS-Security UsernameToken after successfully authenticating a user.

General settings

To configure general settings, complete the following fields:

Name:
Enter an appropriate name for the filter.

Actor:
The UsernameToken is inserted into the WS-Security block identified by the specified SOAP *Actor*.

Credential details

To configure the credential details, complete the following fields:

Username:
Enter the name of the user included in the UsernameToken. By default, the `authentication.subject.id` message attribute is stored, which contains the name of an authenticated user.

Include Nonce:
Select this option if you wish to include a nonce in the UsernameToken. A nonce a random number that is typically used to help prevent replay attacks.

Include Password:
Select this option if you wish to include a password in the UsernameToken.

Password:
If the **Include Password** check box is selected, the API Gateway Explorer inserts the user's password into the generated WS-Security UsernameToken. It can insert **Clear** or **SHA1 Digest** version of the password, depending on which radio

button you select. Oracle recommends the digest form of the password to avoid potential eavesdropping.

You can either explicitly enter the password for this user in the **Password** field, or use a message attribute by selecting the **Wildcard** option, and entering the message attribute in the field provided. By default, the `authentication.subject.password` attribute is used, which contains the password used by the user to authenticate to the API Gateway Explorer.

Advanced options

To configure advanced options, complete the following field:

Indent:
Select this option to add indentation to the generated `UsernameToken` and `Signature` blocks. This makes the security tokens more human-readable.

Set User Name

Overview

The **Set User Name** filter is used to configure the user name, password for this user, and user attributes that can be used when generating SAML authentication, authorization, and attribute assertions.

Configuration

Complete the following fields on the **Set User Name** filter screen, which is available from the **Insert Security Token** category of filters:

Name:
Enter a name for the filter.

User Name:
Enter the user name that will be used when generating security tokens, e.g. SAML authentication assertions, WS-Security UsernameTokens, etc.

Password:
Enter a password for this user.

Credential Format:
Specify the format of the credential given in the **User Name** field above. The format can be either a User name or an X.509 Distinguished Name.

User Attributes:
User attributes can be added by clicking on the **Add** button. Enter a **Name**, **Value**, and **Namespace** in the fields provided. These attributes can then be added to a SAML attribute assertion using the **Insert SAML Attribute Assertion** filter.

Insert SAML authorization assertion

Overview

After successfully authorizing a client, the API Gateway Explorer can insert a Security Assertion Markup Language (SAML) authorization assertion into the SOAP message. Assuming all other security filters in the policy are successful, the assertion will eventually be consumed by a downstream web service.

The following example of a signed SAML authorization assertion might be useful when configuring this filter.

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://.../soap/envelope/">
 <soap:Header xmlns:wsse="http://.../secext">
  <wsse:Security>
   <saml:Assertion
     xmlns:saml="urn:oasis:names:tc:SAML:1.0:assertion"
     AssertionID="oracle-1056130475340"
     Id="oracle-1056130475340"
     IssueInstant="2003-06-20T17:34:35Z"
     Issuer="CN=Sample User,...........,C=IE"
     MajorVersion="1"
     MinorVersion="0">
    <saml:Conditions
      NotBefore="2003-06-20T16:20:10Z"
      NotOnOrAfter="2003-06-20T18:20:10Z"/>
    <saml:AuthorizationDecisionStatement
      Decision="Permit"
      Resource="http://www.oracle.com/service">
     <saml:Subject>
      <saml:NameIdentifier
        Format="urn:oasis:names:tc:SAML:1.0:assertion#X509SubjectName">
         sample
      </saml:NameIdentifier>
     </saml:Subject>
    </saml:AuthorizationDecisionStatement>
    <dsig:Signature xmlns:dsig="http://.../xmldsig#" id="Sample User">
      <!-- XML SIGNATURE INSIDE ASSERTION -->
    </dsig:Signature>
   </saml:Assertion>
  </wsse:Security>
 </soap:Header>
 <soap:Body>
  <ns1:getTime xmlns:ns1="urn:timeservice">
  </ns1:getTime>
 </soap:Body>
</soap:Envelope>
```

General settings

Configure the following field:

Name:
Enter an appropriate name for the filter.

Assertion details settings

Configure the following fields on the **Assertion Details** tab:

Issuer Name:

Select the certificate containing the Distinguished Name (DName) to use as the Issuer of the SAML assertion. This DName is included in the SAML assertion as the value of the `Issuer` attribute of the `<saml:Assertion>` element. For an example, see the sample SAML assertion above.

Expire In:
Specify the lifetime of the assertion in this field. The lifetime of the assertion lasts from the time of insertion until the specified amount of time has elapsed.

Drift Time:
The **Drift Time** is used to account for differences in the clock times of the machine hosting the API Gateway Explorer (that generate the assertion) and the machines that consume the assertion. The specified time is subtracted from the time at which the API Gateway Explorer generates the assertion.

SAML Version:
You can create SAML 1.0, 1.1, and 2.0 attribute assertions. Select the appropriate version from the list.

Important

SAML 1.0 recommends the use of the `http://www.w3.org/TR/2001/REC-xml-c14n-20010315` XML Signature Canonicalization algorithm. When inserting signed SAML 1.0 assertions into XML documents, it is quite likely that subsequent signature verification of these assertions will fail. This is due to the side effect of the algorithm including inherited namespaces into canonical XML calculations of the inserted SAML assertion that were not present when the assertion was generated.

For this reason, Oracle recommend that SAML 1.1 or 2.0 is used when signing assertions as they both uses the exclusive canonical algorithm `http://www.w3.org/2001/10/xml-exc-c14n#`, which safeguards inserted assertions from such changes of context in the XML document. For more information, see the `oasis-sstc-saml-core-1.0.pdf` and the `sstc-saml-core-1.1.pdf` documents, both of which are available at http://www.oasis-open.org.

Resource:
Enter the resource for which you want to obtain the authorization assertion. You should specify the resource as a URI (for example, `http://www.oracle.com/TestService`). The name of the resource is then included in the assertion.

Action:
You can specify the operations that the user can perform on the resource using the **Action** field. This entry is a comma-separated list of permissions. For example, the following is a valid entry: `read,write,execute`.

Assertion location settings

The options on the **Assertion Location** tab specify where the SAML assertion is inserted in the message. By default, the SAML assertion is added to the WS-Security block with the current SOAP actor/role. The following options are available:

Append to Root or SOAP Header:
Appends the SAML assertion to the message root for a non-SOAP XML message, or to the SOAP Header for a SOAP message. For example, this option may be suitable for cases where this filter may process SOAP XML messages or non-SOAP XML messages.

Add to WS-Security Block with SOAP Actor/Role:
Adds the SAML assertion to the WS-Security block with the specified SOAP actor (SOAP 1.0) or role (SOAP 1.1). By default, the assertion is added with the current SOAP actor/role only, which means the WS-Security block with no actor. You can select a specific SOAP actor/role when available from the list.

XPath Location:
To insert the SAML assertion at an arbitrary location in the message, you can use an XPath expression to specify the exact location in the message. You can select XPath expressions from the list. The default is the `First WSSE Security Element`, which has an XPath expression of `//wsse:Security`. You can add, edit, or remove expressions by clicking the relevant button. For more details, see the *Configure XPath expressions* topic.

You can also specify how exactly the SAML assertion is inserted using the following options:

- **Append to node returned by XPath expression** (the default)
- **Insert before node returned by XPath expression**
- **Replace node returned by XPath expression**

Insert into Message Attribute:
Specify a message attribute to store the SAML assertion from the list (for example, `saml.assertion`). Alternatively, you can also enter a custom message attribute in this field (for example, `my.test.assertion`). The SAML assertion can then be accessed downstream in the policy.

Subject confirmation method settings

The settings on the **Subject Confirmation Method** tab determine how the `<SubjectConfirmation>` block of the SAML assertion is generated. When the assertion is consumed by a downstream Web service, the information contained in the `<SubjectConfirmation>` block can be used to authenticate either the end-user that authenticated to the API Gateway Explorer, or the issuer of the assertion, depending on what is configured.

The following is a typical `<SubjectConfirmation>` block:

```
<saml:SubjectConfirmation>
  <saml:ConfirmationMethod>
    urn:oasis:names:tc:SAML:1.0:cm:holder-of-key
  </saml:ConfirmationMethod>
    <dsig:KeyInfo xmlns:dsig="http://www.w3.org/2000/09/xmldsig#">
      <dsig:X509Data>
        <dsig:X509SubjectName>CN=oracle</dsig:X509SubjectName>
        <dsig:X509Certificate>
          MIICmzCCAY ...... mB9CJEw4Q=
        </dsig:X509Certificate>
      </dsig:X509Data>
    </dsig:KeyInfo>
  </saml:SubjectConfirmation>
</saml:SubjectConfirmation>
```

The following configuration fields are available on the **Subject Confirmation Method** tab:

Method:
The selected value determines the value of the `<ConfirmationMethod>` element. The following table shows the available methods, their meanings, and their respective values in the `<ConfirmationMethod>` element:

Method	Meaning	Value
Holder Of Key	The API Gateway Explorer includes the key used to prove that the API Gateway Explorer is the holder of the key, or it includes a reference to the key.	`urn:oasis:names:tc:SAML:1.0:cm:holder-of-key`
Bearer	The subject of the assertion is the bearer of the assertion.	`urn:oasis:names:tc:SAML:1.0:cm:bearer`
SAML Artifact	The subject of the assertion is the user that presented a SAML artifact to the API Gateway Explorer.	`urn:oasis:names:tc:SAML:1.0:cm:artifact`

Method	Meaning	Value
Sender Vouches	Use this confirmation method to assert that the API Gateway Explorer is acting on behalf of the authenticated end-user. No other information relating to the context of the assertion is sent. It is recommended that both the assertion *and* the SOAP Body must be signed if this option is selected. These message parts can be signed by using the **Sign Message** filter (see *XML signature generation*).	`urn:oasis:names:tc:SAML:1.0:cm:bearer`

Note

You can also leave the **Method** field blank, in which case no `<ConfirmationMethod>` block is inserted into the assertion.

Holder-of-Key Configuration:
When you select `Holder-of-Key` as the SAML subject confirmation in the **Method** field, you must configure how information about the key is included in the message. There are a number of configuration options available depending on whether the key is a symmetric or asymmetric key.

Asymmetric Key

To use an asymmetric key as proof that the API Gateway Explorer is the holder-of-key entity, you must select the **Asymmetric Key** radio button, and then configure the following fields on the **Asymmetric** tab:

- **Certificate from Store**:
  To select a key that is stored in the certificate store, select this option and click the **Signing Key** button. On the **Select Certificate** screen, select the box next to the certificate that is associated with the key that you want to use.
- **Certificate from Message Attribute**:
  Alternatively, the key may have already been used by a previous filter in the policy (for example, to sign a part of the message). In this case, the key is stored in a message attribute. You can specify this message attribute in this field.

Symmetric Key

To use a symmetric key as proof that the API Gateway Explorer is the holder of key, select the **Symmetric Key** radio button, and then configure the fields on the **Symmetric** tab:

- **Generate Symmetric Key, and Save in Message Attribute**:
  If you select this option, the API Gateway Explorer generates a symmetric key, which is included in the message before it is sent to the client. By default, the key is saved in the `symmetric.key` message attribute.
- **Symmetric Key in Message Attribute**:
  If a previous filter (for example, a **Sign Message** filter) has already used a symmetric key, you can to reuse this key as proof that the API Gateway Explorer is the holder-of-key entity. You must enter the name of the message attribute in the field provided, which defaults to `symmetric.key`.
- **Encrypt using Certificate from Certificate Store**:
  When a symmetric key is used, you must assume that the recipient has no prior knowledge of this key. It must be included in the message so that the recipient can validate the key. To avoid meet-in-the-middle style attacks, where a

hacker could eavesdrop on the communication channel between the API Gateway Explorer and the recipient and gain access to the symmetric key, the key must be encrypted so that only the recipient can decrypt the key. One way of doing this is to select the recipient's certificate from the certificate store. By encrypting the symmetric key with the public in the recipient's certificate, the key can only be decrypted by the recipient's private key, to which only the recipient has access. Select the **Signing Key** button and then select the recipient's certificate on the **Select Certificate** dialog.

- **Encrypt using Certificate from Message Attribute**:
  Alternatively, if the recipient's certificate has already been used (perhaps to encrypt part of the message) this certificate is stored in a message attribute. You can enter the message attribute in this field.
- **Symmetric Key Length**:
  Enter the length (in bits) of the symmetric key to use.
- **Key Wrap Algorithm**:
  Select the algorithm to use to encrypt (*wrap*) the symmetric key.

Key Info

The **Key Info** tab must be configured regardless of whether you have elected to use symmetric or asymmetric keys. It determines how the key is included in the message. The following options are available:

- **Do Not Include Key Info**:
  Select this option to not include a `<KeyInfo>` section in the SAML assertion.
- **Embed Public Key Information**:
  If this option is selected, details about the key are included in a `<KeyInfo>` block in the message. You can include the full certificate, expand the public key, include the distinguished name, and include a key name in the `<KeyInfo>` block by selecting the appropriate boxes. When selecting the **Include Key Name** field, you must enter a name in the **Value** field, and then select the **Text Value** or **Distinguished Name Attribute** radio button, depending on the source of the key name.
- **Put Certificate in Attachment**:
  Select this option to add the certificate as an attachment to the message. The certificate is then referenced from the `<KeyInfo>` block.
- **Security Token Reference**:
  The Security Token Reference (STR) provides a way to refer to a key contained in a SOAP message from another part of the message. It is often used in cases where different security blocks n a message use the same key material and it is considered an overhead to include the key more than once in the message.
  When this option is selected, a `<wsse:SecurityTokenReference>` element is inserted into the `<KeyInfo>` block. It references the key material using a `URI` to point to the key material and a `ValueType` attribute to indicate the type of reference used. For example, if the STR refers to an encrypted key, you should select `EncryptedKey` from the list, whereas if it refers to a `BinarySecurityToken`, you should select `X509v3` from the list. Other options are available to enable more specific security requirements.

Advanced settings

Configure the following fields on the **Advanced** tab:

Select Required Layout Type:
WS-Policy and SOAP Message Security define a set of rules that determine the layout of security elements that appear in the WS-Security header within a SOAP message. The SAML assertion is inserted into the WS-Security header according to the layout option selected here. The available options correspond to the WS-Policy Layout assertions of `Strict`, `Lax`, `LaxTimestampFirst`, and `LaxTimestampLast`.

Insert SAML Attribute Statement:
You can specify to insert a SAML attribute statement into the generated SAML authorization assertion. If this option is selected, a SAML attribute assertion is generated using attributes stored in the `attribute.lookup.list` message attribute and subsequently inserted into the assertion. The `attribute.lookup.list` attribute must have been populated previously by an attribute lookup filter for the attribute statement to be generated successfully.

Indent:
Select this method to ensure that the generated signature is properly indented.

Security Token Reference:
The generated SAML authorization assertion can be encapsulated within a <SecurityTokenReference> block. The following example demonstrates this:

```
<soap:Header>
 <wsse:Security
     xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/12/secext"
     soap:actor="oracle">
  <wsse:SecurityTokenReference>
   <wsse:Embedded>
    <saml:Assertion
      xmlns:saml="urn:oasis:names:tc:SAML:1.0:assertion"
      AssertionID="oracle-1056130475340"
      Id="oracle-1056130475340"
      IssueInstant="2003-06-20T17:34:35Z"
      Issuer="CN=Sample User,...........,C=IE"
      MajorVersion="1"
      MinorVersion="0">
     <saml:Conditions
       NotBefore="2003-06-20T16:20:10Z"
       NotOnOrAfter="2003-06-20T18:20:10Z"/>
     <saml:AuthorizationDecisionStatement
       Decision="Permit"
       Resource="http://www.oracle.com/service">
      <saml:Subject>
       <saml:NameIdentifier
         Format="urn:oasis:names:tc:SAML:1.0:assertion#X509SubjectName">
          sample
       </saml:NameIdentifier>
      </saml:Subject>
     </saml:AuthorizationDecisionStatement>
     <dsig:Signature xmlns:dsig="http://.../xmldsig#" id="Sample User">
       <!-- XML SIGNATURE INSIDE ASSERTION -->
     </dsig:Signature>
    </saml:Assertion>
   </wsse:Embedded>
  </wsse:SecurityTokenReference>
 </wsse:Security>
</soap:Header>
```

To add the SAML assertion to a <SecurityTokenReference> block as in the example above, select the **Embed SAML assertion within Security Token Reference** option. Otherwise, select **No Security Token Reference**.

Content type filtering

Overview

The *SOAP Messages with Attachments* specification introduced a standard for transmitting arbitrary files along with SOAP messages as part of a multipart MIME message. In this way, both XML and non-XML data, including binary data, can be encapsulated in a SOAP message. The more recent Direct Internet Message Encapsulation (DIME) specification describes another way of packaging attachments with SOAP messages.

API Gateway Explorer can accept or block multipart messages with certain MIME or DIME content types. For example, you can configure a **Content Type** filter to block multipart messages with `image/jpeg` type parts.

Allow or deny content types

The **Content Type Filtering** screen lists the content types that are allowed or denied by this filter.

Allow Content Types:
Use this option if you wish to *accept* most content types, but only want to reject a few specific types. To allow or deny incoming messages based on their content types, complete the following steps:

1. Select the **Allow content types** radio button to allow multipart messages to be routed onwards. If you wish to allow all content types, you do not need to select any of the MIME types in the list.
2. To deny multipart messages with certain MIME or DIME types as parts, select those types here. Multipart messages containing the selected MIME or DIME type parts will be rejected.

Deny Content Types:
If you wish to *block* multipart messages containing most content types, but want to allow a small number of content types, select this option. To reject multipart messages based on the content types of their parts, complete the following steps:

1. Select the **Deny content types** radio button to reject multipart messages. If you wish to block all multipart messages, you do not need to select any of the MIME or DIME types in the list.
2. To allow messages with parts of a certain MIME or DIME type, select the checkbox next to those types. Multipart messages with parts of the MIME or DIME types selected here will be allowed. All other MIME or DIME types will be denied.

MIME and DIME types can be added by clicking the **MIME/DIME Registered Types** button. The next section describes how to add, edit, and remove MIME/DIME types.

Configure MIME/DIME types

The **MIME/DIME Settings** dialog enables you to configure new and existing MIME types. When a type is added, you can configure the API Gateway Explorer to accept or block multipart messages with parts of this type.

Click the **Add** button to add a new MIME/DIME type, or highlight a type in the table, and select the **Edit** button to edit an existing type. To delete an existing type, select that type in the list, and click the **Remove** button. You can edit or add types using the **Configure MIME/DIME Type** dialog.

Enter a name for the new type in the **MIME or DIME Type** field, and the corresponding file extension in the **Extension** field.

Content validation

Overview

This tutorial describes how the API Gateway Explorer can examine the contents of an XML message to ensure that it meets certain criteria. It uses boolean XPath expressions to evaluate whether or not a specific element or attribute contains has a certain value.

For example, you can configure XPath expressions to make sure the value of an element matches a certain string, to check the value of an attribute is greater (or less) than a specific number, or that an element occurs a fixed amount of times within an XML body.

There are two ways to configure XPath expressions on this screen. Please click the appropriate link below:

- Manual XPath Configuration
- XPath Wizard

Manual XPath configuration

To manually configure a **Content Validation** rule using XPath:

1. Enter a meaningful name for this XPath content filter.
2. Click the **Add** button to add a new XPath expression. Alternatively, you can select a previously configured XPath expression from the drop-down list.
3. In order to resolve any prefixes within the XPath expression, the namespace mappings (i.e. **Prefix**, **URI**) should be entered in the table.

As an example of how this screen should be configured, consider the following SOAP message:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
 <soap:Header>
  <dsig:Signature xmlns:dsig="http://www.w3.org/2000/09/xmldsig#" id="sig1">
       ..............
       ..............
       ..............
       ..............
  </dsig:Signature>
 </soap:Header>
 <soap:Body>
  <prod:product xmlns:prod="http://www.company.com">
   <prod:name>SOA Product</prod:name>
   <prod:company>Company</prod:company>
   <prod:description>WebServices Security</prod:description>
  </prod:product>
 </soap:Body>
</soap:Envelope>
```

The following XPath expression evaluates to true if the `<company>` element contains the value `Company`:
XPath Expression: `//prod:company[text()='Company']`

In this case, you must define a mapping for the *prod* namespace as follows:

Prefix	URI
`prod`	`http://www.company.com`

In another example, the element to be examined by the XPath expression belongs to a default namespace. Consider the following SOAP message:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
 <soap:Header>
  <dsig:Signature xmlns:dsig="http://www.w3.org/2000/09/xmldsig#" id="sig1">
      ..............
      ..............
      ..............
      ..............
  </dsig:Signature>
 </soap:Header>
 <soap:Body>
  <product xmlns="http://www.company.com">
   <name>SOA Product</name>
   <company>Company</company>
   <description>Web Services Security</description>
  </product>
 </soap:Body>
</soap:Envelope>
```

The following XPath expression evaluates to true if the `<company>` element contains the value `Company`:
XPath Expression: `//ns:company[text()='Company']`

Because the `<company>` element belongs to the default (`xmlns`) namespace (`http://www.company.com`, you must make up an arbitrary prefix (`ns`) for use in the XPath expression, and assign it to `http://www.company.com`. This is necessary to distinguish between potentially several default namespaces which may exist throughout the XML message. The following mapping illustrates this:

Prefix	URI
ns	http://www.company.com

XPath wizard

The **XPath Wizard** assists administrators in creating correct and accurate XPath expressions. The wizard enables administrators to load an XML message and then run an XPath expression on it to determine what nodes are returned. To launch the **XPath Wizard**, click the **XPath Wizard Button** on the **XPath Expression** dialog.

To use the XPath Wizard, enter (or browse to) the location of an XML file in the **File** field. The contents of the XML file are displayed in the main window of the wizard. Enter an XPath expression in the **XPath** field, and click the **Evaluate** button to run the XPath against the contents of the file. If the XPath expression returns any elements (or returns true), those elements are highlighted in the main window.

If you are not sure how to write the XPath expression, you can select an element in the main window. An XPath expression to isolate this element is automatically generated and displayed in the **Selected** field. If you wish to use this expression, select the **Use this path** button, and click **OK**.

HTTP Status

Overview

This filter is responsible for verifying the HTTP status code in a response message. This enables you to filter messages based on their HTTP status code. For example, if the incoming message matches matches the specified HTTP status code, you could route the message to a specified service, otherwise continue in the policy.

HTTP status codes are returned in the *status-line* of an HTTP response. The following are some typical examples:

```
HTTP/1.1 200 OK
HTTP/1.1 400 Bad Request
HTTP/1.1 500 Internal Server Error
```

For details on how to set an HTTP status code in a response message, see the *HTTP status code* topic.

Configuration

You can verify the HTTP status code using either of the following options:

Option	Description
HTTP status in following range	Select an HTTP status code range from the drop-down list (for example, `Success Code 2xx`).
HTTP status equals	Specify the status code in the field provided (for example, `500` for an internal server error).

For a complete list of status codes, see the
HTTP Specification [http://www.w3.org/Protocols/rfc2616/rfc2616-sec6.html].

Has SOAP Body?

Overview

The **Has SOAP Body?** is a pre-configured instance of the **Content Validation** filter. For convenience purposes, it contains a pre-configured XPath expression that simply checks for the <soap:Body> element. The XPath expression is:

```
/soap:Envelope/soap:Body
```

Configuration

This filter should not need to be configured since its sole purpose is to check for the presence of the <soap:Body> element. However, you can change the XPath expression if necessary. For more information on configuring XPath expressions, see the *Content validation* topic.

Is SOAP Fault?

Overview

The **Is SOAP Fault?** is a pre-configured instance of the **Content Validation** filter. For convenience purposes, it contains a pre-configured XPath expression that simply checks for the <soap:Fault> element. The XPath expression is:

```
/soap:Envelope/soap:Body/soap:Fault
```

Configuration

This filter should not need to be configured since its sole purpose is to check to make sure if the message is a SOAP Fault. However, you can change the XPath expression if necessary. For more information on configuring XPath expressions, see the *Content validation* topic.

HTTP header validation

Overview

The API Gateway Explorer can check HTTP header values for threatening content. This ensures that only properly configured name-value pairs appear in the HTTP request headers. *Regular expressions* are used to test HTTP header values. This enables you to make decisions on what to do with the message (for example, if the HTTP header value is `X`, route to service `X`).

You can configure the following sections on the **Validate HTTP Headers** screen:

- **Enter Regular Expression**:
  HTTP header values can be checked using regular expressions. You can select regular expressions from the global **White list** or enter them manually. For example, if you know that an HTTP header must have a value of `ABCD`, a regular expression of `^ABCD$` is an exact match test.
- **Enter Threatening Content Regular Expression**:
  You can select threatening content regular expressions from the global **Black list** to run against all HTTP headers in the message. These regular expressions identify common attack signatures (for example, SQL injection attacks).

You can configure the global **White list** and **Black list** libraries of regular expressions under the **Libraries** node in the Policy Studio tree.

Configure HTTP header regular expressions

The **Enter Regular Expression** table displays the list of configured HTTP header names together with the **White list** of regular expressions that restrict their values. For this filter to run successfully, *all* required headers must be present in the request, and *all* must have values matching the configured regular expressions.

The **Name** column shows the name of the HTTP header. The **Regular Expression** column shows the name of the regular expression that the API Gateway Explorer uses to restrict the value of the named HTTP header. A number of common regular expressions are available from the global **White list** library.

Configure a regular expression
You can configure regular expressions by selecting the **Add**, **Edit**, and **Delete** buttons. The **Configure Regular Expression** dialog enables you to add or edit regular expressions to restrict the values of HTTP headers. To configure a regular expression, perform the following steps:

1. Enter the name of the HTTP header in the **Name** field.
2. Select whether this header is **Optional** or **Required** using the appropriate radio button. If it is **Required**, the header *must* be present in the request. If the header is not present, the filter fails. If it is **Optional**, the header does not need to be present for the filter to pass.
3. You can enter the regular expression to restrict the value of the HTTP header manually or select it from the global **White list** library of regular expressions in the **Expression Name** drop-down list. A number of common regular expressions are provided (for example, alphanumeric values, dates, and email addresses).
   You can use selectors representing the values of message attributes to compare the value of an HTTP header with the value contained in a message attribute. Enter the $ character in the **Regular Expression** field to view a list of available attributes. At runtime, the selector is expanded to the corresponding attribute value, and compared to the HTTP header value that you want to check. For more details on selectors, see *Select configuration values at runtime*.
4. You can add a regular expression to the library by selecting the **Add/Edit** button. Enter a **Name** for the expression followed by the **Regular Expression**.

Advanced settings

The **Advanced** section enables you to extract a portion of the header value which is run against the regular expression. The extracted substring can be Base64 decoded if necessary. This section is specifically aimed towards HTTP Basic authentication headers, which consist of the `Basic` prefix (with a trailing space), followed by the Base64-encoded username and password. The following is an example of the HTTP Basic authentication header:

```
Authorization: Basic dXNlcjp1c2Vy
```

The Base64-encoded portion of the header value is what you are interested in running the regular expression against. You can extract this by specifying the string that occurs directly before the substring you want to extract, together with the string that occurs directly after the substring.

To extract the Base64-encoded section of the `Authorization` header above, enter `Basic` (with a trailing space) in the **Start substring** field, and leave the **End substring** field blank to extract the entire remainder of the header value.

Important

You must select the start and end substrings to ensure that the exact substring is extracted. For example, in the HTTP Basic example above, you should enter `Basic` (with a trailing space) in the **Start substring** field, and *not* `Basic` (with no trailing space).

By specifying the correct substrings, you are left with the Base64-encoded header value (`dXNlcjp1c2Vy`). However, you still need to Base64 decode it before you can run a regular expression on it. Make sure to select the **Base64 decode** checkbox. The Base64-decoded header value is `user:user`, which conforms to the standard format of the `Authorization` HTTP header. This is the value that you need to run the regular expression against.

The following example shows an example of an HTTP Digest authentication header:

```
Authorization: Digest username="user", realm="oracle.com", qop="auth",
algorithm="MD5", uri="/editor", nonce="Id-00000109924ff10b-0000000000000091",
nc="1", cnonce="ae122a8b549af2f0915de868abff55bacd7757ca",
response="29224d8f870a62ce4acc48033c9f6863"
```

You can extract single values from the header value. For example, to extract the `realm` field, enter `realm="` (including the " character), in the **Start substring** field and " in the **End substring** field. This leaves you with `oracle.com` to run the regular expression against. In this case, there is no need to Base64 decode the extracted substring.

Note

If both **Start substring** and **End substring** fields are blank, the regular expression is run against the entire header value. Furthermore, if both fields are blank and the **Base64 decode** checkbox is selected, the entire header value is Base64 encoded before the regular expression is run against it.

While the above examples deal specifically with the HTTP authentication headers, the interface is generic enough to enable you to extract a substring from other header values.

Configure threatening content regular expressions

The regular expressions entered in this section guard against the possibility of an HTTP header containing malicious content. The **Enter Threatening Content Regular Expression** table lists the **Black list** of regular expressions to run to ensure that the header values do not contain threatening content.

For example, to guard against an SQL `DELETE` attack, you can write a regular expression to identify SQL syntax and add it to this list. The **Threatening Content Regular Expressions** are listed in a table. *All* of these expressions are run against *all* HTTP header values in an incoming request. If the expression matches *any* of the values, the filter fails.

Important

If any regular expressions are configured in the section called “Configure selector-based regular expressions”, these expressions are run *before* the threatening content regular expressions. For example, if you have already configured a regular expression to extract the Base64-decoded attribute value, the threatening content regular expression is run against this value instead of the attribute value stored in the message.

You can add threatening content regular expressions using the **Add** button. You can edit or remove existing expressions by selecting them in the drop-down list, and clicking the **Edit** or **Delete** button.

You can enter the regular expressions manually or select them from the global **Black list** library of threatening content regular expressions. This library is pre-populated with a number of regular expressions that scan for common attack signatures. These include expressions to guard against common SQL injection-style attacks (for example, SQL `INSERT`, SQL `DELETE`, and so on), buffer overflow attacks (content longer than 1024 characters), and the presence of control characters in attribute values (ASCII control characters).

Enter or select an appropriate regular expression to restrict the value of the specified HTTP header. You can add a regular expression to the library by selecting the **Add/Edit** button. Enter a **Name** for the expression followed by the **Regular Expression**.

Regular expression format

This filter uses the regular expression syntax specified by `java.util.regex.Pattern`. For more details, see http://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html

Schema validation

Overview

API Gateway Explorer can check that XML messages conform to the structure or format expected by the Web service by validating those requests against XML schemas. An XML schema precisely defines the elements and attributes that constitute an instance of an XML document. It also specifies the data types of these elements to ensure that only appropriate data is allowed through to the Web service.

For example, an XML schema might stipulate that all requests to a particular Web service must contain a `<name>` element, which contains at most a ten character string. If the API Gateway Explorer receives a message with an improperly formed `<name>` element, it rejects the message.

You can find the **Schema Validation** filter in the **Content Filtering** category of filters in Policy Studio. Drag and drop the filter on to a policy to perform schema validation.

General settings

Configure the following general settings:

Name:
Enter an appropriate name for the filter.

Selecting the schema

To configure the XML schema to validate messages against, click the **Schema to use** tab. You can select to use either a schema from the WSDL for the current Web service, or a specific schema from the global cache of WSDL and XML schema documents.

Selecting which part of the message to match

To configure which part of the message to validate, click the **Part of message to match** tab.

A portion of the XML message can be extracted using an XPath expression. API Gateway Explorer can then validate this portion against the specified XML schema. For example, you might need to validate only the SOAP Body element of a SOAP message. In this case, enter or select an XPath expression that identifies the SOAP Body element of the message. This portion should then be validated against an XML schema that defines the structure of the SOAP Body element for that particular message.

Click the **Add** or **Edit** buttons to add or edit an XPath expression using the **Enter XPath Expression** dialog. To remove an expression select the expression in the **XPath Expression** field and click the **Delete** button.

You can configure XPath expressions manually or using a wizard. For more details, see the *Configure XPath expressions* topic.

Advanced settings

The following settings are available on the **Advanced** tab:

Allow RPC Schema Validation:
When the **Allow RPC Schema Validation** check box is selected, the filter makes a *best attempt* to validate an RPC-encoded SOAP message. An RPC-encoded message is defined in the WSDL as having an operation with the following characteristics:

- The `style` attribute of the `<soap:operation>` element is set to `document`.
- The `use` attribute of the `<soap:body>` element is set to `rpc`.

For details on the possible values for these attributes, see Section 3.5 [http://www.w3.org/TR/wsdl#_soap:body] of the WSDL specification.

The problem with RPC-encoded SOAP messages in terms of schema validation is that the schema contained in the WSDL file does not necessarily fully define the format of the SOAP message, unlike with `document-literal` style messages. With an RPC-encoded operation, the format of the message can be defined by a combination of the SOAP operation name, WSDL message parts, and schema-defined types. As a result, the schema extracted from a WSDL file might not be able to validate a message.

Another problem with RPC-encoded messages is that type information is included in each element that appears in the SOAP message. For such element definitions to be validated by a schema, the type declarations must be removed, which is precisely what the **Schema Validation** filter does if the check box is selected on this tab. It removes the type declarations and then makes a *best attempt* to validate the message.

However, as explained earlier, if some of the elements in the SOAP message are taken from the WSDL file instead of the schema (for example, when the SOAP operation name in the WSDL file is used as the wrapper element beneath the SOAP Body element instead of a schema-defined type), the schema is not able to validate the message.

Inline MTOM Attachments into Message:
Message Transmission Optimization Mechanism (MTOM) provides a way to send binary data to Web services in standard SOAP messages. MTOM leverages the include mechanism defined by XML Optimized Packaging (XOP), whereby binary data can be sent as a MIME attachment (similar to SOAP with Attachments) to a SOAP message. The binary data can then be referenced from within the SOAP message using the `<xop:Include>` element.

The following SOAP request contains a binary image that has been Base64-encoded so that it can be inserted as the contents of the `<image>` element:

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
 <soap:Body>
  <uploadGraphic xmlns="www.example.org">
   <image>/aWKKapGGyQ=</image>
  </uploadGraphic>
 </soap:Body>
</soap:Envelope>
```

When this message is converted to an MTOM message by API Gateway the Base64-encoded content from the `<image>` element is replaced with an `<xop:Include>` element. This contains a reference to a newly created MIME part that contains the binary content. The following request shows the resulting MTOM message:

```
POST /services/uploadImages HTTP/1.1
Host: API Gateway Explorer
Content-Type: Multipart/Related;boundary=MIME_boundary;
    type="application/xop+xml";
    start="<mymessage.xml@example.org>";
    start-info="text/xml"

--MIME_boundary
Content-Type: application/xop+xml;
    charset=UTF-8;
    type="text/xml"
Content-Transfer-Encoding: 8bit
Content-ID: <mymessage.xml@example.org>

<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
 <soap:Body>
  <uploadGraphic xmlns="www.example.org">
   <image>
     <xop:Include xmlns:xop='http://www.w3.org/2004/08/xop/include'
```

```
          href='cid:http://example.org/myimage.gif'/>
   </image>
  </uploadGraphic>
 </soap:Body>
</soap:Envelope>

--MIME_boundary
Content-Type: image/gif
Content-Transfer-Encoding: binary
Content-ID: <http://example.org/myimage.gif>

// binary octets for image

--MIME_boundary
```

When attempting to validate the MTOM message with an XML schema, it is crucial that you are aware of the format of the `<image>` element. Will it contain the Base64-encoded binary data, or will it contain the `<xop:include>` element with a reference to a MIME part?

For example, the XML schema definition for an `<image>` element might look as follows:

```
<xsd:element name="image" maxOccurs="1" minOccurs="1"
                type="xsd:base64Binary"
                xmime:expectedContentTypes="*/*"
                xsi:schemaLocation="http://www.w3.org/2005/05/xmlmime"
                xmlns:xmime="http://www.w3.org/2005/05/xmlmime"/>
```

In this case, the XML schema validator expects the contents of the `<image>` element to be `base64Binary`. However, if the message has been formatted as an MTOM message, the `<image>` element contains a child element, `<xop:Include>` that the schema knows nothing about. This causes the schema validator to report an error and schema validation fails.

To resolve this issue, select the **Inline MTOM Attachments into Message** check box on the **Advanced** tab. At runtime, the schema validator replaces the value of the `<xop:Include>` element with the Base64-encoded contents of the MIME part to which it refers. This means that the message now adheres to the definition of the `<image>` element in the XML schema (the element contains data of type `base64Binary`).

This standard procedure of interpreting XOP messages is described in Section 3.2 Interpreting XOP Packages [http://www.w3.org/TR/2004/CR-xop10-20040826/#interpreting_xop_packages] of the XML-binary Optimized Packaging (XOP) specification.

Reporting schema validation errors

When a schema validation check fails, the validation errors are stored in the `xsd.errors` API Gateway message attribute. You can return an appropriate SOAP Fault to the client by writing out the contents of this message attribute.

For example, you can do this by configuring a **Set Message** filter (for more information, see the *Set message* topic) to write a custom response message back to the client. Place the **Set Message** filter on the failure path of the **Schema Validation** filter. You can enter the following sample SOAP Fault message in the **Set Message** filter. Notice the use of the `${xsd.errors}` message attribute selector in the `<Reason>` element:

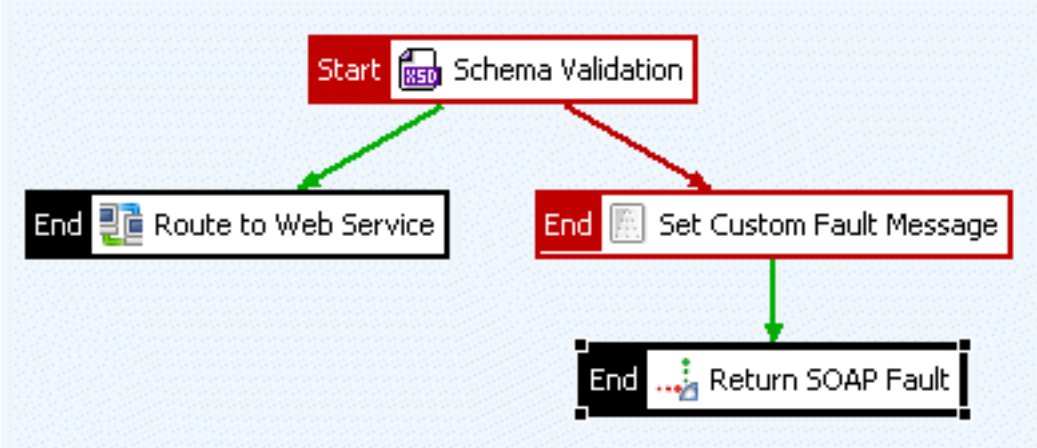
```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
  <env:Body>
    <env:Fault>
      <env:Code>
        <env:Value>env:Receiver</env:Value>
        <env:Subcode>
```

```
          <env:Value xmlns:fault="http://www.Oracle.com/soapfaults">
            fault:MessageBlocked
          </env:Value>
        </env:Subcode>
      </env:Code>
      <env:Reason>
        <env:Text xml:lang="en">
          ${xsd.errors}
        </env:Text>
      </env:Reason>
      <env:Detail xmlns:fault="http://www.Oracle.com/soapfaults"
        fault:type="faultDetails">
      </env:Detail>
    </env:Fault>
  </env:Body>>
</env:Envelope>
```

At runtime, the error reported by the schema validator is set in the message. The following example shows a SOAP Fault containing a typical schema validation error:

```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
  <env:Body>
    <env:Fault>
      <env:Code>
        <env:Value>env:Receiver</env:Value>
        <env:Subcode>
          <env:Value xmlns:fault="http://www.Oracle.com/soapfaults">
            fault:MessageBlocked
          </env:Value>
        </env:Subcode>
      </env:Code>
      <env:Reason>
        <env:Text xml:lang="en">
          [XSD Error: Unknown element 'id' (line: 2, column: 8)]
        </env:Text>
      </env:Reason>
      <env:Detail xmlns:fault="http://www.Oracle.com/soapfaults"
        fault:type="faultDetails">
      </env:Detail>
    </env:Fault>
  </env:Body>>
</env:Envelope>
```

The following figure shows how to use the **Set Message** filter to return a customized SOAP Fault in a policy. If the **Schema Validation** filter succeeds, the message is routed on to the target Web service. However, if the schema validation fails, the **Set Message** filter (named **Set Custom Fault Message**) is invoked. The filter sets the contents of the `xsd.errors` message attribute (the schema validation errors) to the custom SOAP Fault message as shown in the example error. The **Reflect** filter (named **Return SOAP Fault**) then writes the message back to the client.


Start
Schema Validation
End
Route to Web Service
End
Set Custom Fault Message
End
Return SOAP Fault

Validate selector expression

Overview

The **Validate Selector Expression** filter can use regular expressions to check values specified in selectors (for example, message attributes, Key Property Store (KPS), or environment variables). This enables you to make decisions on what to do with the message at runtime. Filters configured in a policy before the **Validate Selector Expression** filter can generate message attributes and store them in the message. For example, you could use the the **Validate Selector Expression** filter to specify that if the attribute value is X, route the message to service X. For more details on selectors, see *Select configuration values at runtime*.

You can configure the following sections on the **Validate Selector Expression** screen:

- **Enter Regular Expression**:
  You can configure selectors that are checked against a regular expression from the global **White list** library, or against a manually configured expression. This check ensures that the value of the selector is acceptable. For example, if you know that a message attribute-based selector named `${my.test.attribute}` must have a value of `ABCD`, a regular expression of `^ABCD$` is an exact match test.
- **Enter Threatening Content Regular Expression**:
  You can select threatening content regular expressions from the global **Black list** to run against each configured selector. These regular expressions identify common attack signatures (for example, SQL injection attacks, ASCII control characters, XML entity expansion attacks, and so on).

You can configure the global **White list** and **Black list** libraries of regular expressions under the **Libraries** node in the Policy Studio tree.

Configure selector-based regular expressions

The **Enter Regular Expression** table displays the list of configured selectors, together with the **White list** of regular expressions that restrict their values. For this filter to run successfully, *all* configured selector checks must have values matching the configured regular expressions.

The **Selector** column shows the name configured for the selector. The **Regular Expression** column shows the name of the regular expression that the API Gateway Explorer uses to restrict the value of the named selector. A number of common regular expressions are available from the global **White list** library.

Configure a Regular Expression

You can configure regular expressions by clicking **Add**, **Edit**, or **Delete**. The **Configure Regular Expression** dialog enables you to add or edit regular expressions to restrict the values of message attributes. To configure a regular expression, perform the following steps:

1. Enter the selector in the **Selector Expression** field (for example, `${my.test.attribute}`).
2. Select whether this attribute is **Optional** or **Required**. If it is **Required**, the attribute *must* be present in the request. If the attribute is not present, the filter fails. If it is **Optional**, the attribute does not need to be present for the filter to pass.
3. You can enter the regular expression to restrict the value of the attribute manually or select it from the global **White list** library of regular expressions in the **Expression Name** drop-down list. A number of common regular expressions are provided (for example, alphanumeric values, dates, and email addresses).
4. You can add a regular expression to the library by selecting the **Add/Edit** button. Enter a **Name** for the expression followed by the **Regular Expression**.

The **Advanced** section enables you to extract a portion of the attribute value that is run against the selector. The extrac-

ted substring can also be Base64 decoded if necessary.

Threatening content regular expressions

The regular expressions entered in this section guard against message attributes containing malicious content. The **Enter Threatening Content Regular Expression** table lists the **Black list** of regular expressions that are run against all message attributes.

For example, to guard against a SQL `DELETE` attack, you can write a regular expression to identify SQL syntax and add to this list. The **Threatening Content Regular Expressions** are listed in a table. *All* of these expressions are run against *all* message attributes configured in the **Regular Expression** table above. If the expression matches *any* attribute values, the filter fails.

Important

If any regular expressions are configured in the section called “Configure selector-based regular expressions”, these expressions are run *before* the threatening content regular expressions. For example, if you have already configured a regular expression to extract the Base64-decoded attribute value, the threatening content regular expression is run against this value instead of the attribute value stored in the message.

Click **Add** to add threatening content regular expressions. You can edit or remove existing expressions by selecting them in the list, and clicking **Edit** or **Delete**. You can enter regular expressions manually or select them from the global **Black list** library of threatening content regular expressions. This library is pre-populated with regular expressions that scan for common attack signatures. These include expressions to guard against common SQL injection-style attacks (for example, SQL `INSERT`, SQL `DELETE`, and so on), buffer overflow attacks (content longer than 1024 characters), and ASCII control characters in attribute values.

Enter or select an appropriate regular expression to scan all message attributes for threatening content. You can add a regular expression to the library by selecting **Add** or **Edit**. Enter a **Name** for the expression followed by the **Regular Expression**.

Add HTTP header

Overview

The API Gateway Explorer can add HTTP headers to a message as it passes through a policy. It can also set a Base64-encoded value for the header. For example, you can use the **Add HTTP Header** filter to add a message ID to an HTTP header. This message ID can then be forwarded to the destination web service, where messages can be indexed and tracked by their IDs. In this way, you can create a complete *audit trail* of the message from the time it is received by the API Gateway Explorer, until it is processed by the back-end system.

Each message being processed by the API Gateway Explorer is assigned a unique transaction ID, which is stored in the `id` message attribute. You can use the `${id}` selector to represent the value of the unique message ID. At runtime, this selector is expanded to the value of the `id` message attribute. For more details on selectors, see *Select configuration values at runtime*.

Configuration

To configure the **Add HTTP Header** filter, complete the following fields:

Name:
Enter an appropriate name for the filter.

HTTP Header Name:
Enter the name of the HTTP header to add to the message.

HTTP Header Value:
Enter the value of the new HTTP header. You can also enter selectors to represent message attributes. At runtime, the API Gateway Explorer expands the selector to the current value of the corresponding message attribute. For example, the `${id}` selector is replaced by the value of the current message ID. Message attribute selectors have the following syntax:
`${messsage_attribute}`

Override existing header:
Select this setting to override the existing header value. This setting is selected by default.

Note

When overriding an existing header, the header can be an HTTP body related header or a general HTTP header. To override an HTTP body related header (for example, `Content-Type`), you must select the **Override existing header** and **Add header to body** settings.

Base64 Encode:
Select this setting to Base64 encode the HTTP header value. For example, you should use this if the header value is an X.509 certificate.

Add header to body:
Select this option to add the HTTP header to the message body.

Add header to HTTP headers attribute:
Select this option to add the HTTP header to the `http.headers` message attribute.

Set HTTP verb

Overview

You can use the **Set HTTP Verb** filter to explicitly set the HTTP verb in the message that is sent from the API Gateway Explorer. By default, all messages are routed onwards using the HTTP verb that the API Gateway Explorer received in the request from the client. If the message originated from a non-HTTP client (for example, JMS), the messages are routed using the HTTP POST verb.

Configuration

Complete the following fields:

Name:
Enter a name for the filter.

HTTP Verb:
Specify the HTTP verb to use in the message that is routed onwards.

Remove attachments

Overview

You can use the **Remove attachment** filter to remove *all* attachments from either a request or a response message, depending on where the filter is placed in the policy.

Configuration

Enter a name for this filter in the **Name** field.

Set message

Overview

The **Set Message** filter replaces the body of the message. The replacement data can be plain text, HTML, XML, or any other text-based markup.

Configuration

Perform the following steps to configure the **Set Message** filter:

1. Enter a name for this filter in the **Name** field.
2. Specify the content type of the new message body in the **Content-Type** field. For example, if the new message body is HTML markup, enter `text/html` in the **Content-Type** field.
3. Enter the new message body in the **Message Body** text area.

   You can use selectors to ensure that current message attribute values are inserted into the message body at the appropriate places. For more information, see the section called “Example of using selectors in the message body”.

   Alternatively, click **Populate** on the right of the window, and select **From file on disk** to load the message contents from a file, or select **From web service operation** to load the message contents from a web service (WSDL file) that you have already imported into the web service repository.

   You can also insert REST API parameters into the message body. Right-click within the message body at the point where the parameter should be inserted and select **Insert > REST API Parameter**.

Example of using selectors in the message body

You can use selectors representing the values of message attributes in the replacement text to insert message-specific data into the message body. For example, you can insert the authenticated user's ID into a `<Username>` element by using a `${authentication.subject.id}` selector as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
 <soap:Header>
  <Username>${authentication.subject.id}</Username>
 </soap:Header>
 <soap:Body>
  <getQuote xmlns="oracle.com">
   <ticker>ORM.L</ticker>
  </getQuote>
 </soap:Body>
</soap:Envelope>
```

Assuming the user authenticated successfully to the API Gateway Explorer, the message body is set as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
 <soap:Header>
  <Username>oracle</Username>
 </soap:Header>
 <soap:Body>
  <getQuote xmlns="oracle.com">
   <ticker>ORM.L</ticker>
  </getQuote>
 </soap:Body>
```

```
</soap:Envelope>
```

For more details on selectors, see *Select configuration values at runtime*.

XML decryption

Overview

The **XML-Decryption** filter is responsible for decrypting data in XML messages based on the settings configured in the **XML-Decryption Settings** filter.

The **XML-Decryption Settings** filter generates the `decryption.properties` message attribute based on configuration settings. The **XML-Decryption** filter uses these properties to perform the decryption of the data.

Configuration

Enter an appropriate name for the filter in the **Name** field.

Auto-generation using the XML decryption wizard

Because the **XML-Decryption** filter must always be paired with an **XML-Decryption Settings** filter, the Policy Studio provides a wizard that can generate both of these filters at the same time. To use the wizard, right-click a policy node under the **Policies** node in the Policy Studio tree, and select **XML Decryption Settings**.

Configure the fields on the **XML Decryption Settings** dialog as explained in the *XML decryption settings* topic. When finished, an **XML-Decryption Settings** filter is created along with an **XML-Decryption** filter.

XML decryption settings

Overview

The API Gateway Explorer can decrypt an XML encrypted message on behalf of its intended recipients. XML Encryption is a W3C standard that enables data to be encrypted and decrypted at the application layer of the OSI stack, thus ensuring complete end-to-end confidentiality of data.

You should use the **XML-Decryption Settings** in conjunction with the **XML-Decryption** filter, which performs the decryption. The **XML-Decryption Settings** generates the `decryption.properties` message attribute, which is required by the **XML-Decryption** filter.

Important

The output of a successfully executed decryption filter is the original unencrypted message. Depending on whether the **Remove EncryptedKey used in decryption** has been enabled, all information relating to the encryption key can be removed from the message. For more details, see Options section.

XML encryption overview

XML encryption facilitates the secure transmission of XML documents between two application endpoints. Whereas traditional transport-level encryption schemes, such as SSL and TLS, can only offer point-to-point security, XML encryption guarantees complete end-to-end security. Encryption takes place at the application-layer and so the encrypted data can be encapsulated in the message itself. The encrypted data can therefore remain encrypted as it travels along its path to the target Web service. Furthermore, the data is encrypted such that only its intended recipients can decrypt it.

To understand how the API Gateway Explorer decrypts XML encrypted messages, you should first examine the format of an XML encryption block. The following example shows a SOAP message containing information about Oracle:

```
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
 <s:Body>
  <getCompanyInfo xmlns="www.oracle.com">
   <name>Company</name>
   <description>XML Security Company</description>
  </getCompanyInfo>
 </s:Body>
</s:Envelope>
```

After encrypting the SOAP Body, the message is as follows:

```
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
 <s:Header>
  <Security xmlns="http://schemas.xmlsoap.org/ws/2003/06/secext" s:actor="Enc">
   <!-- Encapsulates the recipient's key details -->
   <enc:EncryptedKey xmlns:enc="http://www.w3.org/2001/04/xmlenc#"
           Id="00004190E5D1-7529AA14" MimeType="text/xml">
    <enc:EncryptionMethod Algorithm="http://www.w3.org/2001/04xmlenc#rsa-1_5">
     <enc:KeySize>256</enc:KeySize>
    </enc:EncryptionMethod>
    <enc:CipherData>
     <!-- The session key encrypted with the recipient's public key -->
     <enc:CipherValue>
         AAAAAJ/lK ... mrTF8Egg==
     </enc:CipherValue>
    </enc:CipherData>
    <dsig:KeyInfo xmlns:dsig="http://www.w3.org/2000/09/xmldsig#">
     <dsig:KeyName>sample</dsig:KeyName>
```

```
     <dsig:X509Data>
      <!-- The recipient's X.509 certificate -->
      <dsig:X509Certificate>
          MIIEZzCCA0 ... fzmc/YR5gA
      </dsig:X509Certificate>
     </dsig:X509Data>
    </dsig:KeyInfo>
    <enc:CarriedKeyName>Session key</enc:CarriedKeyName>
    <enc:ReferenceList>
     <enc:DataReference URI="#00004190E5D1-5F889C11"/>
    </enc:ReferenceList>
   </enc:EncryptedKey>
  </Security>
 </s:Header>
 <enc:EncryptedData xmlns:enc="http://www.w3.org/2001/04/xmlenc#"
         Id="00004190E5D1-5F889C11" MimeType="text/xml"
         Type="http://www.w3.org/2001/04/xmlenc#Element">
  <enc:EncryptionMethod Algorithm="http://www.w3.org/2001/04xmlenc#aes256-cbc">
   <enc:KeySize>256</enc:KeySize>
  </enc:EncryptionMethod>
  <enc:CipherData>
   <!-- The SOAP Body encrypted with the session key -->
   <enc:CipherValue>
       E2ioF8ib2r ... KJAnrX0GQV
   </enc:CipherValue>
  </enc:CipherData>
  <dsig:KeyInfo xmlns:dsig="http://www.w3.org/2000/09/xmldsig#">
   <dsig:KeyName>Session key</dsig:KeyName>
  </dsig:KeyInfo>
 </enc:EncryptedData>
<s:Envelope>
```

The most important elements are as follows:

- `EncryptedKey`: The `EncryptedKey` element encapsulates all information relevant to the encryption key.
- `EncryptionMethod`: The `Algorithm` attribute specifies the algorithm that is used to encrypt the data. The message data (`EncryptedData`) is encrypted using the Advanced Encryption Standard (AES) *symmetric cipher*, but the session key (`EncryptedKey`) is encrypted with the RSA *asymmetric* algorithm.
- `CipherValue`: The value of the encrypted data. The contents of the `CipherValue` element are always Base64 encoded.
- `KeyInfo`: Contains information about the recipient and his encryption key, such as the key name, X.509 certificate, and Common Name.
- `ReferenceList`: This element contains a list of references to encrypted elements in the message. The `ReferenceList` contains a `DataReference` element for each encrypted element, where the value of a URI attribute points to the `Id` of the encrypted element. In the previous example, you can see that the `DataReference` URI attribute contains the value `#00004190E5D1-5F889C11`, which corresponds with the `Id` of the `EncryptedData` element.
- `EncryptedData`: The XML element(s) or content that has been encrypted. In this case, the SOAP `Body` element has been encrypted, and so the `EncryptedData` block has replaced the SOAP `Body` element.

Now that you have seen how encrypted data can be encapsulated in an XML message, it is important to discuss how this data gets encrypted in the first place. When you understand how data is encrypted, the fields that must be configured to decrypt this data become easier to understand.

When a message is encrypted, only the intended recipient(s) of the message can decrypt it. By encrypting the message with the recipient's public key, the sender can be guaranteed that only the intended recipient can decrypt the message using his private key, to which he has sole access. This is the basic principle behind *asymmetric cryptography*.

In practice, however, encrypting and decrypting data with a public-private key pair is notoriously CPU-intensive and time consuming. Because of this, asymmetric cryptography is seldom used to encrypt large amounts of data. The following steps exemplify a more typical encryption process:

1. The sender generates a one-time *symmetric* (or session) key which is used to encrypt the data. Symmetric key encryption is much faster than asymmetric encryption and is far more efficient with large amounts of data.
2. The sender encrypts the data with the symmetric key. This same key can then be used to decrypt the data. It is therefore crucial that only the intended recipient can access the symmetric key and consequently decrypt the data.
3. To ensure that nobody else can decrypt the data, the symmetric key is encrypted with the recipient's *public key*.
4. The data (encrypted with the symmetric key) and session key (encrypted with the recipient's public key) are then sent together to the intended recipient.
5. When the recipient receives the message he, decrypts the encrypted session key using his *private key*. Because the recipient is the only one with access to the private key, he is the only one who can decrypt the encrypted session key.
6. Armed with the decrypted session key, the recipient can decrypt the encrypted data into its original plaintext form.

Now that you understand how XML Encryption works, it is now time to learn how to configure the API Gateway Explorer to decrypt XML encrypted messages. The following sections describe how to configure the **XML Decryption Settings** filter to decrypt encrypted XML data.

Nodes to decrypt

An XML message may contain several `EncryptedData` blocks. The **Node(s) to Decrypt** section enables you to specify which encryption blocks are to be decrypted. There are two available options:

- Decrypt All Encrypted Nodes
- Use XPath to Select Encrypted Nodes

Decrypt All:
The API Gateway Explorer attempts to decrypt *all* `EncryptedData` blocks contained in the message.

Use XPath:
This option enables the administrator to explicitly choose the `EncryptedData` block that the API Gateway Explorer should decrypt.

For example, the following skeleton SOAP message contains two `EncryptedData` blocks:

```
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
 <s:Header>
    ...
 <s:Header>
 <s:Body>
  <!-- 1st EncryptedData block -->
  <e:EncryptedData xmlns:e="http://www.w3.org/2001/04/xmlenc#"
          Encoding="iso-8859-1" Id="ENC_1" MimeType="text/xml"
          Type="http://www.w3.org/2001/04/xmlenc#Element">
     ...
  </e:EncryptedData>
  <!-- 2nd EncryptedData block -->
  <e:EncryptedData xmlns:e="http://www.w3.org/2001/04/xmlenc#"
          Encoding="iso-8859-1" Id="ENC_2" MimeType="text/xml"
          Type="http://www.w3.org/2001/04/xmlenc#Element">
     ...
  </e:EncryptedData>
 </s:Body>
</s:Envelope>
```

The `EncryptedData` blocks are selected using XPath. You can use the following XPath expressions to select the respective `EncryptedData` blocks:

EncryptedData Block	XPath Expression
1st	`//enc:EncryptedData[@Id='ENC_1']`
2nd	`//enc:EncryptedData[@Id='ENC_2']`

Click the **Add**, **Edit**, or **Delete** buttons to add, edit, or remove an XPath expression.

Decryption key

The **Decryption Key** section enables you to specify the key to use to decrypt the encrypted nodes. As discussed in the section called "XML encryption overview", data encrypted with a public key can only be decrypted with the corresponding private key. The **Decryption Key** settings enable you to specify the private (decryption) key from the `<KeyInfo>` element of the XML Encryption block, or the certificate stored in the Oracle message attribute can be used to lookup the private key of the intended recipient of the encrypted data in the Certificate Store.

Find via KeyInfo in Message:
Select this option if you wish to determine the decryption key to use from the `KeyInfo` section of the `EncryptedKey` block. The `KeyInfo` section contains a reference to the public key used to encrypt the data. You can use this `KeyInfo` section reference to find the relevant private key (from the Oracle Certificate Store) to use to decrypt the data.

Find via certificate from Selector Expression:
Select this option if you do not wish to use the `KeyInfo` section in the message. Enter a selector expression that contains a certificate, (for example, `${certificate}`) whose corresponding private key is stored in the Oracle Certificate Store . Using a selector enables settings to be evaluated and expanded at runtime based on metadata (for example, in a message attribute, a Key Property Store (KPS), or environment variable). For more details, see *Select configuration values at runtime*.

Extract nodes from Selector Expression:
Specify whether to extract nodes from a specified selector expression (for example, `${node.list}`). This setting is not selected by default.

Typically, a **Find Certificate** filter is used in a policy to locate an appropriate certificate and store it in the `certificate` message attribute. When the certificate has been stored in this attribute, the **XML Decryption Settings** filter can use this certificate to lookup the Certificate Store for a corresponding private key for the public key stored in the certificate. To do this, select the `certificate` attribute from the drop-down list.

Options

The following configuration options are available in the **Options** section:

Fail if no encrypted data found:
If this option is selected, the filter fails if no `<EncryptedData>` elements are found within the message.

Remove the EncryptedKey used in decryption:
Select this option to remove information relating to the decryption key from the message. When this option is selected, the `<EncryptedKey>` block is removed from the message.

Important

In cases where the `<EncryptedKey>` block has been included in the `<EncryptedData>` block, it is removed regardless of whether this setting has been selected.

Default Derived Key Label:
If the API Gateway Explorer consumes a `<DerivedKeyToken>`, the default value entered is used to recreate the derived key that is used to decrypt the encrypted data.

Algorithm Suite Required:
Select the WS-Security Policy *Algorithm Suite* that must have been used when encrypting the message. This check ensures that the appropriate algorithms were used to encrypt the message.

Auto-generation using the XML decryption wizard

Because the **XML-Decryption Settings** filter must always be paired with an **XML-Decryption** filter, it makes sense to have a wizard that can generate both of these filters at the same time. To use the wizard, right-click the name of the policy in the tree view of the Policy Studio, and select the **XML Decryption Settings** menu option.

Configure the fields on the **XML Decryption Settings** dialog as explained in the previous sections. When finished, an **XML-Decryption Settings** filter is created along with an **XML-Decryption** filter.

XML encryption

Overview

The **XML-Encryption** filter is responsible for encrypting parts of XML messages based on the settings configured in the **XML-Encryption Settings** filter.

The **XML-Encryption Settings** filter generates the `encryption.properties` message attribute based on configuration settings. The **XML-Encryption** filter uses these properties to perform the encryption of the data.

Configuration

Enter a suitable name for the filter in the **Name** field.

Auto-generation using the XML encryption settings wizard

Because the **XML-Encryption** filter must always be used in conjunction with the **XML-Encryption Settings** and **Find Certificate** filters, the Policy Studio provides a wizard that can generate these three filters at the same time. To use this wizard, right-click a policy node under the **Policies** node in the Policy Studio tree, and select the **XML Encryption Settings** menu option.

For more information on how to configure the **XML Encryption Settings Wizard** see the *XML Encryption Wizard* topic.

XML encryption settings

Overview

The API Gateway Explorer can XML encrypt an XML message so that only certain specified recipients can decrypt the message. XML encryption is a W3C standard that enables data to be encrypted and decrypted at the application layer of the OSI stack, thus ensuring complete end-to-end confidentiality of data.

The **XML-Encryption Settings** should be used in conjunction with the **XML-Encryption** filter, which performs the encryption. The **XML-Encryption Settings** generates the `encryption.properties` message attribute, which is required by the **XML-Encryption** filter.

XML encryption overview

XML encryption facilitates the secure transmission of XML documents between two application endpoints. Whereas traditional transport-level encryption schemes, such as SSL and TLS, can only offer point-to-point security, XML encryption guarantees complete end-to-end security. Encryption takes place at the application-layer, and so the encrypted data can be encapsulated in the message itself. The encrypted data can therefore remain encrypted as it travels along its path to the target Web service.

Before explaining how to configure the API Gateway Explorer to encrypt XML messages, it is useful to examine an XML encrypted message. The following example shows a SOAP message containing information about Oracle:

```
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
 <s:Body>
  <getCompanyInfo xmlns="http://www.oracle.com">
   <name>Company</name>
   <description>XML Security Company</description>
  </getCompanyInfo>
 </s:Body>
</s:Envelope>
```

After encrypting the SOAP Body, the message is as follows:

```
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
 <s:Header>
  <Security xmlns="http://schemas.xmlsoap.org/ws/2003/06/secext" s:actor="Enc">
   <!-- Encapsulates the recipient's key details -->
   <enc:EncryptedKey xmlns:enc="http://www.w3.org/2001/04/xmlenc#"
           Id="00004190E5D1-7529AA14" MimeType="text/xml">
    <enc:EncryptionMethod Algorithm="http://www.w3.org/2001/04xmlenc#rsa-1_5">
     <enc:KeySize>256</enc:KeySize>
    </enc:EncryptionMethod>
    <enc:CipherData>
     <!-- The session key encrypted with the recipient's public key -->
     <enc:CipherValue>
         AAAAAJ/lK ... mrTF8Egg==
     </enc:CipherValue>
    </enc:CipherData>
    <dsig:KeyInfo xmlns:dsig="http://www.w3.org/2000/09/xmldsig#">
     <dsig:KeyName>sample</dsig:KeyName>
     <dsig:X509Data>
      <!-- The recipient's X.509 certificate -->
      <dsig:X509Certificate>
          MIIEZzCCA0 ... fzmc/YR5gA
      </dsig:X509Certificate>
     </dsig:X509Data>
    </dsig:KeyInfo>
    <enc:CarriedKeyName>Session key</enc:CarriedKeyName>
```

```
    <enc:ReferenceList>
     <enc:DataReference URI="#00004190E5D1-5F889C11"/>
    </enc:ReferenceList>
   </enc:EncryptedKey>
  </Security>
 </s:Header>
 <enc:EncryptedData xmlns:enc="http://www.w3.org/2001/04/xmlenc#"
         Id="00004190E5D1-5F889C11" MimeType="text/xml"
         Type="http://www.w3.org/2001/04/xmlenc#Element">
  <enc:EncryptionMethod Algorithm="http://www.w3.org/2001/04xmlenc#aes256-cbc">
   <enc:KeySize>256</enc:KeySize>
  </enc:EncryptionMethod>
  <enc:CipherData>
   <!-- The SOAP Body encrypted with the session key -->
   <enc:CipherValue>
       E2ioF8ib2r ... KJAnrX0GQV
   </enc:CipherValue>
  </enc:CipherData>
  <dsig:KeyInfo xmlns:dsig="http://www.w3.org/2000/09/xmldsig#">
   <dsig:KeyName>Session key</dsig:KeyName>
  </dsig:KeyInfo>
 </enc:EncryptedData>
<s:Envelope>
```

The most important elements are as follows:

- `EncryptedKey`:
  The `EncryptedKey` element encapsulates all information relevant to the encryption key.
- `EncryptionMethod`:
  The `Algorithm` attribute specifies the algorithm used to encrypt the data. The message data (`EncryptedData`) is encrypted using the Advanced Encryption Standard (AES) *symmetric cipher*, but the session key (`EncryptedKey`) is encrypted with the RSA *asymmetric* algorithm.
- `CipherValue`:
  The value of the encrypted data. The contents of the `CipherValue` element are always Base64 encoded.
- `DigestValue`:
  Contains the Base64-encoded message-digest.
- `KeyInfo`:
  Contains information about the recipient and his encryption key, such as the key name, X.509 certificate, and Common Name.
- `ReferenceList`: This element contains a list of references to encrypted elements in the message. It contains a `DataReference` element for each encrypted element, where the value of a URI attribute points to the `Id` of the encrypted element. In the previous example, the `DataReference` URI attribute contains the value `#00004190E5D1-5F889C11`, which corresponds with the `Id` of the `EncryptedData` element.
- `EncryptedData`:
  The XML elements or content that has been encrypted. In this case, the SOAP `Body` element has been encrypted, and so the `EncryptedData` block has replaced the SOAP `Body` element.

Now that you have seen how encrypted data can be encapsulated in an XML message, it is important to discuss how the data is encrypted. When a message is encrypted, it is encrypted in such a manner that only the intended recipients of the message can decrypt it. By encrypting the message with the recipient public key, the sender can be guaranteed that only the intended recipient can decrypt the message using his private key, to which he has sole access. This is the basic principle behind *asymmetric cryptography*.

In practice, however, encrypting and decrypting data with a public-private key pair is a notoriously CPU-intensive and time consuming affair. Because of this, asymmetric cryptography is seldom used to encrypt large amounts of data. The following steps show a more typical encryption process:

1. The sender generates a one-time *symmetric* (or session) key which is used to encrypt the data. Symmetric key encryption is much faster than asymmetric encryption, and is far more efficient with large amounts of data.
2. The sender encrypts the data with the symmetric key. This same key can then be used to decrypt the data. It is therefore crucial that only the intended recipient can access the symmetric key and consequently decrypt the data.
3. To ensure that nobody else can decrypt the data, the symmetric key is encrypted with the recipient's *public key*.
4. The data (encrypted with the symmetric key), and session key (encrypted with the recipient's public key), are then sent together to the intended recipient.
5. When the recipient receives the message, he decrypts the encrypted session key using his *private key*. Because the recipient is the only one with access to the private key, only he can decrypt the encrypted session key.
6. Armed with the decrypted session key, the recipient can decrypt the encrypted data into its original plaintext form.

Now that you understand the structure and mechanics of XML Encryption, you can configure the API Gateway Explorer to encrypt egress XML messages. The next section describes how to configure the tabs on the **XML Encryption Settings** screen.

Encryption key settings

The settings on the **Encryption Key** tab determine the key to use to encrypt the message, and how this key is referred to in the encrypted data. The following configuration options are available:

Important

A symmetric key is used to encrypt the data. This symmetric key is then encrypted (asymmetrically) with the recipient's public key. In this way, only the recipient can decrypt the symmetric encryption key with its private key. When the recipient has access to the unencrypted encryption key, it can decrypt the data.

Generate Encryption Key:
Select this option to generate a symmetric key to encrypt the data with.

Encryption Key from Selector Expression:
If you have already used a symmetric key in a previous filter (for example, a **Sign Message** filter), you can reuse that key to encrypt data by selecting this option and specifying a selector expression to obtain the key (for example, `${symmetric.key}`). Using a selector enables settings to be evaluated and expanded at runtime based on metadata (for example, in a message attribute, a Key Property Store (KPS), or environment variable). For more details, see *Select configuration values at runtime*.

Include Encryption Key in Message:
Select this option if you want to include the encryption key in the message. The encryption key is encrypted for the recipient so that only the recipient can access the encryption key. You may choose not to include the symmetric key in the message if the API Gateway Explorer and recipient have agreed on the symmetric encryption key using some other means.

Specify Method of Associating the Encryption Key with the Encrypted Data:
This section enables you to configure the method by which the encrypted data references the key used to encrypt it. The following options are available:

- ***Point to Encryption Key with Security Token Reference:***
  This option creates a `<SecruityTokenReference>` in the `<EncryptedData>` that points to an `<EncryptedKey>`.
- ***Embed Symmetric Key Inside Encrypted Data:***
  Place the `<xenc:EncryptedKey>` inside the `<xenc:EncryptedData>` element.
- ***Specify Encryption Key via Carried Keyname:***
  Place the encrypted key's carried keyname inside the `<dsig:KeyInfo>`/ `<dsig:KeyName>` of the `<xenc:EncryptedData>`.

- ***Specify Encryption Key via Retrieval Method:***
  Refer to a symmetric key via a retrieval method reference from the <xenc:EncryptedData>.
- ***Symmetric Key Refers to Encrypted Data:***
  The symmetric key refers to <xenc:EncryptedData> using a reference list.

Use Derived Key:
Select this option if you want to derive a key from the symmetric key configured above to encrypt the data. The <enc:EncryptedData> has a <wsse:SecurityTokenReference> to the <wssc:DerivedKeyToken>. The <wssc:DerivedKeyToken> refers to the <enc:EncryptedKey>. Both <wssc:DerivedKeyToken> and <enc:EncryptedKey> are placed inside a <wsse:Security> element.

Key info settings

The **Key Info** tab configures the content of the <KeyInfo> section of the generated <EncryptedData> block. Configure the following fields on this tab:

Do Not Include KeyInfo Section:
This option enables you to omit all information about the certificate that contains the public key that was used to encrypt the data from the <EncryptedData> block. In other words, the <KeyInfo> element is omitted from the <EncryptedData> block. This is useful where a downstream Web service uses an alternative method to decide what key to use to decrypt the message. In such cases, adding certificate information to the message may be regarded as an unnecessary overhead.

Include Certificate:
This is the default option, which places the certificate that contains the encryption key inside the <EncryptedData>. The following example, shows an example of a <KeyInfo> that has been produced using this option:

```
<enc:EncryptedData xmlns:enc="http://www.w3.org/2001/04/xmlenc#">
 <dsig:KeyInfo>
  <dsig:X509Data>
   <dsig:X509SubjectName>CN=Sample...</dsig:X509SubjectName>
   <dsig:X509Certificate>
       MIIEZDCCA0yg
       ....
       RNp9aKD1fEQgJ
   </dsig:X509Certificate>
  </dsig:X509Data>
 </dsig:KeyInfo>
</enc:EncryptedData>
```

Expand Public Key:
The details of the public key used to encrypt the data are inserted into a <KeyValue> block. The <KeyValue> block is only inserted when this option is selected.

```
<enc:EncryptedData xmlns:enc="http://www.w3.org/2001/04/xmlenc#">
   ...
 <dsig:KeyInfo>
  <dsig:X509Data>
   <dsig:X509SubjectName>CN=Sample...</dsig:X509SubjectName>
   <dsig:X509Certificate>
      MIIE ....... EQgJ
   </dsig:X509Certificate>
  </dsig:X509Data>
  <dsig:KeyValue>
   <dsig:RSAKeyValue>
    <dsig:Modulus>
       AMfb2tT53GmMiD
       ...
       NmrNht7iy18=
    </dsig:Modulus>
```

```
    <dsig:Exponent>AQAB</dsig:Exponent>
   </dsig:RSAKeyValue>
  </dsig:KeyValue>
 </dsig:KeyInfo>
</enc:EncryptedData>
```

Include Distinguished Name:
If this checkbox is selected, the Distinguished Name of the certificate that contains the public key used to encrypt the data is inserted in an <X509SubjectName> element as shown in the following example:

```
<enc:EncryptedData xmlns:enc="http://www.w3.org/2001/04/xmlenc#">
  ...
 <dsig:KeyInfo>
  <dsig:X509Data>
   <dsig:X509SubjectName>CN=Sample,C=IE...</dsig:X509SubjectName>
   <dsig:X509Certificate>
       MIIEZDCCA0yg
       ....
       RNp9aKD1fEQgJ
   </dsig:X509Certificate>
  </dsig:X509Data>
 </dsig:KeyInfo>
</enc:EncryptedData>
```

Include Key Name:
This option enables you insert a key identifier, or <KeyName>, to allow the recipient to identify the key to use to decrypt the data. Enter an appropriate value for the <KeyName> in the **Value** field. Typical values include Distinguished Names (DName) from X.509 certificates, key IDs, or email addresses. Specify whether the specified value is a **Text value** of a **Distinguished name attribute** by selecting the appropriate radio button.

```
<enc:EncryptedData xmlns:enc="http://www.w3.org/2001/04/xmlenc#">
   ...
 <dsig:KeyInfo>
  <dsig:KeyName>test@oracle.com</dsig:KeyName>
 </dsig:KeyInfo>
</enc:EncryptedData>
```

Put Certificate in an Attachment:
The API Gateway Explorer supports SOAP messages with attachments. By selecting this option, you can save the certificate containing the encryption key to the file specified in the input field. This file can then be sent along with the SOAP message as a SOAP attachment.

From previous examples, it is clear that the user's certificate is usually placed inside a <KeyInfo> element. However, in this example, the certificate is contained in an attachment, and not in the <EncryptedData>. Clearly, you need a way to reference the certificate from the <EncryptedData> block, so that the recipient can determine what key it should use to decrypt the data. This is the role of the <SecurityTokenReference> block.

The <SecurityTokenReference> block provides a generic mechanism for applications to retrieve security tokens in cases where these tokens are not contained in the SOAP message. The name of the security token is specified in the URI attribute of the <Reference> element.

```
<enc:EncryptedData xmlns:enc="http://www.w3.org/2001/04/xmlenc#">
   ...
 <dsig:KeyInfo>
  <wsse:SecurityTokenReference xmlns:wsse="http://schemas.xmlsoap.org/ws/...">
   <wsse:Reference URI="c:\myCertificate.txt"/>
  </wsse:SecurityTokenReference>
 </dsig:KeyInfo>
</enc:EncryptedData>
```

When the message is sent, the certificate attachment is given a `Content-Id` corresponding to the `URI` attribute of the `<Reference>` element. The following example shows the wire format of the complete multipart MIME SOAP message. It should help illustrate how the `<Reference>` element refers to the `Content-ID` of the attachment:

```
POST /adoWebSvc.asmx HTTP/1.0
Content-Length: 3790
User-Agent: API Gateway Explorer
Accept-Language: en
Content-Type: multipart/related; type="text/xml";
              boundary="----=Multipart-SOAP-boundary"

------=Multipart-SOAP-boundary
Content-Id: soap-envelope
Content-Type: text/xml; charset="utf-8";
SOAPAction=getQuote

<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
   ...
 <enc:EncryptedData xmlns:enc="http://www.w3.org/2001/04/xmlenc#">
    ...
  <dsig:KeyInfo>
   <ws:SecurityTokenReference xmlns:ws="http://schemas.xmlsoap.org/ws/...">
    <ws:Reference URI="c:\myCertificate.txt"/>
   </ws:SecurityTokenReference>
  </dsig:KeyInfo>
 </enc:EncryptedData>
    ...
</s:Envelope>

------=Multipart-SOAP-boundary
Content-Id: c:\myCertificate.txt
Content-Type: text/plain; charset="US-ASCII"

MIIEZDCCA0ygAwIBAgIBAzANBgkqhki
....
7uFveG0eL0zBwZ5qwLRNp9aKD1fEQgJ
------=Multipart-SOAP-boundary-
```

Security Token Reference:
A `<wsse:SecurityTokenReference>` element can be used to point to the security token used to encrypt the data. If you wish to use a `<wsse:SecurityTokenReference>`, enable this option, and select a Security Token Reference type from **Reference Type** drop-down list.

The `<wsse:SecurityTokenReference>`, (in the `<dsig:KeyInfo>`), may contain a `<wsse:Embedded>` security token. Alternatively, the `<wsse:SecurityTokenReference>`, (in the `<dsig:KeyInfo>`), may refer to a certificate using a `<dsig:X509Data>`. Select the appropriate button, **Embed** or **Refer**, depending on whether you want to use an embedded security token or a referred one.

If you have configured the `SecurityContextToken (sct)` mechanism from the **Security Token Reference** drop-down list, you can select to use an **Attached SCT** or an **Unattached SCT**. The default option is to use an **Attached SCT**, which should be used in cases where the SCT refers to a security token inside the `<wsse:Security>` header. If the SCT is located outside the `<wsse:Security>` header, you should select the **Unattached SCT** option.

You can make sure to include a `<BinarySecurityToken>` (BST) that contains the certificate (that contains the encryption key) in the message by selecting the **Include BinarySecurityToken** option. The BST is inserted into the WS-Security header regardless of the type of Security Token Reference selected from the dropdown.

Select **Include TokenType** if you want to add the `TokenType` attribute to the `SecurityTokenReference` element.

Important

When using the Kerberos Token Profile standard, and the API Gateway Explorer is acting as the initiator of a secure transaction, it can use Kerberos session keys to encrypt a message. The `KeyInfo` must be configured to use a Security Token Reference with a `ValueType` of `GSS_Kerberosv5_AP_REQ`. In this case, the Kerberos token is contained in a `<BinarySecurityToken>` in the message.

If the API Gateway Explorer is acting as the recipient of a secure transaction, it can also use the Kerberos session keys to encrypt the message returned to the client. However, in this case, the `KeyInfo` must be configured to use a Security Token Reference with `ValueType` of `Kerberosv5_APREQSHA1`. When this is selected, the Kerberos token is not contained in the message. The Security Token Reference contains a SHA1 digest of the original Kerberos token received from the client, which identifies the session keys to the client.

When using the WS-Trust for SPENGO standard, the Kerberos session keys are not used directly to encrypt messages because a security context with an associated symmetric key is negotiated. This symmetric key is shared by both client and service and can be used to encrypt messages on both sides.

Recipient settings

XML Messages can be encrypted for multiple recipients. In such cases, the symmetric encryption key is encrypted with the public key of each intended recipient and added to the message. Each recipient can then decrypt the encryption key with their private key and use it to decrypt the message.

The following SOAP message has been encrypted for 2 recipients (*oracle_1* and *oracle_2*). The encryption key has been encrypted twice: once for *oracle_1* using its public key, and a second time for *oracle_2* using its public key:

Important

The data itself is only encrypted once, while the encryption key must be encrypted for each recipient. For illustration purposes, only those elements relevant to the above discussion have been included in the following XML encrypted message.

```
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
 <s:Header>
  <Security xmlns="http://schemas.xmlsoap.org/ws/2003/06/secext"
      s:actor="Enc Keys">
   <enc:EncryptedKey xmlns:enc="http://www.w3.org/2001/04/xmlenc#"
        Id="0000418BBB61-A692675C" MimeType="text/xml">
    ...
    <enc:CipherData>
     <!-- Enc key encrypted with oracle_1's public key and base64-encoded -->
     <enc:CipherValue>AAAAAExx1A ... vuAhCgMQ==</enc:CipherValue>
    </enc:CipherData>
    <dsig:KeyInfo xmlns:dsig="http://www.w3.org/2000/09/xmldsig#">
     <dsig:KeyName>oracle_1</dsig:KeyName>
    </dsig:KeyInfo>
    <enc:CarriedKeyName>Session key</enc:CarriedKeyName>
     <enc:ReferenceList>
   <enc:DataReference URI="#0000418BBB61-D4495D9B"/>
    </enc:ReferenceList>
   </enc:EncryptedKey>
   <enc:EncryptedKey xmlns:enc="http://www.w3.org/2001/04/xmlenc#"
        Id="#0000418BBB61-D4495D9B" MimeType="text/xml">
    ...
    <enc:CipherData>
     <!-- Enc key encrypted with oracle_2's public key and base64-encoded -->
     <enc:CipherValue>AAAAABZH+U ... MrMEEM/Ps=</enc:CipherValue>
    </enc:CipherData>
```

```
    <dsig:KeyInfo xmlns:dsig="http://www.w3.org/2000/09/xmldsig#">
     <dsig:KeyName>oracle_2</dsig:KeyName>
    </dsig:KeyInfo>
    <enc:CarriedKeyName>Session key</enc:CarriedKeyName>
    <enc:ReferenceList>
   <enc:DataReference URI="#0000418BBB61-D4495D9B"/>
    </enc:ReferenceList>
   </enc:EncryptedKey>
  </Security>
 </s:Header>
 <enc:EncryptedData xmlns:enc="http://www.w3.org/2001/04/xmlenc#"
      Id="0000418BBB61-D4495D9B" MimeType="text/xml"
      Type="http://www.w3.org/2001/04/xmlenc#Element">
  <enc:EncryptionMethod Algorithm="http://www.w3.org/2001/04xmlenc#aes256-cbc">
   <enc:KeySize>256</enc:KeySize>
  </enc:EncryptionMethod>
  <enc:CipherData>
   <!-- SOAP Body encrypted with symmetric enc key and base64-encoded -->
   <enc:CipherValue>WD0TmuMk9 ... GzYFeq8SM=</enc:CipherValue>
  </enc:CipherData>
  <dsig:KeyInfo xmlns:dsig="http://www.w3.org/2000/09/xmldsig#">
   <dsig:KeyName>Session key</dsig:KeyName>
  </dsig:KeyInfo>
 </enc:EncryptedData>
</s:Envelope>
```

There are two <EncryptedKey> elements, one for each recipient. The <CipherValue> element contains the symmetric encryption key encrypted with the recipient's public key. The encrypted symmetric key must be Base64-encoded so that it can be represented as the textual contents of an XML element.

The <EncryptedData> element contains the encrypted data, along with information about the encryption process, including the encryption algorithm used, the size of the encryption key, and the type of data that was encrypted (for example, whether an element or the contents of an element was encrypted).

Click the **Add** button to add a new recipient for which the data will be encrypted. Configure the following fields on the **XML Encryption Recipient** dialog:

Recipient Name:
Enter a name for the recipient. This name can then be selected on the main **Recipients** tab of the filter.

Actor:
The <EncryptedKey> for this recipient is inserted into the specified SOAP actor/role.

Use Key in Message Attribute:
Specify the message attribute that contains the recipient's public key that is used to encrypt the data. By default, the `certificate` attribute is used. Typically, this attribute is populated by the **Find Certificate** filter, which retrieves a certificate from any one of a number of locations, including the Certificate Store, an LDAP directory, HTTP header, or from the message itself.

If you want to encrypt the message for multiple recipients, you must configure multiple **Find Certificate** filters (or some other filter that can retrieve certificates). Each **Find Certificate** filter retrieves a certificate for a single recipient and store it in a unique message attribute.

For example, a **Find Certificate** filter called **Find Certificate for Recipient1** filter could locate Recipient1's certificate from the Certificate Store and store it in a `certificate_recip1` message attribute. You would then configure a second **Find Certificate** filter called **Find Certificate for Recipient2**, which could retrieve Recipient2's certificate from the Certificate Store and store it in a `certificate_recip2` message attribute.

On the **Recipients** tab of the **XML Encryption Settings** filter, you would then configure two recipients. For the first recipient (Recipient1), you would enter `certificate_recip1` as the location of the encryption key, while for the second re-

cipient (Recipient2), you would specify `certificate_recip2` as the location of the encryption key.

Note

If the API Gateway Explorer fails to encrypt the message for any of the recipients configured on the **Recipients** tab, the filter will fail.

What to encrypt settings

The **What to Encrypt** tab is used to identify parts of the message that must be encrypted. Each encrypted part will be replaced by an `<EncryptedData>` block, which contains all information required to decrypt the block.

You can use *any* combination of **Node Locations**, **XPaths**, and the nodes contained in a **Message Attribute** to specify the nodes that are required to be encrypted.

Important

Note the difference between encrypting the element and encrypting the element content. When encrypting the element, the entire element is replaced by the `<EncryptedData>` block. This is not recommended, for example, if you wish to encrypt the SOAP Body because if this element is removed from the SOAP message, the message may no longer be considered a valid SOAP message.

Element encryption is more suitable when encrypting security blocks, (for example, WS-Security Username tokens and SAML assertions) that may appear in a WS-Security header of a SOAP message. In such cases, replacing the element content (for example, a `<UsernameToken>` element) with an `<EncryptedData>` block will not affect the semantics of the WS-Security header.

If you wish to encrypt the SOAP Body, you should use element content encryption, where the children of the element are replaced by the `<EncryptedData>` block. In this way, the message can still be validated against the SOAP schema.

When using **Node Locations** to identify nodes that are to be encrypted, you can configure whether to encrypt the element or the element contents on the **Locate XML Nodes** dialog. To encrypt the element, select the **Encrypt Node** radio button. Alternatively, to encrypt the element contents, select the **Encrypt Node Content** radio button.

If you are using XPath expressions to specify the nodes that are to be signed, be careful not to use an expression that returns a node and all its contents. The **Encrypt Node** and **Encrypt Node Content** options are also available when configuring XPath expressions on the **Enter XPath Expression** dialog.

Advanced settings

The **Advanced** tab on the main **XML-Encryption Settings** screen enables you to configure some of the more complicated settings regarding XML-Encryption. The following settings are available:

Algorithm Suite Tab:
The following fields can be configured on this tab:

Algorithm Suite:
WS-Security Policy defines a number of *algorithm suites* that group together a number of cryptographic algorithms. For example, a given algorithm suite uses specific algorithms for asymmetric encryption, symmetric encryption, asymmetric key wrap, and so on. Therefore, by specifying an algorithm suite, you are effectively selecting a whole suite of cryptographic algorithms to use.

If you want to use a particular WS-Security Policy algorithm suite, you can select it here. The **Encryption Algorithm** and **Key Wrap Algorithm** fields are automatically populated with the corresponding algorithms for that suite.

Encryption Algorithm:

The encryption algorithm selected is used to encrypt the data. The following algorithms are available:

- AES-256
- AES-192
- AES-128
- Triple DES

Key Wrap Algorithm:
The key wrap algorithm selected here is used to wrap (encrypt) the symmetric encryption key with the recipient's public key. The following key wrap algorithms are available:

- KwRsa15
- KwRsaOaep

Settings Tab:
The following advanced settings are available on this tab:

Generate a Reference List in WS-Security Block:
When this option is selected, a `<xenc:ReferenceList>` that holds a reference to all encrypted data elements is generated. The `<xenc:ReferenceList>` element is inserted into the WS-Security block indicated by the specified actor.

Insert Reference List into EncryptedKey:
When this option is selected, a `<xenc:ReferenceList>` that holds a reference to all encrypted data elements is generated. The `<xenc:ReferenceList>` element is inserted into the `<xenc:EncryptedKey>` element.

Layout Type:
Select the WS-SecurityPolicy layout type that you want the generated tokens to adhere to. This includes elements such as the `<EncryptedData>`, `<EncryptedKey>`, `<ReferenceList>`, `<BinarySecurityToken>`, and `<DerivedKeyToken>` tokens, among others.

Fail if no Nodes to Encrypt:
Select this option if you want the filter to fail if any of the nodes specified on the **What to Encrypt** tab are found in the message.

Insert Timestamp:
This option enables you to insert a WS-Security Timestamp as an encryption property.

Indent:
This option enables you to format the inserted `<EncryptedData>` and `<EncryptedKey>` blocks by indenting the elements.

Insert CarriedKeyName for EncryptedKey:
Select this option to insert a `<CarriedKeyName>` element into the generated `<EncryptedKey>` block.

Auto-generation using the XML encryption settings wizard

Because the **XML-Encryption Settings** filter must always be used in conjunction with the **XML-Encryption** and **Find Certificate** filters, the Policy Studio provides a wizard that can generate these three filters at the same time. Right-click a policy under the **Policies** node in the Policy Studio, and select **XML Encryption Settings**.

For more information on how to configure the **XML Encryption Settings Wizard** see the *XML Encryption Wizard* topic.

XML Encryption Wizard

Overview

There are several filters involved in encrypting a message using XML Encryption. These filters are as follows:

Step	Role
1. Select Public Key	Specify the certificate that contains the public key to use in the encryption. The data will be encrypted such that it can only be decrypted with the corresponding private key.
2. XML Encryption Settings	Specify the recipient of the encrypted data, what data to encrypt, what algorithms to use, and other such options that will affect the way the data is encrypted.

The **XML Encryption Wizard** is available by clicking the on the **Encrypt Request** link on the left-hand side of the Classic Mode of the API Gateway Explorer. The next section describes how to configure the wizard.

Configuration

The first step in configuring the **XML Encryption Wizard** is to select the certificate that contains the public key to use to encrypt the data. Once the data has been encrypted with this public key it will only be able to be decrypted using the corresponding private key. Select the relevant certificate from the list of **Certificates in the Trusted Certificate Store**.

After clicking the **Next** button on the first screen of the wizard, the configuration options for the **XML Encryption Settings** filter are displayed. For more information on configuring this filter, please refer to the *XML encryption settings* topic.

Click the **Finish** button to create the XML-Encryption block within the SOAP message.

XML signature generation

Overview

The API Gateway Explorer can sign both SOAP and non-SOAP XML messages. Attachments to the message can also be signed. The resulting XML signature is inserted into the message for consumption by a downstream web service. At the web service, the signature can be used to authenticate the message sender and verify the integrity of the message.

General settings

Configure the following general setting:

Name:
Enter an appropriate name for the filter.

Signing key settings

On the **Signing Key** tab, you can select either a symmetric or an asymmetric key to sign the message content. Select the appropriate radio button and configure the fields on the corresponding tab.

Asymmetric Key

With an asymmetric signature, the signatory's private key (from a public-private key pair) is used to sign the message. The corresponding public key is then used to verify the signature. The following fields are available for configuration on this tab:

Private Key in Certificate Store:
To use a signing key from the certificate store, select **Key in Store**, and click **Signing Key**. Select a certificate that has the required signing key associated with it. The signing key can also be stored on a Hardware Security Module (HSM). For more details, see *Manage certificates and keys*. The *Distinguished Name* of the selected certificate appears in the `X509SubjectName` element of the XML signature as follows:

```
<dsig:X509SubjectName>
       CN=Sample,OU=R&D,O=Company Ltd.,L=Dublin 4,ST=Dublin,C=IE
    </dsig:X509SubjectName>
```

Private Key from Selector Expression:
Alternatively, the signing key might have already have been used by another filter and stored in a message attribute. To reuse this key, select **Private Key from Selector Expression**, and enter the selector expression (for example, `${asymmetric.key}`). Using a selector enables settings to be evaluated and expanded at runtime based on metadata (for example, in a message attribute, Key Property Store (KPS), or environment variable). For more details, see *Select configuration values at runtime*.

Symmetric Key

With a symmetric signature, the same key is used to sign and verify the message. Typically the client generates the symmetric key and uses it to sign the message. The key must then be transmitted to the recipient so that they can verify the signature. It would be unsafe to transmit an unprotected key along with the message so it is usually encrypted (or wrapped) with the recipient's public key. The key can then be decrypted with the recipient's private key and can then be used to verify the signature. The following configuration options are available on this window:

Generate Symmetric Key, and Save in Message Attribute:
If you select this option, the API Gateway Explorer generates a symmetric key, which is included in the message before it is sent to the client. By default, the key is saved in the `symmetric.key` message attribute.

Symmetric Key from Selector Expression:

If a previous filter (for example, a **Sign Message** filter) has already used a symmetric key, you can to reuse this key as proof that the API Gateway Explorer is the holder-of-key entity. Enter the name of the selector expression in the field provided, which defaults to `${symmetric.key}`. Using a selector enables settings to be evaluated and expanded at runtime based on metadata (for example, in a message attribute, a Key Property Store (KPS), or environment variable). For more details, see *Select configuration values at runtime*.

Include Encrypted Symmetric Key in Message:
As described earlier, the symmetric key is typically encrypted for the recipient and included in the message. However, it is possible that the initiator and recipient of the transaction have agreed on a symmetric key using some out-of-bounds mechanism. In this case, it is not necessary to include the key in the message. However, the default option is to include the encrypted symmetric key in the message. The `<KeyInfo>` section of the signature points to the `<EncryptedKey>`.

Encrypt with Key in Store:
Select this option to encrypt the symmetric key with a public key from the certificate store. Click the **Signing Key** button and then select the certificate that contains the public key of the recipient. By encrypting the symmetric key with this public key, you are ensuring that only the recipient that has access to the corresponding private key will be able to decrypt the encrypted symmetric key.

Encrypt with Key from Selector Expression:
You can also use a key stored in a message attribute to encrypt (or wrap) the symmetric key. Select this radio button and enter the selector expression to obtain the public key you want to use to encrypt the symmetric key with. Using a selector enables settings to be evaluated and expanded at runtime based on metadata (for example, in a message attribute, a Key Property Store (KPS), or environment variable). For more details, see *Select configuration values at runtime*.

Use Derived Key:
A `<wssc:DerivedKeyToken>` token can be used to derive a symmetric key from the original symmetric key held in and `<enc:EncryptedKey>`. The derived symmetric key is then used to actually sign the message, as opposed to the original symmetric key. It must be derived again during the verification process using the parameters in the `<wssc:DerivedKeyToken>`. One of these parameters is the symmetric key held in `<enc:EncryptedKey>`. The following example shows the use of a derived key:

```
<enc:EncryptedKey Id="Id-0000010b8b0415dc-0000000000000000">
  <enc:EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5"/>
  <dsig:KeyInfo>
       ...
  </dsig:KeyInfo>
  <enc:CipherData>
</enc:EncryptedKey>

<wssc:DerivedKeyToken wsu:Id="Id-0000010bd2b8eca1-0000000000000017"
      Algorithm="http://schemas.xmlsoap.org/ws/2005/02/sc/dk/p_sha1">
  <wsse:SecurityTokenReference wsu:Id="Id-0000010bd2b8ed5d-0000000000000018">
    <wsse:Reference URI="#Id Id-0000010b8b0415dc-0000000000000000"
    ValueType="..../oasis-wss-soap-message-security-1.1#EncryptedKey"/>
  </wsse:SecurityTokenReference>
  <wssc:Generation>0</wssc:Generation>
  <wssc:Length>32</wssc:Length>
  <wssc:Label>WS-SecureConverstaionWS-SecureConverstaion</wssc:Label>
  <wssc:Nonce>h9TTWKRylCOz87+mc1/7Pg==</wssc:Nonce>
</wssc:DerivedKeyToken>

<dsig:Signature Id="Id-0000010b8b0415dc-0000000000000004">
  <dsig:SignedInfo>
    <dsig:CanonicalizationMethod
          Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
    <dsig:SignatureMethod
        Algorithm="http://www.w3.org/2000/09/xmldsig#hmac-sha1"/>
    <dsig:Reference>...</dsig:Reference>
  </dsig:SignedInfo>
  <dsig:SignatureValue>...dsig:SignatureValue>
  <dsig:KeyInfo>
    <wsse:SecurityTokenReference wsu:Id="Id-0000010b8b0415dc-0000000000000006">
```

```
      <wsse:Reference
          URI="# Id-0000010bd2b8eca1-0000000000000017"
          ValueType="http://schemas.xmlsoap.org/ws/2005/02/sc/dk"/>
    </wsse:SecurityTokenReference>
  </dsig:KeyInfo>
</dsig:Signature>
```

Symmetric Key Length:
This option enables the user to specify the length of the key to use when performing symmetric key signatures. It is important to realize that the longer the key, the stronger the encryption.

Key Info

This tab configures how the `<KeyInfo>` block of the generated XML signature is displayed. Configure the following fields on this tab:

Do Not Include KeyInfo Section:
This option enables you to omit all information about the signatory's certificate from the signature. In other words, the `KeyInfo` element is omitted from the signature. This is useful where a downstream web service uses an alternative method of authenticating the signatory, uses the signature for the sole purpose of verifying the integrity of the message. In such cases, adding certificate information to the message is an unnecessary overhead.

Include Certificate:
This is the default option which places the signatory's certificate inside the XML signature itself. The following example, shows an example of an XML signature using this option:

```
<dsig:Signature xmlns:dsig="http://www.w3.org/2000/09/xmldsig#" id="Sample">
  ...
 <dsig:KeyInfo>
  <dsig:X509Data>
   <dsig:X509SubjectName>CN=Sample...</dsig:X509SubjectName>
   <dsig:X509Certificate>
       MIIEZDCCA0yg
       ....
       RNp9aKD1fEQgJ
   </dsig:X509Certificate>
  </dsig:X509Data>
 </dsig:KeyInfo>
</dsig:Signature>
```

Expand Public Key:
The details of the signatory's public key are inserted into a `KeyValue` block. The `KeyValue` block is only inserted when this option is selected.

```
<dsig:Signature xmlns:dsig="http://www.w3.org/2000/09/xmldsig#" id="Sample">
   ...
 <dsig:KeyInfo>
  <dsig:X509Data>
   <dsig:X509SubjectName>CN=Sample...</dsig:X509SubjectName>
   <dsig:X509Certificate>
      MIIE ....... EQgJ
   </dsig:X509Certificate>
  </dsig:X509Data>
  <dsig:KeyValue>
   <dsig:RSAKeyValue>
    <dsig:Modulus>
       AMfb2tT53GmMiD
       ...
       NmrNht7iy18=
    </dsig:Modulus>
```

```
    <dsig:Exponent>AQAB</dsig:Exponent>
   </dsig:RSAKeyValue>
  </dsig:KeyValue>
 </dsig:KeyInfo>
</dsig:Signature>
```

Include Distinguished Name:
If this check box is selected, the Distinguished Name of the signatory's X.509 certificate is inserted in an `<X509SubjectName>` element as shown in the following example:

```
<dsig:Signature xmlns:dsig="http://www.w3.org/2000/09/xmldsig#" id="Sample">
  ...
 <dsig:KeyInfo>
  <dsig:X509Data>
   <dsig:X509SubjectName>CN=Sample,C=IE...</dsig:X509SubjectName>
   <dsig:X509Certificate>
       MIIEZDCCA0yg
       ....
       RNp9aKD1fEQgJ
   </dsig:X509Certificate>
  </dsig:X509Data>
 </dsig:KeyInfo>
</dsig:Signature>
```

Include Key Name:
This option allows you insert a key identifier, or `KeyName`, to allow the recipient to identify the signatory. Enter an appropriate value for the `KeyName` in the **Value** field. Typical values include Distinguished Names (DName) from X.509 certificates, key IDs, or email addresses. Specify whether the specified value is a **Text value** of a **Distinguished name attribute** by checking the appropriate radio button.

```
<dsig:Signature xmlns:dsig="http://www.w3.org/2000/09/xmldsig#" id="Sample">
   ...
 <dsig:KeyInfo>
  <dsig:KeyName>test@oracle.com</dsig:KeyName>
 </dsig:KeyInfo>
</dsig:Signature>
```

Put Certificate in an Attachment:
The API Gateway Explorer supports SOAP messages with attachments. By selecting this option, you can save the signatory's certificate to the file specified in the input field. This file can then be sent along with the SOAP message as a SOAP attachment.

From previous examples, it is clear that the user's certificate is usually placed inside a `KeyInfo` element. However, in this example, the certificate is actually contained within an attachment, and not within the XML signature itself. To reference the certificate from the XML signature, so that validating applications can process the signature correctly, is the role of the `SecuriyTokenReference` block.

The `SecurityTokenReference` block provides a generic way for applications to retrieve security tokens in cases where these tokens are not contained within the SOAP message. The name of the security token is specified in the `URI` attribute of the `Reference` element.

```
<dsig:Signature xmlns:dsig="http://www.w3.org/2000/09/xmldsig#" id="Sample">
   ...
 <dsig:KeyInfo>
  <wsse:SecurityTokenReference xmlns:wsse="http://schemas.xmlsoap.org/ws/...">
   <wsse:Reference URI="c:\myCertificate.txt"/>
  </wsse:SecurityTokenReference>
```

```
 </dsig:KeyInfo>
</dsig:Signature>
```

When the message is actually sent, the certificate attachment will be given a "Content-Id" corresponding to the `URI` attribute of the `Reference` element. The following example shows what the complete multipart MIME SOAP message looks like as it is sent over the wire. This illustrates how the `Reference` element actually refers to the "Content-ID" of the attachment:

```
POST /adoWebSvc.asmx HTTP/1.0
Content-Length: 3790
User-Agent: API Gateway Explorer
Accept-Language: en
Content-Type: multipart/related; type="text/xml";
              boundary="----=Multipart-SOAP-boundary"

------=Multipart-SOAP-boundary
Content-Id: soap-envelope
Content-Type: text/xml; charset="utf-8";
SOAPAction=getQuote

<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
   ...
 <dsig:Signature xmlns:dsig="http://www.w3.org/2000/09/xmldsig#" id="Sample">
    ...
  <dsig:KeyInfo>
   <ws:SecurityTokenReference xmlns:ws="http://schemas.xmlsoap.org/ws/...">
    <ws:Reference URI="c:\myCertificate.txt"/>
   </ws:SecurityTokenReference>
  </dsig:KeyInfo>
 </dsig:Signature>
    ...
</s:Envelope>

------=Multipart-SOAP-boundary
Content-Id: c:\myCertificate.txt
Content-Type: text/plain; charset="US-ASCII"

MIIEZDCCA0ygAwIBAgIBAzANBgkqhki
....
7uFveG0eL0zBwZ5qwLRNp9aKD1fEQgJ
------=Multipart-SOAP-boundary-
```

Security Token Reference:
A `<wsse:SecurityTokenReference>` element can be used to point to the security token used in the generation of the signature. Select this option to use this element. The type of the reference must be selected from the **Reference Type** field.

The `<wsse:SecurityTokenReference>`, (within the `<dsig:KeyInfo>`), can contain a `<wsse:Embedded>` security token. Alternatively, the `<wsse:SecurityTokenReference>`, (within the `<dsig:KeyInfo>`), can refer to a certificate via a `<dsig:X509Data>`. Select the appropriate button, **Embed** or **Refer**, depending on whether you want to use an embedded security token or a referred one.

You can make sure to include a `<BinarySecurityToken>` (BST) that contains the certificate used to wrap the symmetric key in the message by selecting the **Include BinarySecurityToken** option. The BST is inserted into the WS-Security header regardless of the type of Security Token Reference selected.

Important

When using the Kerberos Token Profile standard and the API Gateway Explorer is acting as the initiator of a secure transaction, it can use Kerberos session keys to sign a message. The `KeyInfo` must be con-

figured to use a Security Token Reference with a `ValueType` of `GSS_Kerberosv5_AP_REQ`. In this case, the Kerberos token is contained in a `<BinarySecurityToken>` in the message.

If the API Gateway Explorer is acting as the recipient of a secure transaction, it can also use the Kerberos session keys to sign the message returned to the client. However, in this case, the `KeyInfo` must be configured to use a Security Token Reference with `ValueType` of `Kerberosv5_APREQSHA1`. When this `ValueType` is selected, the Kerberos token is not contained in the message. The Security Token Reference contains a SHA1 digest of the original Kerberos token received from the client, which identifies the session keys to the client.

Using the WS-Trust for SPENGO standard, the Kerberos session keys are not used directly to sign messages because a security context with an associated symmetric key is negotiated. This symmetric key is shared by both client and service and can be used to sign messages on both sides.

What to sign settings

The **What to Sign** tab is used to identify parts of the message that must be signed. Each signed part will be referenced from within the generated XML signature. You can use *any* combination of **Node Locations**, **XPaths**, **XPath Predicates**, and the nodes contained in a **Message Attribute** to specify what must be signed.

XML Signing Mechanisms
It is important to consider the mechanisms available for referencing signed elements from within an XML signature. For example, With WSU Ids, an Id attribute is inserted into the root element of the nodeset that is to be signed. The XML signature then references this Id to indicate to verifiers of the signature the nodes that were signed. The use of WSU Ids is the default option because these are WS-I compliant.

Alternatively, a generic Id attribute (not bound to the WSU namespace) can be used to dereference the data. The Id attribute is inserted into the top-level element of the nodeset that is to be signed. The generated XML signature can then reference this Id to indicate what nodes were signed. When XPath transforms are used, an XPath expression that points to the root node of the nodeset that is signed will be inserted into the XML signature. When attempting to verify the signature, this XPath expression must be run on the message to retrieve the signed content.

Id Attribute:
Select the Id attribute used to dereference the signed element in the `dsig:Signature`. The available options are as follows:

- *wsu:Id*
  The default option references the signed data using a `wsu:Id` attribute. A `wsu:Id` attribute is inserted into the root node of the signed nodeset. This Id is then referenced in the generated XML signature as an indication of which nodes were signed. For example:

```
<soap:Envelope xmlns:soap="...">
  <soap:Header>
  <wsse:Security xmlns:wsse="...">
    <dsig:Signature xmlns:dsig="..." Id="Id-00000112e2c98df8-0000000000000004">
      <dsig:SignedInfo>
        <dsig:CanonicalizationMethod
            Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
        <dsig:SignatureMethod
            Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
        <dsig:Reference URI="#Id-00000112e2c98df8-0000000000000003">
          <dsig:Transforms>
            <dsig:Transform
                Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
          </dsig:Transforms>
          <dsig:DigestMethod
              Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
          <dsig:DigestValue>xChPoiWJJrrPZkbXN8FPB8S4U7w=</dsig:DigestValue>
        </dsig:Reference>
      </dsig:SignedInfo>
```

```
      <dsig:SignatureValue>KG4N .... /9dw==</dsig:SignatureValue>
      <dsig:KeyInfo Id="Id-00000112e2c98df8-0000000000000005">
        <dsig:X509Data>
          <dsig:X509Certificate>
            MIID ... ZiBQ==
          </dsig:X509Certificate>
        </dsig:X509Data>
      </dsig:KeyInfo>
    </dsig:Signature>
  </wsse:Security>
 </soap:Header>
 <soap:Body xmlns:wsu="..." wsu:Id="Id-00000112e2c98df8-0000000000000003">
  <vs:getProductInfo xmlns:vs="http://ww.oracle.com">
   <vs:Name>API Gateway Explorer</vs:Name>
   <vs:Version>11.1.2.4.0</vs:Version>
  </vs:getProductInfo>
 </s:Body>
</s:Envelope>
```

In the above example, a `wsu:Id` attribute has been inserted into the `<soap:Body>` element. This `wsu:Id` attribute is then referenced by the `URI` attribute of the `<dsig:Reference>` element in the actual signature. When the signature is being verified, the value of the `URI` attribute can be used to locate the nodes that have been signed.

- *Id*
  Select the `Id` option to use generic Ids (not bound to the WSU namespace) to dereference the signed data. Under this schema, the `URI` attribute of the `<Reference>` points at an Id attribute, which is inserted into the top-level node of the signed nodeset. In the following example, the Id specified in the signature matches the Id attribute inserted into the `<Body>` element, indicating that the signature applies to the entire contents of the SOAP body:

```
<soap:Envelope xmlns:soap="....">
  <soap:Header>
    <dsig:Signature xmlns:dsig="...."
                Id="Id-0000011a101b167c-0000000000000013">
      <dsig:SignedInfo>
        <dsig:CanonicalizationMethod
            Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
        <dsig:SignatureMethod
            Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
        <dsig:Reference URI="#Id-0000011a101b167c-0000000000000012">
          <dsig:Transforms>
            <dsig:Transform
                Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
          </dsig:Transforms>
          <dsig:DigestMethod
              Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
          <dsig:DigestValue>JCy0JoyhVZYzmrLrl92nxfr1+zQ=</dsig:DigestValue>
        </dsig:Reference>
      </dsig:SignedInfo>
      <dsig:SignatureValue>......<dsig:SignatureValue>
      <dsig:KeyInfo Id="Id-0000011a101b167c-0000000000000014">
        <dsig:X509Data>
          <dsig:X509Certificate>......</dsig:X509Certificate>
        </dsig:X509Data>
      </dsig:KeyInfo>
    </dsig:Signature>
 </soap:Header>
 <soap:Body Id="Id-0000011a101b167c-0000000000000012">
  <product version="11.1.2.4.0">
    <name>API Gateway Explorer</name>
    <company>oracle</company>
    <description>SOA Security and Management</description>
  </product>
 </soap:Body>
```

```
</soap:Envelope>
```

- *ID*
  Select this option to use generic IDs (not bound to the WSU namespace) to dereference the signed data. Under this schema, the URI attribute of the `Reference` points at an ID attribute, which is inserted into the top-level node of the signed nodeset. In the following example, the URI specified in the Signature Reference node matches the ID attribute inserted into the `Body` element, indicating that the signature applies to the entire contents of the SOAP body:

```
<soap:Envelope xmlns:soap="....">
  <soap:Header>
    <dsig:Signature xmlns:dsig="....">
      <dsig:SignedInfo>
        <dsig:CanonicalizationMethod
           Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
        <dsig:SignatureMethod
           Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
        <dsig:Reference URI="#Id-0000011a101b167c-0000000000000012">
          <dsig:Transforms>
            <dsig:Transform
              Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
          </dsig:Transforms>
          <dsig:DigestMethod
             Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
          <dsig:DigestValue>JCy0JoyhVZYzmrLrl92nxfr1+zQ=</dsig:DigestValue>
        </dsig:Reference>
      </dsig:SignedInfo>
       <dsig:SignatureValue>......<dsig:SignatureValue>
      <dsig:KeyInfo Id="Id-0000011a101b167c-0000000000000014">
        <dsig:X509Data>
          <dsig:X509Certificate>......</dsig:X509Certificate>
        </dsig:X509Data>
      </dsig:KeyInfo>
    </dsig:Signature>
 </soap:Header>
 <soap:Body ID="Id-0000011a101b167c-0000000000000012">
  <product version="11.1.2.4.0">
    <name>API Gateway Explorer</name>
    <company>Oracle</company>
    <description>SOA Security and Management</description>
  </product>
 </soap:Body>
</soap:Envelope>
```

- *xml:id*
  Select this option to use an `xml:id` to dereference the signed data. Under this schema, the URI attribute of the `Reference` points at an `xml:id` attribute, which is inserted into the top-level node of the signed nodeset. In the following example, the URI specified in the Signature Reference node matches the `xml:id` attribute inserted into the `Body` element, indicating that the signature applies to the entire contents of the SOAP body:

```
<soap:Envelope xmlns:soap="....">
  <soap:Header>
    <dsig:Signature xmlns:dsig="...."
                Id="Id-0000011a101b167c-0000000000000013">
      <dsig:SignedInfo>
        <dsig:CanonicalizationMethod
            Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
        <dsig:SignatureMethod
            Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
        <dsig:Reference URI="#Id-0000011a101b167c-0000000000000012">
          <dsig:Transforms>
            <dsig:Transform
                Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
          </dsig:Transforms>
```

```
          <dsig:DigestMethod
              Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
          <dsig:DigestValue>JCy0JoyhVZYzmrLrl92nxfr1+zQ=</dsig:DigestValue>
        </dsig:Reference>
      </dsig:SignedInfo>
      <dsig:SignatureValue>......<dsig:SignatureValue>
      <dsig:KeyInfo Id="Id-0000011a101b167c-0000000000000014">
        <dsig:X509Data>
          <dsig:X509Certificate>......</dsig:X509Certificate>
        </dsig:X509Data>
      </dsig:KeyInfo>
    </dsig:Signature>
 </soap:Header>
 <soap:Body ID="Id-0000011a101b167c-0000000000000012">
  <product version=11.1.2.4.0>
    <name>API Gateway Explorer</name>
    <company>Oracle</company>
    <description>SOA Security and Management</description>
  </product>
 </soap:Body>
</soap:Envelope>
```

- *No id (use with enveloped signature and XPath 'The Entire Document')*
  Select this option to sign the entire document. In this case, the URI attribute on the `Reference` node of the signature is `""`, which means that no id is used to refer to what is being signed. The `""` URI means that the full document is signed. A signature of this type must be an enveloped signature. On the **Advanced** > **Options** tab, select **Create enveloped signature**. To sign the full document, on the **What to Sign** > **XPaths** tab, select the XPath named `The entire document`.

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:soap="....">
    <soap:Header>
        <wsse:Security
            xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/
               oasis-200401-wss-wssecurity-secext-1.0.xsd">
            <dsig:Signature xmlns:dsig="http://www.w3.org/2000/09/xmldsig#"
                Id="Id-0001346926985531-ffffffff28f6103-1">
                <dsig:SignedInfo>
                    <dsig:CanonicalizationMethod
                      Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#" />
                    <dsig:SignatureMethod
                      Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1" />
                    <dsig:Reference URI="">
                        <dsig:Transforms>
                            <dsig:Transform
                              Algorithm="http://www.w3.org/2000/09/
                                   xmldsig#enveloped-signature" />
                            <dsig:Transform
                              Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#" />
                        </dsig:Transforms>
                        <dsig:DigestMethod
                            Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
                        <dsig:DigestValue>
                            BAz3140AFAfBL/DIj9y+16TEJIU=
                        </dsig:DigestValue>
                    </dsig:Reference>
                </dsig:SignedInfo>
                <dsig:SignatureValue>........</dsig:SignatureValue>
                <dsig:KeyInfo Id="Id-0001346926985531-ffffffff28f6103-2">
                    <dsig:X509Data>
                        <dsig:X509Certificate>........</dsig:X509Certificate>
                    </dsig:X509Data>
                </dsig:KeyInfo>
            </dsig:Signature>
```

```
        </wsse:Security>
    </soap:Header>
    <soap:Body>
       <product version=11.1.2.4.0>
           <name>API Gateway Explorer</name>
           <company>Oracle</company>
           <description>SOA Security and Management</description>
       </product>
    </soap:Body>
</soap:Envelope>
```

Use SAML Ids for SAML Elements:
This option is only relevant if a SAML assertion is required to be signed. If this option is selected, and the signature is to cover a SAML assertion, an `AssertionID` attribute is inserted into a SAML version 1.1 assertion, or an `ID` attribute is inserted into a SAML version 2.0 assertion. The value of this attribute is then referenced from within a <Reference> block of the XML signature. This option is selected by default.

Add and Dereference Security Token Reference for SAML:
This option is only relevant if a SAML assertion is required to be signed. This setting signs the SAML assertion using a Security Token Reference and an STR-Transform. The `Signature` points to the id of the `wsse:SecurityTokenReference`, and applies the STR-Transform. When signing the SAML assertion, this means to sign the XML that the `wsse:SecurityTokenReference` points to, and not the `wsse:SecurityTokenReference`. This option is unselected by default. The following shows an example SOAP header:

```
<soap:Envelope xmlns:soap="....">
  <soap:Header>
     <wsse:Security xmlns:wsse="...." xmlns:wsu="....";
         <dsig:Signature xmlns:dsig=".....">
             <dsig:SignedInfo>
               <dsig:CanonicalizationMethod
                   Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#" />
               <dsig:SignatureMethod
                   Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1" />
               <dsig:Reference
                   URI="#Id-0001347292983847-00000000530a9b1a-1">
                   <dsig:Transforms>
                       <dsig:Transform
                          Algorithm="http://docs.oasis-open.org/wss/2004/01/
                          oasis-200401-wss-soap-message-security-1.0#STR-Transform">
                          <wsse:TransformationParameters>
                             <dsig:CanonicalizationMethod
                               Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#" />
                          </wsse:TransformationParameters>
                       </dsig:Transform>
                   </dsig:Transforms>
                   <dsig:DigestMethod
                       Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
                   <dsig:DigestValue>
                       6/aLwABWfS+9UiX7v39sLJw5MaQ=
                   </dsig:DigestValue>
               </dsig:Reference>
             </dsig:SignedInfo>
             <dsig:SignatureValue>
                .....
             </dsig:SignatureValue>
             <dsig:KeyInfo Id="Id-0001347292983847-00000000530a9b1a-3">
                 <dsig:X509Data>
                    <dsig:X509Certificate>
                       .....
                    </dsig:X509Certificate>
                 </dsig:X509Data>
```

```
            </dsig:KeyInfo>
         </dsig:Signature>
         <wsse:SecurityTokenReference
            wsu:Id="Id-0001347292983847-00000000530a9b1a-1">
            <wsse:KeyIdentifier
                ValueType="http://docs.oasis-open.org/wss/
                  oasis-wss-saml-token-profile-1.0#SAMLAssertionID">
                Id-948d50f1504e0f3703e00000-1
            </wsse:KeyIdentifier>
         </wsse:SecurityTokenReference>
         <saml:Assertion xmlns:saml="...."
             IssueInstant="2012-09-10T16:03:03Z"
             Issuer="CN=AAA Certificate Services, O=Comodo CA Limited,
               L=Salford, ST=Greater Manchester, C=GB"
             MajorVersion="1" MinorVersion="1">
             <saml:Conditions NotBefore="2012-09-10T16:03:02Z"
                 NotOnOrAfter="2012-12-18T16:03:02Z" />
             <saml:AuthenticationStatement
                AuthenticationMethod="urn:oasis:names:tc:SAML:1.0:am:password"
                AuthenticationInstant="2012-09-10T16:03:03Z">
                <saml:Subject>
                    <saml:NameIdentifier
                      Format="urn:oasis:names:tc:SAML:1.1:nameid-format:unspecified">
                      admin
                    </saml:NameIdentifier>
                    <saml:SubjectConfirmation>
                        <saml:ConfirmationMethod>
                            urn:oasis:names:tc:SAML:1.0:cm:sender-vouches
                        </saml:ConfirmationMethod>
                    </saml:SubjectConfirmation>
                </saml:Subject>
             </saml:AuthenticationStatement>
         </saml:Assertion>
      </wsse:Security>
  </soap:Header>
    ....
</soap:Envelope>
```

Where to place signature settings

Append Signature to Root or SOAP Header:
If the message is a SOAP message, the signature will be inserted into the SOAP `Header` element when this radio button is selected. The XML signature will be inserted as an immediate child of the SOAP `Header` element. The following example shows a skeleton SOAP message which has been signed using this option:

```
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
 <s:Header>
  <ws:Security xmlns:ws="http://schemas.xmlsoap.org/..." s:actor="test">
   <dsig:Signature xmlns:dsig="http://www.w3.org/2000/09/..." id="Sample">
     ...
   </dsig:Signature>
  </ws:Security>
 </s:Header>
 <s:Body>
  ...
 </s:Body>
</s:Envelope>
```

If the message is just plain XML, the signature is inserted as an immediate child of the root element of the XML message. The following example shows a non-SOAP XML message signed using this option:

```
<PurchaseOrder>
 <dsig:Signature xmlns:dsig="http://www.w3.org/2000/09/xmldsig#" id="Sample">
  ...
 </dsig:Signature>

 <Items>
 ...
 </Items>
</PurchaseOrder>
```

Place in WS-Security Element for SOAP Actor/Role:
By selecting this option, the XML signature will be inserted into the WS-Security element identified by the specified SOAP *actor* or *role*. A SOAP actor/role is simply a way of distinguishing a particular WS-Security block from others which might be present in the message.

Enter the name of the SOAP actor or role of the WS-Security block in the field. The following SOAP message contains an XML signature within a WS-Security block identified by the "test" actor:

```
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
 <s:Header>
  <ws:Security xmlns:ws="http://schemas.xmlsoap.org/..." s:actor="test">
   <dsig:Signature xmlns:dsig="http://www.w3.org/2000/09/..." id="Sample">
      ...
   </dsig:Signature>
  </ws:Security>
 </s:Header>
 <s:Body>
  ...
 </s:Body>
</s:Envelope>
```

Use XPath Location:
This option is useful in cases where the signature must be inserted into a non-SOAP XML message. In such cases, it is possible to insert the signature into a location pointed to by an XPath expression. Select or add an XPath expression in the field provided, and then specify whether the API Gateway Explorer should insert the signature *before* the location to which the XPath expression points, or *append* it to this location.

Advanced settings

The **Advanced** tab enables you to set the following:

- Additional elements from the message to be signed.
- Algorithms and ciphers used to sign the message parts.
- Various advanced options on the generated XML signature.

Additional

The **Additional** tab allows you to select additional elements from the message that are to be signed. It is also possible to insert a WS-Security Timestamp into the XML signature, if necessary.

Additional Elements to Sign:
The options here allow you to select other parts of the message to sign.

- **Sign KeyInfo Element of Signature**:
  The <KeyInfo> block of the XML signature can be signed to prevent people cut-and-pasting a different <KeyInfo> block into the message, which might point to some other key material, for example.

- **Sign Timestamp**:
  As stated earlier, timestamps are used to prevent replay attacks. However, to guarantee the end-to-end integrity of the timestamp, it is necessary to sign it.

  ## Note

  This option is only enabled when you have elected to insert a Timestamp into the message using the relevant fields on the **Timestamp Options** section below.

- **Sign Attachments**:
  In addition to signing some or all contents of the SOAP message, you can also sign attachments to the SOAP message. To sign all attachments, select **Include Attachments**. A signed attachment is referenced in an XML signature using the *Content-Id* or *cid* of the attachment. The `URI` attribute of the `Reference` element corresponds to this Content-Id. The following example shows how an XML signature refers to a sample attachment. It shows the wire format of the message and its attachment as they are sent to the destination web service. Multiple attachments result in successive `Reference` elements.

```
POST /myAttachments HTTP/1.0
Content-Length: 1000
User-Agent: API Gateway Explorer
Accept-Language: en
Content-Type: multipart/related; type="text/xml";
              boundary="----=Multipart-SOAP-boundary"

------=Multipart-SOAP-boundary
Content-Id: soap-envelope
SOAPAction: none
Content-Type: text/xml; charset="utf-8"

<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
 <s:Header>
  <dsig:Signature id="Sample" xmlns:dsig="http://www.w3.org/2000/09/xmldsig#">
   <dsig:SignedInfo>
    <dsig:CanonicalizationMethod
        Algorithm="http://www.w3.org/2001/10/xml-exc-c14n"/>
    <dsig:SignatureMethod
        Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
    <dsig:Reference URI="cid:moredata.txt">...</dsig:Reference>
   </dsig:SignedInfo>
  </dsig:Signature>
 </s:Header>
 <s:Body>
    ...
 </s:Body>
</s:Envelope>

------=Multipart-SOAP-boundary
Content-Id: moredata.txt
Content-Type: text/plain; charset="UTF-8"

Some more data.
------=Multipart-SOAP-boundary--
```

Transform:
This field is only available when you have selected the **Sign Attachments** box above. It determines the transform used to reference the signed attachments.

Timestamp Options:
It is possible to insert a timestamp into the message to indicate when exactly the signature was generated. Consumers of the signature can then validate the signature to ensure that it is not of date.

The following options are available:

- **No Timestamp**:
  No timestamp is inserted into the signature.
- **Embed in WSSE Security**:
  The `wsu:Timestamp` is inserted into a `wsse:Security` block. The `Security` block is identified by the SOAP actor/role specified on the **Signature** tab.
- **Embed in Signature Property**:
  The `wsu:Timestamp` is placed inside a signature property element in the `dsig:Signature`.

The **Expires In** fields enable the user to optionally specify the `wsu:Expires` for the `wsu:Timestamp`. If all fields are left at `0`, no `wsu:Expires` element is placed inside the `wsu:Timestamp`. The following example shows a `wsu:Timestamp` that has been inserted into a `wsse:Security` block:

```
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
 <s:Header>
   <wsse:Security>
     <wsu:Timestamp wsu:Id="Id-0000011294a0311e-000000000000003d">
      <wsu:Created>2007-05-16T11:22:45Z</wsu:Created>
      <wsu:Expires>2007-05-23T11:22:45Z</wsu:Expires>
    </wsu:Timestamp>
    <dsig:Signature ...>
    ...
    </dsig:Signature ...>
   </wsse:Security>
 </s:Header>
 <s:Body>
    ...
 </s:Body>
</s:Envelope>
```

Algorithm Suite

The fields on this tab determine the combination of cryptographic algorithms and ciphers that are used to sign the message parts.

Algorithm suite:
WS-Security Policy defines a number of *algorithm suites* that group together a number of cryptographic algorithms. For example, a given algorithm suite will use specific algorithms for asymmetric signing, symmetric signing, asymmetric key wrap, and so on. Therefore, by specifying an algorithm suite, you are effectively selecting a whole suite of cryptographic algorithms to use.

To use a particular WS-Security Policy algorithm suite, you can select it here. The **Signature Method**, **Key Wrap Algorithm**, and **Digest Method** fields will then be automatically populated with the corresponding algorithms for that suite.

Signature Method:
The **Signature Method** field enables you to configure the method used to generate the signature. Various strengths of the HMAC-SHA1 algorithms are available from the list.

Key Wrap Algorithm:
Select the algorithm to use to wrap (encrypt) the symmetric signing key. This option need only be configured when you are using a symmetric key to sign the message.

Digest Algorithm:
Select the digest algorithm to you to produce a cryptographic hash of the signed data.

Options

This tab enables you to configure various advanced options on the generated XML signature. The following fields can be configured on this tab:

WS-Security Options:
WSSE 1.1 defines a <SignatureConfirmation> element that can be used as proof that a particular XML signature was processed. A recipient and verifier of an XML signature must generate a <SignatureConfirmation> element for each piece of data that was signed (for each <Reference> in the XML signature). A <SignatureConfirmation> element contains the hash of the signed data and must be signed by the recipient before returning it in the response to the initiator (the original signatory of the data).

When the initiator receives the <SignatureConfirmation> elements in the response, it compares the hash with the hash of the data that it produced initially. If the hashes match, the initiator knows that the recipient has processed the same signature. Select the **Initiator** option if the API Gateway Explorer is the initiator as outlined in the scenario above. The API Gateway Explorer keeps a record of the signed data and compares it to the contents of the <SignatureConfirmation> elements returned from the recipient in the response message.

Alternatively, if the API Gateway Explorer is acting as the recipient in this transaction, you can select the **Responder** radio button to instruct the API Gateway to generate the <SignatureConfirmation> elements and return them to the initiator. The signature confirmations will be added to the WS-Security header.

Layout Type:
Select the WS-SecurityPolicy layout type that you want the XML signature and any generated tokens to adhere to. This includes elements such as <Signature>, <BinarySecurityToken>, and <EncryptedKey>, which can all be generated as part of the signing process.

Fail if No Nodes to Sign:
Check this option if you want the filter to fail if it cannot find any nodes to sign as configured on the **What to Sign** tab.

Add Inclusive Namespaces for Exclusive Canonicalization:
You can include information about the namespaces (and their associated prefixes) of signed elements in the signature itself. This ensures that namespaces that are in the same scope as the signed element, but not directly or visibly used by this element, are included in the signature. This ensures that the signature can be validated as a standalone entity outside of the context of the message from which it was extracted.

Note

The WS-I specification only permits the use of exclusive canonicalization in an XML signature. The <InclusiveNamespaces> element is an attempt to take advantage of some of the behavior of *inclusive* canonicalization, while maintaining the simplicity of *exclusive canonicalization*.

A *PrefixList* attribute is used to list the prefixes of in-scope, but not visibly used elements and attributes. The following example shows how the PrefixList attribute is used in practice:

```
<soap:Envelope xmlns:soap='http://schemas.xmlsoap.org/soap/envelope'>
 <soap:Header>
  <wsse:Security xmlns:wsse='http://docs.oasis-open.org/...'
                 xmlns:wsu='http://docs.oasis-open.org/...'>
   <wsse:BinarySecurityToken wsu:Id='SomeCert'
                             ValueType="http://docs.oasis-open.org/...">
   lui+Jy4WYKGJW5xM3aHnLxOpGVIpzSg4V486hHFe7sH
   </wsse:BinarySecurityToken>
   <ds:Signature xmlns:ds='http://www.w3.org/2000/09/xmldsig#'>
    <ds:SignedInfo>
     <ds:CanonicalizationMethod
         Algorithm='http://www.w3.org/2001/10/xml-exc-c14n#'>
      <c14n:InclusiveNamespaces
          xmlns:c14n='http://www.w3.org/2001/10/xml-exc-c14n#'
          PrefixList='wsse wsu soap' />
```

```
    </ds:CanonicalizationMethod>
    <ds:SignatureMethod
         Algorithm='http://www.w3.org/2000/09/xmldsig#rsa-sha1'/>
    <ds:Reference URI=''>
     <ds:Transforms>
      <dsig:XPath xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
                  xmlns:m='http://example.org/ws'>
       //soap:Body/m:SomeElement
      </dsig:XPath>
      <ds:Transform Algorithm='http://www.w3.org/2001/10/xml-exc-c14n#'>
       <c14n:InclusiveNamespaces
             xmlns:c14n='http://www.w3.org/2001/10/xml-exc-c14n#'
             PrefixList='soap wsu test' />
      </ds:Transform>
     </ds:Transforms>
     <ds:DigestMethod Algorithm='http://www.w3.org/2000/09/xmldsig#sha1' />
     <ds:DigestValue>VEPKwzfPGOxh2OUpoK0bcl58jtU=</ds:DigestValue>
    </ds:Reference>
   </ds:SignedInfo>
   <ds:SignatureValue>+diIuEyDpV7qxVoUOkb5rj61+Zs=</ds:SignatureValue>
   <ds:KeyInfo>
    <wsse:SecurityTokenReference>
     <wsse:Reference URI='#SomeCert' />
    </wsse:SecurityTokenReference>
   </ds:KeyInfo>
  </ds:Signature>
 </wsse:Security>
</soap:Header>
<soap:Body xmlns:wsu='http://docs.oasis-open.org/...'
           xmlns:test='http://www.test.com' wsu:Id='TheBody'>
 <m:SomeElement xmlns:m='http://example.org/ws' attr1='test:fdwfde' />
</soap:Body>
</soap:Envelope>
```

Indent:
Select this method to ensure that the generated signature is properly indented.

Create Enveloped Signature:
By selecting this option, an enveloped XML signature is generated. The following skeleton signed SOAP message shows the enveloped signature:

```
<ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#" id="Sample">
 <ds:SignedInfo>
  <ds:Reference URI="">
   <ds:Transforms>
    <ds:Transform
        Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-signature"/>
   </ds:Transforms>
  </ds:Reference>
 </ds:SignedInfo>
</ds:Signature>
```

This indicates to the application validating the signature that the signature itself should not be included in the signed data. In other words, to validate the signature, the application must first strip out the signature. This is necessary in cases where the entire SOAP envelope has been signed, and the resulting signature has been inserted into the SOAP header. In this case, the signature is over a nodeset which has been altered (the signature has been inserted), and so the signature will break.

Insert CarriedKeyName for EncryptedKey:
Select this option to include a `<CarriedKeyName>` element in the `<EncryptedKey>` block that is generated when using a symmetric signing key.

XML signature verification

Overview

In addition to validating XML signatures for authentication purposes, the API Gateway Explorer can also use XML signatures to prove message integrity. By signing an XML message, a client can be sure that any changes made to the message do not go unnoticed by the API Gateway Explorer. Therefore by validating the XML signature on a message, the API Gateway Explorer can guarantee the *integrity* of the message.

General settings

Configure the following general setting:

Name:
Enter an appropriate name for the filter.

Signature verification settings

The following sections are available on the **Signature Verification** tab:

Signature Location:
Because there may be multiple signatures in the message, you must specify which signature API Gateway Explorer uses to verify the integrity of the message. The signature can be extracted from one of the following:

- From the SOAP header
- Using WS-Security Actors
- Using XPath

Select the appropriate option from the list. For more details, see *Signature location*.

Find Signing Key
The public key used to verify the signature can be taken from the following locations:

- **Via KeyInfo in Message**:
  Typically, a `<KeyInfo>` block is used in an XML signature to reference the key used to sign the message. For example, it is common for a `<KeyInfo>` block to reference a `<BinarySecurityToken>` that contains the certificate associated with the public key used to verify the signature.
- **Via Selector Expression**:
  The certificate used to verify the signature can be extracted from a selector expression. For example, a previous filter (for example, **Find Certificate**) may have already located a certificate and populated the `certificate` message attribute. To use this certificate to verify the signature, specify the selector expression in the field provided (for example, `${certificate}`). Using a selector enables settings to be evaluated and expanded at runtime based on metadata (for example, in a message attribute, KPS, or environment variable). For more details, see *Select configuration values at runtime*.
- **Via Certificate in LDAP**:
  Clients may not always want to include their public keys in their signatures. In such cases, the public key can be retrieved from a specified LDAP directory. This setting enables you to select a previously configured LDAP directory from a list. You can add LDAP connections under the **External Connections** node in the Policy Studio tree. Right-click the **LDAP Connection** tree node, and select **Add an LDAP Connection**.
- **Via Certificate in Store**:
  Similarly, you can retrieve a certificate from the certificate store by selecting this option, and clicking the **Select** button. Select the check box next to the certificate that contains the public key that you want to use to verify the signature, and click **OK**.

What must be signed settings

The **What Must Be Signed** tab defines the content that must be signed for a SOAP message to pass the filter. This ensures that the client has signed something meaningful (part of the SOAP message), instead of arbitrary data that would pass a blind signature validation. This further strengthens the integrity verification process. The nodeset that must be signed can be identified by a combination of XPath expressions, node locations, and the contents of a message attribute. For more details, see *What to sign*.

Note

If all attachments are required to be signed, select **All attachments** to enforce this.

Advanced settings

The following advanced configuration options are available on the **Advanced** tab:

Signature Confirmation:
If this filter is configured as part of an initiator policy, where the API Gateway Explorer acts as the client in a web services transaction, select the **Initiator** option. This means that the filter keeps a record of the signature that it has verified, and checks the `SignatureConfirmation` returned by the recipient.

Alternatively, if the API Gateway Explorer acts as the recipient in the transaction, select the **Recipient** option. In this case, the API Gateway Explorer returns the `SignatureConfirmation` elements in the response to the initiator.

Default Derived Key Label:
If the API Gateway Explorer consumes a `DerivedKeyToken`, use the default value to recreate the derived key.

Algorithm Suite:
Select the WS-Security Policy *Algorithm Suite* that must have been used when signing the message. This check ensures that the appropriate algorithms were used to sign the message.

Fail if No Signatures to Verify:
Select this if you want to configure the filter to fail if no XML signatures are present in the incoming message.

Verify Signature for Authentication Purposes:
You can use the **XML Signature Verification** filter to authenticate an end user. If the message can be successfully validated, it proves that only the private key associated with the public key used to verify the signature was used to sign the message. Because the private key is only accessible to its owner, a successful verification can be used to effectively authenticate the message signer.

Retrieve DOM using Selector Expression:
You can configure this field to verify the response from a SAML PDP. When the API Gateway Explorer receives a response from the SAML PDP, it stores the signature on the response in a message attribute. You can specify this attribute using a selector expression to verify this signature. Using a selector enables settings to be evaluated and expanded at runtime based on metadata (for example, in a message attribute, Key Property Store (KPS), or environment variable). For more details, see *Select configuration values at runtime*.

Remove enclosing WS-Security element on successful verification:
Select this check box if you wish to remove the enclosing WS-Security block when the signature has been successfully verified. This setting is not selected by default.

Kerberos configuration

Overview

The **Kerberos Configuration** screen enables you to configure API Gateway Explorer instance-wide Kerberos settings. The most important setting allows you to upload a Kerberos configuration file to the API Gateway Explorer, which contains information about the location of the Kerberos Key Distribution Center (KDC), encryption algorithms and keys, and domain realms to use.

You can also configure trace options for the various APIs used by the Kerberos system. For example, these include the Generic Security Services (GSS) and Simple and Protected GSSAPI Negotiation (SPNEGO) APIs.

Linux and Solaris platforms ship with a native implementation of the GSS library, which can be leveraged by the API Gateway Explorer. The location of the GSS library can be specified using settings on this screen.

Kerberos configuration file—krb5.conf

The Kerberos configuration file (`krb5.conf`) is required by the Kerberos system to configure the location of the Kerberos KDC, supported encryption algorithms, and default realms.

The file is required by both Kerberos Clients and Services that are configured for the API Gateway Explorer. Kerberos Clients need to know the location of the KDC so that they can obtain a Ticket Granting Ticket (TGT). They also need to know what encryption algorithms to use and to what realm they belong.

A Kerberos Client or Service knows what realm it belongs to because either the realm is appended to the principal name after the `@` symbol. Alternatively, if the realm is not specified in the principal name, it is assumed to be in the `default_realm` as specified in the `krb5.conf` file.

Kerberos Services do not need to talk to the KDC to request a TGT. However, they still require the information about supported encryption algorithms and default realms contained in the `krb5.conf` file. There is only one `default_realm` specified in this file, but you can specify a number of additional named realms. The `default_realm` setting is found in the `[libdefaults]` section of the `krb5.conf` file. It points to a realm in the `[realms]` section. This setting is not required.

A default `krb5.conf` is displayed in the text area, which can be modified where appropriate and then uploaded to the API Gateway Explorer's configuration by clicking the **OK** button. Alternatively, if you already have a `krb5.conf` file that you want to use, browse to this file using the **Load File** button. The contents of the file are displayed in the text area, and can subsequently be uploaded by clicking the **OK** button.

Note

You can also type directly into the text area to modify the `krb5.conf` contents. Please refer to your Kerberos documentation for more information on the settings that can be configured in the `krb5.conf` file.

Advanced settings

The check boxes on this screen enable you to configure various tracing options for the underlying Kerberos API. Trace output is always written to the `/trace` directory of your API Gateway Explorer installation.

Kerberos Debug Trace:
Enables extra tracing from the Kerberos API layer.

SPNEGO Debug Trace:
Turns on extra tracing from the SPNEGO API layer.

Extra Debug at Login:
Provides extra tracing information during login to the Kerberos KDC.

Native GSS library

The Generic Security Services API (GSS-API) is an API for accessing security services, including Kerberos. Implementations of the GSS-API ship with the Linux and Solaris platforms and can be leveraged by the API Gateway Explorer when it is installed on these platforms. The fields on this tab allow you to configure various aspects of the GSS-API implementation for your target platform.

Note

These are instance-wide settings. If use of the native GSS API is selected, it will be used for all Kerberos operations. All Kerberos Clients and Services must therefore be configured to load their credentials natively.

If the native API is used the following will not be supported:

- The SPNEGO mechanism.
- The WS-Trust for SPNEGO standard as it requires the SPNEGO mechanism.
- The SPNEGO over HTTP standard as it requires the SPNEGO mechanism. (It is possible to use the KERBEROS mechanism with this protocol, but this would be non-standard.)
- Signing and encrypting using the Kerberos session keys.

Use Native GSS Library:
Check this checkbox to use the operating system's native GSS implementation. This option only applies to API Gateway Explorer installations on the Linux and Solaris platforms.

Native GSS Library Location:
If you have opted to use the native GSS library, enter the location of the GSS library in the field provided, for example, `/usr/lib/libgssapi.so`. On Linux, the library is called `libgssapi.so`. On Solaris, this library is called `libgss.so`.

Note

This setting is only required when this library is in a non-default location.

Native GSS Trace:
Use this option to enable debug tracing for the native GSS library.

Kerberos client authentication

Overview

You can configure the API Gateway Explorer to act as a Kerberos client by obtaining a service ticket for a specific Kerberos service. The service ticket makes up part of the Kerberos client-side token that is injected into a SOAP message and then sent to the service. If the service can validate the token, the client is authenticated successfully.

You can also configure a **Connection** filter (from the Routing category) to authenticate to a Kerberos service by inserting a client-side Kerberos token into the `Authorization` HTTP header.

Therefore, you should use the **Connection** filter to send the client-side Kerberos token in an HTTP header to the Kerberos service. You should use the **Kerberos Client Authentication** filter to send the client-side Kerberos token in a `BinarySecurityToken` block in the SOAP message. For more information on authenticating to a Kerberos service using a client-side Kerberos token, see the topic on the *Connection* filter.

The **Kerberos Client Authentication** filter is available from the authentication category of filters. Drag and drop this filter on to the policy canvas to configure the filter. The sections below describe how to configure the fields on this filter window.

General settings

Name:
Enter an appropriate name for the filter.

Kerberos client settings

The fields configured on the **Kerberos Client** tab determine how the Kerberos client obtains a service ticket for a specific Kerberos service. The following fields must be configured:

Kerberos Client:
The role of the Kerberos client selected in this field is twofold. First, it must obtain a Kerberos Ticket Granting Ticket (TGT) and second, it uses this TGT to obtain a service ticket for the **Kerberos Service Principal** selected below. The TGT is acquired at server startup, refresh (for example, when a configuration update is deployed), and when the TGT expires.

Click the button on the right, and select a previously configured Kerberos client in the tree. To add a Kerberos client, right-click the **Kerberos Clients** tree node, and select **Add Kerberos Client**.

Kerberos Service Principal:
The Kerberos client selected above must obtain a service ticket from the Kerberos Ticket Granting Server (TGS) for the Kerberos service principal selected in this field. The service principal can be used to uniquely identify the service in the Kerberos realm. The TGS grants a ticket for the selected principal, which the client can then send to the Kerberos service. The principal in the ticket must match the Kerberos service's principal for the client to be successfully authenticated.

Click the button on the right, and select a previously configured Kerberos principal in the tree (for example, the default `HTTP/host Service Principal`). To add a Kerberos principal, right-click the **Kerberos Principals** tree node, and select **Add Kerberos Principal**.

Kerberos Standard:
When using the **Kerberos Client Authentication** filter to insert Kerberos tokens into SOAP messages in order to authenticate to Kerberos services, it can do so according to two different standards:

- Web Services Security Kerberos Token Profile 1.1
- WS-Trust for Simple and Protected Negotiation Protocol (SPNEGO)

When using the Kerberos Token Profile, the client-side Kerberos token is inserted into a `BinarySecurityToken` block within the SOAP message. The Kerberos session key may be used to sign and encrypt the SOAP message using the signing and encrypting filters. When this option is selected, the fields on the **Kerberos Token Profile** tab must be configured.

When the WS-Trust for SPNEGO standard is used, a series of requests and responses occur between the Kerberos client and the Kerberos service in order to establish a secure context. Once the secure context has been established (using WS-Trust and WS-SecureConversation), a further series of requests and responses are used to produce a shared secret key that can be used to sign and encrypt "real" requests to the Kerberos service.

If the **WS-Trust for SPNEGO** option is selected it will not be necessary to configure the fields on the **Kerberos Token Profile** tab. However, the **Kerberos Client Authentication** filter must be configured as part of a complicated policy that is set up to handle the multiple request and response messages that are involved in setting up the secure context between the Kerberos client and service.

Kerberos token profile settings

The fields on this tab need only be configured if the **Kerberos Token Profile** option has been selected on the **Kerberos Client** tab. This tab allows you to configure where to insert the `BinarySecurityToken` within the SOAP message.

Where to Place BinarySecurityToken:
It is possible to insert the `BinarySecurityToken` inside a named WS-Security Actor/Role within the SOAP message or else an XPath expression can be specified to indicate where the token should be inserted.

Select the **WS-Security Element** radio button to insert the token into a WS-Security element within the SOAP Header element. You can either select the default `Current actor/role only` option or enter a named actor/role in the field provided. The `BinarySecurityToken` will be inserted into a WS-Security block for the actor/role specified here.

Alternatively, you should select the **XPath Location** option to use an XPath expression to specify where the `BinarySecurityToken` is to be inserted. Click the **Add** button to add a new XPath expression or select an XPath and click the **Edit** or **Delete** buttons to edit or delete an existing XPath expression. For more information, see the *Configure XPath expressions* topic.

Note

You can insert the `BinarySecurityToken` *before* or *after* the node pointed to by the XPath expression. Select the **Append** or **Before** radio buttons depending on where you want to insert the token relative to the node pointed to by the XPath expression.

BinarySecurityToken Value Type:
Currently, the only supported `BinarySecurityToken` type is the `GSS_Kerberosv5_AP_REQ` type. The selected type will be specified in the generated `BinarySecurityToken`.

Connection

Overview

The **Connection** filter makes the connection to the remote Web service. It relies on connection details that are set by the other filters in the **Routing** category. Because the **Connection** filter connects out to other services, it negotiates the SSL handshake involved in setting up a mutually authenticated secure channel.

General settings

Enter an appropriate name for the filter in the **Name** field. Click the tabs to configure the various aspects of the **Connection** filter.

SSL settings

You can configure SSL settings, such as trusted certificates, client certificates, and ciphers on the **SSL** tab. For details on the fields on this tab, see the section called “SSL settings” in the *Connect to URL* topic.

Authentication settings

You can select credential profiles to use for authentication on the **Authentication** tab. For details on the fields on this tab, see the section called “Authentication settings” in the *Connect to URL* topic.

Additional settings

The **Settings** tab allows you to configure the following additional settings:

- **Retry**
- **Failure**
- **Proxy**
- **Redirect**
- **Headers**

By default, these sections are collapsed. Click a section to expand it.

For details on the fields on this tab, see the section called “Additional settings” in the *Connect to URL* topic.

Connect to URL

Overview

The **Connect to URL** filter is the simplest routing filter to use to connect to a target Web service. To configure this filter to send messages to a Web service, you need only enter the URL of the service in the **URL** field. If the Web service is SSL enabled or requires mutual authentication, you can use the other tabs on the **Connect to URL** filter to configure this.

Depending on how the API Gateway Explorer is perceived by the client, different combinations of routing filters can be used. Using the **Connect to URL** filter is equivalent to using the following combination of routing filters:

- Static Router
- Rewrite URL
- Connection

General settings

Configure the following general settings:

Name:
Enter an appropriate name for the filter.

URL:
Enter the complete URL of the target Web service. You can specify this setting as a selector, which enables values to be expanded at runtime. For more details, see *Select configuration values at runtime*. Defaults to `${http.request.uri}`.

Tip

You can also enter any query string parameters associated with the incoming request message as a selector, for example, `${http.request.uri}?${http.raw.querystring}`.

Request settings

On the **Request** tab, you can use the API Gateway Explorer selector syntax to evaluate and expand request details at runtime. For more details, see *Select configuration values at runtime*. The values specified on this tab are used in the outbound request to the URL.

Method:
Enter the HTTP verb used in the incoming request (for example, `GET`). Defaults to `${http.request.verb}`.

Request Body:
Enter the content of the incoming request message body. Defaults to `${content.body}`.

Important

You must enter the body headers and body content in the **Request Body** text area. For example, enter the `Content-Type` followed by a return and then the required message payload:

```
Content-Type: text/html

<!DOCTYPE html>
<html>
<body>
<h1>Hello World</h1>
</body>
```

```
</html>
```

Request Protocol Headers:
Enter the HTTP headers associated with the incoming request message. Defaults to `${http.headers}`.

SSL settings

Configure the SSL settings on the **SSL** tab. You can select the server certificates to trust on the **Trusted Certificates** tab, and the client certificates on the **Client Certificates** tab.

You can also specify the ciphers that API Gateway Explorer supports in the **Ciphers** field. The API Gateway Explorer sends this list of supported ciphers to the destination server, which selects the highest strength common cipher as part of the SSL handshake. The selected cipher is then used to encrypt the data as it is sent over the secure channel.

Trusted certificates

When API Gateway Explorer connects to a server over SSL, it must decide whether to trust the server's SSL certificate. You can select a list of CA or server certificates from the **Trusted Certificates** tab that are considered trusted by the API Gateway Explorer when connecting to the server specified in the **URL** field on this dialog.

The table on the **Trusted Certificates** tab lists all certificates imported into the API Gateway Explorer Certificate Store. To *trust* a certificate for this particular connection, select the box next to the certificate in the table.

To select all certificates for a particular CA, select the box next to the CA parent node in the table.

Alternatively, you can select the **Trust all certificates in the Certificate Store** option to trust all certificates in the store. This is selected by default.

Client certificates

In cases where the destination server requires clients to authenticate to it using an SSL certificate, you must select a client certificate on the **Client Certificates** tab.

To select a client certificate click the **Signing Key** button, and complete the following fields on the **Select Certificate** dialog:

Choose the certificate to present for mutual authentication (optional):
Select this option to choose a certificate from the Certificate Store. Select the client certificate to use to authenticate to the server specified in the **URL** field.

Authentication settings

The **Authentication** tab enables you to select a client credential profile for authentication. You can use client credential profiles to configure client credentials and provider settings for authentication using API keys, HTTP basic or digest authentication, Kerberos, or OAuth.

Click the browse button next to the **Choose a Credential Profile** field to select a credential profile. You can configure client credential profiles globally under the **External Connections** node in the Policy Studio tree. For more details on configuring client credentials, see the ??? topic.

Additional settings

The **Settings** tab enables you to configure the following additional settings:

- **Retry**

- **Failure**
- **Proxy**
- **Redirect**
- **Headers**

By default, these sections are collapsed. Click a section to expand it.

Retry settings

To specify the retry settings for this filter, complete the following fields:

Perform Retries:
Select whether the filter performs retries. By default, this setting is not selected, no retries are performed, and all **Retry** settings are disabled. This means that the filter only attempts to perform the connection once.

Retry On:
Select the HTTP status ranges on which retries can be performed. If a host responds with an HTTP status code that matches one of the selected ranges, this filter performs a retry. Select one or more ranges in the table (for example, `Client Error 400-499`). For details on adding custom HTTP status ranges, see the next subsection.

Retry Count:
Enter the maximum number of retries to attempt. Defaults to `5`.

Retry Interval (ms):
Enter the time to delay between retries in milliseconds. Defaults to `500` ms.

Add an HTTP status range
To add an HTTP status range to the default list displayed in the **Retry On** table, click the **Add** button. In the **Configure HTTP Status Code** dialog, complete the following fields:

Name	Enter a name for the HTTP status range.
Start status	Enter the first HTTP status code in the range.
End status	Enter the last HTTP status code in the range.

To add one specific status code only, enter the same code in the **Start status** and **End status** fields. Click **OK** to finish. You can manage existing HTTP status ranges using **Edit** and **Delete**.

Failure settings

To specify the failure settings for this filter, complete the following fields:

Consider SLA Breach as Failure:
Select whether to attempt the connection if a configured SLA has been breached. This is not selected by default. If this option is selected, and an SLA breach is encountered, the filter returns `false`.

Save Transaction on Failure (for replay):
Select whether to store the incoming message in the specified directory and file if a failure occurs during processing. This is not selected by default.

File name:
Enter the name of the file that the message content is saved to. You can specify this using a selector, which is expanded to the specified value at runtime. Defaults to `${id}.out`. For more details on selectors, see *Select configuration values at runtime*.

Directory:

Enter the directory that the file is saved to. You can specify this using a selector, which is expanded to the specified value at runtime. Defaults to `${environment.VINSTDIR}/message-archive`, where `VINSTDIR` is the location of a running API Gateway Explorer instance.

Maximum number of files in directory:
Enter the maximum number of files that can be saved in the directory. Defaults to `500`.

Maximum file size:
Enter the maximum file size in MB. Defaults to `1000`.

Include HTTP Headers:
Select whether to include HTTP headers in the file. HTTP headers are not included by default.

Include Request Line:
Select whether to include the HTTP request line from the client in the file. The request line is not included by default.

Call policy on Connection Failure:
Select whether to execute a policy in the event of a connection failure. This is not selected by default.

Connection Failure Policy:
Click the browse button on the right, and select the policy to run in the event of a connection failure in the dialog.

Proxy settings

To specify the proxy settings for this filter, complete the following fields:

Send via Proxy:
Select this option if the API Gateway Explorer must connect to the destination Web Service through an HTTP proxy. In this case, the API Gateway Explorer includes the full URL of the destination Web service in the request line of the HTTP request. For example, if the destination Web service resides at `http://localhost:8080/services`, the request line is as follows:

```
POST http://localhost:8080/services HTTP/1.1
```

If the API Gateway Explorer was not routing through a proxy, the request line is as follows:

```
POST /services HTTP/1.1
```

Proxy Server:
When **Send via Proxy** is selected, you can configure a specific proxy server to use for the connection. Click the browse button next to this field, and select an existing proxy server.

Transparent Proxy (present client's IP address to server):
Enables the API Gateway Explorer as a *transparent proxy* on Linux systems with the `TPROXY` kernel option set. When selected, the IP address of the original client connection that caused the policy to be invoked is used as the local address of the connection to the destination server.

Redirect settings

To specify the redirect settings for this filter, complete the following fields:

Follow Redirects:
Specifies whether the API Gateway Explorer follows HTTP redirects, and connects to the redirect URL specified in the HTTP response. This setting is enabled by default.

Header settings

To specify the header settings for this filter, complete the following fields:

Forward spurious received Content headers:
Specifies whether the API Gateway Explorer sends any content-related message headers when sending an HTTP request with no message body to the HTTP server. For example, select this setting if content-related headers are required by an out-of-band agreement. If there is no body in the outbound request, any content-related headers from the original inbound HTTP request are forwarded. These are extracted from the `http.content.headers` message attribute, generally populated by the API Gateway Explorer for the incoming call. This attribute can be manipulated in a policy using the appropriate filters, if required. This field is not selected by default.

HTTP Host Header:
An HTTP 1.1 client *must* send a `Host` header in all HTTP 1.1 requests. The `Host` header identifies the host name and port number of the requested resource as specified in the original URL given by the client.

When routing messages on to target Web services, the API Gateway Explorer can forward on the `Host` as received from the client, or it can specify the address and port number of the destination Web Service in the `Host` header that it routes onwards.

Select **Use Host header specified by client** to force the API Gateway Explorer to always forward on the original `Host` header that it received from the client. Alternatively, to configure the API Gateway Explorer to include the address and port number of the destination Web service in the `Host` header, select the **Generate new Host header** radio button.

HTTP status code

Overview

This filter sets the HTTP status code on response messages. This enables an administrator to ensure that a more meaningful response is sent to the client in the case of an error occurring in a configured policy.

For example, if a **Relative Path** filter fails, it might be useful to return a `503 Service Unavailable` response. Similarly, if a user does not present identity credentials when attempting to access a protected resource, you can configure API Gateway Explorer to return a `401 Unauthorized` response to the client.

HTTP status codes are returned in the *status-line* of an HTTP response. The following are some typical examples:

```
HTTP/1.1 200 OK
HTTP/1.1 400 Bad Request
HTTP/1.1 500 Internal Server Error
```

Configuration

Name:
Enter an appropriate name for this filter.

HTTP response code status:
Enter the status code returned to the client. For a complete list of status codes, see the HTTP Specification [http://www.w3.org/Protocols/rfc2616/rfc2616-sec6.html].

Insert WS-Addressing information

Overview

The WS-Addressing specification defines a transport-independent standard for including addressing information in SOAP messages. API Gateway Explorer can generate WS-Addressing information based on a configured endpoint in a policy, and then insert this information into SOAP messages.

Configuration

Complete the following fields to configure API Gateway Explorer to insert WS-Addressing information into the SOAP message header.

Name:
Enter an appropriate name for the filter.

To:
The message is delivered to the specified destination.

From:
Informs the destination server where the message originated from.

Reply To:
Indicates to the destination server where it should send response messages to.

Fault To:
Indicates to the destination server where it should send fault messages to.

MessageID:
A unique identifier to distinguish this message from others at the destination server. It also provides a mechanism for correlating a specific request with its corresponding response message.

Action:
The specified action indicates what action the destination server should take on the message. Typically, the value of the WS-Addressing `Action` element corresponds to the SOAPAction on the request message. For this reason, this field defaults to the `soap.request.action` message attribute.

Relates To:
If responses are to be received asynchronously, the specified value provides a method to associate an incoming reply to its corresponding request.

Namespace:
The WS-Addressing namespace to use in the WS-Addressing block.

Send to JMS

Overview

The **Send to JMS** filter enables you to configure a JMS messaging system to which the API Gateway sends messages. You can configure various settings for the message request and response (for example, destination and message type, how the message system should respond, and so on).

API Gateway Explorer provides all the required third-party JAR files for IBM WebSphere MQ and Apache ActiveMQ (both embedded and external).

Note

For other third-party JMS providers only, you must add the required third-party JAR files to the API Gateway Explorer classpath for messaging to function correctly. If the provider's implementation is platform-specific, copy the provider JAR files to `INSTALL_DIR/ext/PLATFORM`.

`INSTALL_DIR` is your API Gateway Explorer installation, and `PLATFORM` is the platform on which API Gateway Explorer is installed (`Win32`, `Linux.i386`, or `SunOS.sun4u-32`). If the provider implementation is platform-independent, copy the JAR files to `INSTALL_DIR/ext/lib`.

Request settings

The **Request** tab specifies the following properties of the request sent to the messaging system:

JMS Service:
Click the browse button on the right, and select an existing JMS service in the tree. To add a JMS Service, right-click the **JMS Services** tree node, and select **Add a JMS Service**.

Destination type:
Select one of the following from the list:

- **Queue**
- **Topic**
- **JNDI lookup**

Defaults to **Queue**.

Destination:
Enter the name of the JMS queue, JMS topic, or JNDI lookup to specify where you want to drop the messages.

Delivery Mode:
Select one of the following delivery modes:

- **Persistent**:
  Instructs the JMS provider to ensure that a message is not lost in transit if the JMS provider fails. A message sent with this delivery mode is logged to persistent storage when it is sent. This is the default mode.
- **Non-persistent**:
  Does not require the JMS provider to store the message. With this mode, the message may be lost if the JMS provider fails.

Priority Level:
You can use message priority levels to instruct the JMS provider to deliver urgent messages first. The ten levels of priority range from 0 (lowest) to 9 (highest). If you do not specify a priority level, the default level is 4. A JMS provider tries to

deliver higher priority messages before lower priority ones but does not have to deliver messages in exact order of priority.

Time to Live:
By default, a message never expires. However, if a message becomes obsolete after a certain period, you may want to set an expiry time (in milliseconds). The default value is `0`, which means the message never expires.

Message ID:
Enter an identifier to be used as the unique identifier for the message. By default, the unique identifier is the ID assigned to the message by API Gateway Explorer (`${id}`). However, you can use a proprietary correlation system, perhaps using MIME message IDs instead of API Gateway Explorer message IDs.

Correlation ID:
Enter an identifier for the message that API Gateway Explorer uses to correlate response messages with the corresponding request messages. Usually, if `${id}` is specified in the **Message ID** field above, it is also used here to correlate request messages with their correct response messages.

Message Type:
This drop-down list enables you to specify the type of data to be serialized and sent in the JMS message to the JMS provider. The option selected depends on what part of the message you want to send to the consumer. For example, to send the message body, select the option to format the body according to the rules defined in the SOAP over JMS [http://www.w3.org/TR/soapjms/] recommendation. Alternatively, to serialize a list of name-value pairs to the JMS message, choose the option to create a `MapMessage`.

Select one of thd following serialization options:

- **Use content.body attribute to create a message in the format specified in the SOAP over Java Message Service recommendation**:
  If this option is selected, messages are formatted according to the SOAP over JMS [http://www.w3.org/TR/soapjms/] recommendation. This is the default option because in most cases the message body is routed to the messaging system. When this option is selected, a `javax.jms.BytesMessage` is created and a JMS property containing the content type `text/xml`) is set on the message.
- **Create a MapMessage from the java.lang.Map in the attribute named below**:
  Select this option to create a `javax.jms.MapMessage` from the API Gateway Explorer message attribute named below that consists of name-value pairs.
- **Create a BytesMessage from the attribute named below**:
  Select this option to create a `javax.jms.BytesMessage` from the API Gateway Explorer message attribute named below.
- **Create an ObjectMessage from the java.lang.Serializable in the attribute named below**:
  Select this option to create a `javax.jms.ObjectMessage` from the API Gateway Explorer message attribute named below.
- **Create a TextMessage from the attribute named below**:
  Select this option to create a `javax.jms.TextMessage` from the message attribute named below.
- **Use the javax.jms.Message stored in the attribute named below**:
  If a `javax.jms.Message` has already been stored in a message attribute, select this option, and enter the name of the attribute in the field below.

Attribute Name:
Enter the name of the API Gateway Explorer message attribute that holds the data that is to be serialized to a JMS message and sent over the wire to the JMS provider. The type of the attribute named here must correspond to that selected in the **Message Type** drop-down field above.

Custom Message Properties:
You can set custom properties for messages in addition to those provided by the header fields. Custom properties may be required to provide compatibility with other messaging systems. You can use message attribute selectors as property values. For example, you can create a property called `AuthNUser`, and set its value to `${authenticated.subject.id}`. Other applications can then filter on this property (for example, only consume mes-

sages where `AuthNUser` equals `admin`). To add a new property, click **Add**, and enter a name and value in the fields provided on the **Properties** dialog.

Use the following policy to change JMS request message:
This setting enables you to customize the JMS message before it is published to a JMS queue or topic. Click the browse button on the right, and select a configured policy in the dialog. The selected policy is then invoked before the JMS request is sent to the queuing system.

When the selected policy is invoked, the JMS request message is available on the white board in the `jms.outbound.message` message attribute. You can therefore call JMS API methods to manipulate the JMS request further. For example, you could configure a policy containing a **Scripting Language** filter that runs a script such as the following against the JMS message:

```
function invoke(msg) {
 var jmsMsg = msg.get("jms.outbound.message");
 jmsMsg.setIntProperty("My_JMS_Report", 123);
 return true;
}
```

Response settings

The **Response** tab specifies whether API Gateway Explorer uses asynchronous or synchronous communication when talking to the messaging system. For example, to use asynchronous communication, you can select the **Do not set response** option. If synchronous communication is required, you can select to read the response from a temporary queue or from a named queue or topic.

You can also specify whether API Gateway Explorer waits on a response message from a queue or topic from the messaging system. API Gateway Explorer sets the `JMSReplyTo` property on each message that it sends. The value of the `JMSReplyTo` property is the temporary queue, queue, or topic selected in this dialog. It is the responsibility of the application that consumes the message from the queue (JMS consumer) to send the message back to the destination specified in `JMSReplyTo`.

API Gateway Explorer sets the `JMSCorrelationID` property to the value of the **Correlation ID** field on the **Request** tab to correlate requests messages to their corresponding response messages. If you select to use a temporary queue or temporary topic, this is created when API Gateway Explorer starts up.

Configure how messaging system should respond:
Select where the response message is to be placed using one of the following options:

- **Do not set response**:
  Select this option if you do not expect or do not care about receiving a response from the JMS provider.
- **Use temporary queue**:
  Select this option to instruct the JMS provider to place the response message on a temporary queue. In this case, the temporary queue is created when API Gateway Explorer starts up. Only API Gateway Explorer can read from the temporary queue, but any application can write to it. API Gateway Explorer uses the value of the `JMSReplyTo` header to indicate the location where it reads responses from.
- **Use queue**:
  If you want the JMS provider to place response messages on a queue, select this option, and enter the queue name in the text box. This is used in the `JMSReplyTo` field of the response message.
- **Use topic**:
  If you want the JMS provider to place response messages on a topic, select this option, and enter the topic name in the text box. This is used in the `JMSReplyTo` field of the response message.
- **Use named queue or topic (JNDI)**:
  If you want the JMS provider to place response messages on a named queue or topic using JNDI lookup, select this option, and enter the JNDI name for the queue or topic in the text box. This is used in the `JMSReplyTo` field of the response message.

Wait for response:
If **Do not set response** is not selected, you can select whether API Gateway Explorer waits to receive a response:

- **Wait with timeout (ms)**:
  API Gateway Explorer waits a specific time period to receive a response before it times out. If API Gateway Explorer times out waiting for a response, the **Messaging System** filter fails. Enter the timeout value in milliseconds. The default value of `10000` means that API Gateway Explorer waits for a response for 10 seconds. The accepted range of values is `10000`–`20000` ms.
- **Selector for response**:
  If **Wait with timeout (ms)** is selected, you can enter a selector expression that specifies a response message. The expression entered specifies the messages that the consumer is interested in receiving. By using a selector, the task of filtering the messages is performed by the JMS provider instead of by the consumer.

  The selector is a string that specifies an expression whose syntax is based on the SQL92 conditional expression syntax. The API Gateway Explorer instance only receives messages whose headers and properties match the selector. For more details on selectors, see *Select configuration values at runtime*.

Important

The JMS consumer automatically returns the results of the invoked policy to the JMS destination specified in the `JMSReplyTo` header in the request. This means that you do not need to send a reply using the **Send to JMS** filter.

If the incoming JMS message contains a `JMSReplyTo` header, the queue or topic expects a response. So when the JMS consumer policy completes, API Gateway Explorer sends a message to the `JMSReplyTo` source in reverse. For example, the consumer reads the JMS message, and populates an attribute with a value inferred from the message type to Java (for example, from `TextMessage` to `String`). When the policy completes, the consumer looks up this attribute, an infers the JMS response message type based on the object type stored in the message.

Rewrite URL

Overview

You can use the **Rewrite URL** filter to specify the path on the remote machine to send the request to. This filter normally used in conjunction with a **Static Router** filter, whose role is to supply the host and port of the remote service. For more details, see the *Static router* topic.

Configuration

Configure the following fields on the **Rewrite URL** filter configuration window:

Name:
Enter an appropriate name for the filter in the **Name** field.

URL:
Enter the relative path of the web service in the **URL** field. API Gateway Explorer combines the specified path with the host and port number specified in the **Static Router** filter to build up the complete URL to route to.

Alternatively, you can perform simple URL rewrites by specifying a fully qualified URL into the **URL** field. You can then use a **Dynamic Router** to route the message to the specified URL.

Route to SMTP

Overview

You can use the **SMTP** filter to relay messages to an email recipient using a configured SMTP server.

General settings

Complete the following general settings:

Name:
Specify a descriptive name for this SMTP server.

SMTP Server Settings:
Click the browse button and select a preconfigured SMTP server in the tree.

Message settings

Complete the following fields in the **Message settings** section:

To:
Enter the email address of the recipients of the messages. You can enter multiple addresses by separating each one using a semicolon. For example:

```
joe.soap@example.com;joe.bloggs@example.com;john.doe@example.com
```

From:
Enter the email address of the senders of the messages. You can enter multiple addresses by separating each one using a semicolon.

Subject:
Enter some text as the subject of the email messages.

Send content in body:
Select this option to send the message content in the body of the message. This is selected by default.

Send content as attachment:
Select this option to send the message content as an attachment.

Send content in body and as attachment:
Select this option to send the message content in the body of the message and as an attachment.

Attachment name:
If you selected **Send content as attachment** or **Send content in body and as attachment**, enter a name for the attachment in this field. The default is `${id}.bin`. For more details, see *Select configuration values at runtime*.

Static router

Overview

API Gateway Explorer uses the information configured in the **Static Router** filter to connect to a machine that is hosting a web service. You should use the **Static Router** filter in conjunction with a **Rewrite URL** filter to specify the path to send the message to on the remote machine. For more details, see the *Rewrite URL* topic.

Configuration

You must configure the following fields must be configured on the **Static Router** configuration window:

Name:
Enter a name for the filter.

Host:
Enter the host name or IP address of the remote machine that is hosting the destination Web service.

Port:
Enter the port on which the remote service is listening.

HTTP:
Select this option if API Gateway Explorer should send the message to the remote machine over plain HTTP.

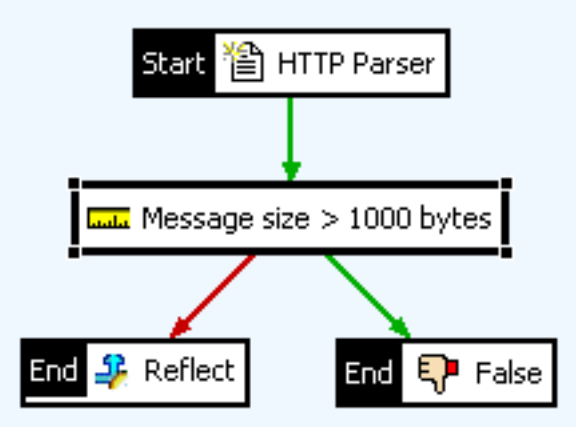
HTTPS:
Select this option if API Gateway Explorer should send the message to the remote machine over a secure channel using SSL. You can use a **Connection** filter to configure API Gateway Explorer to mutually authenticate to the remote system.

False filter

Overview

You can use the **False** filter to force a path in the policy to return false. This can be useful to create a *false positive* path in a policy.

The following policy parses the HTTP request and then runs a **Message Size** filter on the message to make sure that the message is no larger than 1000 bytes. To make sure that the message cannot be greater than this size, you can connect a **False** filter to the *success* path of the **Message Size** filter. This means that an exception is raised if a message exceeds 1000 bytes in size.



Configuration

Enter a name for the filter in the **Name** field.

Find certificate

Overview

The **Find Certificate** filter locates a certificate and sets it in the message for use by other certificate-based filters. Certificates can be extracted from the user store, message attributes, HTTP headers, or attachments.

Configuration

By default, API Gateway Explorer stores the extracted certificate in the `certificate` message attribute. However, it can store the certificate in any message attribute, including any arbitrary attribute (for example, `user_certificate`). The certificate can be extracted from this attribute by a successor filter in the policy.

Name:
Enter an appropriate name for the filter.

Attribute Name:
Enter or select the name of the message attribute to store the extracted certificate in. When the target message attribute has been selected, the next step is to specify the location of the certificate from one of the following options.

Certificate Store:
Click the **Select** button, and select a certificate from the certificate store.

User or Wildcard:
This field represents an alternative way to specify what user certificate is used. An explicitly named user certificate is used, or you can specify a selector to locate a user name or DName, which can then be used to locate the certificate.

Selector Expression:
You can specify a selector by enclosing the message attribute that contains the user name or DName in curly brackets, and prefixing this with $. For example:

```
${authentication.subject.id}
```

This selector means that API Gateway Explorer uses the certificate belonging to the subject of the authentication event in subsequent certificate-related filters. The certificate is set to the `certificate` message attribute. Using selectors is a more flexible way of locating certificates than specifying the user directly. For more details on selectors, see *Select configuration values at runtime*.

HTTP Header Name:
Enter the name of the HTTP header that contains the certificate.

Attachment Name:
Enter the name of the attachment (`Content-Id`) that contains the certificate. Alternatively, you can enter a selector in this field to represent the value of a message attribute.

Certificate Alias or Wildcard:
Enter the alias name of the certificate. Alternatively, you can enter a selector to represent the value of a message attribute. For more details on selectors, see *Select configuration values at runtime*.

Pause processing

Overview

The **Pause** filter is mainly used for testing purposes. A **Pause** filter causes a policy to sleep for a specified amount of time.

Configuration

Enter an appropriate name for the filter in the **Name** field. When the filter is executed in a policy, it sleeps for the time specified in the **Pause for** field. The sleep time is specified in milliseconds.

Scripting language filter

Overview

The **Scripting Language** filter uses the Java Specification Request (JSR) 223 [http://java.sun.com/developer/technicalArticles/J2SE/Desktop/scripting/] to embed a scripting environment in the API Gateway Explorer core engine. This enables you to write bespoke JavaScript or Groovy code to interact with the message as it is processed by the API Gateway Explorer. You can get, set, and evaluate specific message attributes with this filter.

Some typical uses of the **Scripting Language** filter include the following:

- Check the value of a specific message attribute
- Set the value of a message attribute
- Remove a message attribute
- DOM processing on the XML request or response

Write a script

To write a script filter, you must implement the `invoke()` method. This method takes a `com.vordel.circuit.Message` object as a parameter and returns a boolean result.

The API Gateway Explorer provides a **Script Library** that contains a number of prewritten `invoke()` methods to manipulate specific message attributes. For example, there are `invoke()` methods to check the value of the `SOAPAction` header, remove a specific message attribute, manipulate the message using the DOM, and assign a particular role to a user.

You can access the script examples provided in the **Script library** by clicking the **Show script library** button on the filter's main configuration window.

Important

When writing the JavaScript or Groovy code, you should note the following:

- You must implement the `invoke()` method. This method takes a `com.vordel.circuit.Message` object as a parameter, and returns a boolean.
- You can obtain the value of a message attribute using the `getProperty` method of the `Message` object.
- Do not perform attribute substitution as follows:

  ```
  msg.get("my.attribute.a") == msg.get("my.attribute.b")
  ```

  This is not thread safe and can cause performance issues.

Use local variables

The API Gateway Explorer is a multi-threaded environment, therefore, at any one time multiple threads can be executing code in a script. When writing JavaScript or Groovy code, always declare variables locally using `var`. Otherwise, the variables are global, and global variables can be updated by multiple threads.

For example, always use the following approach:

```
var myString = new java.lang.String("hello word");
for (var i = 100; i < 100; i++) {
```

```
    java.lang.System.out.println(myString + java.lang.Integer.toString(i));
}
```

Do not use the following approach:

```
myString = new java.lang.String("hello word");
for (i = 100; i < 100; i++) {
    java.lang.System.out.println(myString + java.lang.Integer.toString(i));
}
```

Using the second example under load, you cannot guarantee which value is output because both of the variables (`myString` and `i`) are global.

Add your script JARs to the classpath

You must add your custom JavaScript or Groovy JAR files to your API Gateway Explorer classpath and to the list of runtime dependencies in Policy Studio.

Add your script JARs to the API Gateway Explorer classpath

Because the scripting environment is embedded in the API Gateway Explorer engine, it has access to all Java classes on the API Gateway Explorer classpath, including all JRE classes. If you wish to invoke a Java object, you must place its corresponding class file on the API Gateway Explorer classpath. The recommended way to add classes to the API Gateway Explorer classpath is to place them (or the JAR files that contain them) in the `INSTALL_DIR/ext/lib` folder. For more details, see the `readme.txt` in this folder.

Add your script JARs to Policy Studio

Your custom JavaScript or Groovy script JARs must also be compiled and validated in Policy Studio. To add your JARs files to the list of runtime dependencies in Policy Studio, perform the following steps:

1. In the Policy Studio main menu, select **Window** > **Preferences** > **Runtime Dependencies**.
2. Click **Add** to select your script JAR file(s) and any other required third-party JARs.
3. Click **Apply** when finished. Copies of these JAR files are added to the `plugins` directory in your Policy Studio installation.
4. You must restart Policy Studio and the server for these changes to take effect. You should restart Policy Studio using the `policystudio -clean` command.

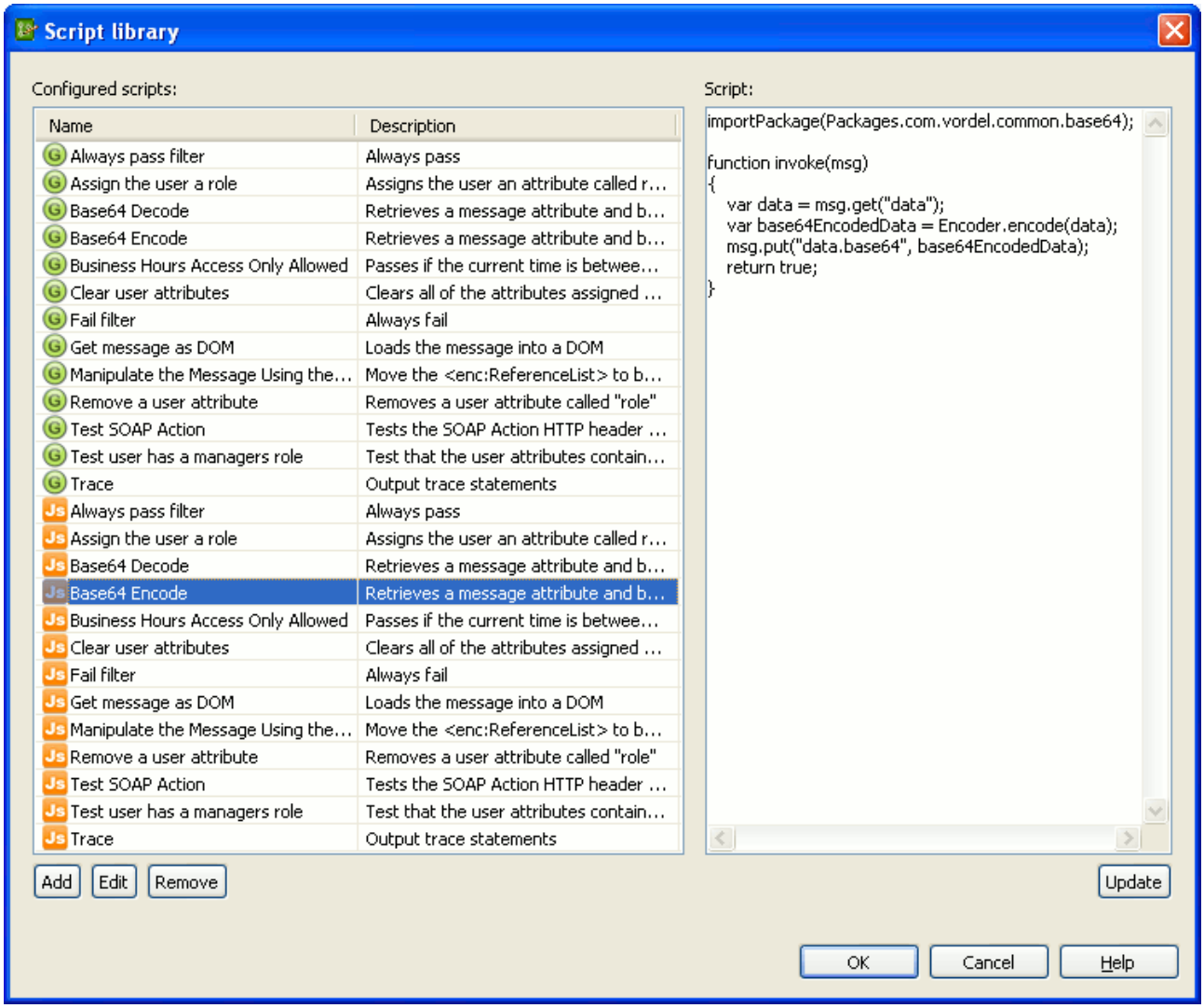
Configure a script filter

You can write or edit the JavaScript or Groovy code in the text area on the **Script** tab. A JavaScript function skeleton is displayed by default. Use this skeleton code as the basis for your JavaScript code. You can also load an existing JavaScript or Groovy script from the **Script library** by clicking the **Show script library** button.

On the **Script library** dialog, click any of the **Configured scripts** in the table to display the script in the text area on the right. You can edit a script directly in this text area. Make sure to click the **Update** button to store the updated script to the **Script library**.

Add a script to the library

You can add a new script to the library by clicking the **Add** button, which displays the **Script Details** dialog. Enter a **Name** and a **Description** for the new script in the fields provided. By default, the **Language** field is set to JavaScript, but you can also select Groovy from the drop-down list. You can then write the script in the **Script** text area.



Note

Regular expressions specified in **Scripting Language** filters use the regular expression engine of the relevant scripting language. For example, JavaScript-based filters use JavaScript regular expressions. This includes regular expressions inside XSDs defined by the W3C XML Schema standard. Other API Gateway Explorer filters that use regular expressions are based on `java.util.regex.Pattern`, unless stated otherwise.

Test Case Shortcut

Overview

The **Test Case Shortcut** filter allows you to re-use the functionality of one Test Case from another. So, for example, it is possible to create one Test Case called "Security Tokens" that inserts various security tokens into the message. Another Test Case could be created called "Add HTTP Headers", which adds a series of HTTP headers to a message. It would then be possible to create a third Test Case that calls the other 2 Test Cases using **Test Case Shortcut** filters.

Configuration

Complete the following fields to configure the **Test Case Shortcut** filter:

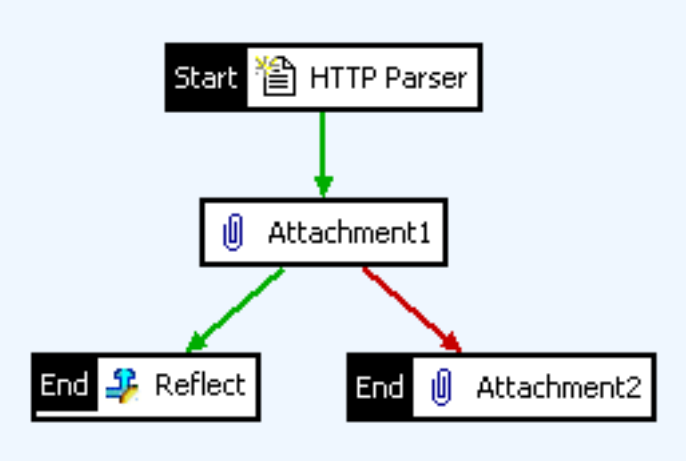
Name:
Enter a friendly name for the filter here.

Test Case Shortcut:
Select another Test Case that you want to re-use from the tree. The Test Case in which this **Test Case Shortcut** filter is placed will call into the selected Test Case when it is run.

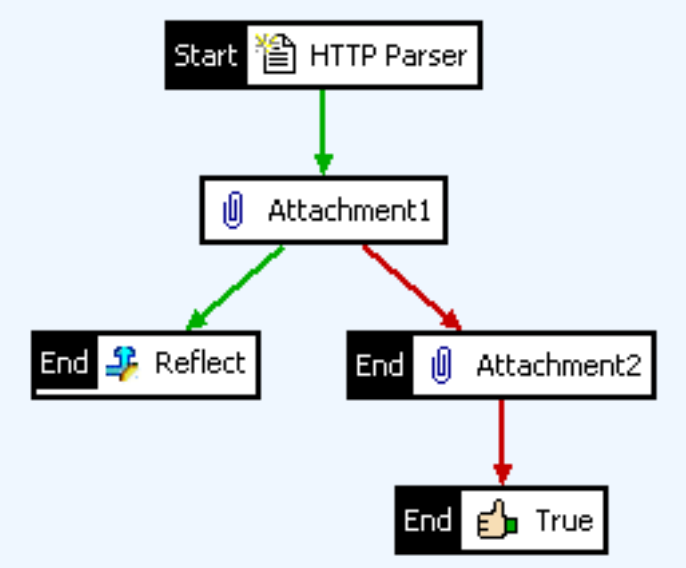
True filter

Overview

You can use the **True** filter to force a path in a policy to return true. For example, this can be useful to prevent a path from ending on a false case and consequently throwing an exception. The following policy parses the HTTP request, and then runs **Attachment1** on the message. If **Attachment1** passes, the message is echoed back to the client by the **Reflect** filter. However, if **Attachment1** fails, the **Attachment2** filter is run on the message. Because this is an *end* node, if this filter fails, an exception is thrown.



By adding a **True** filter to the **Attachment2** filter, this path always ends on a true case, and so does not throw an exception if **Attachment2** fails.



Configuration

Enter an appropriate name for the filter in the **Name** field.

Retrieve WSDL files from a UDDI registry

Overview

You can use WSDL files to register web services in the **Web Service Repository** and to add WSDL documents and XML schemas to the global cache. Policy Studio can retrieve a WSDL file from the file system, from a URL, or from a UDDI registry. This topic explains how to retrieve a WSDL file from a UDDI registry.

You can also browse a UDDI registry in Policy Studio directly without registering a WSDL file. Under the **Business Services** node in the tree, right-click the **Web Service Repository** node, and select **Browse UDDI Registry**.

UDDI concepts

Universal Description, Discovery and Integration (UDDI) is an OASIS-led initiative that enables businesses to publish and discover Web services on the Internet. A business publishes services that it provides to a public XML-based registry so that other businesses can dynamically look up the registry and discover these services. Enough information is published to the registry to enable other businesses to find services and communicate with them. In addition, businesses can also publish services to a private or semi-private registry for internal use.

A business registration in a UDDI registry includes the following components:

- **Green Pages**:
  Contains technical information about the services exposed by the business
- **Yellow Pages**:
  Categorizes the services according to standard taxonomies and categorization systems
- **White Pages**:
  Gives general information about the business, such as name, address, and contact information

You can search the UDDI registry according to a whole range of search criteria, which is of key importance to Policy Studio. You can search the registry to retrieve the WSDL file for a particular service. Policy Studio can then use this WSDL file to create a policy for the service, or to extract a schema from the WSDL to check the format of messages attempting to use the operations exposed by the Web service.

For a more detailed description of UDDI, see the UDDI specification. In the meantime, the next section gives high-level definitions of some of the terms displayed in the Policy Studio interface.

UDDI definitions

Because UDDI terminology is used in Policy Studio windows, such as the **Import WSDL** wizard, the following list of definitions explains some common UDDI terms. For more detailed explanations, see the UDDI specification.

businessEntity
This represents all known information about a particular business (for example, name, description, and contact information). A `businessEntity` can contain a number of `businessService` entities. A `businessEntity` may have an `identifierBag`, which is a list of name-value pairs for identifiers, such as Data Universal Numbering System (DUNS) numbers or taxonomy identifiers. A `businessEntity` may also have a `categoryBag`, which is a list of name-value pairs used to tag the `businessEntity` with classification information such as industry, product, or geographic codes. There is no mapping for a WSDL item to a `businessEntity`. When a WSDL file is published, you must specify a `businessEntity` for the `businessService`.

businessService
A `businessService` represents a logical service classification, and is used to describe a Web service provided by a business. It contains descriptive information in business terms outlining the type of technical services found in each `businessService` element. A `businessService` may have a `categoryBag`, and may contain a number of `bindingTemplate` entities. In the WSDL to UDDI mapping, a `businessService` represents a `wsdl:service`. A `busi-`

nessService has a businessEntity as its parent in the UDDI registry.

bindingTemplate
A bindingTemplate contains pointers to the technical descriptions and the access point URL of the Web service, but does not contain the details of the service specification. A bindingTemplate may contain references to a number of tModel entities, which do include details of the service specification. In the WSDL to UDDI mapping, a bindingTemplate represents a wsdl:port.

tModel
A tModel is a Web service type definition, which is used to categorize a service type. A tModel consists of a key, a name, a description, and a URL. tModels are referred to by other entities in the registry. The primary role of the tModel is to represent a technical specification (for example, WSDL file). A specification designer can establish a unique technical identity in a UDDI registry by registering information about the specification in a tModel. Other parties can express the availability of Web services that are compliant with a specification by including a reference to the tModel in their bindingTemplate data.

This approach facilitates searching for registered Web services that are compatible with a particular specification. tModels are also used in identifierBag and categoryBag structures to define organizational identity and various classifications. In this way, a tModel reference represents a relationship between the keyed name-value pairs to the supername, or namespace in which the name-value pairs are meaningful. A tModel may have an identifierBag and a categoryBag. In the WSDL to UDDI mapping, a tModel represents a wsdl:binding or wsdl:portType.

Identifier
The purpose of identifiers in a UDDI registry is to enable others to find the published information using more formal identifiers such as DUNS numbers, Global Location Numbers (GLN), tax identifiers, or any other kind of organizational identifiers, regardless of whether these are private or shared.

The following are identification systems used commonly in UDDI registries:

Identification System	Name	tModel Key
Dun and Bradstreet D-U-N-S Number	dnb-com:D-U-N-S	uuid:8609C81E-EE1F-4D5A-B202-3EB13AD01823
Thomas Registry Suppliers	thomasregister-com:supplierID	uuid:B1B1BAF5-2329-43E6-AE13-BA8E97195039

Category
Entities in the registry may be categorized according to categorization system defined in a tModel (for example, geographical region). The businessEntity, businessService, and tModel types have an optional categoryBag. This is a collection of categories, each of which has a name, value, and tModel key.

The following are categorization systems used commonly in UDDI registries:

Categorization System	Name	tModel Key
UDDI Type Taxonomy	uddi-org:types	uuid:C1ACF26D-9672-4404-9D70-39B756E62AB4
North American Industry Classification System (NAICS) 1997 Release	ntis-gov:naics:1997	uuid:C0B9FE13-179F-413D-8A5B-5004DB8E5BB2

Example tModel mapping for WSDL portType

The following shows an example `tModel` mapped for a WSDL `portType`:

```
<tModel tModelKey="uuid:e8cf1163-8234-4b35-865f-94a7322e40c3" >
    <name>
       StockQuotePortType
    </name>
    <overviewDoc>
         <overviewURL>
             http://location/sample.wsdl
         <overviewURL>
    <overviewDoc>

    <categoryBag>
         <keyedReference
             tModelKey="uuid:d01987d1-ab2e-3013-9be2-2a66eb99d824"
         keyName="portType namespace"
             keyValue="http://example.com/stockquote/" />
         <keyedReference
             tModelKey="uuid:6e090afa-33e5-36eb-81b7-1ca18373f457"
         keyName="WSDL type"
             keyValue="portType" />
    </categoryBag>
</tModel>
```

In this example, the `tModel` name is the same as the local name of the WSDL `portType` (in this case, `StockQuote-PortType`), and the `overviewURL` links to the WSDL file. The `categoryBag` specifies the WSDL namespace, and shows that the `tModel` is for a `portType`.

Configure a registry connection

You first need to select the UDDI registry that you want to search for WSDL files. Complete the following fields to select or add a UDDI registry:

Select Registry:
Select an existing UDDI registry to browse for WSDL files from the **Registry** drop-down list. To configure the location of a new UDDI registry, click **Add**. Similarly, to edit an existing UDDI registry location, click **Edit**. Then configure the fields in the **Registry Connection Details** dialog. For more details, see *Connect to a UDDI registry*.

Select Locale:
You can select an optional language locale from this list. The default is `No Locale`.

WSDL search

When you have configured a UDDI registry connection, you can search the registry using a variety of different search options on the **Search** tab. **WSDL Search** is the default option. This enables you to search for the UDDI entries that the WSDL file is mapped to. You can also do this using the **Advanced Search** option. The following different types of WSDL searches are available:

WSDL portType (UDDI tModel):
Searches for a `uddi:tModel` that corresponds to a `wsdl:portType`. You can enter optional search criteria for specific categories in the `uddi:tModel` (for example, **Namespace of portType**).

WSDL binding (UDDI tModel):
Searches for a `uddi:tModel` that corresponds to a `wsdl:binding`. You can enter optional search criteria for specific categories in the `uddi:tModel` (for example, **Name of binding**, or **Binding Transport Type**).

WSDL service (UDDI businessService):

Searches for a `uddi:businessService` that corresponds to a `wsdl:service`. You can enter optional search criteria for specific categories in the `uddi:businessService` (for example, **Namespace of service**).

WSDL port (UDDI bindingTemplate):
Searches for a `uddi:bindingTemplate` that corresponds to a `wsdl:port`. This search is more complex because a `serviceKey` is required to find a `uddi:bindingTemplate`. This means that at least two queries are carried out, first to find the `uddi:businessService`, and another to find the `uddi:bindingTemplate`.

For example, a `bindingTemplate` contains a reference to the `tModel` for the `wsdl:portType`. You can use the `tModel` key to find all implementations (`bindingTemplates`) for that `wsdl:portType`. The search looks for `businessServices` that have `bindingTemplates` that refer to the `tModel` for the `wsdl:portType`. Then with the `serviceKey`, you can find the `bindingTemplate` that refers to the `tModel` for the `wsdl:portType`.

In all cases, click **Next** to start the WSDL search. The **Search Results** tree shows the `tModel` URIs as top-level nodes. These URIs are all WSDL URIs, and you can use these to generate policies on import by selecting the URI, and clicking the **Finish** button.

You can click any of the nodes in the tree to display detailed properties about that node in the table below the **Search Results** tree. The properties listed depend on the type of the node that is selected. You can also right-click a node to edit it (for example, add a description, add a category or identifier node, or delete a duplicate node).

Quick search

The **Quick Search** option enables you to search the UDDI registry for a specific `tModel` name or category.

tModel Name:
You can enter a **tModel Name** for a fine-grained search. This is a partial or full name pattern with wildcard searching as specified by the *SQL-92 LIKE* specification. The wildcard characters are percent %, and underscore _, where an underscore matches any single character and a percent matches zero or more characters.

Categories:
You can select one of the following options to search by:

wsdlSpec	Search for `tModels` classified as `wsdlSpec` (`uddi-org:types` category set to `wsdlSpec`). This is the default.
Reusable WS-Policy Expressions	Search for `tModels` classified as reusable WS-Policy Expressions.
All	Search for all `tModels`.

Click **Next** to start the search. The **Search Results** tree shows the `tModel` URIs as top-level nodes. These URIs are all WSDL URIs, and you can use these to generate policies on import by selecting the URI, and clicking the **Finish** button.

You can click any of the nodes in the tree to display detailed properties about that node in the table below the **Search Results** tree. The properties listed depend on the type of the node that is selected. You can also right-click a node to edit it (for example, add a description, add a category or identifier node, or delete a duplicate node).

Name search

The **Name Search** option enables you to search for a `businessEntity`, `businessService` or `tModel` by name. In the **Select Registry Data Type**, select one of the following UDDI entity levels to search for:

- **businessEntity**
- **businessService**
- **tModel**

You can enter a name in the **Name** field to narrow the search. You can also use wildcards in the name. The name applies to a `businessEntity`, `businessService`, or `tModel`, depending on which registry entity type has been selected. If no name is entered, all entities of the selected type are retrieved.

Click the **Search** button to start the search. The search results display the matching entities in the tree. For example, if you enter `MyTestBusiness` for **Name**, and select **businessEntity**, this searches for a `businessEntity` with the name `MyTestBusiness`. Child nodes of the matching `businessEntity` nodes are also shown. `tModels` are displayed in the results if any child nodes of the `businessEntity` refer to `tModels`. Only referred to `tModels` are displayed. The same applies if you search for a `businessService`. If you select `tModel`, and search for `tModels`, only `tModels` are displayed.

Important

The `tModel` URIs shown in the resulting tree may not all be categorized as `wsdlSpec` according to the `uddi-org:types` categorization system. You can choose to load any of these URIs as a WSDL file, but you are warned if it is not categorized as `wsdlSpec`.

As before, you can click any node in the results tree to display properties about that node in the table. You can also right-click a node to edit it (for example, add a description, add a category or identifier node, or delete a duplicate node).

UDDI v3 name searches

By default, a UDDI v3 name search is an exact match. To perform a search using wildcards (for example, %, _, and so on), you must select the **approximateMatch** find qualifier in the **Advanced Options** tab. This applies to anywhere you enter a name for search purposes (for example, **Name Search**, **Quick Search**, and **Advanced Search**).

Advanced search

The **Advanced Search** option enables you to search the UDDI registry using any combination of **Names**, **Keys**, **tModels**, **Discovery URLs**, **Categories**, and **Identifiers**. You can also specify the entity level to search for in the tree. All of these options combine to provide a very powerful search facility. You can specify search criteria for any of the categories listed above by right-clicking the folder node in the **Enter Search Criteria** tree, and selecting the **Add** menu option. You can enter more than one search criteria of the same type (for example, two **Key** search criteria).

Important

The `tModel` URIs shown in the resulting tree may not all be categorized as `wsdlSpec` according to the `uddi-org:types` categorization system. You can choose to load any of these URIs as a WSDL file, but you are warned if it is not categorized as `wsdlSpec`.

The following options enable you to add a search criteria for each of the types listed in the **Enter Search Criteria** tree. All search criteria are configured by right-clicking the folder node, and selecting the **Add** menu option.

Names:
Enter a name to be used in the search in the **Name** field in the **Name Search Criterion** dialog. For example, the name could be the **businessEntity** name. The name is a partial or full name pattern with wildcards allowed as specified by the *SQL-92 LIKE* specification. The wildcard characters are percent %, and underscore _, where an underscore matches any single character and a percent matches zero or more characters. A name search criterion can be used for `businessEntity`, `businessService`, and `tModel` level searches.

Keys:
In the **Key Search Criterion** dialog, you can specify a key to search the registry for in the **Key** field. The key value is a Universally Unique Identifier (UUID) value for a registry object. You can use the **Key Search Criterion** on all levels of searches. If one or more keys are specified with no other search criteria, the keys are interpreted as the keys of the selected type of registry object and used for a direct lookup, instead of a find/search operation. For example, if you enter

key1 and key2, and select the businessService entity type, the search retrieves the businessService object with key key1, and another businessService with key key2.

If you enter a key with other search criteria, a key criterion is interpreted as follows:

- For a businessService entity lookup, the key is the businessKey of the services.
- For a bindingTemplate entity lookup, the key is the serviceKey of the binding templates.
- Not applicable for any other object type.

tModels:
You can enter a key in the **tModel Key** field on the **tModel Search Criterion** window. The key entered should correspond to the UUID of the tModel associated with the type of object you are searching for. A tModel search criterion may be used for businessEntity, businessService, and bindingTemplate level searches.

Discovery URLs:
Enter a URL in the **Discovery URL** field on the **Discovery URL Search Criterion** dialog. The **Use Type** field is optional, but can be used to further fine-grain the search by type. You can use a Discovery URL search criterion for businesEntity level searches only.

Categories:
Select a previously configured categorization system from the **Type** drop-down list in the **Category Search Criterion** dialog. This pre-populated with a list of common categorization systems. You can add a new categorization system using the **Add** button.

In the **Add/Edit Category** dialog, enter a **Name**, **Description**, and **UUID** for the new category type in the fields provided. When the categorization system has been selected or added, enter a value to search for in the **Value** field. The **Name** field is optional.

Identifiers:
Select a previously configured identification system from the **Type** drop-down list in the **Identifier Search Criterion** dialog. This is pre-populated with well-known identification systems. To add a new identification system, click the **Add** button.

In the **Add/Edit Identifier** dialog, enter a **Name**, **Description**, and **UUID** for the new identifier in the fields provided.

Select Registry Data Type:
Select one of the following UDDI entity levels to search for:

- **businessEntity**
- **businessService**
- **bindingTemplate**
- **tModel**

The search also displays child nodes of the matching nodes. tModels are also returned if they are referred to.

Advanced options

This tab enables you to configure various aspects of the search conditions specified on the previous tabs. The following options are available:

UDDI Find Qualifier:	Description:
andAllKeys	By default, identifier search criteria are ORed together. This setting ensures that they are ANDed instead. This is already the default for categoryBag and tModelBag.
approximateMatch (v3)	This applies to a UDDI v3 registry only. Specifies wildcard searching as defined

UDDI Find Qualifier:	Description:
	by the `uddi-org:approximatematch:SQL99 tModel`, which means approximate matching where percent sign (%) indicates any number of characters, and underscore (_) indicates any single character. The backslash character (\) is an escape character for the percent sign, underscore and backslash characters. This option adjusts the matching behavior for `name`, `keyValue` and `keyName` (where applicable).
binarySort (v3)	This applies to a UDDI v3 registry only. Enables greater speed in sorting, and causes a binary sort by name, as represented in Unicode codepoints.
bindingSubset (v3)	This applies to a UDDI v3 registry only. Specifies that the search uses only `categoryBag` elements from contained `bindingTemplate` elements in the registered data, and ignores any entries found in the `categoryBag` that are not direct descendents of registered `businessEntity` or `businessService` elements.
caseInsensitiveMatch (v3)	This applies to a UDDI v3 registry only. Specifies that that the matching for `name`, `keyValue` and `keyName` (where applicable) should be performed without regard to case.
caseInsensitiveSort (v3)	This applies to a UDDI v3 registry only. Specifies that the result set should be sorted without regard to case. This overrides the default case sensitive sorting behavior.
caseSensitiveMatch (v3)	This applies to a UDDI v3 registry only. Specifies that that the matching for `name`, `keyValue` and `keyName` (where applicable) should be case sensitive. This is the default behavior.
caseSensitiveSort (v3)	This applies to a UDDI v3 registry only. Specifies that the result set should be sorted with regard to case. This is the default behavior.
combineCategoryBags	Makes the `categoryBag` entries of a `businessEntity` behave as if all `categoryBags` found at the `businessEntity` level and in all contained or referenced `businessServices` are combined. Searching for a category yields a positive match on a registered business if any of the `categoryBags` contained in a `businessEntity` (including the `categoryBags` in contained or referenced `businessServices`) contain the filter criteria.
diacriticInsensitiveMatch (v3)	This applies to a UDDI v3 registry only. Specifies that matching for `name`, `keyValue` and `keyName` (where applicable) should be performed without regard to diacritics. Support for this qualifier by nodes is optional.
diacriticSensitiveMatch (v3)	This applies to a UDDI v3 registry only. Specifies that matching for `name`, `keyValue` and `keyName` (where applicable) should be performed with regard to diacritics. This is the default behavior.
exactMatch (v3)	This applies to a UDDI v3 registry only. Specifies that only entries with `name`, `keyValue` and `keyName` (where applicable) that exactly match the name argument passed in, after normalization, are returned. This qualifier is sensitive to case and diacritics where applicable. This is the default behavior.
exactNameMatch (v2)	This applies to a UDDI v2 registry only. Specifies that the name entered as part of the search criteria must exactly match the name specified in the UDDI registry.
orAllKeys	By default, `tModel` and category search criteria are ANDed. This setting ORs these criteria instead.
orLikeKeys	When a bag container contains multiple `keyedReference` elements (`categoryBag` or `identifierBag`), any `keyedReference` filters from the same namespace (for example, with the same `tModelKey` value) are ORed together rather than ANDed. For example, this enables you to search for `any of these four values from this namespace, and any of these`

UDDI Find Qualifier:	Description:
	`two values from this namespace`.
serviceSubset	Causes the component of the search that involves categorization to use only the `categoryBags` from directly contained or referenced `businessServices` in the registered data. The search results return only those businesses that match based on this modified behavior, in conjunction with any other search arguments provided.
signaturePresent (v3)	This applies to a UDDI v3 registry only. This restricts the result to entities that contain, or are contained in, an XML Digital `Signature` element. The `Signature` element should be verified by the client. This option, or the presence of a `Signature` element, should only be used to refine a search result, and should not be used as a verification mechanism by UDDI clients.
sortByDateAsc (v3)	This applies to a UDDI v3 registry only. Sorts the results alphabetically in order of ascending date.
sortByDateDsc (v3)	This applies to a UDDI v3 registry only. Sorts the results alphabetically in order of descending date.
sortByNameAsc	Sorts the results alphabetically in order of ascending name.
sortByNameDsc	Sorts the results alphabetically in order of descending name.
suppressProjectedServices (v3)	This applies to a UDDI v3 registry only. Specifies that service projections must not be returned when searching for services or businesses. This option is enabled by default when searching for a service without a `businessKey`.
UTS-10 (v3)	This applies to a UDDI v3 registry only. Specifies sorting of results based on the Unicode Collation Algorithm on elements normalized according to Unicode Normalization Form C. A sort is performed according to the Unicode Collation Element Table in conjunction with the Unicode Collation Algorithm on the name field, and normalized using Unicode Normalization Form C. Support for this qualifier by nodes is optional.

Publish

Click the **Publish** radio button to view the **Published UDDI Entities Tree View**. This enables you to manually publish UDDI entities to the specified UDDI registry (for example, `businessEntity`, `businessService`, `bindingTemplate`, and `tModel` entities). You must already have the appropriate permissions to write to the UDDI registry.

Add a businessEntity

To add a business, perform the following steps:

1. Right-click the tree view, and select **Add businessEntity**.
2. In the **Business** dialog, enter a **Name** and **Description** for the business.
3. Click **OK**.
4. You can right-click the new `businessEntity` node to add child UDDI entities in the tree (for example, `businessService`, `Category`, and `Identifier` entities).

Add a tModel

To add a `tModel`, perform the following steps:

1. Right-click the tree view, and select **Add tModel**.
2. In the **tModel** dialog, enter a **Name**, **Description**, and **Overview URL** for the `tModel`. For example, you can use the **Overview URL** to specify the location of a WSDL file.
3. Click **OK**.
4. You can right-click the new `tModel` node to add child UDDI entities in the tree (for example, `Category` and `Identifier` entities).

As before, you can click any node in the results tree to display properties about that node in the table. You can also right-click a node to edit it (for example, add a description, add a category or identifier node, or delete a duplicate node). At any stage, you can click the **Clear** button on the right to clear the entire contents of the tree. This does not delete the contents of the registry.

For more details on UDDI entities such as `businessEntity` and `tModel`, see the section called “UDDI definitions”. For details on how to publish web services automatically using a wizard, see ???.

Connect to a UDDI registry

Overview

This topic explains how to configure a connection to a UDDI registry in the **Registry Connection Details** dialog. It explains how to configure connections to UDDI v2 and UDDI v3 registries, and how to secure a connection over SSL.

Configure a registry connection

Configure the following fields in the **Registry Connection Details** dialog:

Registry Name:
Enter the display name for the UDDI registry.

UDDI v2:
Select this option to use UDDI v2.

UDDI v3:
Select this option to use UDDI v3.

Inquiry URL:
Enter the URL on which to search the UDDI registry (for example, `http://HOSTNAME:PORT/uddi/inquiry`).

Publish URL:
Enter the URL on which to publish to the UDDI registry, if required (for example, `http://HOSTNAME:PORT/uddi/publishing`).

Security URL (UDDI v3):
For UDDI v3 only, enter the URL for the security service, if required (for example, `http://HOSTNAME:PORT/uddi/security.wsdl`).

Important

For UDDI v3, the **Inquiry URL**, **Publish URL**, and **Security URL** specify the URLs of the WSDL for the inquiry, publishing, and security Web services that the registry exposes. These fields can use the same URL if the WSDL for each service is at the same URL.

For example, a WSDL file at `http://HOSTNAME:PORT/uddi/uddi_v3_registry.wsdl` can contain three URLs:

- `http://HOSTNAME:PORT/uddi/inquiry`
- `http://HOSTNAME:PORT/uddi/publishing`
- `http://HOSTNAME:PORT/uddi/security`

These are the service endpoint URLs that Policy Studio uses to browse and publish to the registry. These URLs are not set in the connection dialog, but discovered using the WSDL. However, for UDDI v2, WSDL is *not* used to discover the service endpoints, so you must enter these URLs directly in the connection dialog.

Max Rows:
Enter the maximum number of entries returned by a search. Defaults to `20`.

Registry Authentication:
The registry authentication settings are as follows:

Type	This optional field applies to UDDI v2 only. The only supported authentication

	type is `UDDI_GET_AUTHTOKEN`.
Username	Enter the user name required to authenticate to the registry, if required.
Password	Enter the password for this user, if required.

The user name and password apply to UDDI v2 and v3. These are generally required for publishing, but depend on the configuration on the registry side.

HTTP Proxy:
The HTTP proxy settings apply to UDDI v2 and v3:

Proxy Host	If the UDDI registry location entered above requires a connection to be made through an HTTP proxy, enter the host name of the proxy.
Proxy Port	If a proxy is required, enter the port on which the proxy server is listening.
Username	If the proxy has been configured to only accept authenticated requests, Policy Studio sends this user name and password to the proxy using HTTP Basic authentication.
Password	Enter the password to use with the user name specified in the field above.

HTTPS Proxy:
The HTTPS proxy settings apply to UDDI v2 and v3:

SSL Proxy Host	If the **Inquiry URL** or **Publish URL** uses the HTTPS protocol, the SSL proxy host entered is used instead of the HTTP proxy entered above. In this case, the HTTP proxy settings are not used.
Proxy Port	Enter the port that the SSL proxy is listening on.

Secure a connection to a UDDI registry

You can communicate with the UDDI registry over SSL. All communication may not need to be over SSL (for example, you may wish publish over SSL, and perform inquiry calls without SSL). For UDDI v2 and v3, you can use a mix of `http` and `https` URLs for WSDL and service address locations.

You can specify some or all of the **Inquiry URL**, **Publish URL**, and **Security URL** settings as `https` URLs. For example, with UDDI v3, you could use a single URL like the following:

```
https://HOSTNAME:PORT/uddi/wsdl/uddi_v3_registry.wsdl
```

If any URLs (WSDL or service address location) use `https`, you must configure the Policy Studio so that it trusts the registry SSL certificate.

Configure Policy Studio to trust a registry certificate

For an SSL connection, you must configure the registry server certificate as a trusted certificate. Assuming mutual authentication is not required, the simplest way to configure an SSL connection between Policy Studio and UDDI registry is to add the registry certificate to the Policy Studio default truststore (the `cacerts` file). You can do this by performing the

following steps in Policy Studio:

1. Select the **Certificates and Keys** > **Certificates** node in the Policy Studio tree.
2. Click **Create/Import**, and click **Import Certificate**.
3. Browse to the UDDI registry SSL certificate file, and click **Open**.
4. Click **Use Subject** on the right of the **Alias Name** field, and click **OK**. The registry SSL certificate is now imported into the certificate store, and must be added to the Java keystore.
5. Click **Keystore** on the **Certificate** window.
6. Click **Browse** next to the **Keystore** field.
7. Browse to the following file:
   `INSTALL_DIR/policystudio/jre/lib/security/cacerts`
8. Click **Open**, and enter the **Keystore password**. The default password is: `changeit`.
9. Click **Add to Keystore**.
10. Browse to the registry SSL certificate imported earlier, select it, and click **OK**.
11. Restart Policy Studio. You should now be able to connect to the registry over SSL.

Configure mutual SSL authentication

If mutual SSL authentication is required (if Policy Studio must authenticate to the registry), Policy Studio must have an SSL private key and certificate. In this case, you should create a keystore containing the Policy Studio key and certificate. You must configure Policy Studio to load this file. For example, edit the `INSTALL_DIR/policystudio/policystudio.ini` file, and add the following arguments:

```
-Djavax.net.ssl.keyStore=/home/oracle/osr-client.jks
-Djavax.net.ssl.keyStorePassword=changeit
```

This example shows an `osr-client.jks` keystore file used with Oracle Service Registry (OSR), which is the UDDI registry provided by Oracle.

Note

You can also use Policy Studio to create a new keystore (`.jks`) file. Click **New keystore** instead of browsing to the `cacerts` file as described earlier.

Configure XPath expressions

Overview

The API Gateway Explorer uses XPath expressions in a number of ways, for example, to locate an XML signature in a SOAP message, to determine what elements of an XML message to validate against an XML schema, to check the content of a particular element within an XML message, amongst many more uses.

There are two ways to configure XPath expressions:

- Manual configuration
- XPath wizard

Manual configuration

If you are already familiar with XPath and wish to configure the expression manually, complete the following fields, using the examples below if necessary:

1. Enter or select a name for the XPath expression in the **Name** drop-down list.
2. Enter the XPath expression to use in the **XPath Expression** field.
3. In order to resolve any prefixes within the XPath expression, the namespace mappings (**Prefix**, **URI**) should be entered in the table.

Consider the following example SOAP message: >

```
 <?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Header>
  <dsig:Signature xmlns:dsig="http://www.w3.org/2000/09/xmldsig#" id="sample">
      ..............
      ..............
      ..............
      ..............
  </dsig:Signature>
</soap:Header>
  <soap:Body>
    <prod:product xmlns:prod="http://www.oracle.com">
      <prod:name>SOA Product*</prod:name>
      <prod:company>Company</prod:company>
      <prod:description>WebServices Security</prod:description>
    </prod:product>
  </soap:Body>
</soap:Envelope>
```

The following XPath expression evaluates to true if the *<name>* element contains the value *API Gateway Explorer*:
XPath Expression: `//prod:name[text()='API Gateway Explorer']`

In this case, it is necessary to define a mapping for the *prod* namespace as follows:

Prefix	**URI**
`prod`	`http://www.oracle.com`

In another example, the element to be examined belongs to a default namespace. Consider the following SOAP message:

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Header>
  <dsig:Signature xmlns:dsig="http://www.w3.org/2000/09/xmldsig#" id="sample">
      ..............
      ..............
      ..............
      ..............
  </dsig:Signature>
</soap:Header>
  <soap:Body>
    <product xmlns="http://www.company.com">
      <name>SOA Product</name>
      <company>Company</company>
      <description>WebServices Security</description>
    </product>
  </soap:Body>
</soap:Envelope>
```

The following XPath expression evaluates to true if the <*company*> element contains the value *Company*:
XPath Expression: `//ns:company[text()='Company']`

The <*company*> element actually belongs to the default (`xmlns`) namespace (`http://www.company.com`. This means that it is necessary to make up an arbitrary prefix, `ns`, for use in the XPath expression and assign it to `http://www.company.com`. This is necessary to distinguish between potentially several default namespaces, which may exist throughout the XML message. The following mapping illustrates this:

Prefix	URI
`ns`	`http://www.oracle.com`

Return a nodeset

Both of the examples above dealt with cases where the XPath expression evaluated to a Boolean value. For example, the expression in the above example asks does the <`company`> element in the `http://www.oracle.com` namespace contain a text node with the value `oracle`?.

It is sometimes necessary to use the XPath expression to return a subset of the XML message. For example, when using an XPath expression to determine what nodes should be signed in a signed XML message, or when retrieving the nodeset to validate against an XML Schema.

The API Gateway Explorer ships with such an XPath expression: one that returns `All Elements inside SOAP Body` To view this expression, select it from the **Name** field. It appears as follows:
XPath Expression: `/soap:Envelope/soap:Body//*`

This XPath expression simply returns all child elements of the SOAP <`Body`> element. To construct and test more complicated expressions, administrators are advised to use the **XPath Wizard**.

XPath wizard

The XPath wizard assists administrators in creating correct and accurate XPath expressions. The wizard allows adminis-

trators to load an XML message and then run an XPath expression on it to determine what nodes are returned. To launch the XPath wizard, click the **XPath Wizard** button on the **XPath Expression** dialog.

To use the XPath wizard, simply enter (or browse to) the location of an XML file in the **File** field. The contents of the XML file will appear in the main window of the wizard. Enter an XPath expression in the **XPath** field and click the **Evaluate** button to run the XPath against the contents of the file. If the XPath expression returns any elements (or returns true), those elements will be highlighted in the main window.

If you are not sure how to write the XPath expression, you can select an element in the main window. An XPath expression to isolate this element is automatically generated and displayed in the **Selected** field. To use this expression, select the **Use this path** button, and click **OK**.

Signature location

Overview

A given XML message can contain several XML signatures. Consider an XML document (for example, a company policy approval form) that must be digitally signed by a number of users (for example, department managers) before being submitted to the ultimate web service (for example, a company policy approval web service). Such a message will contain several XML signatures by the time it is ready to be submitted to the web service.

In such cases, where multiple signatures will be present within a given XML message, it is necessary to specify which signature the API Gateway Explorer should use in the validation process. For more information on validating XML signatures, see *XML signature verification*.

Configuration

The API Gateway Explorer can extract the signature from an XML message using several different methods:

- WS-Security block
- SOAP message header
- Advanced (XPath)

Select the most appropriate method from the **Signature Location** field. Your selection will depend on the types of SOAP messages that you expect to receive. For example, if incoming SOAP messages will contain an XML signature within a WS-Security block, you should choose this option from the list.

Use WS-Security actors

If the signature is present in a WS-Security block:

1. Select `WS-Security block` from the **Signature Location** field.
2. Select a SOAP actor from the **Select Actor/Role(s)** field. Each actor uniquely identifies a separate WS-Security block. By selecting `Current actor/role only` from the list, the WS-Security block with no actor is taken.
3. In cases where there might be multiple signatures within the WS-Security block, it is necessary to extract one using the **Signature Position** field.

The following is a skeleton version of a message where the XML signature is contained in the *sample* WS-Security block, (`soap-env:actor="sample"`):

```
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
 <s:Header>
  <wsse:Security xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/04/secext"
       s:actor="sample">
   <dsig:Signature xmlns:dsig="http://www.w3.org/2000/09/xmldsig#" id="s1">
    ....
   </dsig:Signature>
  </wsse:Security>
 </s:Header>
 <s:Body>
  <ns1:getTime xmlns:ns1="urn:timeservice">
  </ns1:getTime>
 </s:Body>
</s:Envelope>
```

Use SOAP header

If the signature is present in the SOAP header:

1. Select `SOAP message header` from the **Signature Location** field.
2. If there is more than one signature in the SOAP header, then it is necessary to specify which signature the API Gateway Explorer should use. Specify the appropriate signature by setting the **Signature Position** field.

The following is an example of an XML message where the XML signature is contained within the SOAP header:

```
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
 <s:Header>
  <dsig:Signature xmlns:dsig="http://www.w3.org/2000/09/xmldsig#" id="s1">
      ....
  </dsig:Signature>
 </s:Header>
 <s:Body>
  <ns1:getTime xmlns:ns1="urn:timeservice">
  </ns1:getTime>
 </s:Body>
</s:Envelope>
```

Use XPath expression

Finally, an XPath expression can be used to locate the signature.

1. Select `Advanced (XPath)` from the **Signature Location** field.
2. Select an existing XPath expression from the list, or add a new one by clicking on the **Add** button. XPath expressions can also be edited or removed with the **Edit** and **Remove** buttons.

The default *First Signature* XPath expression takes the first signature from the SOAP header. The expression is as follows:

```
//s:Envelope/s:Header/dsig:Signature[1]
```

To edit this expression, click the **Edit** button to display the **Enter XPath Expression** dialog.

An example of a SOAP message containing an XML signature in the SOAP header is provided below. The following XPath expression instructs the API Gateway Explorer to extract the first signature from the SOAP header:

```
//s:Envelope/s:Header/dsig:Signature[1
```

Because the elements referenced in the expression (`Envelope` and `Signature`) are *prefixed* elements, you must define the namespace mappings for each of these elements as follows:

Prefix	**URI**
`s`	`http://schemas.xmlsoap.org/soap/envelope/`
`dsig`	`http://www.w3.org/2000/09/xmldsig#`

```
<?xml version="1.0" encoding="UTF-8"?>
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
 <s:Header>
  <dsig:Signature xmlns:dsig="http://www.w3.org/2000/09/xmldsig#" id="s1">
```

```
    ....
  </dsig:Signature>
 </s:Header>
 <s:Body>
  <product xmlns="http://www.oracle.com">
   <name>SOA Product*</name>
   <company>Company</company>
   <description>Web Services Security</description>
  </product>
 </s:Body>
</s:Envelope>
```

When adding your own XPath expressions, you must be careful to define any namespace mappings in a manner similar to that outlined above. This avoids any potential clashes that might occur where elements of the same name, but belonging to different namespaces are present in an XML message.

What to sign

Overview

The **What To Sign** section enables the administrator to define the exact content that must be signed for a SOAP message to pass the corresponding filter. The purpose of this configuration section is to ensure that the client has signed something meaningful (part of the SOAP message) instead of some arbitrary data that would pass a blind signature validation.

This prevents clients from simply pasting technically correct, but unrelated signatures into messages in the hope that they pass any blind signature verification. For example, the user may be able to generate a valid XML Signature over any arbitrary XML document. Then by including the signature and XML portion into a malicious SOAP message, the signature passes a blind signature validation, and the harmful XML is allowed to reach the Web service.

The **What To Sign** section ensures that clients must sign a part of the SOAP message, and therefore prevents them from pasting arbitrary XML Signatures into the message. This section enables you to use any combination of **Node Locations**, **XPath Expressions**, **XPath Predicates**, and/or **Message Attribute** to specify message content that must be signed. This topic describes how to configure each of the corresponding tabs displayed in this section.

ID configuration

With WSU IDs, an ID attribute is inserted into the root element of the nodeset that is to be signed. The XML Signature then references this ID to indicate to verifiers of the signature the nodes that were signed. The use of WSU IDs is the default option because they are WS-I compliant.

Alternatively, a generic ID attribute (that is not bound to the WSU namespace) can be used to dereference the data. The ID attribute is inserted into the top-level element of the nodeset to be signed. The generated XML Signature can then reference this ID to indicate what nodes were signed.

You can also use `AssertionID` attributes when signing SAML assertions. The following options provide more details and examples of the different styles of IDs that are available.

Use WSU IDs:
Select this option to reference the signed data using a `wsu:Id` attribute. In this case, a `wsu:Id` attribute is inserted into the root node of the nodeset that is signed. This id is then referenced in the generated XML Signature as an indication of what nodes were signed. The following example shows the correlation:

```
<s:Envelope xmlns:s="...">
  <s:Header>
  <wsse:Security xmlns:wsse="...">
    <dsig:Signature xmlns:dsig="..." Id="Id-00000112e2c98df8-0000000000000004">
      <dsig:SignedInfo>
        <dsig:CanonicalizationMethod
            Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
        <dsig:SignatureMethod
            Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
        <dsig:Reference URI="#Id-00000112e2c98df8-0000000000000003">
          <dsig:Transforms>
            <dsig:Transform
                Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
          </dsig:Transforms>
          <dsig:DigestMethod
              Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
          <dsig:DigestValue>xChPoiWJJrrPZkbXN8FPB8S4U7w=</dsig:DigestValue>
        </dsig:Reference>
      </dsig:SignedInfo>
      <dsig:SignatureValue>KG4N .... /9dw==</dsig:SignatureValue>
      <dsig:KeyInfo Id="Id-00000112e2c98df8-0000000000000005">
        <dsig:X509Data>
```

```
          <dsig:X509Certificate>
            MIID ... ZiBQ==
          </dsig:X509Certificate>
        </dsig:X509Data>
      </dsig:KeyInfo>
    </dsig:Signature>
  </wsse:Security>
  </s:Header>
   <s:Body xmlns:wsu="..." wsu:Id="Id-00000112e2c98df8-0000000000000003">
  <vs:getProductInfo xmlns:vs="http://ww.oracle.com">
   <vs:Name>API Gateway Explorer</vs:Name>
   <vs:Version>11.1.2.4.0</vs:Version>
  </vs:getProductInfo>
 </s:Body>
</s:Envelope>
```

In the above example, a `wsu:Id` attribute has been inserted into the `<s:Body>` element. This `wsu:Id` attribute is then referenced by the `URI` attribute of the `<dsig:Reference>` element in the actual Signature.

When the Signature is being verified, the value of the `URI` attribute can be used to locate the nodes that have been signed.

Use IDs:
Select this option to use generic IDs (that are not bound to the WSU namespace) to dereference the signed data. Under this schema, the `URI` attribute of the `<Reference>` points at an ID attribute, which is inserted into the top-level node of the nodeset that is signed. Take a look at the following example, noting how the ID specified in the Signature matches the ID attribute that has been inserted into the `<Body>` element, indicating that the Signature applies to the entire contents of the SOAP Body.

```
lt;soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Header>
    <dsig:Signature xmlns:dsig="http://www.w3.org/2000/09/xmldsig#"
                Id="Id-0000011a101b167c-0000000000000013">
      <dsig:SignedInfo>
        <dsig:CanonicalizationMethod
            Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
        <dsig:SignatureMethod
            Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
        <dsig:Reference URI="#Id-0000011a101b167c-0000000000000012">
          <dsig:Transforms>
            <dsig:Transform
                Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
          </dsig:Transforms>
          <dsig:DigestMethod
              Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
          <dsig:DigestValue>JCy0JoyhVZYzmrLrl92nxfr1+zQ=</dsig:DigestValue>
        </dsig:Reference>
      </dsig:SignedInfo>
      <dsig:SignatureValue>......<dsig:SignatureValue>
      <dsig:KeyInfo Id="Id-0000011a101b167c-0000000000000014">
        <dsig:X509Data>
          <dsig:X509Certificate>......</dsig:X509Certificate>
        </dsig:X509Data>
      </dsig:KeyInfo>
    </dsig:Signature>
  </soap:Header>
 <soap:Body Id="Id-0000011a101b167c-0000000000000012">
  <product version="11.1.2.4.0">
    <name>API Gateway Explorer</name>
    <company>Oracle</company>
    <description>SOA Security and Management</description>
  </product>
```

```
 </soap:Body>
</soap:Envelope>
```

Use SAML IDs for SAML Elements:
This ID option is specifically intended for use where a SAML assertion is to be signed. When this option is selected, an `AssertionID` attribute is inserted into a SAML 1.1 assertion, or a more generic ID attribute is used for a SAML 2.0 assertion.

Node locations

Node locations are perhaps the simplest way to configure the message content that must be signed. The table on this screen is pre-populated with a number of common SOAP security headers, including the SOAP Body, WS-Security block, SAML assertion, WS-Security UsernameToken and Timestamp, and the WS-Addressing headers. For each of these headers, there are several namespace options available. For example, you can sign both a SOAP 1.1 and/or a SOAP 1.2 block by distinguishing between their namespaces.

On the **Node Locations** tab, you can select one or more nodesets to sign from the default list. You can also add more default nodesets by clicking the **Add** button. Enter the **Element Name**, **Namespace**, and **Index** of the nodeset in the fields provided. The **Index** field is used to distinguish between two elements of the same name that occur in the same message.

XPath configuration

You can use an XPath expression to identify the nodeset (the series of elements) that must be signed. To specify that nodeset, select an existing XPath expression from the table, which contains several XPath expressions that can be used to locate nodesets representing common SOAP security headers, including SAML assertions. Alternatively, you can add a new XPath expression using the **Add** button. XPath expressions can also be edited and removed with the **Edit** and **Remove** buttons.

An example of a SOAP message is provided below. The following XPath expression indicates that all the contents of the SOAP body, including the `Body` element itself, should be signed:

```
/soap:Envelope/soap:Body/descendant-or-self::node()
```

You must also supply the namespace mapping for the `soap` prefix, for example:

Prefix	URI
`soap`	`http://schemas.xmlsoap.org/soap/envelope/`

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Header>
  </soap:Header>
  <soap:Body>
    <product xmlns="http://www.oracle.com">
      <name>SOA Product</name>
      <company>Company</company>
      <description>Web services Security</description>
    </product>
  </soap:Body>
</soap:Envelope>
```

XPath predicates

Select this option if you wish to use an XPath transform to reference the signed content. You must select an XPath predicate from the table to do this. The table is prepopulated with several XPath predicates that can be used to identify common security headers that occur in SOAP messages, including SAML assertions.

To illustrate the use of XPath predicates, the following example shows how the SOAP message is signed when the default *Sign SOAP Body* predicate is selected:

```
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
 <s:Body>
  <vs:getProductInfo xmlns:vs="http://www.oracle.com">
   <vs:Name>API Gateway Explorer</vs:Name>
   <vs:Version>11.1.2.4.0</vs:Version>
  </vs:getProductInfo>
 </s:Body>
</s:Envelope>
```

The default XPath expression (Sign SOAP Body) identifies the contents of the SOAP `Body` element, including the `Body` element itself. The following is the XML Signature produced when this XPath predicate is used:

```
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
 <s:Header>
  <dsig:Signature id="Sample" xmlns:dsig="http://www.w3.org/2000/09/xmldsig#">
   <dsig:SignedInfo>
     ...
    <dsig:Reference URI="">
     <dsig:Transforms>
      <dsig:Transform
          Algorithm="http://www.w3.org/TR/1999/REC-xpath-19991116">
       <dsig:XPath xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
         ancestor-or-self::soap:Body
       </dsig:XPath>
      </dsig:Transform>
      <dsig:Transform Algorithm="http://www.w3.org/2001/10/xml-exc-c14n"/>
     </dsig:Transforms>
       ...
    </dsig:Reference>
   </dsig:SignedInfo>
    ...
  </dsig:Signature>
 </s:Header>
 <s:Body>
  <vs:getProductInfo xmlns:vs="http://ww.oracle.com">
   <vs:Name>API Gateway Explorer</vs:Name>
   <vs:Version>11.1.2.4.0</vs:Version>
  </vs:getProductInfo>
 </s:Body>
</s:Envelope>
```

This XML Signature includes an extra `Transform` element, which has a child `XPath` element. This element specifies the XPath predicate that validating applications must use to identify the signed content.

Message attribute

Finally, you can use the contents of a message attribute to determine what must be signed in the message. For example, you can configure a ??? filter to extract certain content from the message and store it in a particular message attribute. You can then specify this message attribute on the **Message Attribute** tab.

To do this, select the **Extract nodes from message attribute** check box, and enter the name of the attribute that contains the nodes in the field provided.

Select configuration values at runtime

Overview

A selector is a special syntax that enables API Gateway Explorer configuration settings to be evaluated and expanded at runtime based on metadata values (for example, from message attributes, a Key Property Store (KPS), or environment variables). The selector syntax uses the Java Unified Expression Language (JUEL) to evaluate and expand the specified values. Selectors provide a powerful feature when integrating with other systems or when customizing and extending the API Gateway Explorer.

Selector syntax

The API Gateway Explorer selector syntax uses JUEL to evaluate and expand the following types of values at runtime:

- Message attribute properties configured in message filters, for example:

  ```
  ${authentication.subject.id}
  ```

- Environment variables specified in `envSettings.props` and `system.properties` files, for example:

  ```
  ${env.PORT.MANAGEMENT}
  ```

- Values stored in a KPS table, for example:

  ```
  ${kps.CustomerProfiles[JoeBloggs].age}
  ```

Important

Do not use hyphens (`-`) in selector expressions. Hyphens are not supported by the Java-based selector syntax. You can use underscores (_) instead.

Access fields

A message attribute selector can refer to a field of that message (for example `certificate`), and you can use `.` characters to access subfields. For example, the following selector expands to the `username` field of the object stored in the `profile` attribute in the message:

```
${profile.username}
```

You can also access fields indirectly using square brackets (`[` and `]`). For example, the following selector is equivalent to the previous example:

```
${profile[field]}
```

You can specify literal strings as follows:

```
${profile["a field name with spaces"]}
```

For example, the following selector uses the `kathy.adams@acme.com` key value to look up the `User` table in the KPS, and returns the value of the `age` property:

```
${kps.User["kathy.adams@acme.com"].age}
```

Note

For backwards compatibility with the `.` spacing characters used in previous versions of the API Gateway Explorer, if a selector fails to resolve with the above rules, the flat, dotted name of a message attribute still works. For example, `${content.body}` returns the item stored with the `content.body` key in the message.

Special selector keys

The following top-level keys have a special meaning:

Key	Description
`kps`	Subfields of the `kps` key reflect the alias names of KPS tables in the API Gateway Explorer group. Further indexes represent properties of an object in a table (for example, `${kps.User["kathy.adams@acme.com"].age}`).
`env`, `system`	In previous versions, fields from the `envSettings.props` and `system.properties` files had restrictions on prefixes. The selector syntax does not require the `env` and `system` prefixes in these files. For example, `${env.` selects settings from `envSettings.props`, and the rest of the selector chooses any properties in it. However, for compatibility, if a setting in either file starts with this prefix, it is stripped away so the selectors still behave correctly with previous versions.

Resolve selectors

Each `${...}` selector string is resolved step-by-step, passing an initial context object (for example, `Message`). The top-level key is offered to the context object, and if it resolves the field (for example, the message contains the named attribute), the resolved object is indexed with the next level of key. At each step, the following rules apply:

1. At the top level, test the key for the global values (for example, `kps`, `system`, and `env`) and resolve those specially.
2. If the object being indexed is a Dictionary, KPS, or Map, use the index as a key for the item's normal indexing mechanism, and return the resulting lookup.
3. If all else fails, attempt Java reflection on the indexed object.

Note

Method calls are currently only supported using Java reflection. There are currently no supported functions as specified by the Unified Expression Language (EL) standard. For more details on JUEL, see http://juel.sourceforge.net/.

Example selector expressions

This section lists some example selectors that use expressions to evaluate and expand their values at runtime.

Message attribute

The following message attribute selector returns the HTTP `User-Agent` header:

```
${http.headers["User-Agent"]}
```

For example, this might expand to the following value:

```
Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/535.7 (KHTML, like Gecko) Chrome/16.0.912.77
Safari/535.7
```

Environment variable

In a default configuration, the following environment variable selector returns port 8091:

```
${env.PORT.MANAGEMENT + 1}
```

Key Property Store

The following selector looks up a KPS table with an alias of `User`:

```
${kps.User[http.querystring.id].firstName}
```

This selector retrieves the object whose key value is specified by the `id` query parameter in the incoming HTTP request, and returns the value of the `firstName` property in that object.

The following selector explicity provides the key value, and returns the value of the `age` property:

```
${kps.User["kathy.adams@acme.com"].age}
```

In this example, the ASCII `"` character is used to delimit the key string.

The following selector looks up a KPS table with a composite secondary key of `firstName,lastName`:

```
${kps.User[http.querystring.firstName][http.querystring.lastName].email}
```

In this example, the key values are received as query parameters in the incoming HTTP request. The selector returns the value of the `email` property from the resulting object.

Examples using reflection

The following message attribute selector returns the CGI argument from an HTTP URL (for example, returns `bar` for `http://localhost/request.cgi?foo=bar`):

```
${http.client.getCgiArgument("foo")}
```

This returns the name of the top-level element in an XML document:

```
${content.body.getDocument().getDocumentElement().getNodeName()}
```

This returns true if the HTTP response code lies between 200 and 299:

```
${http.response.status / 200 == 2}
```

Tip

You can use the **Trace** filter to determine the appropriate selector expressions to use for specific message attributes. When configured after another filter, the **Trace** filter outputs the available message attributes and their Java type (for example, `Map` or `List`). For details on `com.vordel` classes, see:

```
<install-dir>/apigateway/docs/javadoc/index.html
```

For example, for the `OAuth2AccessToken` class, you can use selector expressions such as `${accesstoken.getAdditionalInformation()}`.

Extract message attributes

There are a number of API Gateway Explorer filters that extract message attribute values (for example, **Extract Certificate Attributes** and **Retrieve from HTTP Header**). Using selectors to extract message attributes offers a more flexible alternative to using such filters. For more details on using selectors instead of these filters, contact Oracle Support.