# API Gateway

Version 7.6.2

14 July 2020

# Policy Developer Filter Reference

DRAFT



This documentation describes the following Axway software:

Axway API Gateway 7.6.2

# Contents

# Preface

This guide describes the filters that you can use when developing policies in Policy Studio, and how to configure them.

## Who should read this guide

This guide is intended for policy developers.

Familiarity with Axway products is recommended.

## How to use this guide

This guide should be used alongside the *API Gateway Policy Developer Guide* and the other guides in the API Gateway documentation set.

Filters are divided into filter categories. Each of the following sections describe how to configure the filters in that filter category in Policy Studio:

**Note**

- For information on filters in the API Management category, see the *API Manager User Guide*.
- For information on filters in the OAuth and OpenID Connect categories, see the *API Gateway OAuth User Guide*.

# Related documentation

The AMPLIFY API Management solution enables you to create, publish, promote, and manage Application Programming Interfaces (APIs) in a secure and scalable environment. For more information, see the *AMPLIFY API Management Getting Started Guide*.

The following reference documents are also available on the Axway Documentation portal at https://docs.axway.com:

- *Supported Platforms*

  Lists the different operating systems, databases, browsers, and thick client platforms supported by each Axway product.

- *Interoperability Matrix*

  Provides product version and interoperability information for Axway products.

# Support services

The Axway Global Support team provides worldwide 24 x 7 support for customers with active support agreements.

Email support@axway.com or visit Axway Support at https://support.axway.com.

See "Get help with API Gateway" in the *API Gateway Administrator Guide* for the information that you should be prepared to provide when you contact Axway Support.

# Training services

Axway offers training across the globe, including on-site instructor-led classes and self-paced online learning. For details, go to: http://www.axway.com/support-services/training

# Accessibility

Axway strives to create accessible products and documentation for users.

This documentation provides the following accessibility features:



## Screen reader support

- Alternative text is provided for images whenever necessary.
- The PDF documents are tagged to provide a logical reading order.

## Support for high contrast and accessible use of colors

- The documentation can be used in high-contrast mode.
- There is sufficient contrast between the text and the background color.
- The graphics have the right level of contrast and take into account the way color-blind people perceive colors.

# Updates and revisions

This guide includes the following documentation changes.

## Changes in version 7.6.2

- Updated the topic on configuring an OCSP client filter to describe new options for time validation. For more information, see *OCSP client* on page 178.
- Added information on how to insert an XML node containing a `namespace`. For more information, see *Add XML node* on page 237
- Restructured the API Gateway Analytics information and added links to the new *API Gateway Analytics User Guide*.
- Updated the topic on the Generic Error filter to describe new options for customized generic errors. For more information, see *Generic error handling* on page 306.
- Updated information on the Smooth Rate Limiting algorithm options. For more information, see "Throttling" in the *API Gateway Policy Developer Filter Reference*.

## Changes in version 7.6.1

- Added information on specifying multiple input schemas when executing data maps. For details, see *Transform with data map* on page 244.

## Changes in version 7.6.0

- Removed references to the following:
  - API Gateway server-side support for Windows
  - API Gateway Appliance
  - API Gateway Analytics
- Updated the topic on the Throttling filter with new settings for the floating time window and smooth rate limiting algorithms. For more details, see *Throttling* on page 212.

# Amazon Web Services filters 1

The following filters enable you to integrate with Amazon Web Services:



# Send to Amazon SQS

## Overview

Amazon Simple Queue Service (SQS) is a hosted message queuing service for distributing messages amongst machines. API Gateway acts as a client to SQS and can send messages to SQS. You can use the **Send to Amazon SQS** filter to send messages to an SQS queue.

For more information on Amazon SQS, go to http://aws.amazon.com/sqs/.

## General settings

Configure the following settings on the **Send to Amazon SQS** window:

**Name**:
Enter a suitable name for the filter to display in a policy.

### *AWS settings*

**AWS Credential**:
Click the browse button to select your AWS security credentials (API key and secret) for Amazon SQS.

**Region**:
Select the region in which to access the SQS service. You can choose from the following options:

- US East (Northern Virginia)
- US West (Oregon)
- US West (Northern California)
- EU (Ireland)
- Asia Pacific (Singapore)

- Asia Pacific (Tokyo)
- Asia Pacific (Sydney)
- South America (Sao Paulo)
- US GovCloud

**Client settings**:
Click the browse button to select the AWS client configuration to be used by API Gateway when connecting to Amazon SQS. For more information on configuring client settings, see "Configure AWS client settings" in the *API Gateway Policy Developer Guide*.

# Send message settings

Configure the following settings on the **Send Message** tab:

**Queue name**:
Enter the name of the queue to send the message to. The default name is `publishQueue`. To create a new queue with the specified name, click the **Create** option.

**Send the message payload**:
Select this option to send the message payload to the queue.

**Or send the value of the attribute below to SQS**:
Select this option to send the value of an attribute to the queue. Complete the following fields:

- **Attribute Name**:
  Enter the name of the attribute.
- **Content Type**:
  Enter the content type to be used for sending the message to SQS (for example, `text/plain`).
- **Content Encoding**:
  Enter the content encoding.

# Advanced settings

On the **Advanced** tab you can configure how to handle messages that are larger than 256KB in size. Configure these fields:

**Split message into smaller ones**:
Select this option to split the message into smaller messages before sending it to the queue.

**Store in S3 and place pointer on SQS queue**:
Select this option to store the payload in Amazon S3 and place a pointer to S3 in the queue. Configure the S3 settings as described in *S3 settings* on page 30.

## Further information

For more detailed information on Amazon Web Services integration, see the *AWS Integration Guide* available from Axway Support.

# Upload to Amazon S3

## Overview

Amazon Simple Storage Service (S3) is an online storage web service that you can use to store and retrieve any amount of data. API Gateway acts as a client to S3 and can upload data to S3. You can use the **Upload to Amazon S3** filter to upload data to Amazon S3.

For more information on Amazon S3, go to http://aws.amazon.com/s3/.

## General settings

Configure the following settings on the **Upload to Amazon S3** window:

**Name**:
Enter a suitable name for the filter to display in a policy.

### *AWS settings*

**AWS Credential**:
Click the browse button to select your AWS security credentials (API key and secret) for Amazon S3.

**Region**:
Select the region in which to store your data. You can choose from the following options:

- US East (Northern Virginia)
- US West (Oregon)
- US West (Northern California)
- EU (Ireland)
- Asia Pacific (Singapore)
- Asia Pacific (Tokyo)
- Asia Pacific (Sydney)
- South America (Sao Paulo)
- US GovCloud

**Client settings**:
Click the browse button to select the AWS client configuration to be used by API Gateway when connecting to Amazon S3. For more information on configuring client settings, see "Configure AWS client settings" in the *API Gateway Policy Developer Guide*.

## *S3 settings*

**Bucket name**:
Enter the name of the bucket in which to store the data. To create a new bucket with the specified name, click the **Create** option.

**Object key**:
Enter the object key for the object to be stored. Alternatively, you can enter a selector that is expanded at runtime. For more details on selectors, see "Select configuration values at runtime" in the *API Gateway Policy Developer Guide*.

**Encryption key**:
Click the browse button to select an encryption key for the object.

**How to store**:
Select how to store the object. You can choose from the following options:

- Standard – This is the standard S3 storage option.
- Reduced Redundancy – This is a storage option within Amazon S3 for storing non-critical, reproducible data at lower levels of redundancy than standard storage.
- Glacier – This is a low-cost storage option for data archival.

# Further information

For more detailed information on Amazon Web Services integration, see the *AWS Integration Guide* available from Axway Support.

# Attribute filters 2

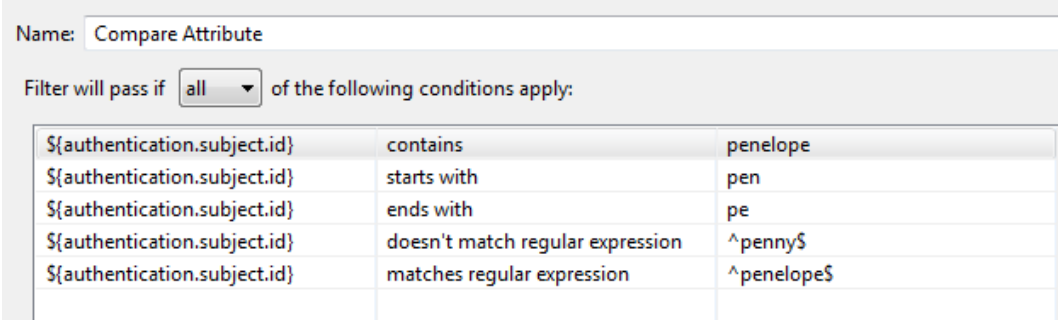
The following filters enable you to work with message attributes:



# Compare attribute

## Overview

The **Compare Attribute** filter enables you to compare the value of a specified message attribute on the API Gateway white board with the values specified in the filter. For example, the following filter only passes if the `${authentication.subject.id}` message attribute has a value of `penelope`:




# Configuration

Configure the following fields:

**Name**:
Enter an appropriate name for this filter to display in a policy.

**Filter will pass if**:
Select **all** or **one** of the specified conditions to apply. Defaults to **all**. Click the **Add** button at the bottom right to specify a rule condition. In the **Attribute filter rule** dialog, perform the following steps:

1. Enter a message attribute selector in the **Value from** text box on the left (for example, `${http.request.verb}` or `${my.customer.attribute}`).
2. Select one of the following rule conditions from the list:
    - `contains`
    - `doesn't contain`
    - `doesn't match regular expression`
    - `ends with`
    - `is`
    - `is not`
    - `matches regular expression`
    - `starts with`
3. Enter a value to compare with in the text box on the right (for example, `POST`). Alternatively, you can enter a selector that is expanded at runtime (for example, `${http.request.uri}`). For more details on selectors, see "Select configuration values at runtime" in the *API Gateway Policy Developer Guide*.
4. Click **OK**.

Finally, to edit or delete an existing rule condition, select it in the table, and click the appropriate button.

# Extract REST request attributes

## Overview

This filter extracts the values of query string parameters and HTTP headers from a REST request and stores them in separate message attributes. The request can be an HTTP GET or HTTP POST request. This filter is in the **Attributes** category in Policy Studio. For details on creating a REST request, see *Create REST request* on page 242.

### *HTTP GET requests*

The following example shows an incoming HTTP GET request with query string and HTTP headers:

```
GET /services?name=Niall&location=Dublin&location=Pembroke%20St HTTP/1.1
Host:mail.google.com
User-Agent:Mozilla/5.0 (Windows; U; Windows NT 6.1; en-GB; rv:1.9.2.15)
Gecko/20110303 Firefox/3.6.15
Accept:text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language:en-gb,en;q=0.5
Accept-Encoding:gzip,deflate
Accept-Charset:ISO-8859-1,utf-8;q=0.7,*;q=0.7
```

Using this example, when **Request Querystring** is selected, the **Extract REST Request Attributes** filter generates the following attributes:

```
http.header.Host = mail.google.com
http.header.User-Agent = Mozilla/5.0 (Windows; U; Windows NT 6.1; en-GB;
rv:1.9.2.15)
Gecko/20110303 Firefox/3.6.15
http.header.Accept = text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
http.header.Accept-Language = en-gb,en;q=0.5
http.header.Accept-Encoding = gzip,deflate
http.header.Accept-Charset = ISO-8859-1,utf-8;q=0.7,*;q=0.7
params.query.name = Niall
params.query.location.1 = Dublin
params.query.location.2 = Pembroke St
```

This filter extracts all parameters from an incoming REST request, and stores them in separate message attributes so that they can be validated easily.

**Note** For multi-valued query string parameters, each value is given an incremental index. For example, the multi-valued `location` parameter results in the creation of the `params.query.location.1` and `params.query.location.2` message attributes.

### *HTTP POST requests*

When you POST a form to the API Gateway, the parameters are placed in the message body, and not in the query string. However, the **Extract REST Request Attributes** filter treats posted parameters the same as normal query parameters, and also adds them to the `params.query` message attribute in a similar way to the HTTP GET request. For example, an HTTP POST message body contains the following:

```
grant_type=password&username=johndoe&password=A3ddj3w
```

This means that the `${params.query.grant_type}` message attribute selector contains a value of `password`.

## Configuration

Configure the following fields on the **Extract REST Request Attributes** window:

**Name**:
Enter an appropriate name for this filter to display in a policy.

**Request Querystring**:
Select whether to extract the values of query string parameters from an HTTP POST or GET request. These are simple name-value pairs (for example, `Name=Joe Bloggs`). This setting is selected by default.

**HTTP Headers**:
Select whether to extract the HTTP header values from an HTTP POST or GET request (selected by default).

**Decode Extracted Attributes**:
Select whether to decode URI paths that have been percent-encoded (for example, using `%2F` for `/`). This setting enables compatibility with previous API Gateway versions, which decoded URI paths, and is not selected by default. For example, this means that URI path components such as the following stay in a raw state:

```
/s8koID4%2FAd6AqgADSghC%2Bg%3D%3D/book%20repo/first%20book.pdf
```

This results in:

```
path[0] = ""
path[1] = "s8koID4%2FAd6AqgADSghC%2Bg%3D%3D"
path[2] = "book%20repo"
path[3] = "first%20book.pdf"
```

When this setting is selected, the URI path is decoded. This results in:

```
path[0] = ""
path[1] = "s8koID4%/Ad6AqgADSghC+g=="
path[2] = "book repo"
path[3] = "first book.pdf"
```

# Extract WSS header

## Overview

The **Extract WSS Header** filter extracts a WS-Security `Header` block from a message. The extracted security header is stored in the `authentication.ws.wsblockinfo` message attribute.

To process this security header later in the policy, you can specify this message attribute in the configuration window for the specific processing filter. For example, to sign the security header, you can specify the `authentication.ws.wsblockinfo` message attribute in the **What to Sign** section of the **XML Signature Generation** filter. Open the **Message Attribute** tab on the **What to Sign** window, and specify this attribute to sign the security header.

### *Timestamp validity*

The **Extract WSS Header** filter implicitly checks the `wsu:Timestamp` in the WSS `Header` block, if present. It checks the `Expires` and `Created` time to determine whether the current time is between the following values:

```
[Created time - drift time], [Expires time + drift time]
```

The drift time is taken from the value set in **Environment Configuration** > **Server Settings > General > Token drift time (secs)**, which defaults to 300 seconds. This filter fails if the extracted WSS header block contains an invalid timestamp.

## Configuration

Configure the following fields on the **Extract WSS Header** filter configuration window:

**Name**:
Enter an intuitive name for this filter to display in a policy (for example, `ExtractCurrent Actor WSS Header`).

**Actor or Role**:
Specify the name of the SOAP Actor or Role of the WS-Security header that you want to extract. Remember, the WS-Security header is stored in the `authentication.ws.wsblockinfo` message attribute.

**Remove enclosing WS-Security element**:
This option removes the enclosing `wsse:Security` element from the message.

# Extract WSS timestamp

## Overview

You can use the **Extract WSS Timestamp** filter to extract a WSS header timestamp from a message. The timestamp is stored in a specified message attribute so that it can be processed later in a policy. This filter requires the WSS header block to have been extracted previously. For more details, see *Extract WSS header* on page 35.

Typically, the **Validate Timestamp** filter is used to retrieve the timestamp from the specified message attribute and validate it. The **Validate Timestamp** filter is available from the **Content Filtering** filter category. For more details, see *Validate timestamp* on page 231.

## Configuration

Configure the following fields on the **Extract WSS Timestamp** filter configuration window:

**Name**:
Enter an appropriate name for this filter to display in a policy.

**Message Attribute to Contain the Timestamp**:
When API Gateway extracts the WSS header timestamp from the message at runtime, it stores the timestamp in the specified message attribute. To validate the timestamp later in the policy, you *must* specify this message attribute in the configuration window for the **Validate Timestamp** filter.

# Extract WSS UsernameToken element

## Overview

You can use the **Extract WSS Username Token** filter to extract a WS-Security `UsernameToken` from a message if it exists. The extracted `UsernameToken` token is stored in the `wss.usernameToken` message attribute.

To process the `UsernameToken` later in the policy, you can specify this message attribute in the configuration window for the processing filter. For example, to sign the `UsernameToken`, you can simply specify the `wss.usernameToken` message attribute in the **What to Sign** section of the **XML Signature Generation** filter. Open the **Message Attribute** tab on the **What to Sign** window, and specify this attribute to sign the user name token.

## Configuration

Configure the following field on the **Extract WSS Username Token** filter configuration window:

**Name**:
Enter an appropriate name for the filter. Remember that the WS-Security `UsernameToken` is stored in the `wss.usernameToken` message attribute.

# Get cookie

## Overview

An HTTP cookie is data sent by a server in an HTTP response to a client. The client can then return an updated cookie value in subsequent requests to the server. For example, this enables the server to store user preferences, manage sessions, track browsing habits, and so on.

The **Get Cookie** filter is used to read the `Cookie` and `Set-Cookie` HTTP headers. The `Cookie` header is used when a client sends a cookie to a server. The `Set-Cookie` header is used when the server instructs the client to store a cookie.

For more details, see *Create cookie* on page 241.

## Configuration

Configure the following fields on the **Get Cookie** filter configuration window:

**Filter Name**:
Enter an appropriate name for this filter to display in a policy..

**Cookie Name**:
Enter a regular expression that matches the name of the cookie. This value can use wildcards. Defaults to the `.*` wildcard.

**Remove all Cookie Headers from Message after retrieval**:
When this setting is selected, all `Cookie` and `Set-Cookie` headers are removed from the message after retrieving the target cookie. This setting is not selected by default.

# Attribute storage

When a cookie is retrieved, it is stored in the appropriate API Gateway message attribute. The following message attributes are used to store cookies:

| Cookie Header Type | Message Attribute Name |
|---|---|
| `Cookie` | `cookie.`*`cookie_name`*`.value` (for example, `cookie.mytest.value`) |
| `Set-Cookie` | `cookie.`*`cookie_name`*`.`*`cookie_attribute_name`* (for example, `cookie.mytest.header`) |

## *Set-Cookie attribute list*

The `Set-Cookie` HTTP header includes the following cookie attributes (reflected in the `Set-Cookie` message attribute name):

| Cookie Attribute Name | Description |
|---|---|
| `header` | The HTTP header name. |
| `value` | The value of the cookie. |
| `domain` | The domain name for this cookie. |
| `path` | The path on the server to which the browser returns this cookie. |
| `maxage` | The maximum age of the cookie in days, hours, minutes, and/or seconds. |
| `secure` | Whether sending this cookie is restricted to a secure protocol. This setting is not selected by default, which means that it can be sent using any protocol. |
| `HTTPOnly` | Whether the browser should use cookies over HTTP only. This setting is not selected by default. |

# 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 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.

You can 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.axway.com">
            <saml:AttributeValue>admin</saml:AttributeValue>
          </saml:Attribute>
          <saml:Attribute Name="email" NameFormat="http://www.axway.com">
            <saml:AttributeValue>joe@axway.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</name>
        <company>Axway</company>
        <description>Web Services Security</description>
      </product>
    </soap:Body>
</soap:Envelope>
```

# Assertion details settings

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 API Gateway (that generate the assertion) and the machines that consume the assertion. The specified time is subtracted from the time at which API Gateway 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.

**Note** 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 might 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, Axway 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. 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 is suitable for cases where this filter might 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 "Configure XPath expressions" in the *API Gateway Policy Developer Guide*.

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 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, 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=axway</dsig:X509SubjectName>
      <dsig:X509Certificate>
        MIICmzCCAY ...... mB9CJEw4Q=
      </dsig:X509Certificate>
    </dsig:X509Data>
  </dsig:KeyInfo>
</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 includes the key used to prove that the API Gateway 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. | `urn:oasis:names:tc:SAML:1.0:cm:artifact` |

| Method | Meaning | Value |
|---|---|---|
| Sender Vouches | Use this confirmation method to assert that the API Gateway 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 (see *XML signature generation* on page 322). | `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 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:
To use an asymmetric key as proof that the API Gateway 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 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, or environment variable). For more details, see "Select configuration values at runtime" in the *API Gateway Policy Developer Guide*.

Symmetric Key:
To use a symmetric key as proof that the API Gateway 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 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, an **XML Signature Generation** filter) has already used a symmetric key, you can reuse this key as proof that the API Gateway is the holder-of-key entity. Enter the name of the selector expression (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, or environment variable). For more details, see "Select configuration values at runtime" in the *API Gateway Policy Developer Guide*.
- **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 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 Selector Expression**:
  Alternatively, if the recipient's certificate has already been used (perhaps to encrypt part of the message), the certificate 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, or environment variable). For more details, see "Select configuration values at runtime" in the *API Gateway Policy Developer Guide*.
- **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 list, whereas if it refers to a `BinarySecurityToken` , select `X509v3` from the list. 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="axway">
    <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.example.com">
            <saml:AttributeValue>admin</saml:AttributeValue>
          </saml:Attribute>
          <saml:Attribute Name="email" NameFormat="http://www.example.com">
            <saml:AttributeValue>joe@example.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 attributes with JSON path

## Overview

JSON Path is an XPath like query language for JSON (JavaScript Object Notation) that enables you to select nodes in a JSON document. The **Retrieve Attributes with JSON Path** filter enables you to retrieve specified message attributes from a JSON message using JSON Path expressions.

For more details on JSON Path, go to http://code.google.com/p/jsonpath/.

## Configuration

Configure the following fields on the **Retrieve Attributes with JSON Path** filter window:

**Name**:
Enter an appropriate name for this filter to display in a policy.

**Extract attributes using the following JSON Path expressions**:
Specify the list of attributes for the API Gateway to retrieve using appropriate JSON Path expressions.

To add an attribute to the list, click the **Add** button, and enter the following values in the dialog:

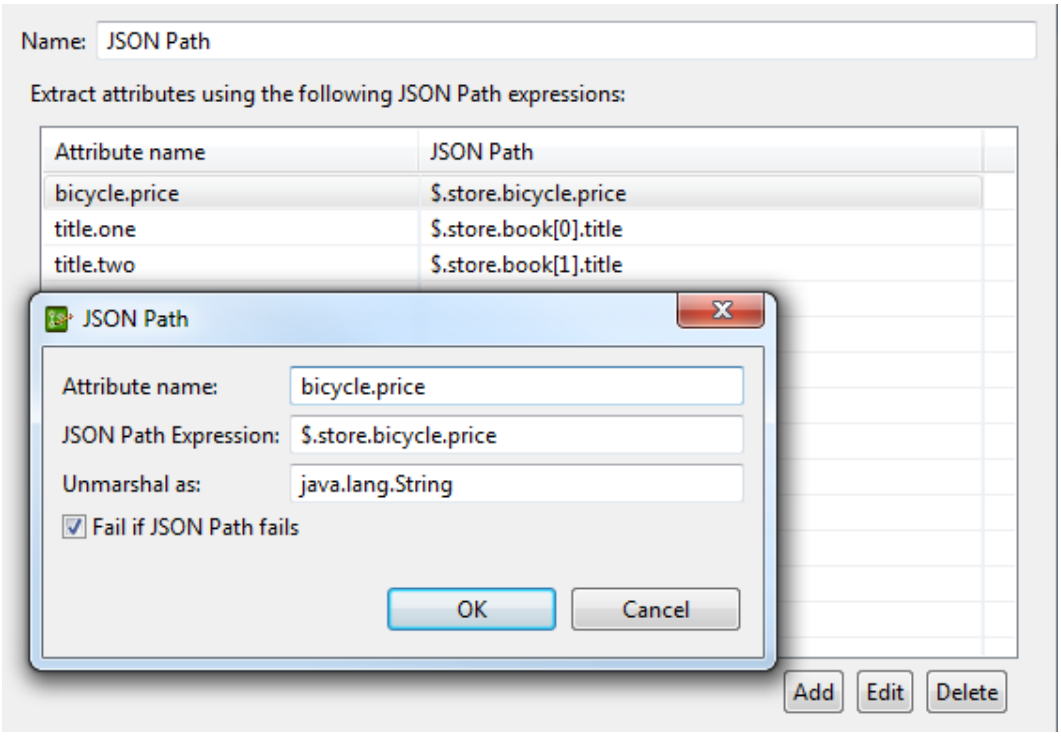
- **Attribute name**:
  Enter the message attribute name that you wish to extract using JSON Path (for example, `bicycle.price`).
- **JSON Path Expression**:
  Enter the JSON Path expression that you wish to use to extract the message attribute (for example, `$.store.bicycle.price`). Policy Studio prompts if you enter an unsupported JSON Path expression.
- **Unmarshal as**:
  Enter the data type to unmarshal the message attribute value as (defaults to `java.util.List`). For example, typical data types to unmarshal as include the following:
  - `java.lang.String`: Enter this value when using this filter to extract a single value and store it in an API Gateway attribute on the message whiteboard.
  - `com.fasterxml.jackson.databind.JsonNode`: Enter this value when using this filter to store the content in a message attribute and then place it in a JSON message using a **JSON Add Node** filter.
- **Fail if JSON Path Fails**:
  Select whether the filter should fail if the specified JSON Path expression fails. This option is not selected by default.

**Note** If no attributes are specified, the API Gateway retrieves all the attributes in the message and sets them to the `attribute.lookup.list` attribute.

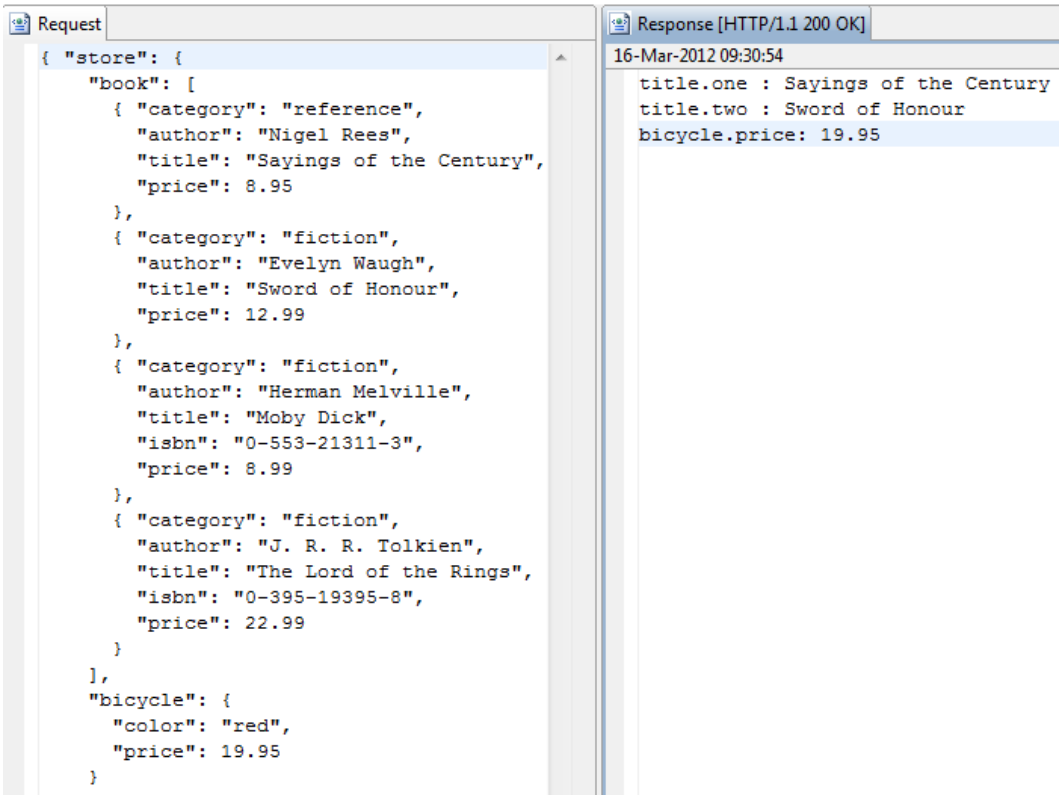
# JSON Path examples

The following are some examples of using the **Retrieve Attributes with JSON Path** filter to retrieve data from a JSON message.

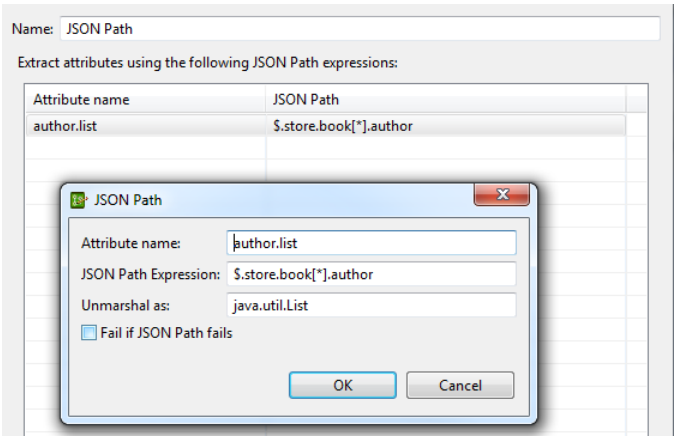
**Retrieving attributes**
The following example retrieves three different data items from the JSON message and stores them in the specified message attributes as strings:



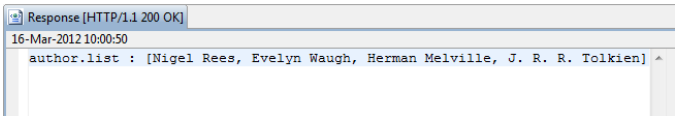

When the extracted attributes are added to the `content.body` message attribute, the following example shows the corresponding request and response message in Axway API Tester:



**Retrieving multiple attributes in a list**
The following example retrieves all the authors from the JSON message and stores them in the specified message attribute as a `List`:




The following example shows the corresponding request and response in Axway API Tester:




## Exceptions

The **Retrieve Attributes with JSON Path** filter aborts with a `CircuitAbortException` if the content body of the payload is not in valid JSON format.

# Retrieve attribute from directory server

The API Gateway can leverage an existing directory server by querying it for user profile data. The **Retrieve From Directory Server** filter can look up a user and retrieve that user's attributes represented as a list of search results. Each element of the list represents a list of multivalued attributes returned from the directory server.

## Database settings

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

**LDAP Directory**:
The API Gateway queries the selected Lightweight Directory Access Protocol (LDAP) directory for user attributes. An LDAP connection is retrieved from a pool of connections at runtime. Click the browse button to select the LDAP directory to query. To use an existing LDAP directory, (for example, `Sample Active Directory Connection`), select it in the tree.

To add an LDAP directory, right-click the **LDAP Connections** tree node, and select **Add an LDAP Connection**. Alternatively, you can add LDAP connections under the **Environment Configuration > External Connections** node in the Policy Studio tree. For more details on how to configure LDAP connections, see "Configure LDAP directories" in the *API Gateway Policy Developer Guide*.

## *Retrieve Unique User Identity*

The **Retrieve Unique User Identity** section enables you to select the user whose profile the API Gateway looks up in the directory server. The user ID can be taken from a message attribute or looked up from an LDAP directory. Configure the following fields:

**From Selector Expression**:
Select this option if the user ID is stored in a message attribute, and specify the selector expression used to obtain its value at runtime (for example, `${authentication.subject.id}`). A user's credentials are stored in the `authentication.subject.id` message attribute after authenticating to the API Gateway, so this is the most likely attribute to enter in this field.

Typically, this contains the Distinguished Name (DName) or user name of the authenticated user. The name extracted from the specified message attribute is used to query the directory server. For more details on selector expressions, see "Select configuration values at runtime" in the *API Gateway Policy Developer Guide*.

**From LDAP Search**:
In cases where you have not already obtained the user's identity and the `authentication.subject.id` attribute has not been prepopulated by a prior authentication filter, you must configure the API Gateway to retrieve the user's identity from an LDAP search. Click **Configure Directory Search** to configure the search criteria to use to retrieve the user's unique DName from the LDAP repository. The User Search dialog is used to search a given LDAP directory for a unique user according to the criteria configured in the following fields:

- **Base Criteria**:

  The value entered here tells the API Gateway where it should begin searching the LDAP directory. For example, it might be appropriate to search for a given user under the `C=IE` tree in the LDAP hierarchy.

- **Query Search Filter**:

  The value entered here is what the API Gateway uses to determine whether it has obtained a successful match. Since you are searching for a specific user, you can use the user name of an authenticated user (the value of the `authentication.subject.id` message attribute) to lookup in the LDAP directory. You must also specify the object class that defines users for the particular type of LDAP directory that you are searching against. For example, object classes representing users amongst common LDAP directories are `inetOrgPerson`, `givenName`, and `User`.

  For example, to search for an authenticated user against Microsoft's Active Directory, you might specify the following as the **Query Search Filter**:

  ```
  (objectclass=User)(cn=${authentication.subject.id})
  ```

This example uses a selector to obtain the ID of the authenticated subject at runtime. For more details on selectors, see "Select configuration values at runtime" in the *API Gateway Policy Developer Guide*.

The string representation of the LDAP search filter must comply with the `RFC2254` grammar and format. However, you might use the following Java system property in `jvm.xml` to ignore `RFC2254` escaping mechanism in the filter:

```
<ConfigurationFragment>
 <SystemProperty name="avoidLDAPQueryEscaping" value="true" />
</ConfigurationFragment>
```

- **Search Scope**:

  These settings specify the depth of the LDAP tree to search. This depends largely on the structure of your LDAP directory.

## *Retrieve Attributes*

The **Retrieve Attributes** section instructs the API Gateway to search the LDAP tree to locate a specific user profile. When the appropriate profile is retrieved, the API Gateway extracts the specified user attributes. Configure the following fields:

**Base Criteria**:
You can specify where the API Gateway should begin searching the LDAP directory using a selector. The selector represents the value of a message attribute that is expanded at runtime. The two most likely message attributes to be used here are the authenticated user's ID and Distinguished Name. Select one of the predefined selectors from the list:

- `${authentication.subject.id}`
- `${authentication.subject.dname}`

Alternatively, you can enter a selector representing other message attributes using the same syntax. For more details on selectors, see "Select configuration values at runtime" in the *API Gateway Policy Developer Guide*.

**Search Filter**:
This is the name given by the particular LDAP directory to the *User* class. This depends on the type of LDAP directory configured. You can also use a selector to represent the value of a message attribute. For example, you can use the `user.role` attribute to store the user class. The syntax for using the selector representing this attribute is as follows:

```
(objectclass=${user.role})
```

**Search Scope**:
If the API Gateway retrieves a user profile node from the LDAP tree, the option selected here dictates the level that the API Gateway searches the node to. The available options are:

- **Object level**
- **One level**

- **Sub-tree**

**Unique Result**:
Select this option to force the API Gateway to retrieve a unique user profile from the LDAP directory. This is useful in cases where the LDAP search has returned several profiles.

**Attribute Name**:
The **Attribute Name** table lists the attributes the API Gateway retrieves from the user profile. If no attributes are listed, the API Gateway extracts all user attributes. In both cases, retrieved attributes are set to the `attribute.lookup.list` message attribute. Click **Add** to add the name of an attribute to extract from the returned user profile. Enter the attribute name to extract from the profile in the **Attribute Name** field of the **Attribute Lookup** dialog.

**Note** It is important to be aware of the following:

- If the search returns results for more than one user, and the **Unique Result** option is enabled, an error is generated. If this option is not enabled, all attributes are merged.
- If an attribute is configured that does not exist in the repository, no error is generated.
- If no attributes are configured, all attributes present for the user are retrieved.

# Advanced settings

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

**Enable legacy attribute naming for retrieved attributes**:
Specifies whether to enable legacy naming of retrieved message attributes. This field is not selected by default. Prior to version 7.1, retrieved attributes were stored in message attributes in the following format:

```
user.<retrieved_attribute_name>
```

For example, `${user.email}`, `${user.role}`, and so on. If the retrieved attribute was multi-valued, you would access the values using `${user.email.1}` or `${user.email.2}`, and so on. In version 7.1 and later, by default, you can query for multivalued retrieved attributes using an array syntax (for example, `${user.email[0]}`, or `${user.email[1]}`, and so on). Select this setting to use the legacy format for attribute naming instead.

**Example of output attribute format with legacy attribute naming**
The following table shows the output attribute format when legacy attribute naming is selected:

| Prefix for message attribute name (optional) | Output attribute format (when attribute name is memberOf) |
|---|---|
| `user` (default) | • `attribute.lookup.list`: Map of retrieved attributes<br>• `user.memberOf`: When retrieves only a single value for the given attribute<br>• `user.memberOf.*` (for example, `user.memberOf.1`, `user.memberOf.2`, and so on): When retrieves multiple values for the given attribute<br>• `${user.memberOf}`: Example selector |
| None | • `attribute.lookup.list`: Map of retrieved attributes<br>• `memberOf`: When retrieves only a single value for the given attribute<br>• `memberOf.*` (for example, `memberOf.1`, `memberOf.2`, and so on): When retrieves multiple values for the given attribute<br>• `${user.memberOf}`: Example selector |

**Example of output attribute format without legacy attribute naming**
The following table shows the output attribute format when legacy attribute naming is not selected:

| Prefix for message attribute name (mandatory) | Output attribute format (when attribute name is user.memberOf) |
|---|---|
| `user` (default) | • `user`: List of search results, where each element of the list corresponds to search results (pairs of attribute names and values)<br>• Example selector: `${user[0].memberOf[0]}` |

**Prefix for message attribute**:
You can specify an optional prefix for message attribute names. The default prefix is `user`. For more details, see **Enable legacy attribute naming for retrieved attributes**.

**Note** When legacy attribute naming is not enabled, if you call the **Retrieve from Directory Server** filter multiple times in a row, the results are appended to the original `user` attributes, instead of replacing them.

For example, in single request, two **Retrieve from Directory Server** filters consecutively query an LDAP server for attributes, and both use the default prefix name of `user`. However, the attribute results of the second LDAP call do not replace the first results, and the `user` attribute gets a second entry instead. This means that to retrieve the attributes returned by the second call, you must use `${user[1].anotherattribute[0]}`.

# Retrieve 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 window:

**Name**:
Enter an appropriate name for this filter to display in a policy.

**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 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.

# Retrieve attribute from SAML attribute assertion

## Overview

A SAML (Security Assertion Markup Language) attribute assertion contains information about a user in the form of a series of attributes. The **Retrieve from SAML Attribute Assertion** filter can retrieve these attributes and store them in the `attribute.lookup.list` message attribute.

The following SAML attribute assertion contains three attributes, `"role"`, `"email"`, and `"dept"`. The **Retrieve from SAML Attribute Assertion** filter stores all three attributes and their values in the `attribute.lookup.list` message attribute.

```
<?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.axway.com">
            <saml:AttributeValue>admin</saml:AttributeValue>
          </saml:Attribute>
          <saml:Attribute Name="email" NameFormat="http://www.axway.com">
            <saml:AttributeValue>joe@axway.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</name>
        <company>Axway</company>
        <description>Web Services Security</description>
      </product>
    </soap:Body>
</soap:Envelope>
```

# Details settings

The following fields are available on the **Details** tab:

**SOAP Actor/Role**:
If you expect the SAML assertion to be embedded within a WS-Security block, you can identify this block by specifying the SOAP Actor or Role of the WS-Security header that contains the assertion.

**XPath Expression**:
Alternatively, if the assertion is not contained within a WS-Security block, you can enter an XPath expression to locate the attribute assertion. XPath expressions can be added by selecting the **Add** button. Expressions can be edited and deleted by selecting an XPath expression and clicking the **Add** and **Delete** buttons respectively.

**SAML Namespace**:
Select the SAML namespace that must be used on the SAML assertion for this filter to succeed. If you do not wish to check the namespace, select the `Do not check version` option from the list.

**SAML Version**:
Enter the SAML version that the assertion must adhere to by entering the major version in the first field, followed by the minor version in the second field. For example, for SAML 2.0, enter `2` in the first field and `0` in the second field.

**Drift Time**:
When the API Gateway receives a SAML attribute assertion, it first checks to make sure that it has not expired. The lifetime of the assertion is specified using the `NotBefore` and `NotOnOrAfter` attributes of the `<Conditions>` element in the assertion itself. The API Gateway makes sure that the time at which it validates the assertion is between the `NotBefore` and `NotOnOrAfter` times.

The **Drift Time** is used to account for differences in the clock time of the machine that generated the assertion and the machine hosting the API Gateway. The time specified here is subtracted from the time at which the API Gateway attempts to validate the assertion.

# Trusted issuer settings

You can use the table on this tab to select the issuers that you consider trusted. In other words, this filter only accepts assertions that have been issued by the SAML authorities selected here.

Click the **Add** button to display the **Trusted Issuers** window. Select the Distinguished Name of a SAML authority whose certificate has been added to the certificate store and click the **OK** button. Repeat this step to add more SAML authorities to the list of trusted issuers.

# Subject settings

The API Gateway can perform some very basic authentication checks on the subject or sender of the assertion using the options available on the **Subject** tab. The API Gateway can compare the subject of the assertion (for example, the `<NameIdentifier>`) to one of the following values:

- **The subject of the authentication filter**:
  Select this option if the user specified in the `<NameIdentifier>` element must match the user that authenticated to the API Gateway. The subject of the authentication event is stored in the `authentication.subject.id` message attribute.
- **The following value**:
  This option can be used if the `<NameIdentifier>` must match a user-specified value. Select this radio button and enter the value in the field provided.
- **Neither of the above**:
  If the **Neither of the above** radio button is selected, the API Gateway does not attempt to match the `<NameIdentifier>` to any value.

# Lookup attributes settings

The **Lookup Attributes** tab is used to determine what attributes the API Gateway should extract from the SAML attribute assertion. Extracted attributes and their values are set to the `attribute.lookup.list` message attribute.

The table lists the attributes that the API Gateway extracts from the assertion and set to the `attribute.lookup.list`.

Alternatively, check the **Extract all of the attributes from the SAML assertion** check box to configure the API Gateway to extract all attributes from the assertion. All attributes are set to the `attribute.lookup.list` message attribute.

To configure a specific attribute to look up in the message, click the **Add** button to display the **Attribute Lookup** dialog. Enter the value of the "Name" attribute of the `<Attribute>` element in the **Attribute name** field. Enter the value of the "NameFormat" attribute of the `<Attribute>` element in the **Namespace** field.

# Retrieve attribute from SAML PDP

## Overview

The API Gateway can request information about an authenticated end user in the form of user *attributes* from a SAML PDP (Policy Decision Point) using the SAML Protocol (SAMLP). In such cases, the API Gateway presents evidence to the PDP in the form of some user credentials, such as the Distinguished Name of a client's X.509 certificate.

The PDP looks up its configured user store and retrieves attributes associated with that user. The attributes are inserted into a SAML attribute assertion and returned to the API Gateway in a SAMLP response. The assertion or SAMLP response is usually signed by the PDP.

When the API Gateway receives the SAMLP response, it performs a number of checks on the response, such as validating the PDP signature and certificate, and examining the assertion. It can also insert the SAML attribute assertion into the original message for consumption by a downstream web service.

## Request settings

The **Request** tab describes how the API Gateway should package the SAMLP request before sending it to the SAML PDP.

You can configure the following fields on the **Request** tab:

**SAML PDP URL Set**:
You can configure a group of SAML PDPs to which the API Gateway connects in a round-robin fashion if one or more of the PDPs are unavailable. This is known as a SAML PDP URL set. Click the button on the right, and select a previously configured SAML PDP URL set in the tree. To add a URL set, right-click the **SAML PDP URL Sets** tree node, and select **Add a URL Set**. Alternatively, you can configure a SAML PDP URL set under the **Environment Configuration** > **External Connections** node in the Policy Studio tree.

**SOAP Action**:
Enter the SOAP action required to send SAMLP requests to the PDP. Click the **Use Default** button to use the following default SOAP action as specified by SAMLP:

```
http://www.oasis-open.org/committees/security
```

**SAML Version**:
Select the SAML version to use in the SAMLP request.

**Signing Key**:
If the SAMLP request is to be signed, click the **Signing Key** button, and select the appropriate signing key from the certificate store.

## SAML subject settings

You can describe the *subject* of the SAML assertion on the **SAML Subject** tab. Complete the following fields:

**Subject Selector Expression**:
Enter a selector expression for the message attribute that contains the user name of an authenticated user. The default value is `${authentication.subject.id}`.

**Subject Format**:
Select the format of the subject selected in the **Subject Selector Expression** field above.

**Note** There is no need to select a format here if the **Subject Attribute** field is set to `authentication.subject.id`.

## Subject confirmation settings

The settings on the **Subject Confirmation** 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, 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=axway</dsig:X509SubjectName>
      <dsig:X509Certificate>
        MIICmzCCAY ...... mB9CJEw4Q=
      </dsig:X509Certificate>
    </dsig:X509Data>
  </dsig:KeyInfo>
</saml:SubjectConfirmation>
```

You must configure the following fields on the **Subject Confirmation** 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 | A <SubjectConfirmation> is inserted into the SAMLP request. The <SubjectConfirmation> contains a <dsig:KeyInfo> section with the certificate of the user selected to sign the SAMLP request. The user selected to sign the SAMLP request must be the authenticated subject ( authentication.subject.id). Select the **Include Certificate** option if the signer's certificate is to be included in the SubjectConfirmation block. Alternatively, select the **Include Key Name** option if only the key name is to be included. | urn:oasis:names:tc:SAML:1.0:cm: holder-of-key |
| Bearer | A <SubjectConfirmation> is inserted into the SAMLP request. | urn:oasis:names:tc:SAML:1.0:cm: bearer |
| SAML Artifact | A <SubjectConfirmation> is inserted into the SAMLP request. | urn:oasis:names:tc:SAML:1.0:cm: artifact |
| Sender Vouches | A <SubjectConfirmation> is inserted into the SAMLP request. The SAMLP request must be signed by a user. | urn:oasis:names:tc:SAML:1.0:cm: bearer |

If the **Method** field is left blank, no <ConfirmationMethod> block is inserted into the assertion.

**Include Certificate**:
Select this option to include the SAML subject's certificate in the <KeyInfo> section of the <SubjectConfirmation> block.

**Include Key Name**:
Alternatively, if you do not want to include the certificate, you can select this option to only include the key name in the <KeyInfo> section.

### *Attributes*

You can list a number of user attributes to include in the SAML attribute assertion that is generated by the API Gateway. If no attributes are explicitly listed in this section, the API Gateway inserts all attributes associated with the user (all user attributes in the `attribute.lookup.list` message attribute) in the assertion.

To add a specific attribute to the SAML attribute assertion, click the **Add** button. A user attribute can be configured using the **Attribute Lookup** dialog. Enter the name of the attribute that is added to the assertion in the **Attribute name** field. Enter the namespace that is associated with this attribute in the **Namespace** field. You can edit and remove previously configured attributes using the **Edit** and **Remove** buttons.

## Response settings

The **Response** tab configures the SAMLP response returned from the SAML PDP. The following fields are available:

**SOAP Actor/Role**:
If the SAMLP response from the PDP contains a SAML attribute assertion, the API Gateway can extract it from the response and insert it into the downstream message. The SAML assertion is inserted into the WS-Security block identified by the specified SOAP actor/role.

**Drift Time**:
The SAMLP request to the PDP is time stamped by the API Gateway. To account for differences in the times on the machines running the API Gateway and the SAML PDP the specified time is subtracted from the time at which the API Gateway generates the SAMLP request.

# Retrieve attribute from Tivoli

You can use the **Retrieve from Tivoli** filter when you need to retrieve user attributes independently from authorizing the user against Tivoli Access Manager for e-business. This filter is found in the **Attributes** category.

For details on prerequisites for integration with IBM Tivoli, see the *API Gateway Authentication and Authorization Integration Guide*.

## Configuration

Complete the following fields to configure the **Retrieve from Tivoli** filter:

**Name**:
Enter an appropriate name for the filter to display in a policy.

**User ID**:
Enter the ID of a user to retrieve attributes for. You can enter a static user name, Distinguished Name (DName), or selector representing a message attribute. The selector is expanded to the value of the message attribute at runtime.

For example, you can enter `${authentication.subject.id}`. This means that the ID of the authenticated user, which is normally a DName, is used to retrieve attributes for. For this to work correctly, you must configure an authentication filter to run before this filter in the policy. For more details on selectors, see "Select configuration values at runtime" in the *API Gateway Policy Developer Guide*.

**Attributes**:
You can specify a list of user attributes to retrieve from the Tivoli server. To add individual attributes to be retrieved, click **Add** and enter the attributes in the dialog. If you want all attributes to be retrieved, leave the table blank.

**Tivoli Configuration Files**:
A Tivoli configuration file that contains all the required connection details is associated with a particular API Gateway instance. Click **Settings** to display the **Tivoli Configuration** dialog. On the **Tivoli Configuration** dialog, select the API Gateway instance which connection details you want to edit. For more details on configuring this wizard, see "Configure Tivoli connections" in the *API Gateway Policy Developer Guide*.

# 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 window:

**Name**:
Enter an appropriate name for this filter to display in a policy.

**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 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 is added to the content body.

# Retrieve attribute from database

The API Gateway can retrieve user attributes from a specified database, or write user attributes to a specified database. It can do this by running an SQL query on the database, or by invoking a stored procedure call. The query results are represented as a list of properties. Each element in the list represents a query result row returned from the database for the specified SQL query or stored procedure call. These properties represent pairs of attribute names and values for each column in the row.

## Database settings

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

**Database Location**:
The API Gateway searches the selected database for the user's attributes. Click the browse button to select the database to search. To use an existing database connection (for example, `Default Database Connection`), select it in the tree. To add a database connection, right-click the **Database Connections** tree node, and select **Add DB connection**. Alternatively, you can add

database connections under the **Environment Configuration** > **External Connections** node in the Policy Studio tree view. For more information on configuring database connections, see "Configure database connections" in the *API Gateway Policy Developer Guide*.

**Database Statements**:
The **Database Statements** table lists the currently configured SQL queries or stored procedure calls. These queries and calls retrieve certain user attributes from the database selected in the **Database Location** field. You can edit and delete existing queries by selecting them from the list and clicking the **Edit** and **Delete** buttons. Queries are executed in the order they are listed in the filter. You can change the order of execution using the **Up** and **Down** buttons. For more information on how to configure a **Database Query**, see "Configure database queries" in the *API Gateway Policy Developer Guide*.

# Advanced settings

On the **Advanced** tab, configure the following fields in the **User Attribute Extraction** section:

**Place query results into user attribute list**:
Select whether to place database query results in message attributes. When selected, the message attribute names are generated based on the message attribute prefix and the attribute name. For example, if the specified prefix is `user` and the attributes are `PHONE` and `EMAIL`, the `user.PHONE` and `user.EMAIL` attributes are generated. This setting is selected by default.

**Associate attributes with user ID returned by selector**:
When the **Place query results into message attribute list** setting is selected, you can specify a user ID to associate with the user attributes. For example, if the user name is stored as `admin` in the database, you must select the message attribute containing the value `admin`. The API Gateway then looks up the database using this name. By default, the user ID is stored in the `${authentication.subject.id}` message attribute.

Configure the following fields on the **Attribute Naming** section:

**Enable legacy attribute naming for retrieved attributes**:
Specifies whether to enable legacy naming of retrieved message attributes. This field is not selected by default. Prior to version 7.1, retrieved attributes were stored in message attributes in the following format:

```
user.<retrieved_attribute_name>
```

For example, `${user.email}`, `${user.role}`, and so on. If the retrieved attribute was multivalued, you would access the values using `${user.email.1}` or `${user.email.2}`, and so on. In version 7.1 and later, by default, you can query for multivalued retrieved attributes using an array syntax (for example, `${user.email[0]}`, or `${user.email[1]}`, and so on). Select this setting to use the legacy format for attribute naming instead.

**Example of output attribute format with legacy attribute naming**
The following table shows the output attribute format when legacy attribute naming is selected:

| Place query results into user attribute list | Prefix for message attribute name (optional) | Output attribute format (when attribute name is PHONE) |
|---|---|---|
| Selected (default) | `user` (default) | • `attribute.lookup.list` : Map of retrieved attributes<br>• `user.PHONE`: Attribute value<br>• `${user.PHONE}`: Example selector |
| Selected (default) | None | • `attribute.lookup.list` : Map of retrieved attributes<br>• `PHONE`: Attribute value<br>• `${PHONE}`: Example selector |
| Not selected | `user` (default) | • `user.PHONE`: Attribute value<br>• `${user.PHONE}`: Example selector |
| Not selected | None | • `PHONE`: Attribute value<br>• `${PHONE}`: Example selector |

**Example of output attribute format without legacy attribute naming**
The following table shows the output attribute format when legacy attribute naming is not selected:

| Place query results into user attribute list | Prefix for message attribute name (mandatory) | Output attribute format (when attribute name is PHONE) |
|---|---|---|
| Selected (default) | `user` (default) | • `user`: List of properties, where each corresponds to a retrieved row (attribute name and value pair)<br>• `${user[0].PHONE}`: Example selector |
| Not selected | `user` (default) | • `user.PHONE`: List of properties, where each corresponds to a retrieved row (attribute name and value pair)<br>• `${user.PHONE[0]}`: Example selector |

**Prefix for message attribute**:
Specifies an optional prefix for message attribute names used to store query results. The default prefix is `user`. For more details, see **Place query results into user attribute list** and **Enable legacy attribute naming for retrieved attributes**.

**Attribute name for stored procedure out parameters**:
You can also specify an attribute name for stored procedure out parameters. The default prefix is `out.param.value`.

**Case for attribute names**:
You can specify whether attribute names are in lower case or upper case. The default is lower case.

Configure the following fields on the **Result Set Options** section:

**Fail on empty result set**:
Specify whether this filter fails if the result set is empty. This setting is not selected by default.

**Attribute name for result set size**:
Specify the attribute name used to store the size of the result set. Defaults to `db.result.count`.

# Retrieve attribute from user store

## Overview

The API Gateway user store contains user profiles, including attributes relating to each user. After a user has successfully authenticated to the API Gateway, the **Retrieve From User Store** filter can retrieve attributes belonging to that user from the user store.

## Database settings

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

**User ID**:
Select or enter the name of the message attribute that contains the name of the user to look up in the user store. For example, if the user name is stored as `admin`, select the message attribute containing the value `admin`. The API Gateway then looks up the user in the user store using this name.

**Attributes**:
Enter the list of attributes that the API Gateway should retrieve if it successfully looks up the user specified by the **User ID** field. To add attributes, click the **Add** button. Similarly, to edit or remove existing attributes, click the **Edit** or **Remove** buttons.

## Advanced settings

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

**Enable legacy attribute naming for retrieved attributes**:
Specifies whether to enable legacy naming of retrieved message attributes. This field is not selected by default. Prior to version 7.1, retrieved attributes were stored in message attributes in the following format:

```
user.<retrieved_attribute_name>
```

For example, `${user.email}`, `${user.role}`, and so on. If the retrieved attribute was multivalued, you would access the values using `${user.email.1}` or `${user.email.2}`, and so on. In version 7.1 and later, by default, you can query for multivalued retrieved attributes using an array syntax (for example, `${user.email[0]}`, or `${user.email[1]}`, and so on). Select this setting to use the legacy format for attribute naming instead.

**Prefix for message attribute**:
You can specify an optional prefix for message attribute names. The default prefix is `user`.

**Fail on empty result set**:
Specify whether this filter fails if the result set is empty. This setting is not selected by default.

# Authentication filters 3

The following filters enable you to perform authentication with API Gateway:



# Attribute authentication

## Overview

In cases when user credentials are passed to the API Gateway in a non-standard way, the credentials can be copied into API Gateway message attributes, and authenticated against a specified authentication repository (for example, API Gateway user store, LDAP directory, or database) using an **Attribute Authentication** filter. For example, assume user credentials are passed to API Gateway in the following XML message:

```
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
  <s:Body>
    <ns:User xmlns:ns="http://www.user.com">
      <ns:Username>1</ns:Username>
      <ns:Password>2</ns:Password>
    </ns:User>
  </s:Body>
</s:Envelope>
```

In this example, the standard methods of passing credentials (for example, HTTP basic or digest authentication, SAML assertions, and WS-Security Username tokens) are bypassed, and the client sends the user name and password as parameters in a simple SOAP message.

When the API Gateway receives this message, it can extract the value of the `<Username>` and `<Password>` elements using an XPath expression configured in the **Retrieve from Message** filter. This filter uses an XPath expression to retrieve the value of an element or attribute, and can then store this value in the specified message attribute.

You can configure an instance of this filter to retrieve the value of the `<Username>` attribute, and store it in the `authentication.subject.id` message attribute. Similarly, you can configure another filter to retrieve the value of the `<Password>`, and store it in the `authentication.subject.password` message attribute.

The **Attribute Authentication** filter can then use the user name and password values stored in these message attributes to authenticate the user against the specified authentication repository.

# Configuration

Complete the following fields to configure this filter:

**Name**:
Enter an appropriate name for this filter to display in a policy.

**Username**:
Specify the API Gateway message attribute that contains the user name of the user to be authenticated. The default attribute is the `authentication.subject.id` attribute, which is typically used to store a user name.

**Password**:
Enter the API Gateway message attribute that contains the password of the user to authenticate. The default message attribute is `authentication.subject.password`, which typically stores a password.

**Credential Format**:
Select the format of the credential stored in the API Gateway message attribute specified in the **Username** field above. By default, `User Name` is selected.

**Repository Name**:
Select an existing repository to authenticate the user against from the list. Alternatively, you can configure a new authentication repository by clicking the **Add** button. For more details on configuring the various types of repository supported by the API Gateway, see the *API Gateway Policy Developer Guide*.

# API key authentication

## Overview

API keys are supplied by client users and applications calling REST APIs to track and control how the APIs are used (for example, to meter access and prevent abuse or malicious attack). The **Authenticate API Key** filter enables you to securely authenticate an API key with the API Gateway.

API keys include a key ID that identifies the client responsible for the API service request. This key ID is not a secret, and must be included in each request. API keys can also include a confidential secret key used for authentication, which should only be known to the client and to the API service. You can use the **Authenticate API Key** filter to specify where to find the API key ID and secret key in the request message, and to specify timestamp and expiry options.

An example use case for this filter would be a client accessing a REST API service to invoke specific methods (for example, `startVM()` or `stopVM()`). To invoke these methods, you are required to provide your API key ID and secret key to the API Gateway. You can keep the secret key private by sending the request over HTTPS.

Alternatively, you can use the secret key to generate an HMAC digital signature. This means that the secret key is not sent in the request, but is inferred instead, because the message must have been signed using the required secret key. When the API service receives the request, it uses the API key ID to look up the corresponding secret key, and uses it to validate the signature and confirm the request sender.

The API Gateway supports the following API key types:

- Simple API keys including a key ID only. The API key ID is included in all requests to authenticate the client.
- Amazon Web Services style API keys including a key ID and a secret key, which are used together to securely authenticate the client. The API key ID is included in all requests to identify the client. The secret key is known only to the client and the API Gateway.

For more details on authenticating Amazon Web Services API keys, go to:
http://s3.amazonaws.com/doc/s3-developer-guide/RESTAuthentication.html

## General settings

Configure the following general settings:

**Name**:
Enter a suitable name for this filter in your policy.

**KPS Alias**:
Enter the alias name of the Key Property Store (KPS) used to store the API keys. For more details, see "Key Property Store" in the *API Gateway Policy Developer Guide*. Defaults to the example `ClientRegistry` supplied with the API Gateway. For details on storing API keys in the Axway Client Application Registry, see the *API Gateway OAuth User Guide*.

**Field Containing Secret**:
Enter the name of the field in the KPS that contains the secret. Defaults to `secretKey`.

# API key settings

Configure the following fields on the **API Key** tab:

**Where to find API key**:
To specify where to find the API key in the request message, select one of the following options:

- **API key is located in**:

  Select one of the following from the list:

  - `Query String`
  - `Header`
  - `Parameter`

  The default option is `Query String`. Enter the name in the text box. Defaults to `KeyId`.

- **API key is in Authorization header with format**:

  Select one of the following Authorization headers from the list:

  - `Amazon AWS s3 Authorization Header - "AWS apiKey + ":" + base64(signature)"`
  - `HTTP Basic Authentication Header - "Basic base64 (apiKey:secret)"`

  Defaults to the `Amazon AWS s3 Authorization Header`.

- **API key can be found using the following selector**:

  Enter the selector value that specifies the location of the API key. For details on selectors, see "Select configuration values at runtime" in the *API Gateway Policy Developer Guide*. Defaults to `${http.client.getCgiArgument("KeyId")}`.

**Where to find Secret key**:
To specify where to find the secret key in the request message, select the **Extract Secret** setting, and select one of the following options:

- **Secret key is in**:

  Select one of the following from the list:

  - `Query String`
  - `Header`
  - `Parameter`

  The default option is `Query String`. Enter the name in the text box. Defaults to `SecretKey`.

- **Secret key is in Authorization header with format**:

  Select the Authorization header from the list. Defaults to `HTTP Basic Authentication Header - "Basic base64(apiKey:secret)"`.

- **Secret key can be inferred from signature**:

  The client can use the secret key to generate a digital signature that is included in the request. When the API Gateway receives the request, it uses the API key ID to identify the client and look up the corresponding secret key in the Axway Client Application Registry. The secret key is then used to validate the signature and authenticate the client. To specify the signature format, select one of the following from the list:

  - `Amazon AWS s3 Authorization Header Authentication - "AWS apiKey + ":" + base64(signature)"`
  - `Amazon AWS s3 REST Authentication - "?Signature=<base64(signature)> &Expires=<seconds since epoch>&AWSAccessKeyId=<aws-id>"`

  Defaults to `Amazon AWS s3 Authorization Header Authentication`.

- **Secret key can be found using the following selector**:

  Enter the selector value that specifies the location of the secret key. For details on selectors, see "Select configuration values at runtime" in the *API Gateway Policy Developer Guide*. Defaults to `${http.client.getCgiArgument("SecretKey")}`.

**Authenticate API key and secret**:
Select whether to authenticate both the API key ID and the secret key. This means that the client must supply the API key ID and the secret key in the request message. This setting is selected by default.

# Advanced settings

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

**Validate Timestamp**:
Select whether to validate the API key timestamp using the settings specified below. This setting is unselected by default.

**Timestamp is located in**:
To specify where the timestamp is located in the request message, select one of the following from the list:

- `Header`
- `Parameter`
- `Query String`

The default option is `Header`. Enter the name in the text box. Defaults to `Date`.

**Timestamp format is** :
To specify the timestamp format, select one of the following from the list:

- `Simple Date Format`
- `Milliseconds since epoch`
- `Seconds since epoch`

The default option is `Simple Date Format`. Enter the format in the text box. Defaults to `EEE, dd MMM yyyy HH:mm:ss zzz`.

**Timestamp Drift +/-**:
You can specify a drift time in milliseconds to allow differences in the clock times between the machine on which the API key was generated and the machine on which the API Gateway is running. Defaults to +-`60000` milliseconds (one minute).

**Validate Expires**:
Select whether to validate the API key expiry details using the settings specified below. This setting is unselected by default.

**Expires is located in**:
To specify the location of the expiry details in the request message, select one of the following from the list:

- `Query String`
- `Header`
- `Parameter`

The default option is `Query String`. Enter the name in the text box. Defaults to `Expires`.

**Expires format is**:
To specify the format of the expiry details, select one of the following from the list:

- `Milliseconds since epoch`
- `Seconds since epoch`
- `Simple Date Format`

The default option is `Milliseconds since epoch`. Enter the format in the text box.

**Timestamp Drift +/-**:
You can specify a drift time in milliseconds to allow differences in the clock times between the machine on which the API key was generated and the machine on which the API Gateway is running. Defaults to `60000` milliseconds (one minute).

# Check session

## Overview

The **Check Session** filter checks for the presence of a valid cookie-based HTTP session. This filter tries to locate a valid session based on the value of a specified cookie name. If the session is found, the filter retrieves the user and sets it in the `authentication.subject.id` attribute.

The **Check Session** filter should be used with the **Create Session** and **End Session** filters to manage HTTP sessions. For more details, see:

- *Create session* on page 74
- *End session* on page 76

## Configuration

Complete the following fields to configure this filter:

**Name**:
Enter an appropriate name for this filter to display in a policy.

**Session cookie**:
Enter the name of the HTTP cookie used for the session. This filter tries to locate a valid session based on the value of the specified cookie name. The cookie name is output in the generated `http.session.cookie.name` message attribute. Defaults to `VIDUSR`.

# Create session

## Overview

The **Create Session** filter enables the API Gateway to create an HTTP session and configure various session attributes (for example, expiry, domain, and security). This filter requires an `authentication.subject.id` attribute for the user, and stores it in the HTTP session. This

session ID is used to create a cookie with a specified name, which is stored in the generated `http.session.cookie.name` attribute. The cookie is then sent to the user specified by the `authentication.subject.id` attribute.

The **Create Session** filter should be used with the **Check Session** and **End Session** filters to manage HTTP sessions. For more details, see:

- *Check session* on page 74
- *End session* on page 76

**Tip** The **Create Session** filter offers a more flexible approach to managing HTTP sessions than using the **HTTP Form-Based Authentication** filter. For example, the form-based approach does not include the ability to check or end sessions, and sessions are auto-renewed on each invocation of the filter. For more details, see *HTML form-based authentication* on page 80.

# Configuration

Complete the following fields to configure this filter:

**Name**:
Enter an appropriate name for this filter to display in a policy.

**Expiration time of session in milliseconds**:
Enter the HTTP session expiry timeout in milliseconds. When the session reaches the specified lifetime, it is automatically invalidated, and can no longer pass the **Check Session** filter.

**Session cookie**:
Enter the name of the HTTP cookie used for the session. This filter uses the HTTP session ID to create the cookie named by this field. The specified cookie name is output in the generated `http.session.cookie.name` message attribute. Defaults to `VIDUSR`.

**Session cookie domain**:
Enter the domain value for the `Set-Cookie` header (for example, `example.com`).This informs the browser that cookies should be sent back to the server for the specified domain only.

**Session cookie path**:
Enter the path value for the `Set-Cookie` header (for example, `/sales`). This informs the browser that cookies should be sent back to the server for the specified path only. Defaults to `/`.

**Session sent over SSL only**:
Select whether the session uses SSL only. When selected, this adds a `Secure` flag to the cookie.

**HTTP-only cookie**:
Select whether the session uses HTTP only. When selected, this adds an `HTTPOnly` flag to the cookie.

# End session

## Overview

The **End Session** filter terminates a cookie-based HTTP session. This filter tries to locate the session based on the value of a specified cookie name, and then invalidates it.

The **End Session** filter should be used with the **Create Session** and **Check Session** filters to manage HTTP sessions. For more details, see:

- *Create session* on page 74
- *Check session* on page 74

## Configuration

Complete the following fields to configure this filter:

**Name**:
Enter an appropriate name for this filter to display in a policy.

**Remove session cookie**:
Select whether to try and remove the session cookie. When selected, the API Gateway sends a new cookie with the expiry time set in the past. You must also set the same domain and path values that were used to create the session using the **Create Session** filter.

**Session cookie**:
Enter the name of the HTTP cookie used for the session. This filter tries to locate the session specified by this cookie name. This is output in the generated `http.session.cookie.name` message attribute. Defaults to `VIDUSR`.

**Session cookie domain**:
When **Remove session cookie** is selected, enter the same domain that was used to create the session in the **Create Session** filter (for example, `example.com`). This removes the cookie for the specified domain only.

**Session cookie path**:
When **Remove session cookie** is selected, enter the same path that was used to create the session in the **Create Session** filter (for example, `/sales`). This removes the cookie for the specified path only. Defaults to `/`.

# CA SOA Security Manager authentication

## Overview

CA SOA Security Manager can authenticate end users and authorize them to access protected web resources. When the API Gateway receives a message containing user credentials, it can forward the message to CA SOA Security Manager where the passed credentials are extracted from the message to authenticate the end user. When the message has been passed to CA SOA Security Manager, it can authenticate the user by the following methods:

- **XML Document Credential Collector**:
  Gathers credentials from the message and maps them to fields within a user directory.
- **XML Digital Signature**:
  Validates the X.509 certificate contained within an XML-Signature on the message.
- **WS-Security**:
  Extracts user credentials from WS-Security tokens contained in the message.
- **SAML Session Ticket**:
  Consumes a SAML session ticket from an HTTP header, SOAP envelope, or session cookie to authenticate the end user.

By delegating the authentication decision to CA SOA Security Manager, the API Gateway acts as a Policy Enforcement Point (PEP). It *enforces* the decisions made by the CA SOA Security Manager, which acts a Policy Decision Point (PDP). For more details, see the *CA SOA Security Manager Policy Configuration Guide*.

## Prerequisites

Integration with CA SOA Security Manager requires CA TransactionMinder SDK version 6.0 or later. You must add the required third-party binaries to your API Gateway and Policy Studio installations.

**Add third-party binaries to API Gateway**

To add third-party binaries to API Gateway, perform the following steps:

1. Add the binary files as follows:
   - Add `.jar` files to the `INSTALL_DIR/apigateway/ext/lib` directory.
   - Add `.so` files to the `INSTALL_DIR/apigateway/<platform>/lib` directory.
2. Restart API Gateway.

**Add third-party binaries to Policy Studio**

To add third-party binaries to Policy Studio, perform the following steps:

1. Select **Window > Preferences > Runtime Dependencies** in the Policy Studio main menu.
2. Click **Add** to select a JAR file to add to the list of dependencies.
3. Click **Apply** when finished. A copy of the JAR file is added to the `plugins` directory in your Policy Studio installation.
4. Click **OK**.
5. Restart Policy Studio with the `-clean` option. For example:

```
> cd INSTALL_DIR/policystudio/
> policystudio -clean
```

# Configuration

**Name**:
Enter a name for this filter to display in a policy.

**Agent Name**:
To act as a PEP for the CA SOA Security Manager, the API Gateway must have been set up as a *SOA Agent* with the Policy Server. For more details on how to do this, see the *CA SOA Security Manager Agent Configuration Guide*.

Click the button on the right to select a previously configured agent to connect to SOA Security Manager. This name *must* correspond with the name of an agent previously configured in the SOA Security Manager **Policy Server**. At runtime, the API Gateway connects as this agent to a running instance of SOA Security Manager.

To add an agent, right-click the **SiteMinder/SOA Security Manager Connections** tree node, and select **Add a SOA Security Manager Connection**. Alternatively, you can add SOA Security Manager connections under the **Environment Configuration** > **External Connections** node in the Policy Studio tree. For details on how to configure SOA Security Manager connections, see "Configure SiteMinder/SOA Security Manager connections" in the *API Gateway Policy Developer Guide*.

## *Message details settings*

While authenticating the user against CA SOA Security Manager, the user can also be authorized for a specified action on a particular resource. Configure the following fields in the **Message Details** section:

**Resource**:
Enter the name of the resource for which you want to ensure that the user has access to. By default, the `http.request.uri` message attribute is used, which contains the relative path on which the request was received by the API Gateway.

**Action**:
Specify the action that the user is attempting to perform on the specified resource. The API Gateway checks the user's entitlements in CA SOA Security Manager to ensure that the user is allowed to perform this action on the resource entered above. By default, the `http.request.verb` message attribute is used, which stores the HTTP verb used by the client when sending up the message.

**Protocol**:
Enter the protocol used by the client to access the requested resource. Users can have different access rights depending on their roles in the organization. For example, managers might be allowed to FTP to a given resource, but more junior employees might only be allowed to GET a resource using HTTP. Defaults to `http`.

**Headers**:
To carry out further authorization checks on the message, it is possible to forward the HTTP headers associated with the client message to the CA SOA Security Manager. By default, the `http.headers` message attribute is used to ensure that the original client headers are send to the CA SOA Security Manager.

# XmlToolkit.properties file

The `XmlToolkit.properties` file contains default properties used by the SOA agent, such as the URL of the CA SOA Manager, an identifier for the SOA agent, and an indication to the SOA Manager if it should perform fine-grained resource identification or not. The `XmlToolkit.properties` file can be found in the `/lib/modules/soasm` directory of your API Gateway installation.

```
#Wed Jul 18 15:02:16 BST 2007
WSDMResourceIdentification=yes
WS_UT_CREATION_EXPIRATION_MINUTES=60
```

The following properties are available:

- `WSDMResourceIdentification`

  This value cannot be configured from the Policy Studio, and so can only be set directly in the properties file. If this property is set to `no` (or if the properties file cannot be found) only a coarse-grained resource identification is performed on the requested URL. If this value is set to `yes`, a fine-grained resource identification including the requested URL, web service name, and SOAP operation (`<url>/<web service name>/<soap operation>`).

- `WS_UT_CREATION_EXPIRATION_MINUTES`

  Specifies the WS-Username Token age limit restriction in minutes. This setting helps prevent against replay attacks. The default token age limit is 60 minutes. See the following section for more information on modifying this setting.

### *Configure the user name and password digest token age restriction*

By default, the WS-Security authentication scheme imposes a 60 minute restriction on the age of user name and password digest tokens to protect against replay attacks.

You can configure a different value for the token age restriction for the API Gateway by setting the `WS_UT_CREATION_EXPIRATION_MINUTES` parameter in the `XmlToolkit.properties` file for that API Gateway. To configure the API Gateway to use a non-default age restriction for user name and password token authentication, complete the following steps:

1. Navigate to the`<INSTALL_DIR>/system/lib/modules/soasm` directory, where `INSTALL_DIR` points to the root of your API Gateway installation.
2. Open the `XmlToolkit.properties` file in a text editor.
3. Add the following line, where `token_age_limit` specifies the token age limit in minutes:

```
WS_UT_CREATION_EXPIRATION_MINUTES=token_age_limit
```

4. Save and close the `XmlToolkit.properties` file.
5. Restart API Gateway.

**Note** It is important to note the following:

- The properties file is written to the `/lib/modules/soasm` directory when a SOA Security Manager Authentication or Authorization filter is loaded at startup, or on server refresh (for example, when a configuration update is deployed), but only if the file does not already exist in this location.
- If the properties file already exists in the `/lib/modules/soasm` directory, the `WSDMResourceIdentification` property is *not* overwritten. In other words, the user is allowed to manually set this property independently of the Policy Studio.
- If the `WSDMResourceIdentification` property does not exist, it is given a default value of `yes` and written to the file.

# HTML form-based authentication

## Overview

HTML form-based authentication enables users to supply their user name and password details in an HTML form, and submit them to login to a system. Using HTML form-based authentication, normal HTTP authentication features such as HTTP basic or HTTP digest are not used. Instead, the user name and password are typically sent as HTML `<FORM>` data in an HTTP `POST` over SSL.

When the **HTML Form based Authentication** filter is configured, the API Gateway can authenticate the user details specified in the HTML form against a user profile stored in the API Gateway local repository, a database, or an LDAP directory. The **HTML Form based Authentication** filter also enables you to specify how HTTP sessions are managed (for example, session expiry, and applicable API Gateway domain or relative path).

**Tip** For an alternative approach to HTTP session management, which also includes the ability to check or to end sessions, see *Create session* on page 74.

# General settings

These settings enable you to configure general details such as the names of the HTML form fields, format of user credentials, and repository to validate credentials against. Complete the following settings:

**Name**:
Enter an appropriate name for the filter to display in a policy.

**Username**:
Enter the name of the HTML form field in which the user enters their user name. Defaults to `username`.

**Password**:
Enter the name of the HTML form field in which the user enters their password. Defaults to `password`.

**Format of Authentication Credentials**:
You must specify the format of the user credentials presented by the client because the API Gateway has no way of telling one credential format from another. Select one of the following from the list:

- `User Name`
- `X.509 Distinguished Name`

The selected format is then used internally by the API Gateway when performing authorization lookups against third-party Identity Management servers.

**Repository Name**:
This specifies the name of the authentication repository where all user profiles are stored. This can be in the API Gateway's local repository, a database, or an LDAP directory. Select a preconfigured repository from the list (for example, `Local User Store`).

You can add a new repository by right-clicking the appropriate node under **Environment Configuration** > **External Connections** > **Authentication Repository Profiles** (for example, **Database Repositories**), and selecting **Add a new Repository**. For more details on authentication repositories, see the *API Gateway Policy Developer Guide*.

## Session settings

The settings on the **Session** tab enable you to configure how HTTP sessions between the HTML form client and the API Gateway are managed. Complete the following settings:

**Create a session**:
Select whether to create an HTTP session. This setting is selected by default.

**Expiry of session in milliseconds**:
Enter the period of time in milliseconds before the session expires. Defaults to `600000` (10 minutes).

**Session cookie**:
Enter the name of the cookie used to manage the session. The default is `VIDUSR`.

**Session applicable for this domain**:
Enter the API Gateway domain name to which the session applies (for example, `dmz`).

**Session applicable for this path**:
Enter the API Gateway relative path to which the session applies. Defaults to `/`.

**Session sent over SSL only**:
Select whether the session is sent over an SSL connection only. This setting is not selected by default.

**HTTP Only cookie**:
Select the check box to add a HTTPOnly flag to the session cookie. This is not selected by default.

## Invalid login attempts settings

The settings on the **Invalid Login Attempts** tab enable you to specify how to handle invalid login attempts. You can choose to lock user accounts, ban IP addresses, or both, if a specified number of invalid attempts are made in a specified time period. The invalid attempt information is also stored in a cache.

**Note** If you are using two or more instances of HTTP basic, HTTP digest, or HTML form-based authentication filters in the same policy, and they share the same invalid attempts cache, you must use the same invalid attempts settings on each of the filters.

For more details on the settings on this tab, see *Invalid attempts* on page 84.

# HTTP Basic authentication

In HTTP Basic authentication, a client authenticate using an `Authorization` header. When you include a **HTTP Basic Authentication** filter in a policy, API Gateway can use the `Authorization` header to authenticate the client against a user profile stored in an authentication repository.

HTTP Basic authentication has two approaches:

- **Challenge-response handshake**:

  This is typical of situations where a browser is talking to a web server, for example, when a user uses the browser to access a web resource. The browser does not know that the server requires HTTP Basic authentication, so the initial request to the server does not include an `Authorization` header. The server responds with an `HTTP 401` response code, instructing the browser to authenticate by sending the `Authorization` header. The browser displays a dialog to the user to enter a user name and password combination, and passes the provided credentials to the `Authorization` header. The browser then sends a second request including the `Authorization` header to the server for authentication.

- **Direct authentication**:
  This is used mainly for machine-to-machine transactions with no human intervention. The client sends up the `Authorization` HTTP Basic authentication header in its first request to the server, so the challenge-response handshake is omitted.

Before being passed to the `Authorization` header, the credentials are concatenated, base64-encoded, for example:

```
Authorization:Basic dm9yZGVsOnZvcmRlbA==
```

The realm presented in the challenge for HTTP Basic authentication is the realm currently specified in **Environment Configuration > Server Settings > General**. For more details, see the *API Gateway Administrator Guide*.

# General settings

When you configure the **HTTP Basic Authentication** filter, you specify where API Gateway finds the user profiles for authentication. API Gateway can look up user profiles in a local repository on API Gateway, a database, or an LDAP directory. For details on adding users to a local repository, see "Manage API Gateway users" in the *API Gateway Policy Developer Guide*.

Configure the following settings:

**Credential Format**:
This defines the format of the user name presented to API Gateway during the HTTP Basic handshake that API Gateway uses for authorization checks against third-party Identity Management servers. API Gateway has no way of telling one format from another, so you must specify the format of the credential the client presents. The most common formats are user name or Distinguished Name (DName).

**Allow client challenge**:
This option is selected by default, and the client performs the HTTP Basic authentication challenge-response handshake with API Gateway. If you deselect this option, the client always sends the `Authorization` header to API Gateway for direct authentication.

**Allow retries**:
If you select this option, the user can try to enter their user name and password again if, for example, the authentication fails or has not yet been provided. The number of retries is controlled by the browser (usually three times).

**Remove HTTP authentication header**:
Select this option to remove the `Authorization` header from the downstream message. If this option is not selected, the incoming `Authorization` header is forwarded to the destination web service.

**Repository Name:**
Select the authentication repository where the user profiles are stored. This can be a local repository, a database, or an LDAP directory. For more details on authentication repositories, see "Configure authentication repositories" in the *API Gateway Policy Developer Guide*.

**Note** If you want to authenticate against an LDAP repository, you must set the authentication type of the LDAP connection to `Simple` . HTTP Basic authentication is not compatible with the other LDAP authentication types (`None`, `External` or `Digest-MD5`).

# Invalid attempts

The settings for invalid attempts specify how to handle invalid attempts: how many times authentication can be attempted, and what happens after this limit is crossed. By default, the number of invalid attempts is not limited.

You can select to lock user accounts, ban IP addresses, or both, and define how long that restriction lasts. In addition, you can define how many invalid attempts can be made and over what time period.

To ban IP addresses, you must also define the attribute that contains the IP address in the **Key is** field.

**Store invalid attempt information in cache:**
To store information on invalid attempts in cache, select this option, and choose the cache (local or distributed) you want to use, or add a new one. For more details on caches, see "Global caches" in the *API Gateway Policy Developer Guide*.

## *Restrictions when using the same invalid attempts cache in multiple filters*

You can configure a policy including more than one HTTP authentication filters (**HTTP Basic Authentication**, **HTTP Digest Authentication**, or **HTML Form based Authentication**), for example, to allow users to authenticate with either HTTP Basic or HTTP digest authentication. If the filters in your policy use the same cache for invalid attempts, you must use the same invalid attempts settings in all the filters sharing a cache. This ensures that if an user breaches the invalid attempts threshold, the lockout behavior and the user experience remains consistent.

# HTTP digest authentication

## Overview

A client can authenticate to API Gateway with a user name and password digest using *HTTP digest authentication*. When an **HTTP Digest Authentication** filter is configured, API Gateway requests the client to present a user name and password digest as part of the *HTTP digest challenge-response* mechanism. API Gateway can then authenticate this user against a user profile stored in the API Gateway's local repository.

The realm presented in the challenge for HTTP digest authentication is the realm currently specified in **Environment Configuration** > **Server Settings > General**. For more information on API Gateway settings, see the *API Gateway Administrator Guide*.

## General settings

The **HTTP Digest Authentication** filter enables you to specify where API Gateway can find user profiles for authentication purposes. API Gateway can look up user profiles in the API Gateway's local repository. For more information on adding users to the local repository, see "Manage API Gateway users" in the *API Gateway Policy Developer Guide*.

Complete the following settings:

**Name:**
Enter an appropriate name for the filter to display in a policy.

**Credential Format**:
The user name presented to API Gateway during the HTTP digest handshake can be of many formats, usually user name or Distinguished Name (DName). Because API Gateway has no way of inherently telling one format from another (for example, the client's user name could be a DName), you must specify the format of the credential presented by the client. This format is then used internally by API Gateway when performing authorization lookups against third-party Identity Management servers.

**Session Timeout**:
As part of the HTTP digest authentication protocol, API Gateway must generate a *nonce* (number used once) value, and send it to the client. The client uses this nonce to create the digest of the user name and password. However, it should only be allowed a certain amount of time to do so. The **Session Timeout** field specifies the length of time (in milliseconds) for which the nonce is valid.

**Allow retries**:
Select this option to allow the user to retry their user name and password in the browser when an HTTP 401 response code is received (for example, if authentication fails, or is not yet provided). The number of times that the browser displays the user name and password dialog when an HTTP 401 is received is controlled by the browser (usually three times). This setting is not selected by default.

**Remove HTTP authentication header:**
Select this option to remove the HTTP `Authorization` header from the downstream message. If this option is not selected, the incoming `Authorization` header is forwarded on to the destination web service.

**Repository Name**:
Select the name of the local authentication repository where all user profiles are stored.

You can add a new repository under the **Environment Configuration** > **External Connections** node. Right-click the **Local Repositories** node under **Authentication Repositories**, and select **Add a new repository**. For more details on authentication repositories, see the *API Gateway Policy Developer Guide*.

### *Invalid attempts*

The **Invalid Attempts** section enables you to specify how to handle invalid attempts. You can choose to lock user accounts, ban IP addresses, or both, if a specified number of invalid attempts are made in a specified time period. The invalid attempt information is also stored in a cache.

**Note** If you are using two or more instances of HTTP basic, HTTP digest, or HTML form-based authentication filters in the same policy, and they share the same invalid attempts cache, you must use the same invalid attempts settings on each of the filters.

For more details on the fields in this section, see *Invalid attempts* on page 84.

# HTTP header authentication

## Overview

You can use the **HTTP Header** filter in cases where the API Gateway receives end user authentication credentials in an HTTP header. A typical scenario would see the end user (or message originator) authenticating to an intermediary. The intermediary authenticates the end user and, to propagate the end-user credentials to the destination web service, the intermediary inserts the credentials into an HTTP header and forwards them onwards.

When the API Gateway receives the message, it performs the following tasks:

- Authenticates the sender of the message (the intermediary)
- Extracts the *end user* identity from the token in the HTTP header for use in subsequent authorization filters

**Note** In the case outlined above, the API Gateway does *not* attempt to reauthenticate the end user. It trusts that the intermediary has already authenticated the end user, and so the API Gateway does not authenticate the user again. However, it is good practice to authenticate the message sender (the intermediary). Any subsequent authorization filters use the end user credentials that were passed in the HTTP header.

## Configuration

The following configuration fields are available on this window:

**Name**:
Enter an appropriate name for this filter to display in a policy.

**HTTP Header Name**:
Enter the name of the HTTP header that contains the end user credentials.

**HTTP Header Type**:
Select the type of credentials that are passed in the named HTTP header. The following types are supported:

- X.509 Distinguished Name
- Certificate
- User Name

# IP address authentication

## Overview

You can configure the API Gateway to allow or deny machines, or groups of machines, access to resources based on their IP addresses. The main table on the window shows the IP addresses from which the API Gateway accepts or denies messages depending on what is configured.

The **IP Address** authentication filter uses the value stored in the `http.request.clientaddr` message attribute to determine whether to allow or deny access. This message attribute contains the remote host address from the TCP socket used in the connection between the client and the API Gateway.

## Configuration

Configure the following fields:

**Name**:
Enter a name for the filter to display in a policy.

**IP Addresses**:
You can add IP addresses by clicking the **Add** button, which displays the **Add IP Filter** dialog. Enter an **IP Address** and **Subnet Mask** to indicate a network to filter.

Messages sent from hosts belonging to this network are accepted or rejected based on what is configured in the section below. A **Subnet Mask** of `255.255.255.255` can be used to filter specific IP addresses. For more details, see *Configure subnet masks* on page 88.

**Note** If requests are made across a proxy, portal, or other such intermediary, the API Gateway filters on the IP address of the intermediary. Therefore, you should enter the IP address of the intermediary on this window, and not that of the user or client machine.

You can edit and remove existing IP addresses by selecting the **Edit** and **Remove** buttons.

**Access**:
Depending on whether the **Allow Access** or **Deny Access** radio button is checked, the IP addresses listed in the table are allowed or denied access to the web service.

# Configure subnet masks

An IP address is normally represented by a string of four numbers separated by periods (for example, `192.168.0.20`). Each number is normally represented as the decimal equivalent of an 8-bit binary number, which means that each number can take any value between 0 (all 8 bits cleared) and 255 (all 8 bits set).

A *subnet mask* (or netmask) is also a set of four number blocks separated by periods, each of which has a value in the range 0-255. Every IP address consists of two parts: the network address and the host number. The netmask is used to determine the size of these two parts. The positions of the bits set in the netmask represent the space reserved for the network address, while the bits that are cleared represent the space reserved for the host number. The netmask determines the range of IP addresses.

The following examples illustrate how netmasks work in practice.

## *Example 1 - Specify a range of IP addresses*

To allow requests from the following IP addresses:

`192.168.0.16`, `192.168.0.17`, `192.168.0.18`, and `192.168.0.19`.

Use the following address and netmask combination:

```
192.168.0.16/255.255.255.252
```

In more detail, the binary representation of the netmask is as follows:

```
11111111.11111111.11111111.11111100
```

The top 30 bits of the netmask indicate the network and the last 2 bits refer to the host on the network. These last 2 bits allow 4 different addresses as shown in the worked example below.

When the API Gateway receives a request from a certain IP address, the API Gateway performs a logical AND on the client IP address and the configured netmask. It also does a logical AND with the IP address entered in the IP Address filter and the configured subnet mask. If the AND-ed binary values are the same, the request from the IP address can be considered in the same network range as that configured in the filter.

The following worked example illustrates the mechanics of the IP address filtering. It assumes that you have entered the following in the IP Address and Netmask fields in the IP Address filter:

| Field | Value |
|---|---|
| **IP Address** | `192.168.0.16` |
| **Net Mask** | `255.255.255.252` |

```
Step 1: AND the IP address and Netmask configured in the IP Address Filter:
11000000.10100000.00000000.00010000 (192.168.0.16)
AND
11111111.11111111.11111111.11111100 (255.255.255.252)
========================================
11000000.10100000.00000000.00010000
Step 2: Request is received from 192.168.0.18:
11000000.10100000.00000000.00010010 (192.168.0.18)
AND
11111111.11111111.11111111.11111100 (255.255.255.252)
========================================
11000000.10100000.00000000.00010000
===> AND-ed value is equal to the result for 192.168.0.16.
===> Therefore the client IP address is inside the configured range.
Step 3: Request is received from 192.168.0.20:
11000000.10100000.00000000.00010100 (192.168.0.20)
AND
11111111.11111111.11111111.11111100 (255.255.255.252)
========================================
11000000.10100000.00000000.00010100
===> AND-ed value is NOT equal to the result for 192.168.0.16.
===> Therefore the client IP address is NOT inside the configured range.
```

## *Example 2 – Specify an exact IP address*

You can also specify an exact IP address by using a netmask of`255.255.255.255`. When this netmask is used, only requests from this client IP address is allowed or blocked, depending on what is configured in the filter. This example assumes that the following details have been configured in the IP Address filter:

| Field | Value |
|---|---|
| **IP Address** | `192.168.0.36` |
| **Net Mask** | `255.255.255.255` |

```
Step 1: AND the IP address and Netmask configured in the IP Address Filter:
11000000.10100000.00000000.00100100 (192.168.0.36)
AND
11111111.11111111.11111111.11111111 (255.255.255.255)
=======================================
11000000.10100000.00000000.00100100
Step 2: Request is received from client with IP address of 192.168.0.37:
11000000.10100000.00000000.00100101 (192.168.0.37)
AND
11111111.11111111.11111111.11111111 (255.255.255.255)
=======================================
11000000.10100000.00000000.00100101
===> AND-ed value is NOT equal to the result for 192.168.0.36
===> Therefore the client IP address is NOT inside the configured range.
```

# Insert SAML authentication assertion

## Overview

After successfully authenticating a client, the API Gateway 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="axway-1056477425082"
    Id="axway-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>
```

# 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 (that generate the assertion) and the machines that consume the assertion. The specified time is subtracted from the time at which the API Gateway 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.

**Note** 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, Axway recommend that SAML 1.1 or 2.0 is used when signing assertions as they both use 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. 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 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 "Configure XPath expressions" in the *API Gateway Policy Developer Guide*.

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, 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=axway</dsig:X509SubjectName>
      <dsig:X509Certificate>
        MIICmzCCAY ...... mB9CJEw4Q=
      </dsig:X509Certificate>
    </dsig:X509Data>
  </dsig:KeyInfo>
</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 includes the key used to prove that the API Gateway 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` |

| Method | Meaning | Value |
|---|---|---|
| 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. | `urn:oasis:names:tc:SAML:1.0:cm:artifact` |
| Sender Vouches | Use this confirmation method to assert that the API Gateway 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 (see *XML signature generation* on page 322). | `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 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** window, select the check box next to the certificate that is associated with the key 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, or environment variable). For more details, see "Select configuration values at runtime" in the *API Gateway Policy Developer Guide*.

Symmetric Key:
To use a symmetric key as proof that the API Gateway 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 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, an **XML Signature Generation** filter) has already used a symmetric key, you can reuse this key as proof that the API Gateway is the holder-of-key entity. Enter the name of the selector expression (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, or environment variable). For more details, see "Select configuration values at runtime" in the *API Gateway Policy Developer Guide*.
- **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 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 Selector Expression**:
  Alternatively, if the recipient's certificate has already been used (perhaps to encrypt part of the message), the certificate 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, or environment variable). For more details, see "Select configuration values at runtime" in the *API Gateway Policy Developer Guide*.
- **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 check 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 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

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 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 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="axway">
    <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="axway" 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="Axway-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 timestamp

## Overview

In any secure communications protocol, it is crucial that secured messages do not have an indefinite life span. In secure web services transactions, a WS-Utility (WSU) timestamp can be inserted into a WS-Security Header to define the lifetime of the message in which it is placed. A message containing an expired timestamp should be rejected immediately by any web service that consumes the message.

Typically, the timestamp contains `Created` and `Expires` times, which combine to define the lifetime of the timestamp. The following shows an example `wsu:Timestamp`:

```
<wsu:Timestamp xmlns:wsu="http://schemas.xmlsoap.org/ws/2002/07/utility">
  <wsu:Created>2009-03-16T16:32:22Z</wsu:Created>
  <wsu:Expires>2009-03-16T16:42:22Z</wsu:Expires>
</wsu:Timestamp>
```

Because the WS-Utility timestamp is inserted into the WS-Security header block, it is also referred to as a WSS timestamp. For example, see *Extract WSS timestamp* on page 36.

## Configuration

Complete the following fields to configure the API Gateway to insert a timestamp into the message:

**Name**:
Enter an intuitive name for the filter to display in a policy.

**Actor**:
The timestamp is inserted into the WS-Security header identified by the SOAP Actor selected here.

**Expires In**:
Configure the lifetime of the timestamp (and hence the message into which the timestamp is inserted) by specifying the expiry time of the assertion. The expiry time is expressed in days, hours, minutes, and seconds.

**Layout Type**:
In cases where the timestamp must adhere to a particular layout as mandated by the WS-Policy `<Layout>` assertion, you must select the appropriate layout type. A web service that enforces a WS-Policy might reject the message if the layout of security elements in the SOAP header is incorrect. Therefore, you must ensure that you select the correct layout type.

# Insert WS-Security UsernameToken

## Overview

When a client has been successfully authenticated, the API Gateway 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 using HTTP digest authentication. After successfully authenticating the user, the API Gateway 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="axway"
  xmlns:wsu="http://schemas.xmlsoap.org/ws/2003/06/utility">
    <wsu:Created>2006.01.13T-10:42:43Z</wsu:Created>
    <wsse:Username>axway</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 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 to display in a policy.

**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 to include a nonce in the UsernameToken. A nonce is 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 inserts the user's password into the generated WS-Security UsernameToken. It can insert a **Clear** or **SHA1 Digest** version of the password, depending on which radio button you select. Axway 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 selector in the field provided. The default is `${authentication.subject.password}`, which contains the user password to authenticate to the API Gateway.

### *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.

# Kerberos client authentication

## Overview

You can configure the API Gateway to act as a Kerberos client and to obtain a service ticket for a specific Kerberos service. The service ticket makes up part of the Kerberos client-side token that is sent to the Kerberos service. If the service can validate the token, the client is authenticated successfully.

For more details on different Kerberos setups with API Gateway, see *API Gateway Kerberos Integration Guide*.

There are two filters you can use to configure the client-side transaction:

- Use a **Connection** filter to authenticate to a Kerberos service by inserting a client-side Kerberos token into the `Authorization` HTTP header. For more information on authenticating to a Kerberos service using a client-side Kerberos token, see *Connection* on page 387.
- Use a **Kerberos Client** filter to send the client-side Kerberos token in a `BinarySecurityToken` block in the SOAP message.

To add a **Kerberos Client** filter, open the **Authentication** category, and drag the filter onto the policy canvas. The following sections describe how to configure the different fields of this 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.

Configure the following fields:

**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 selected **Kerberos Service Principal**. The TGT is acquired at server startup, server refresh (for example, when an update to configuration is deployed), and when the TGT expires.

To select which Kerberos client to use, click the **...** button, and select a previously configured Kerberos client. To add a Kerberos client, right-click **Kerberos Clients**, and select **Add Kerberos Client**. You can also add Kerberos clients under **Environment Configuration > External Connections** in the node tree. For more details, see "Configure Kerberos clients" in the *API Gateway Policy Developer Guide*.

**Kerberos Service Principal**:
The Kerberos client must obtain a service ticket from the Kerberos Ticket Granting Server (TGS) for the Kerberos service principal you set in this field. The TGS grants the Kerberos client a ticket for the selected principal, and the client can then send this ticket to the Kerberos service. The principal in the ticket must match the Kerberos service's principal for the client to be successfully authenticated.

The service principal name (SPN) can be used to uniquely identify the Kerberos service in the Kerberos realm.

To select which Kerberos service principal to use, 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 **Kerberos Principals**, and select **Add Kerberos Principal**. You can also add Kerberos principals under **Environment Configuration > External Connections** in the node tree. For more details, see "Configure Kerberos principals" in the *API Gateway Policy Developer Guide*.

**Kerberos Standard**:
When using the **Kerberos Client** filter to insert Kerberos tokens into SOAP messages, the Kerberos client can authenticate to Kerberos services using to two standards:

- **Web Services Security Kerberos Token Profile 1.1** – When using the Kerberos Token Profile, the client-side Kerberos token is inserted into a `BinarySecurityToken` block in the SOAP message. If you select this option, you must configure the fields on the **Kerberos Token Profile** tab. You can use signing and encryption filters to sign and encrypt the SOAP message using the Kerberos session key.
- **WS-Trust for Simple and Protected Negotiation Protocol (SPNEGO)** – When using the WS-Trust for SPNEGO standard, a series of requests and responses occur between the Kerberos client and the Kerberos service to establish a secure context using WS-Trust and WS-SecureConversation. After establishing the secure context, a further series of requests and responses produce a shared secret key that can be used to sign and encrypt *real* requests to the Kerberos service.If you select this option, it is not necessary to configure the fields on the **Kerberos Token Profile** tab, but you must configure the **Kerberos Client** filter as a part of a complicated policy set up to handle the multiple request and response messages involved in establishing the secure context between the Kerberos client and service.

## Kerberos token profile settings

You only need to configure the fields on the **Kerberos Token Profile** tab if you set **Kerberos Standard** to **Web Services Security Kerberos Token Profile 1.1** on the **Kerberos Client** tab. This tab allows you to configure where to insert the `BinarySecurityToken` in the SOAP message.

**Where to Place BinarySecurityToken**:
You can insert the `BinarySecurityToken` inside a named WS-Security Actor/Role in the SOAP message, or you can specify an XPath expression to indicate where the token should be inserted.

To insert the token into a WS-Security element in the SOAP Header element, select **WS-Security Element**. The `BinarySecurityToken` is inserted into a WS-Security block for the specified actor/role. You can use the default option `Current actor/role only`, or enter a named actor/role in the field provided.

Alternatively, to use an XPath expression to specify where to insert the `BinarySecurityToken`, select **XPath Location**. Click the **Add** button to add a new XPath expression, or select an existing XPath expression from the list. You can also edit or delete existing expressions, if needed. For more information, see "Configure XPath expressions" in the *API Gateway Policy Developer Guide*.

**Note** You can control inserting the `BinarySecurityToken` relative to the node pointed to by the XPath expression. Select the **Append** to insert the token after or **Before** to insert the token before the node.

**BinarySecurityToken Value Type**:
Currently, the only supported `BinarySecurityToken` type is `GSS_Kerberosv5_AP_REQ`. The selected type is specified in the generated `BinarySecurityToken`.

# Kerberos service authentication

## Overview

The API Gateway can act as a Kerberos service to consume Kerberos tokens sent from a client in the HTTP header or in the message itself. The Kerberos client must have obtained a ticket from the Ticket Granting Server (TGS) for this Kerberos service. The service ticket makes up part of the Kerberos client-side token that is sent to the Kerberos service. If the service can validate the token, the client is authenticated successfully.

For more details on different Kerberos setups with API Gateway, see *API Gateway Kerberos Integration Guide*.

To add a **Kerberos Service** filter, open the **Authentication** category, and drag the filter onto the policy canvas. The following sections describe how to configure the different fields of this filter.

# General settings

The fields configured on the **Kerberos Client** tab determine how the Kerberos service consumes a Kerberos token from a specific Kerberos client.

Configure the following fields:

**Kerberos Service**:
The **Kerberos Service** you select in this field is responsible for consuming the Kerberos client's Kerberos token. The Kerberos client must have obtained a ticket for the Kerberos service's principal name to be able to authenticate to the Kerberos service.

Click the **...** button, and select a previously configured Kerberos service. To add a Kerberos service, right-click **Kerberos Services**, and select **Add Kerberos Service**. You can also add Kerberos services under **Environment Configuration > External Connections** in the node tree. For more details, see "Configure Kerberos services" in the *API Gateway Policy Developer Guide*.

# Kerberos standard settings

Configure the following fields on the **Kerberos Standard** tab:

**Kerberos Standard**:
Select one of the following Kerberos standards:

- Kerberos Token Profile
- WS-Trust for SPNEGO
- SPNEGO over HTTP

**Note** The Kerberos Service filter consumes the Kerberos client-side tokens regardless of whether the token is sent at the message layer in the SOAP message, or at the transport layer in an HTTP header.

**Client Token Location for Message-Level Standards**:
The Kerberos service ticket can be sent in the `Authorization` HTTP header, or inside the message itself (for example, inside a `<BinarySecurityToken>` element). Alternatively, the Kerberos token can be in a message attribute.

Select one of the following options:

- **Message Body**:
  Select this option if you expect the Kerberos service ticket to be contained in the message. You must enter an XPath expression to point to the expected location of the Kerberos token. You can select some default expressions that point to common locations from the list. To add a new XPath expression, click **Add**. You can also edit or delete existing expressions, if needed. For more details, see "Configure XPath expressions" in the *API Gateway Policy Developer Guide*.
- **Selector Expression**:
  When using the **WS-Trust for SPNEGO** standard, the **Consume WS-Trust** filter places the client-side Kerberos token inside the `ws.trust.spnego.token` message attribute.

# Message level settings

You can configure settings adhering to the message-level standards (for example, Kerberos Token Profile and WS-Trust for SPNEGO) on the **Message Level** tab.

**Extract Session Keys**:
You must select this option to use the Kerberos/SPNEGO session keys to sign, encrypt, or decrypt a message in a subsequent filter. This option is only available when the token is extracted from the message body.

**Key Length**:
When using **WS-Trust for SPNEGO** standard, the **Kerberos Service** filter generates a new symmetric key and wraps it using the Kerberos session key. This setting determines the length of the new symmetric key.

**Cache Security Context Session Key**:
The service-side policy might need to cache the session key in order to process (decrypt and verify) multiple requests from a Kerberos client. Use this field to select a cache for the session key.

# Transport level settings

The options on the **Transport Level** tab are specific to Kerberos tokens received over HTTP, and are only relevant if you selected to use **SPNEGO Over HTTP** standard.

**Cookie Name**:
The initial handshake between a Kerberos client and a Kerberos service can sometimes involve the exchange of a series of request and responses until the secure context has been established. In such cases, you can use a HTTP cookie to keep track of the context across multiple request and response messages. Enter the name of the cookie in the text field.

**Allow Client Challenge**:
In some cases, a Kerberos client might not authenticate (send the `Authorization` HTTP header) to the Kerberos service on first request. The Kerberos service then responds with an HTTP 401 response code, instructing the client to authenticate to the server by sending the `Authorization` header. The Kerberos client sends a second request, this time with the `Authorization` header containing the relevant Kerberos token. Select this option to allow this kind of negotiation between the Kerberos client and service.

**Client Sends Body Only After Context is Established**:
The Kerberos client might wait to mutually authenticate the Kerberos service before sending the body of the message. Select this option to enable the Kerberos service to accept the body after the context has been established if the Kerberos client provides the known cookie. The cookies are cached in the cache you configure.

# Advanced SPNEGO settings

The settings on the **Advanced SPNEGO** tab apply only to the **WS-Trust for SPENGO** and **SPENGO over HTTP** standards.

**Cache Partially Established Contexts**:
In theory, a Kerberos client and a Kerberos service may need to send and receive a number of tokens between each other to authenticate. In this case, the **Kerberos Service** filter must cache the partially established context for each Kerberos client. The contexts are only cached during the establishment of the context.

In practice, however, a single client-side Kerberos token is normally enough to establish a context on the service-side, making this setting redundant.

# SAML authentication

## Overview

A Security Assertion Markup Language (SAML) *authentication assertion* is issued as proof of an authentication event. Typically, an end user authenticates to an intermediary, who generates a SAML authentication assertion to prove that it has authenticated the user. The intermediary inserts the assertion into the message for consumption by a downstream web service.

When the API Gateway receives a message containing a SAML authentication assertion, it does not attempt to authenticate the end user again. Instead, it authenticates the sender of the assertion (the intermediary) to ensure that only the intermediary could have issued the assertion, and then validates the authentication details contained in the assertion. Therefore, the API Gateway performs the following tasks in this scenario:

- Authenticates the sender of the message (the intermediary)
- Extracts the end user's identity from the authentication assertion and validates the authentication details

The **SAML Authentication** filter performs the second task. A separate authentication filter must be placed before this filter in the policy to authenticate the sender of the assertion. The end user's identity is used in any subsequent authorization filters.

The following sample SOAP message contains a SAML authentication assertion:

```
<?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="axway-1056477425082"
    Id="axway-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>
```

# Details settings

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

**SOAP Actor/Role**:
If you expect the SAML assertion to be embedded in a WS-Security block, you can identify this block by specifying the SOAP Actor or Role of the WS-Security header that contains the assertion.

**XPath Expression**:
Alternatively, if the assertion is not contained in a WS-Security block, you can enter an XPath expression to locate the authentication assertion. You can configure XPath expressions using the **Add**, **Edit**, and **Delete** buttons.

**SAML Namespace**:
Select the SAML namespace that must be used on the SAML assertion for this filter to succeed. If you do not wish to check the namespace, select the `Do not check version` option from the list.

**SAML Version**:
Specify the SAML version that the assertion must adhere to by entering the major version in the first field, and the minor version in the second field. For example, for SAML 2.0, enter `2` in the first field and `0` in the second field.

**Drift Time**:
The *drift time*, specified in seconds, is used when checking the validity dates on the authentication assertion. The drift time allows for differences between the clock times of the machine on which the assertion was generated and the machine hosting the API Gateway.

**Remove enclosing WS-Security element on successful validation**:
Select this check box to remove the WS-Security block that contains the SAML assertion after the assertion has been successfully validated.

## Trusted issuer settings

You can use the table on the **Trusted Issuers** tab to select the issuers that you consider trusted. In other words, this filter only accepts assertions that have been issued by the SAML authorities selected here.

Click the **Add** button to display the **Trusted Issuers** window. Select the Distinguished Name of a SAML authority whose certificate has been added to the certificate store, and click **OK**. Repeat this step to add more SAML authorities to the list of trusted issuers.

# SAML PDP authentication

## Overview

The API Gateway can request an authentication decision from a Security Assertion Markup Language (SAML) Policy Decision Point (PDP) for an authenticated client using the SAML Protocol (SAMLP). In such cases, the API Gateway presents evidence to the PDP in the form of some user credentials, such as the Distinguished Name of a client's X.509 certificate.

The PDP decides whether to authenticate the end user. It then creates an authentication assertion, signs it, and returns it to the API Gateway in a SAMLP response. The API Gateway can then perform a number of checks on the response, such as validating the PDP signature and certificate, and examining the assertion. It can also insert the SAML authentication assertion into the message for consumption by a downstream web service.

# Request settings

The **Request** tab describes how the API Gateway should package the SAMLP request before sending it to the SAML PDP.

You can configure the following fields on the **Request** tab:

**SAML PDP URL Set**:
You can configure a group of SAML PDPs to which the API Gateway connects in a round-robin fashion if one or more of the PDPs are unavailable. This is known as a SAML PDP URL set. Click the button on the right, and select a previously configured SAML PDP URL set in the tree. To add a URL set, right-click the **SAML PDP URL Sets** tree node, and select **Add a URL Set**. Alternatively, you can configure a SAML PDP URL set under the **Environment Configuration** > **External Connections** node in the Policy Studio tree.

**SOAP Action**:
Enter the SOAP action required to send SAMLP requests to the PDP. Click the **Use Default** button to use the following default SOAP action as specified by SAMLP:

```
http://www.oasis-open.org/committees/security
```

**SAML Version**:
Select the SAML version to use in the SAMLP request.

**Signing Key**:
If the SAMLP request is to be signed, click the **Signing Key** button, and select the appropriate signing key from the certificate store.

## *SAML subject settings*

You can describe the *subject* of the SAML assertion on the **SAML Subject** tab. Complete the following fields:

**Subject Selector Expression**:
Enter a selector expression for the message attribute that contains the user name of an authenticated user. The default value is `${authentication.subject.id}`.

**Subject Format**:
Select the format of the subject selected in the **Subject Selector Expression** field above.

**Note** There is no need to select a format here if the **Subject Selector Expression** field is set to `authentication.subject.id`.

## Subject confirmation settings

The settings on the **Subject Confirmation** 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, 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=axway</dsig:X509SubjectName>
      <dsig:X509Certificate>
        MIICmzCCAY ...... mB9CJEw4Q=
      </dsig:X509Certificate>
    </dsig:X509Data>
  </dsig:KeyInfo>
</saml:SubjectConfirmation>
```

You must configure the following fields on the **Subject Confirmation** 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 | A <SubjectConfirmation> is inserted into the SAMLP request. The <SubjectConfirmation> contains a <dsig:KeyInfo> section with the certificate of the user selected to sign the SAMLP request. The user selected to sign the SAMLP request must be the authenticated subject ( authentication.subject.id).<br>Select the **Include Certificate** option if the signer's certificate is to be included in the SubjectConfirmation block. Alternatively, select the **Include Key Name** option if only the key name is to be included. | urn:oasis:names:tc:SAML:1.0:cm:<br>holder-of-key |
| Bearer | A <SubjectConfirmation> is inserted into the SAMLP request. | urn:oasis:names:tc:SAML:1.0:cm:<br>bearer |
| SAML Artifact | A <SubjectConfirmation> is inserted into the SAMLP request. | urn:oasis:names:tc:SAML:1.0:cm:<br>artifact |
| Sender Vouches | A <SubjectConfirmation> is inserted into the SAMLP request. The SAMLP request must be signed by a user. | urn:oasis:names:tc:SAML:1.0:cm:<br>bearer |

If the **Method** field is left blank, no <ConfirmationMethod> block is inserted into the assertion.

**Include Certificate**:
Select this option to include the SAML subject's certificate in the <KeyInfo> section of the <SubjectConfirmation> block.

**Include Key Name**:
Alternatively, to not include the certificate, select this option to only include the key name in the <KeyInfo> section.

## Response settings

The **Response** tab configures the SAMLP response returned from the SAML PDP. The following fields are available:

**SOAP Actor/Role**:
If the SAMLP response from the PDP contains a SAML authentication assertion, the API Gateway can extract it from the response and insert it into the downstream message. The SAML assertion is inserted into the WS-Security block identified by the specified SOAP actor/role.

**Drift Time**:
The SAMLP request to the PDP is time stamped by the API Gateway. To account for differences in the times on the machines running the API Gateway and the SAML PDP the specified time is subtracted from the time at which the API Gateway generates the SAMLP request.

# SSL authentication

## Overview

A client can mutually authenticate to the API Gateway through the exchange of X.509 certificates. An X.509 certificate contains identity information about its owner and is digitally signed by the Certificate Authority (CA) that issued it.

A client presents such a certificate to the API Gateway while the initial SSL/TLS session is being negotiated, in other words, during the *SSL handshake*. The **SSL Authentication** filter extracts this information from the client certificate and sets it as message attributes. These attributes can then be used by subsequent filters in the policy.

The **SSL Authentication** filter can be used as a decision-making node on the policy. For example, it can be used to determine a path through a policy based on how users authenticate to the API Gateway.

# STS client authentication

## Overview

The **Security Token Service Client** filter enables the API Gateway to act as a client to a Security Token Service (STS). An STS is a third-party web service that authenticates clients by validating credentials and issuing security tokens across different formats (for example, SAML, Kerberos, or X.509). The API Gateway can use the **Security Token Service Client** filter to request security tokens from an STS using WS-Trust. WS-Trust specifies the protocol for issuing, exchanging, and validating security tokens.

An STS has its own security requirements for authenticating and authorizing requests for tokens. This means that the API Gateway might need to insert tokens, digitally sign, and encrypt the request that it sends to the STS for the required token. Because the STS is exposed as a web service, it should have a WSDL file with WS-Policies that describe its security requirements.

For example, the API Gateway can use the **Security Token Service Client** filter to request tokens that it cannot issue itself, and which might be required by an endpoint service. The endpoint service might require tokens to be signed by a particular authority (STS), or there might be a requirement for a token that contains a key encrypted for the endpoint service, and which only the STS can generate. You can also use the **Security Token Service Client** filter to virtualize an STS using the API Gateway.

# Example request

Using WS-Trust, requests for tokens are placed in a `RequestSecurityToken` (RST) element in the SOAP `Body` element. The STS returns the requested token in a `RequestSecurityTokenResponse` (RSTR) element in the SOAP `Body`. The following example is an extract from a token request message sent from the API Gateway to the STS:

```
<soap:Body
  xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/
  oasis-200401-wss-wssecurity-utility-1.0.xsd"
  wsu:Id="Id-0000012e71431904-00000000011d5641-19">
    <wst:RequestSecurityToken
      xmlns:wst="http://docs.oasis-open.org/ws-sx/ws-trust/200512"
      Context="Id-0000012e71431904-00000000011d5641-15">
        <wst:RequestType>
          http://docs.oasis-open.org/ws-sx/ws-trust/200512/Issue
        </wst:RequestType>
        <wst:TokenType>
          http://docs.oasis-open.org/wss/oasis-wss-saml-token-profile-1.1#SAMLV1.1
        </wst:TokenType>
        <wst:KeyType>
          http://docs.oasis-open.org/ws-sx/ws-trust/200512/SymmetricKey
        </wst:KeyType>
        <wst:Entropy>
            <wst:BinarySecret
              Type="http://schemas.xmlsoap.org/ws/2005/02/trust/SymmetricKey">
              WLQmo5mRYiBRqq2D7677Dg==
            </wst:BinarySecret>
        </wst:Entropy>
        <wsp:AppliesTo
          xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy">
            <wsa:EndpointReference
              xmlns:wsa="http://www.w3.org/2005/08/addressing">
                <wsa:Address>default</wsa:Address>
            </wsa:EndpointReference>
        </wsp:AppliesTo>
    </wst:RequestSecurityToken>
```

```
</soap:Body>
```

In this simple example, the client (API Gateway) requests a SAML token with a symmetric `KeyType`. The SAML token is requested for an endpoint service named `default`. An optional `OnBehalfOf` token is not supplied.

# Request settings

Configure the following general request settings on the **Request** tab:

**Request Type**:
Select one of the following request types:

- **Issue**: A request to issue a token. This is the default request type.
- **Validate**: A request to validate a token.

**Token Type to Request**:
Select the token type to request from the STS (for example, `SAML 1.0`, `SAML 1.1`, `SAML 2.0`, or `UsernameToken`). You can also request a custom token type by entering the custom token URI (for example, `http://www.mycustomtoken.com/EmailToken`). The default is `SAML 1.1`.

## *Issue: POP Key*

A *proof-of-possession* (POP) security token contains secret data used to demonstrate authorized use of an associated security token. Typically, the POP data is encrypted with a key known only to the recipient of the POP token. For **Issue** requests, you can configure the following POP key settings on the **Issue: POP Key** tab:

**Proof of Possession Key Type**:
Select the POP key type for the token you are requesting. This only applies to certain types of tokens (for example, SAML tokens). Select one of the following key types from the list:

| | |
|---|---|
| `SymmetricKey` | When a SAML token is requested with a symmetric POP key, the SAML assertion returned by the STS has a subject confirmation type of `holder-of-key`. The subject confirmation data contains a symmetric key encrypted for the endpoint service. The API Gateway (the client) can request the SAML token from the STS with the endpoint service specified as the token scope, so the STS knows what certificate to use to encrypt the symmetric key it places in the SAML assertion's subject confirmation data.<br>The API Gateway cannot decrypt the symmetric key in the SAML assertion because it is encrypted for the endpoint service. The STS passes the symmetric key to the requesting API Gateway in the RSTR so that the API Gateway also has the symmetric key. It can then use the SAML assertion (symmetric key) to sign the message to the endpoint service, proving that it holds the key in the SAML assertion. The endpoint service can verify the signature because it can decrypt the key in the SAML assertion. This is the default POP key type. |
| `PublicKey` | When a SAML token is requested with a public asymmetric POP key, the SAML assertion returned by the STS has a subject confirmation type of `holder-of-key`. The subject confirmation data contains a public key or certificate. The API Gateway (the client) can also use this SAML assertion to sign messages to the endpoint service using the related private key, thus proving they hold the key referenced in the SAML assertion. The public key in the SAML assertion is not encrypted because it is not sensitive data.<br>This SAML assertion can be used to sign messages to multiple endpoint services because it does not contain a key encrypted for a specific service. The API Gateway can specify the public key used in the **Public Proof of Possession Key** settings. This public key can be associated with a certificate in the certificate store, or generated on-the-fly using the **Generate Key** filter. For more details, see *Generate key* on page 270. |
| `Bearer` | When a SAML token is requested with a bearer POP key, the SAML assertion returned by the STS has a subject confirmation type of `bearer`. In this case, the SAML token does not contain a POP key. |

**Note** An STS can also generate a SAML token with a subject confirmation type of `sender-vouches`. In this case, the endpoint service trusts the client directly, the SAML assertion does not need to be signed by the STS. The client signs the SAML assertion and the SOAP `Body` before sending the message to the endpoint service. This type of SAML assertion does not map to a value for **Proof of Possession Key Type**, but can be returned from the STS if no key type is specified.

**Key Size**:
Enter the key size in bits to indicate the desired strength of the security. Defaults to `256` bits.

**Entropy Type**:
If the **Proof of Possession Key Type** requested is a `SymmetricKey`, you must specify an **Entropy Type**. If the API Gateway provides *entropy*, this means that it provides some of the binary

material used to generate the symmetric key. In general, both the API Gateway and the STS provide some entropy for the symmetric key (a computed key). However, either side can also fully generate the symmetric key. Select one of the following options:

- **None**: The API Gateway does not provide any entropy, so the STS must fully generate the symmetric key.
- **Binary Secret**: The API Gateway provides entropy in the form of a Base64-encoded binary secret (or key). You must specify a **Binary Secret Type**. For details, see the next setting.
- **Encrypted Key**: The API Gateway provides entropy in the form of an `EncryptedKey` element. You must configure an **XML-Encryption** filter in the policy, which applies security before creating the WS-Trust message. This filter generates a symmetric key and encrypts it, but does not encrypt any data. The key must be encrypted with the STS certificate.

**Binary Secret Type**:
If the **Entropy Type** is **Binary Secret**, you must specify a **Binary Secret Type**. Select one of the following:

- Nonce: The API Gateway generates a nonce value and places it in the RST.
- SymmetricKey: The **Binary Secret Message Attribute** value must be specified. In this case, this is the name of the message attribute that contains the symmetric key passed to the STS to be used as entropy for generating the POP symmetric key. The type of this message attribute must be `byte[]` when the **Binary Secret Type** is `SymmetricKey`.
- AsymmetricKey: The **Binary Secret Message Attribute** value must be specified. In this case, this is the name of the message attribute that contains the private asymmetric key passed to the STS to be used as entropy for generating the POP symmetric key. The type of this message attribute must be `byte[]`, `PrivateKey`, `KeyPair`, or `X509Certificate` when the **Binary Secret Type** is `AsymmetricKey`. In each case, the private key is used.

**Binary Secret Message Attribute**:
Enter or select the message attribute that contains the binary secret. This setting is required when the **Binary Secret Type** is `SymmetricKey` or `AsymmetricKey`.

**Computed Key Algorithm**:
When both the API Gateway and STS provide entropy values for the symmetric POP key, you can specify a computed key algorithm (for example, `PSHA1`). This is used when the key resulting from the token request is not directly returned, and is computed.

**Public Proof of Possession Key**:
If the **Proof of Possession Key Type** requested is a `PublicKey`, you can specify what public key to include in the token using the following settings:

- **Use Key Format**: Select how the `UseKey` element in the RST formats the public key from the list (for example, `PublicKey`, `Certificate`, `BinarySecurityToken`, and so on).
- **Use Key Selector Expression**: Select or enter the selector expression that contains the public key. The public key can be of type `X509Certificate`, `PublicKey`, or `KeyPair`.

## *Issue: On Behalf Of Token*

For **Issue** requests, you can optionally configure the `OnBehalfOf` token for the RST. If an `OnBehalfOf` token is in the RST, this means you are requesting a token on behalf of the subject identified by the token or endpoint reference in the `OnBehalfOf` element. You can configure the following settings on the **Issue: On Behalf Of Token** tab:

**On Behalf Of**:
Select one of the following options:

- **None**: No `OnBehalfOf` token is specified. This is the default.
- **Token**: The token is embedded directly under the `<OnBehalfOf>` element in the RST.
- **EmbeddedSTR**: The token is placed in the `<OnBehalfOf><SecurityTokenReference><Embedded>` element in the RST.
- **Endpoint Reference**: A reference to the token is placed in the `<OnBehalfOf><SecurityTokenReference>` element. The token is placed in the WS-Security header.

**On Behalf Of Token Selector Expression**:
Enter the selector expression for the message attribute that contains the `OnBehalfOf` token. This can be a `UsernameToken`, SAML token, X.509 certificate, and so on. The type of this message attribute can be `Node`, `List` of Nodes, `String`, or `X509Certificate`. This message attribute must be populated using a filter configured in the policy that applies security before creating the WS-Trust message. For example, this includes a filter to extract a `UsernameToken` from the incoming message, or a **Find Certificate** filter.

**Endpoint Address**:
When the **On Behalf Of** type is **Endpoint Reference**, no token is placed in the `OnBehalfOf` element. Instead, you can enter an endpoint address in this field that identifies the subject on whose behalf you are requesting the token.

**Identity Type**:
When the **On Behalf Of** type is **Endpoint Reference**, you can select an identity type from the list (for example, `DNSName`, `ServicePrincipaName`, or `UserPrincipalName`).

**Identity**:
When the **Identity Type** is set to `DNSName`, `ServicePrincipaName`, or `UserPrincipalName`, you must specify a value in this field.

**Identity Selector Expression**:
When the selected **Identity Type** is one of `PublicKey`, `Certificate`, `BinarySecurityToken`, `SecurityTokenReference_x509v3`, or `SecurityTokenReference_ThumbprintSHA1`, you must specify a selector expression in this field. This specifies the name of the message attribute that contains the certificate for the subject on whose behalf you are requesting the token. The type of this message attribute must be `X509Certificate`.

## Issue: Token Scope and Lifetime

For **Issue** requests, you can optionally specify details for the scope of the requested token (for example, the endpoint service this token is used for). These details are placed in the `AppliesTo` element of the RST. You can configure the following settings on the **Issue: Token Scope and Lifetime** tab:

**Endpoint Address**:
Enter an address for the endpoint.

**Identity Type**:
Select an identity type from the list (for example, `Certificate`, `BinarySecurityTokenDNSName`, `ServicePrincipalName`, or `UserPrincipalName`).

**Identity**:
When the **Identity Type** is set to `DNSName`, `ServicePrincipaName`, or `UserPrincipalName`, you must specify a value in this field.

**Identity Selector Expression**:
When the **Identity Type** selected is one of `PublicKey`, `Certificate`, `BinarySecurityToken`, `SecurityTokenReference_x509v3`, or `SecurityTokenReference_ThumbprintSHA1`, you must specify a selector expression in this field. This specifies the name of the message attribute that contains the certificate for the endpoint service that the token is sent to. The type of this message attribute must be `X509Certificate`.

**Expires In**:
Specify when the token is due to expire in the fields provided.

**Lifetime Format**:
Enter the date and time format in which the token lifetime is specified. Defaults to `yyyy-MM-dd'T'HH:mm:ss.SSS'Z'`.

**Note** The STS can choose to ignore the token lifetime specified in the RST.

## Validate:Target

If the request type is set to **Validate**, you can use the **Validate:Target** tab to specify the token that you require the STS to validate. In this case, the STS does not issue a token. It validates the token passed to it in the RST and returns a status. The STS response is placed in the `sts.validate.code` and `sts.validate.reason` message attributes.

You can configure the following settings on the **Validate:Target** tab:

**Token**:
Specifies that the token is placed directly under the `<ValidateTarget>` element in the RST.

**EmbeddedSTR**:
Specifies that the token is placed in the `<ValidateTarget><SecurityTokenReference><Embedded>` element.

**STR**:
Specifies that a reference to the token is placed in the `<ValidateTarget><SecurityTokenReference>` element. The token is placed in the WS-Security header.

**Validate Target Selector Expression**:
Enter a selector expression for the message attribute that contains the token to validate. The attribute type can be `Node`, a `List` of Nodes, or `String`. This message attribute must be populated using a filter configured in the policy that applies security before creating the WS-Trust message. For example, you can run a filter to extract a SAML token from the incoming message.

# Policies settings

The **Policies** tab enables you to specify the policies that the **Security Token Service Client** filter delegates to. You can configure the following settings on this tab by clicking the button next to each field:

**Policy apply security before creating the WS-Trust message**:
Specifies the policy that runs before the **Security Token Service Client** filter creates the RST (the WS-Trust request message for the STS). The filters in this policy are used to set up message attribute values that the STS client filter requires (for example, the `OnBehalfOf` token).

**Policy to apply security to the WS-Trust request**:
Specifies the policy that runs after the **Security Token Service Client** filter has created the RST. The filters in this policy can sign and/or encrypt the message as required by the STS. It can also inject other security tokens into the WS-Security header if required.

**Policy to apply security to the WS-Trust response**:
Specifies the policy that runs to apply security to the WS-Trust response. This policy runs when the response is received from the STS. The filters in this policy can decrypt and verify signatures on the response message.

# Routing settings

When routing to an STS, you can specify a direct connection to the web service endpoint by entering a URL on the **Routing** tab. Alternatively, when the routing behavior is more complex, you can delegate to a custom routing policy to handle the added complexity. The options on the **Routing** tab allow for these alternative routing configurations.

**Use the following URL**:
Select this option to route to the specified URL. You can enter the URL in the text box, or specify the URL as a selector so that the URL is built dynamically at runtime from the specified message attributes (for example `${host}:${port}`, or

`${http.destination.protocol}://${http.destination.host}:${http.destination.port}`). For more details on selectors, see "Select configuration values at runtime" in the *API Gateway Policy Developer Guide*.

You can configure SSL settings, credential profiles for authentication, and other settings for the direct connection using the tabs in the **Connection Details** group. For more details, see *Connect to URL* on page 380.

**Delegate routing to the following policy**:
Select this option to use a dedicated routing policy to send messages on to the STS. Click the browse button next to the **Routing policy** field to select the policy to use to route messages.

**No routing**:
Select this option to only allow request reflection for test purposes.

# Response settings

The **Response** tab enables you to specify options for processing the response message from the STS. You can configure the following settings on this tab:

**Verify returned security token type**:
When selected, the filter checks that the `TokenType` returned is what was requested. This is selected by default.

**Put security token into message attribute**:
When specified, the token returned from the STS is placed in the specified message attribute. The type of this attribute is `String`. Defaults to `sts.security.token`. An element version of the token is placed in a message attribute named `attrname.element`.

**Insert security token into original message in SOAP Actor/Role**:
When specified, the token returned from the STS is inserted into the original message. This is the original message received by the API Gateway (was the current message before the **Security Token Service Client** filter ran). Defaults to `Current actor/role only`.

**Extract Token Lifetime**:
When selected, the token lifetime is extracted from the response, and the `sts.token.lifetime.created` and `sts.token.lifetime.expires` message attributes are populated. This setting is selected by default.

# Advanced settings

The **Advanced** tab enables you to specify the following options:

**Versions and Namespaces**:
The version and namespace options are as follows:

- **WS-Trust Version**: Specifies the WS-Trust namespace to use in the generated RST. Defaults to `WS-Trust 1.3`.

- **SOAP version**: Specifies the SOAP version to use in the generated RST. Defaults to `SOAP 1.1`.
- **WS-Addressing Namespace**: Specifies the WS-Addressing namespace to use in the generated RST. Defaults to `http://www.w3.org/2005/08/addressing`.
- **WS-Policy Namespace**: Specifies the WS-Policy namespace to use in the generated RST. Defaults to `WS-Policy 1.2`.
- **WS-Security Actor**: Specifies the actor in which to place tokens that are referred to from the RST using STRs (for example, `OnBehalfOf`). Defaults to `Current actor/role only`.

**Algorithms**:
The algorithm options are as follows:

- **Canonicalization Algorithm**: When selected, additional elements are added to the RST, which specify a client-requested canonicalization algorithm (for example, `ExC14n`).
- **Encryption Algorithm**: When selected, additional elements are added to the RST, which specify a client-requested encryption algorithm (for example, `Aes256`).
- **Encrypt with**: When selected, specifies the encryption algorithm with which to encrypt the RSTR (for example, `Aes256`).
- **Sign with**: When selected, specifies the signature algorithm with which to digitally sign the RSTR (for example, `RsaSha256`).

**Advanced Settings**:
The advanced options are as follows:

**Content-Type**: Specifies the `Content-Type` of the message to be sent to the STS. For example, for Microsoft Windows Communication Foundation (WCF), select `application/soap+xml`. Defaults to `text/xml`.

**Store and restore original message**: When selected, the original message is saved before messages sent from the API Gateway to the STS and messages sent from the STS to the API Gateway are processed. It is then reinstated after this filter finishes processing the STS response. This is the default behavior.

For debug purposes, you might wish to return the STS response from your policy. In this case, deselect this setting, and the current message after this filter completes should then be the STS response. You might also wish to debug the RST (the request to the STS), and return that from your policy. In this case, disable this setting, click the **Routing** tab, and select the **No routing** option.

# WS-Security UsernameToken authentication

## Overview

A WS-Security *UsernameToken* enables an end user identity to be passed over multiple hops before reaching the destination web service. The user identity is inserted into the message and is available for processing at each hop on its path.

The client user name and password are encapsulated in a WS-Security `<wsse:UsernameToken>`. When the API Gateway receives this token, it can perform one of the following tasks, depending on the requirements:

- Ensure that the timestamp on the token is still valid
- Authenticate the user name against a repository
- Authenticate the user name and password against a repository

The following sample SOAP message contains two `<wsse:UsernameToken>` blocks:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Header>
    <wsse:Security xmlns:wsse="http://schemas.xmlsoap.org/ws/2003/06/secext">
        <wsse:UsernameToken wsu:Id="sample"
          xmlns:wsu="http://schemas.xmlsoap.org/ws/2003/06/utility">
            <wsse:Username>sample</wsse:Username>
            <wsse:Password Type="wsse:PasswordText">axway</wsse:Password>
            <wsu:Created>2004-05-19T08:44:51Z</wsu:Created>
        </wsse:UsernameToken>
    </wsse:Security>
    <wsse:Security soap:actor="axway"
      xmlns:wsse="http://schemas.xmlsoap.org/ws/2003/06/secext">
        <wsse:UsernameToken wsu:Id="axway"
          xmlns:wsu="http://schemas.xmlsoap.org/ws/2003/06/utility">
            <wsse:Username>axway</wsse:Username>
            <wsse:Password Type="wsse:PasswordText">axway</wsse:Password>
            <wsu:Created>2004-05-19T08:46:04Z</wsu:Created>
        </wsse:UsernameToken>
    </wsse:Security>
</soap:Header>
<soap:Body>
    <getHello xmlns="http://www.axway.com"/>
</soap:Body>
</soap:Envelope>
```

This topic explains how to use the **WS-Security UsernameToken** filter to authenticate users using a WS-Security `<wsse:UsernameToken>`.

# General settings

To configure general settings, complete the following fields.

## *Actor Details*

**Actor**:
The example SOAP message at the top of this page contains two `<wsse:UsernameToken>` blocks. You must specify which block contains the `<wsse:UsernameToken>` used to authenticate the end user. Specify the SOAP Actor/Role of the WS-Security block that contains the token.

**Credential Format**:
The API Gateway can authenticate users against a user profile repository based on User Names, X.509 Distinguished Names, or email addresses. Unfortunately, the WS-Security specification does not provide a means of specifying the type of `<wsse:UsernameToken>`, and so it is necessary for the administrator to do so using the **Credential Format** field. The type specified here is used internally by the API Gateway in subsequent authorization filters.

## *Token Validation*

Each `wsse:UsernameToken` contains a timestamp inserted into the `<wsu:Created>` element. Using this timestamp together with the details entered in this section, the API Gateway can determine whether the WS-Security `UsernameToken` has expired. The `<wsu:Created>` element is as follows:

```
<wsse:UsernameToken wsu:Id="axway"
  xmlns:wsu="http://schemas.xmlsoap.org/ws/2003/06/utility">
    <wsu:Created>2006.01.13T-10:42:43Z</wsu:Created>
        ...
</wsse:UsernameToken>
```

To configure token validation settings, complete the following fields:

**Drift Time**:
Specified in seconds to account for differences in the clock times between the machine on which the token was generated and the machine running the API Gateway. Using the *start time*, *end time*, and *drift time*, the token is considered valid if the current time falls between the following times:

```
[start - drift] and [start + drift + end]
```

**Validity Period**:
Specifies the lifetime of the token, where the value of the `<wsu:Created>` element represents the *start time* of the assertion, and the time period entered represents the *end time*.

**Timestamp Required**:
Select this option to ensure that the UsernameToken contains a timestamp. If no timestamp is found in the UsernameToken, a SOAP Fault is returned.

## Nonce Settings

**Nonce Required**:
Select this option to ensure that the UsernameToken contains a `<wsse:Nonce>` element. This is a randomly generated number that is added to the message. You can use the combination of a timestamp and a nonce to help prevent replay attacks.

**Select cache to store WSS UsernameToken nonces in**:
Click the button on the right, and select the cache that stores the nonce value. Defaults to the local `WSS UsernameToken Nonce Cache`.

To add a cache, right-click the **Caches** tree node, and select **Add Local Cache** or **Add Distributed Cache**. Alternatively, you can configure caches under the **Environment Configuration** > **Libraries** node in the Policy Studio tree. For more details on caches, see the *API Gateway Policy Developer Guide*.

## Token Validation via Repository

Having validated the timestamp on the token, the API Gateway can then optionally authenticate the user name and password contained in the token. The following options are available:

- **No Verification**
  No verification of the user name and password is performed. Only the timestamp on the token is validated. This is the default behavior.
- **Verify Username Only**
  Only the user name is looked up in the selected repository. If the user name is found in this repository, the user is authenticated. Select the **No password allowed** check box to block messages that contain a UsernameToken with a `<wsse:Password>` element.
- **Verify Username and Password**
  The user name is looked up in the selected repository and is only authenticated if the corresponding password matches the one configured in the repository. If you select this option, you must select the type of the password. Both clear text and digest formats are supported. Select the appropriate option.

**Repository Name**:
The API Gateway attempts to authenticate users against the selected **Authentication Repository**. User profiles can be stored in the local store, a database, or an LDAP directory. For more details on authentication repositories, see the *API Gateway Policy Developer Guide*.

## *Advanced*

**Remove enclosing WS-Security element on successful validation**:
Select this option to remove the WS-Security block that contains the UsernameToken after the token has been successfully authenticated. For example, in the above sample SOAP message that contains two `<wsse:UsernameToken>` elements in two different WS-Security blocks, you could configure the API Gateway to remove one of these on successful authentication.

# Authorization filters 4

The following filters enable you to perform authorization with API Gateway:



# RSA Access Manager authorization

RSA Access Manager (formerly RSA ClearTrust) provides identity management and access control services for web applications. It centrally manages access to web applications, ensuring that only authorized users are allowed access to resources.

The **Access Manager** filter enables integration with RSA Access Manager. This filter can query Access Manager for authorization information for a particular user on a given resource. In other words, API Gateway asks Access Manager to make the authorization decision. If the user has been given authorization rights to the web service, the request is allowed through to the service. Otherwise, the request is rejected.

## Prerequisites

You must copy RSA Access Manager libraries to API Gateway, so you must have RSA Access Manager installed on a server.

1. Copy the following files from the `lib` directory on your RSA Access Manager installation:
    - `axm-core-6.2.jar`
    - `cryptojce-6.1.jar`
    - `cryptojcommon-6.1.jar`
    - `jcm-6.1.jar`

2. Add the files to the `INSTALL_DIR/apigateway/ext/lib` directory on API Gateway:
3. Restart API Gateway.

# General settings

Configure the following general settings.

## *Connection Details*

The **Connection Details** section enables you to specify a group of Access Manager servers to connect to in order to authenticate clients. You can select a group of Access Manager servers to provide failover in cases where one or more servers are not available.

**Connection Group Type**:
API Gateway can connect to a group of Access Manager authorization servers or dispatcher servers. When multiple Access Manager authorization servers are deployed for load-balancing purposes, API Gateway should first connect to a dispatcher server, which returns a list of active authorization servers. An attempt is then made to connect to one of these authorization servers using round-robin DNS. If the first dispatcher server in the connection group is not available, API Gateway attempts to connect to the dispatcher server with the next highest priority in the group, and so on.

If a dispatcher server has not been deployed, API Gateway can connect directly to an authorization server. If the authorization server with the highest priority in the connection group is not available, API Gateway attempts to connect to the authorization server with the next highest priority, and so on. Select the type of the connection group (**Authorization Server** or **Dispatcher Server**). All servers in the group must be of the same type.

**Connection Group**:
Click the button on the right, and select the connection group to use for authenticating clients. To add a connection group, right-click the **RSA Access Manager Connection Sets** tree node, and select **Add a Connection Set**. Alternatively, you can configure a connection set under the **Environment Configuration > External Connections** node in the Policy Studio tree. For more details, see "Configure connection groups" in the *API Gateway Policy Developer Guide*.

## *Authorization Details*

The **Authorization Details** section describes the resource for which the user is requesting access.

- **Server**:
  Enter the name of the server that is hosting the requested resource. The name entered must correspond to a preconfigured server name in Access Manager.
- **Resource**:
  Enter the name of the requested resource. This resource must be preconfigured in Access Manager.

Alternatively, you can enter a selector representing a message attribute in the **Resource** field. API Gateway expands this selector at runtime to the value of the corresponding message attribute. API Gateway message attribute selectors take the following format:

```
${message.attribute}
```

The following example of a typical SOAP message received by API Gateway shows how this works:

```
POST /services/timeservice HTTP/1.0
Host:localhost:8095
Content-Length:374
SOAPAction:TimeService
Accept-Language:en-US
Content-Type:text/XML; utf-8
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
    <soap:Body>
        <ns1:getTime xmlns:ns1="urn:timeservice"></ns1:getTime>
    </soap:Body>
</soap:Envelope>
```

The following table shows an example of selector expansion:

| Selector | Expanded To |
|---|---|
| `${http.request.uri}` | `/services/timeservice` |

For more details on selectors, see "Select configuration values at runtime" in the *API Gateway Policy Developer Guide*.

# Attribute authorization

## Overview

The purpose of the filters in the **Attributes** filter group is to extract user attributes from various sources. You can use these filters to retrieve attributes from the message, an LDAP directory, a database, the API Gateway user store, HTTP headers, a SAML attribute assertion, and so on.

After retrieving a set of user attributes, API Gateway stores them in the `attribute.lookup.list` message attribute, which is essentially a map of name-value pairs. It is the role of the **Attributes** authorization filter to check the value of these attributes to authorize the user.

# Configuration

Configure the following fields on the **Attributes** configuration window:

**Name**:
Enter a suitable name for this filter to display in a policy.

**Attributes**:
The **Attributes** table lists the checks that the API Gateway performs on user attributes stored in the `attribute.lookup.list` message attribute. The API Gateway performs the following checks:

- The entries in the table are OR-ed together so that if any one of them succeeds, the filter returns a pass.
- The attribute checks listed in the table are run in series until one of them passes.
- You can add a number of attribute-value pairs to a single attribute check by separating them with commas (for example, `company=axway, department=engineering, role=engineer`).
- If multiple attribute-value pairs are present in a given attribute check, these pairs are AND-ed together so that the overall attribute check only passes if all the attribute-value pairs pass. For example, if the attribute check comprises, `department=engineering, role=engineer`, this check only passes if both attributes are found with the correct values in the `attribute.lookup.list` message attribute.

To add an attribute check to the **Attributes** table, click **Add**, and enter attributes in the dialog. For attribute checks involving attributes extracted from a SAML attribute assertion, you must specify the namespace of the attribute given in the assertion. For example, API Gateway can extract the `role` attribute from the following SAML `<Attribute Statement>`, and store it in the `attribute.lookup.list` map:

```
<saml:AttributeStatement>
  <saml:Attribute Name="role" NameFormat="http://www.company.com">
    <saml:AttributeValue>admin</saml:AttributeValue>
  </saml:Attribute>
  <saml:Attribute Name="email" NameFormat="http://www.company.com">
    <saml:AttributeValue>joe@company.com</saml:AttributeValue>
  </saml:Attribute>
  <saml:Attribute Name="dept" NameFormat="">
    <saml:AttributeValue>engineering</saml:AttributeValue>
  </saml:Attribute>
</saml:AttributeStatement>
```

The `NameFormat` attribute of the `<Attribute>` gives the namespace of the attribute name. You must enter this namespace (together with a corresponding prefix) in the **Add Attributes** dialog. For example, to extract the `role` attribute from the SAML attribute statement above, enter

`pre:role=admin` in the **Attribute Requirement** field. Then you must also map the `pre` prefix to the `http://www.company.com` namespace, as specified by the `NameFormat` attribute in the attribute statement.

# Axway PassPort authorization

## Overview

Axway PassPort provides a central repository, identity broker, and security audit point for Business-to-Business Integration (B2Bi) or Managed File Transfer (MFT) solutions. Axway PassPort centralizes and simplifies provisioning and management for your entire online ecosystem, enabling secure collaboration between applications, divisions, customers, suppliers, and regulatory bodies.

This topic explains how to configure the settings in the **Axway PassPort Authorization** filter, which are used to configure integration between Axway PassPort and API Gateway.

### *Axway CSD*

An Axway Component Security Descriptor (CSD) file is an XML file that defines resources, privileges, and roles for each component. For more details, see the *Axway PassPort* user documentation. The **Axway PassPort Authorization** filter checks if the specified user has the privileges to perform the action on the specified resource. The CSD file defines the available actions that a resource supports. It might also define privileges and roles, which can also be created in the PassPort administration user interface.

## Configuration

Configure the following settings:

**Name**:
Enter an appropriate name for this filter to display in a policy.

**User ID**:
Enter the ID of the user to authorize. You can enter a static name or a selector that specifies a message attribute. The selector is expanded at runtime to the value of the message attribute. For more details on specifying settings as selectors, see "Select configuration values at runtime" in the *API Gateway Policy Developer Guide*. Defaults to `${authentication.subject.id}`.

**Resource**:
Enter the name of the resource for which the user is seeking authorization. This resource must have been defined in the `<ResourceDefinition>` in the PassPort repository CSD. You can enter a static name or a selector that specifies a message attribute. The selector is expanded at runtime to the value of the message attribute. Defaults to `${http.request.uri}`.

**Action**:
Enter the action being performed on the resource for which authorization is sought. This action must have been defined in the `<AvailableActions>` section of the PassPort repository CSD. You can enter a static name or a selector that specifies a message attribute. The selector is expanded at runtime to the value of the message attribute. Defaults to `${http.request.verb}`.

**PassPort Repository**:
Select an existing connection to an Axway PassPort repository to use for authorization. To configure a connection in the Policy Studio tree, right-click **Environment Configuration** > **External Connections** > **Authentication Repository Profiles** > **Axway PassPort Repository**, and select **Add a new Repository**. For more details, see the *API Gateway Policy Developer Guide*.

# CA SOA Security Manager authorization

## Overview

CA SOA Security Manager can authenticate end users and authorize them to access protected web resources. The API Gateway can interact directly with CA SOA Security Manager by asking it to make authorization decisions on behalf of end users that have successfully authenticated to the API Gateway. CA SOA Security Manager decides whether to authorize the user, and relays the decision back to the API Gateway where the decision is enforced. The API Gateway acts as a Policy Enforcement Point (PEP) in this situation, enforcing the authorization decisions made by the CA SOA Security Manager, which acts a Policy Decision Point (PDP).

**Note** A **CA SOA Security Manager** authentication filter must be invoked before a **CA SOA Security Manager** authorization filter in a given policy. In other words, the end user must authenticate to CA SOA Security Manager before they can be authorized for a protected resource.

## Prerequisites

Integration with CA SOA Security Manager requires CA TransactionMinder SDK version 6.0 or later. You must add the required third-party binaries to your API Gateway and Policy Studio installations.

**Add third-party binaries to API Gateway**

To add third-party binaries to API Gateway, perform the following steps:

1. Add the binary files as follows:
    - Add `.jar` files to the `INSTALL_DIR/apigateway/ext/lib` directory.
    - Add `.so` files to the `INSTALL_DIR/apigateway/<platform>/lib` directory.
2. Restart API Gateway.

**Add third-party binaries to Policy Studio**

To add third-party binaries to Policy Studio, perform the following steps:

1. Select **Window > Preferences > Runtime Dependencies** in the Policy Studio main menu.
2. Click **Add** to select a JAR file to add to the list of dependencies.
3. Click **Apply** when finished. A copy of the JAR file is added to the `plugins` directory in your Policy Studio installation.
4. Click **OK**.
5. Restart Policy Studio with the `-clean` option. For example:

```
> cd INSTALL_DIR/policystudio/
> policystudio -clean
```

## Configuration

Configure the following fields on the **CA SOA Security Manager** authorization filter:

**Name**:
Enter an appropriate name for the filter to display in a policy.

**Attributes**:
If the end user is successfully authorized, the attributes listed here are looked up in CA SOA Security Manager, and returned to the API Gateway. These attributes are stored in the `attributes.lookup.list` message attribute. They can be retrieved at a later stage to generate a SAML attribute assertion.

Select the **Set attributes for SAML Attribute token** check box, and click the **Add** button to specify an attribute to fetch from CA SOA Security Manager.

# Certificate attribute authorization

## Overview

The API Gateway can authorize access to a web service based on the X.509 attributes of an authenticated client's certificate. For example, a simple **Certificate Attributes** filter might only authorize clients whose certificates have a Distinguished Name (DName) containing the following attribute: `O=axway`. In other words, only `axway` users are authorized to access the web service.

An X.509 certificate consists of a number of fields. The `Subject` field is most relevant. It gives the DName of the client to which the certificate belongs. A DName is a unique name given to an X.500 directory object. It consists of a number of attribute-value pairs called Relative Distinguished Names (RDNs). Some of the most common RDNs and their explanations are as follows:

- `CN`: CommonName
- `OU`: OrganizationalUnit

- O: Organization
- L: Locality
- S: StateOrProvinceName
- C: CountryName

For example, the following is the DName of the *sample.p12* client certificate supplied with API Gateway:

```
CN=Sample Cert, OU=R&D, O=Company Ltd., L=Dublin 4, S=Dublin, C=IE
```

Using the **Certificate Attributes** filter, it is possible to authorize clients based on attributes (for example, the CN, OU, or C in the DName).

# Configuration

Configure the following settings:

**Name**:
Enter an appropriate name for the filter to display in a policy.

**X.509 Attributes**:
To add a new X.509 attribute check, click the **Add** button. In the **Add X.509 Attributes** dialog, enter a comma-separated list of name-value pairs representing the X.509 attributes and their values (for example, OU=dev,O=Company).

The new attribute check is displayed in the **X.509 Attributes** table. You can edit and delete existing entries by clicking the **Edit** and **Remove** buttons.

The **X.509 Attributes** table lists a number of attribute checks to be run against the client certificate. Each entry tests a number of certificate attributes in such a way that the check only passes if all of the configured attribute values match those in the client certificate. In effect, the attributes listed in a single attribute check are AND-ed together.

For example, imagine the following is configured as an entry in the **X.509 Attributes** table:

```
OU=Eng, O=Company Ltd
```

If the API Gateway receives a certificate with the following DName, this attribute check passes because *all* the configured attributes match those in the certificate DName:

```
CN=User1, OU=Eng, O=Company Ltd, L=D4, S=Dublin, C=IECN=User2, OU=Eng, O=Company
Ltd, L=D2, S=Dublin, C=IE
```

However, if the API Gateway receives a certificate with the following DName, the attribute check fails because the attributes in the DName do not match *all* the configured attributes (the OU attribute has the wrong value):

```
CN=User1, OU=qa, O=Company Ltd, L=D4, S=Dublin, C=IE
```

The **X.509 Attributes** table can contain several attribute check entries. In such cases, the attribute checks (the entries in the table) are OR-ed together, so that if any of the checks succeed, the overall **Certificate Attributes** filter succeeds.

To summarize:

- Attribute values within an attribute check only succeed if *all* the configured attribute values match those in the DName of the client certificate.
- The filter succeeds if *any* of the attribute checks listed in the **X.509 Attributes** table succeed.

# Entrust GetAccess authorization

## Overview

Entrust GetAccess provides identity management and access control services for web resources. It centrally manages access to web applications, enabling users to benefit from a single sign-on capability when accessing the applications that they are authorized to use.

The **GetAccess** filter enables integration with Entrust GetAccess. This filter can query GetAccess for authorization information for a particular user for a given resource. In other words, the API Gateway asks GetAccess to make the authorization decision. If the user has been given authorization rights to the web service, the request is allowed through to the service. Otherwise, the request is rejected.

## GetAccess WS-Trust STS settings

This tab enables you to configure how the API Gateway authenticates to the GetAccess WS-Trust Security Token Service (STS). You can configure the API Gateway to connect to a group of GetAccess STS servers in a round-robin fashion. This provides the necessary failover capability when one or more STS servers are not available.

Configure the following fields:

- **URL Group**:
  Click the button on the right, and select an STS URL group in the tree. This group consists of a number of GetAccess STS servers to which the API Gateway round-robins connection attempts. To add a URL group, right-click the **Entrust GetAccess URL Sets** node, and select **Add a URL Set**. Alternatively, you can configure a URL connection set under the **Environment Configuration** > **External Connections** node in the Policy Studio tree. For more details, see "Configure URL groups" in the *API Gateway Policy Developer Guide*.

- **Drift Time**:
  Having successfully authenticated to a GetAccess STS server, the STS server issues a SAML authentication assertion and returns it to the API Gateway. When checking the validity period of the assertion, the specified **Drift Time** is used to account for a possible difference between the time on the STS server and the time on the machine hosting the API Gateway.
- **WS-Trust STS Attribute Field Name**:
  Specify the field name for the `Id` field in the WS-Trust request. The default is `Id`.

# GetAccess SAML PDP settings

When the API Gateway has successfully authenticated to a GetAccess STS server, it can then obtain authorization information about the end user from the GetAccess SAML PDP. The authorization details are returned in a SAML authorization assertion, which is then validated by the API Gateway to determine whether the request should be denied.

Configure the following fields:

- **URL Group**:
  Click the button on the right, and select an SAML PDP URL group in the tree. This group consists of a number of GetAccess SAML PDP servers to which the API Gateway round-robins connection attempts. To add a URL group, right-click the **Entrust GetAccess URL Sets** node, and select **Add a URL Set**. Alternatively, you can configure a URL connection set under the **Environment Configuration** > **External Connections** node in the Policy Studio tree. For more details, see "Configure URL groups" in the *API Gateway Policy Developer Guide*.
- **Drift Time**:
  The specified **Drift Time** is used to account for the possible difference between the time on the GetAccess SAML PDP and the time on the machine hosting the API Gateway. This comes into effect when validating the SAML authorization assertion.
- **Resource**:
  This is the resource for which the client is requesting access. You can enter a selector representing a message attribute, which is looked up and expanded to a value at runtime. For example, to specify the original path on which the request was received by the API Gateway as the resource, enter the selector `${http.request.uri}`. For more details on selectors, see "Select configuration values at runtime" in the *API Gateway Policy Developer Guide*.
- **Actor/Role**:
  To add the SAML authorization assertion to the downstream message, select a SOAP actor/role to indicate the WS-Security block where the assertion is added. Leave this field blank if the assertion is not to be added to the message.

# Insert SAML authorization assertion

## Overview

After successfully authorizing a client, the API Gateway 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 is eventually 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="axway-1056130475340"
      Id="axway-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.axway.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>
```

# 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 (that generate the assertion) and the machines that consume the assertion. The specified time is subtracted from the time at which the API Gateway 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.

**Note** 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, Axway recommend that SAML 1.1 or 2.0 is used when signing assertions as they both use 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. 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.

**Resource**:
Enter the resource for which to obtain the authorization assertion. You should specify the resource as a URI (for example, `http://www.axway.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 "Configure XPath expressions" in the *API Gateway Policy Developer Guide*.

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 either the end-user that authenticated to the API Gateway, 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=axway</dsig:X509SubjectName>
      <dsig:X509Certificate>
        MIICmzCCAY ...... mB9CJEw4Q=
      </dsig:X509Certificate>
    </dsig:X509Data>
  </dsig:KeyInfo>
</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 includes the key used to prove that the API Gateway 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 SAMLArtifact to the API Gateway. | `urn:oasis:names:tc:SAML:1.0:cm:artifact` |

| Method | Meaning | Value |
|---|---|---|
| Sender Vouches | Use this confirmation method to assert that the API Gateway 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 (see *XML signature generation* on page 322). | `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 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** window, select the check box next to the certificate that is associated with the key 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, or environment variable). For more details, see "Select configuration values at runtime" in the *API Gateway Policy Developer Guide*.

Symmetric Key:
To use a symmetric key as proof that the API Gateway 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 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, an **XML Signature Generation** filter) has already used a symmetric key, you can reuse this key as proof that the API Gateway is the holder-of-key entity. Enter the name of the selector expression (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, or environment variable). For more details, see "Select configuration values at runtime" in the *API Gateway Policy Developer Guide*.
- **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 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 Selector Expression**:
  Alternatively, if the recipient's certificate has already been used (perhaps to encrypt part of the message), the certificate 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, or environment variable). For more details, see "Select configuration values at runtime" in the *API Gateway Policy Developer Guide*.
- **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 check 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 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

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 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 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="axway">
      <wsse:SecurityTokenReference>
```

```
      <wsse:Embedded>
      <saml:Assertion
        xmlns:saml="urn:oasis:names:tc:SAML:1.0:assertion"
        AssertionID="axway-1056130475340"
        Id="axway-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.axway.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**.

# LDAP attribute authorization

## Overview

The **LDAP RBAC** filter combines Lightweight Directory Access Protocol (LDAP) with Role-Based Access Control (RBAC). This filter enables you to authorize a backend service based on user roles stored using LDAP. You can use the **LDAP RBAC** filter to read an attribute from LDAP, and compare it against some known values (for example, if `role` contains `engineering`, authorize the user). This filter combines functionality available in the **Retrieve from Directory Server** and **Compare Attribute** filters.

The **LDAP RBAC** filter enables you to define LDAP connection and search settings, and to configure how specified message attributes are processed. This filter also enables you to configure optional settings such as results caching and actions to take if a returned attribute is multivalued.

# General settings

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

**Connection**:
Click the button on the right to select your preconfigured LDAP directory server (for example, `openldap.qa.axway.com`). For details on how to configure LDAP servers, see "Configure LDAP directories" in the *API Gateway Policy Developer Guide*.

**Search Base**:
Enter the Distinguished Name (DN) to use as the base from which the search starts (for example, `o=Axway,l=Dublin 4,st=Dublin,C=IE`).

**Filter**:
Enter the search filter to use. For example:

```
(&(objectclass=inetOrgPerson)(cn=${authentication.subject.id}))
```

**Scope**:
Select one of the following search scopes from the list.

- **Object**: Searches on the base DN only (compare)
- **One Level**: Searches the direct children of the base DN
- **Subtree**: Searches the base DN and all its descendants

Defaults to **Subtree**.

**Attribute validation rules**:
When the search completes, the attributes returned in the results are processed by the rules in the **Attribute validation rules** table. This processing is the same as the **Compare Attribute** filter. You can logically `AND` and `OR` rules together in the **Filter will pass if** list by selecting **all** or **one**.

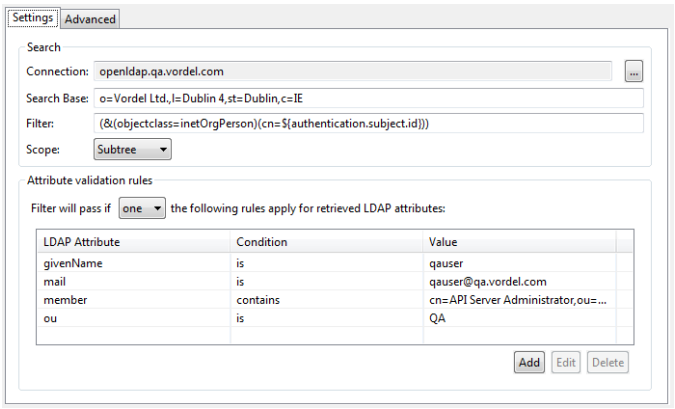
For example, if **Filter will pass if** is set to **all**, and Rule A, Rule B, and Rule C all evaluate to true, the filter passes. However, if Rule A evaluates to false, the filter fails. If the **Filter will pass** is set to **one**, and Rule A and Rule B evaluate to false, but Rule C evaluates to true, the filter passes. However, if Rule C evaluates to false, the filter fails.

Select **all** or **one** of the specified conditions to apply. Click the **Add** button at the bottom right to specify a rule condition. In the **Attribute filter rule** dialog, perform the following steps:

1. Enter a message attribute name in the **LDAP attribute named** field (for example, `member` or `mail`).

2. Select one of the following rule conditions from the list:
    - `contains`
    - `doesn't contain`
    - `doesn't match regular expression`
    - `ends with`
    - `is`
    - `is not`
    - `matches regular expression`
    - `starts with`
3. Enter a value to compare with in the text box on the right (for example, `POST`). Alternatively, you can enter a selector that is expanded at runtime (for example, `${http.request.uri}`). For more details on selectors, see "Select configuration values at runtime" in the *API Gateway Policy Developer Guide*.
4. Click **OK**.

The following figure shows some example search settings and attribute validation rules:




**Tip** When using this filter to determine if a user is a member of a `groupOfNames`, all the member attributes are concatenated together. The string containing the member attributes can be compared using a regular expression value provided in **Attribute validation rules**.

Because each attribute is not checked individually, you must create the regular expression string appropriately. For example, an expression such as `(?i:^.*${cert.subject.id}.*$)` allows for extra characters before and after the string searched for.

# Advanced settings

You can configure the following optional settings on the **Advanced** tab:

**Cache settings**:
Select whether to cache the LDAP search results. This setting is selected by default.

**Store results in the cache**:
Click the button on the right to select the preconfigured cache in which to store results. For more details on caches, see the *API Gateway Policy Developer Guide*. Select one of the following settings:

- **Use the LDAP search filter as cache key**: Uses the LDAP search filter configured on the **Settings** tab as the cache key.
- **Or use the following value as the cache key**: Enter a specific value for the cache key.

**If returned attribute contains multiple values**:
Select one of the following settings:

- **Concatenate values with the following**: Enter the character used to concatenate multiple attribute values. A comma is used by default.
- **Use value at index**: Enter the index number of the attribute value to use. Defaults to `0`.

# SAML authorization

## Overview

A Security Assertion Markup Language (SAML) authorization assertion contains proof that a certain user has been authorized to access a specified resource. Typically, such assertions are issued by a SAML Policy Decision Point (PDP) when a client requests access to a specified resource. The client must present identity information to the PDP, which ensures that the client does have permission to access the resource. The PDP then issues a SAML authorization assertion stating whether the client is allowed access the resource.

When the API Gateway receives a request containing such an assertion, it performs the following validation on the assertion details:

- Ensures the resource in the assertion matches that configured in the **SAML Authorization** filter
- Checks the client's access permissions for the resource
- Ensures the assertion has not expired

If the information validates, the API Gateway authorizes the message for the resource specified in the assertion.

The following example of a SAML authorization assertion might be useful when configuring the **SAML Authorization** filter.

```
<saml:Assertion xmlns:saml="urn:oasis:names:tc:SAML:1.0:assertion"
  xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
```

```
  MajorVersion="1" MinorVersion="0"
  AssertionID="192.168.0.131.1010924615489"
  Issuer="AA" IssueInstant="2002-03-26 16:23:35">
  <saml:Conditions NotBefore="2002-04-18T09:19:00Z"
    NotOnOrAfter="2003-06-28T09:21:00Z"/>
  <saml:AuthorizationDecisionStatement
    Resource="http://www.abc.org/services/getPrice"
    Decision="Permit">
      <saml:Action>Read</saml:Action>
  </saml:AuthorizationDecisionStatement>
</saml:Assertion>
```

# Details settings

The following fields are available on the **Details** tab:

**SOAP Actor/Role**:
There might be several authorization assertions contained in a message. You can identify the assertion to validate by entering the name of the SOAP actor/role of the WS-Security header that contains the assertion.

**XPath Expression**:
Alternatively, you can enter an XPath expression to locate the authorization assertion. You can configure XPath expressions using the **Add**, **Edit**, and **Delete** buttons.

**SAML Namespace**:
Select the SAML namespace that must be used on the SAML assertion for this filter to succeed. To not check the namespace, select the `Do not check version` option from the list.

**SAML Version**:
Enter the SAML version that the assertion must adhere to by entering the major version in the first field, followed by the minor version in the second field. For example, for SAML version 2.0, enter `2` in the first field and `0` in the second field.

**Drift Time**:
The *drift time*, specified in seconds, is used when checking the validity dates on the authorization assertion. The drift time allows for differences between the clock times of the machine on which the assertion was generated and the machine hosting the API Gateway.

**Remove enclosing WS-Security element on successful validation**:
Select this check box to remove the WS-Security block that contains the SAML assertion after the assertion has been successfully validated.

# Trusted issuer settings

You can use the table on the **Trusted Issuers** tab to select the issuers that you consider trusted. In other words, this filter only accepts assertions that have been issued by the selected SAML authorities.

Click the **Add** button to display the **Trusted Issuers** dialog. Select the Distinguished Name of a SAML authority whose certificate has been added to the certificate store, and click **OK**. Repeat this step to add more SAML authorities to the list of trusted issuers.

## Optional settings

The optional settings enable further examination of the contents of the authorization assertion. The assertion can be checked to ensure that the authorized subject matches a specified value, and that the resource specified in the assertion matches the one entered here.

The API Gateway can verify that the subject in the SAML assertion (the `<NameIdentifier>` ) matches one of the following options:

- **The subject of the authentication filter**
- **The following value**: (for example, `user@axway.com`)
- **Neither of the above**

The API Gateway examines the `<Resource>` tag inside the SAML authorization assertion. By default, it compares the `<Resource>` to the `destination.uri` attribute that is set in the policy. If they are identical, this filter passes. Otherwise, it fails.

You can enter a value for the resource in the **Resource** field. The API Gateway then compares the `<Resource>` in the assertion to this value instead of the `destination.uri` attribute. The filter passes if the `<Resource>` value matches the value entered in the **Resource** field.

# SAML PDP authorization

## Overview

API Gateway can request an authorization decision from a Security Assertion Markup Language (SAML) Policy Decision Point (PDP) for an authenticated client using the SAML Protocol (SAMLP). In such cases, the API Gateway presents evidence to the PDP in the form of some user credentials, such as the Distinguished Name of a client's X.509 certificate.

The PDP decides whether the user is authorized to access the requested resource. It then creates an authorization assertion, signs it, and returns it to the API Gateway in a SAML Protocol response. The API Gateway can then perform a number of checks on the response, such as validating the PDP signature and certificate, and examining the assertion. It can also insert the SAML authorization assertion into the message for consumption by a downstream web service.

# Request settings

The **Request** tab describes how the API Gateway packages the SAMLP request before sending to the SAML PDP.

You can configure the following fields on the **Request** tab:

**SAML PDP URL Set**:
You can configure a group of SAML PDPs to which the API Gateway connects in a round-robin fashion if one or more of the PDPs are unavailable. This is known as a SAML PDP URL set. Click the button on the right, and select a previously configured SAML PDP URL set in the tree. To add a URL set, right-click the **SAML PDP URL Sets** tree node, and select **Add a URL Set**. Alternatively, you can configure a SAML PDP URL set under the **Environment Configuration** > **External Connections** node in the Policy Studio tree.

**SOAP Action**:
Enter the SOAP action required to send SAMLP requests to the PDP. Click the **Use Default** button to use the following default SOAP action as specified by SAMLP:

```
http://www.oasis-open.org/committees/security
```

**SAML Version**:
Select the SAML version to use in the SAMLP request.

**Signing Key**:
If the SAMLP request is to be signed, click the **Signing Key** button, and select the appropriate signing key from the certificate store.

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

**Evidence**:
SAMLP stipulates that proof of identity in the form of a SAML authentication assertion must be presented to the SAML PDP as part of the SAMLP request. API Gateway can use an existing SAML authentication assertion that is already present in the message, or generate one based on the user that authenticated to it.

Select the **Use SAML Assertion in message** option to include an existing assertion in the SAMLP request. Specify the actor/role of the WS-Security block where the assertion is found in the **SOAP Actor/Role** field.

Alternatively, select the **Create SAML Assertion from authenticated client** radio button to generate a new authentication assertion for inclusion in the SAMLP request. To sign the newly generated assertion with a private key from the certificate store, click **Signing Key**. Alternatively, you can enter a selector expression for the signing certificate (for example, `${certificate}`).

The specified **Drift Time** is subtracted from the time that API Gateway generates the authentication assertion. This accounts for any possible difference in the times of the machines hosting the SAML PDP and the API Gateway.

## SAML subject settings

You can describe the *subject* of the SAML assertion on the **SAML Subject** tab. Complete the following fields:

**Subject Selector Expression**:
Enter a selector expression for the message attribute that contains the user name of an authenticated user. The default value is `${authentication.subject.id}`.

**Subject Format**:
Select the format of the subject selected in the **Subject Selector Expression** field above.

**Note** There is no need to select a format here if the **Subject Attribute** field is set to `authentication.subject.id`.

## Subject confirmation settings

The settings on the **Subject Confirmation** 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, 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=axway</dsig:X509SubjectName>
      <dsig:X509Certificate>
        MIICmzCCAY ...... mB9CJEw4Q=
      </dsig:X509Certificate>
    </dsig:X509Data>
  </dsig:KeyInfo>
</saml:SubjectConfirmation>
```

You must configure the following fields on the **Subject Confirmation** 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 | A `<SubjectConfirmation>` is inserted into the SAMLP request. The `<SubjectConfirmation>` contains a `<dsig:KeyInfo>` section with the certificate of the user selected to sign the SAMLP request. The user selected to sign the SAMLP request must be the authenticated subject ( `authentication.subject.id`). Select the **Include Certificate** option if the signer's certificate is to be included in the `SubjectConfirmation` block. Alternatively, select the **Include Key Name** option if only the key name is to be included. | `urn:oasis:names:tc:SAML:1.0:cm:holder-of-key` |
| Bearer | A `<SubjectConfirmation>` is inserted into the SAMLP request. | `urn:oasis:names:tc:SAML:1.0:cm:bearer` |
| SAML Artifact | A `<SubjectConfirmation>` is inserted into the SAMLP request. | `urn:oasis:names:tc:SAML:1.0:cm:artifact` |
| Sender Vouches | A `<SubjectConfirmation>` is inserted into the SAMLP request. The SAMLP request must be signed by a user. | `urn:oasis:names:tc:SAML:1.0:cm:bearer` |

If the **Method** field is left blank, no `<ConfirmationMethod>` block is inserted into the assertion.

**Include Certificate**:
Select this option to include the SAML subject's certificate in the `<KeyInfo>` section of the `<SubjectConfirmation>` block.

**Include Key Name**:
Alternatively, if you do not want to include the certificate, you can select this option to only include the key name in the `<KeyInfo>` section.

## Response settings

The **Response** tab enables you to configure the API Gateway to perform a number of checks on the SAMLP response from the PDP by examining the contents of various key elements in the authorization assertion.

**SOAP Actor/Role**:
If the SAMLP response from the PDP contains a SAML authorization assertion, the API Gateway can extract it from the response and insert it into the downstream message. The SAML assertion is inserted into the WS-Security block identified by the specified SOAP actor/role.

**Drift Time**:
The SAMLP request to the PDP is time stamped by the API Gateway. To account for differences in the times on the machines running the API Gateway and the SAML PDP the specified time is subtracted from the time at which the API Gateway generates the SAMLP request.

**Subject in the assertion (the NameIdentifier) must match**:
The authorization assertion can be checked to ensure that the authorized subject matches a specified value, and that the resource specified in the assertion matches the one entered here. API Gateway can verify that the subject in the SAML assertion (the `NameIdentifier` ) matches one of the following options:

- **The subject of the authentication filter**
- **The following value** (for example, `CN=sample, O=Company, C=ie`)
- **Neither of the above**

# Tivoli authorization

IBM Tivoli Access Manager for e-business (TAM) provides authentication and access control services for web resources. It also stores policies describing the access rights of users.

API Gateway can integrate with this product through the **Tivoli** filter. This filter can query Tivoli for authorization information for a particular user on a given resource. In other words, API Gateway asks Tivoli to make the authorization decision. If the user has been given authorization rights to the web service, the request is allowed through to the service. Otherwise, the request is rejected.

## Prerequisites

Before you can configure the **Tivoli** filter, you must configure API Gateway for integration with TAM. For more details, see the *API Gateway Authentication and Authorization Integration Guide*

# Configuration

Configure the following fields on the **Tivoli Authorization** window:

**Name**:
Enter an appropriate name for the filter to display in a policy.

**Object Space**:
The object space represents the resource for which the client must be authorized. Enter the name of the resources in the **Object Space** field. You can also enter selectors that represent the values of message attributes. At runtime, API Gateway expands the selector to the current value of the corresponding message attribute. For example, to specify the original path on which the request was received by API Gateway as the resource, enter the selector `${http.request.uri}`. For more details on selectors, see "Select configuration values at runtime" in the *API Gateway Policy Developer Guide*.

**Permissions**:
Clients can access a resource with a number of permissions such as read, write, and execute. A client is only authorized to access the requested resource if it has the relevant permissions checked in the table. To edit the existing permissions, click **Edit**.

**Attributes**:
You can specify a list of user attributes to retrieve from the Tivoli server. To add attributes to be retrieved that are not listed, click **Add**, and enter the attribute name in the dialog. If you want to retrieve all attributes, leave the table blank, and select **Set attributes for SAML Attribute token**. You can then use these attributes in a **Insert SAML Attribute Assertion** filter at a later stage. If you do not want to retrieve any attributes, leave the table blank and deselect **Set attributes for SAML Attribute token**.

**Note** The permissions for the `primary` action group are available by default. You can also configure custom action groups and make them available for selection in the filter. The groups created here reflect custom groups created on the Tivoli server. To create a new group with custom action bits, click the **Edit** button to display the **Tivoli Action Group** dialog.

Enter a name for the group in the **Name** field. Click **Add** to add a new action bit to the group. The **Tivoli Action** dialog is displayed. You must enter an **Action Bit** (for example, `r`) and a **Description** (for example, `Read permission`) for the new action bit. Click **OK** on the **Tivoli Action** dialog to return to the **Tivoli Action Group** dialog.

Add as many action bits as required to your new group before clicking **OK** on the **Tivoli Action Group** dialog. The new action bits are then available for selection in the table on the main filter window.

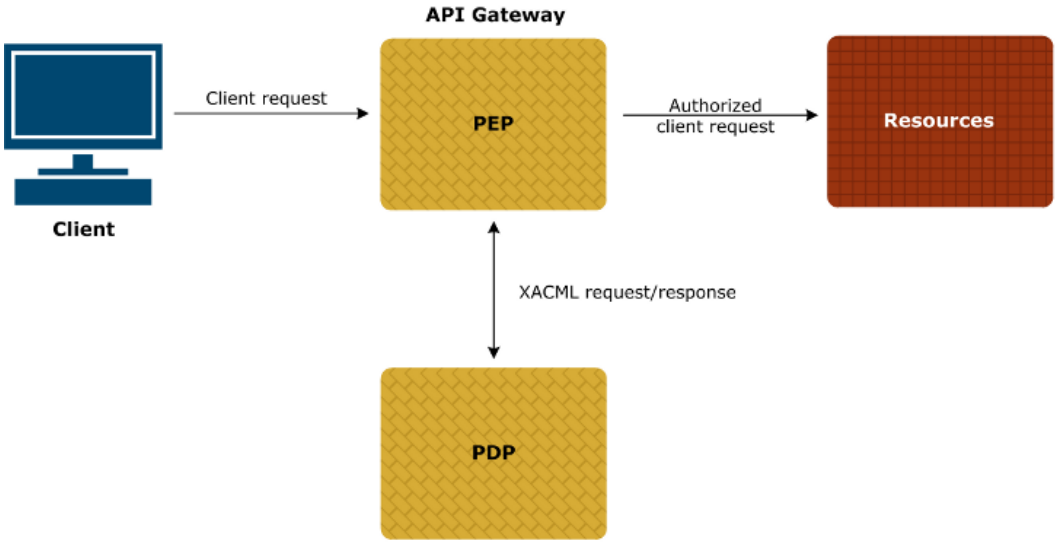
**Tivoli Configuration Files**:
A Tivoli configuration file that contains all the required connection details is associated with a particular API Gateway instance. Click **Settings** to display the **Tivoli Configuration** dialog.

On the **Tivoli Configuration** dialog, select the API Gateway instance whose connection details you want to configure, then follow the steps as outlined in the *API Gateway Authentication and Authorization Integration Guide*.

# XACML PEP authorization

## Overview

The eXtensible Access Control Markup Language (XACML) Policy Enforcement Point (PEP) filter enables you to configure the API Gateway to act as a PEP. The API Gateway intercepts a user request to a resource, and enforces the decision from the Policy Decision Point (PDP). The API Gateway queries the PDP to see if the user has access to the resource, and depending on the PDP response, allows the filter to pass or fail. Possible PDP responses include `Permit`, `Deny`, `NotApplicable`, and `Indeterminate`.




### *Workflow*

In more detail, when the **XACML PEP** filter is configured in the API Gateway, the workflow is as follows:

1. The client sends a request for the resource to the XACML PEP filter.
2. The PEP filter stores the original client request, and generates the XACML request.
3. The PEP filter delegates message-level security to the polices configured on the **XACML** tab.
4. The PEP filter routes the XACML request to the PDP using details configured on the **Routing** tab.
5. The PDP decides if access should be granted, and sends the XACML response back to the API Gateway.
6. The PEP filter validates the response from the PDP.
7. By default, if the response is `Permit`, the PEP filter passes, and the original client request for the resource is authorized, and the policy flow continues on the success path.

### *Further information*

For more details on XACML, see the XACML specification at:
http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf

## Example XACML request

The following example XACML request is used to illustrate the XACML request configuration settings explained in this topic:

```
<Request xmlns="urn:oasis:names:tc:xacml:2.0:context:schema:os"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <Subject>
      <Attribute AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"
        DataType="http://www.w3.org/2001/XMLSchema#string">
        <AttributeValue>admin</AttributeValue>
      </Attribute>
      <Attribute AttributeId=”department"
        DataType="http://www.w3.org/2001/XMLSchema#string">
        <AttributeValue>sysadmin</AttributeValue>
      </Attribute>
    </Subject>
    <Resource>
      <Attribute AttributeId="urn:oasis:names:tc:xacml:1.0:resource:resource-id"
        DataType="http://www.w3.org/2001/XMLSchema#string">

<AttributeValue>http://localhost:8280/services/echo/echoString</AttributeValue>
      </Attribute>
    </Resource>
    <Action>
      <Attribute AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"
        DataType="http://www.w3.org/2001/XMLSchema#string">
        <AttributeValue>read</AttributeValue>
      </Attribute>
    </Action>
    <Environment/>
</Request>
```

## XACML settings

The **XACML** tab specifies configuration settings for the generated XACML request. Configure the following fields on this tab:

**XACML Version**:
Select the XACML version from the list. Defaults to `XACML2_0`.

**Create XACML Request Assertion with the following attributes**:
Click the **Add** button on the following tabs to add attributes to the XACML request:

| | |
|---|---|
| **Subject** | Represents the entity making the access request (wants access to the resource). The `Subject` element can contain multiple `Attribute` elements used to identify the `Subject`. Each `Attribute` element has two attributes: `AttributeId` and `DataType`. You can define your own `AttributeId` or use those provided by the XACML specification. For more details on adding attributes, see the next subsection. |
| **Resource** | Defines the data, service, or system component that the `Subject` wants to access. The `Resource` element contains one or more attributes of the resource to which subjects request access. There can be only one `Resource` element per XACML request. A specific `Resource` is identified by the `Attribute` child element. In the *Example XACML request* on page 154, the `Subject` wants to access the following `Resource`: `http://localhost:8280/services/echo/echoString` |
| **Action** | Contains one or more attributes of the action that subjects wish to perform on the resource. There can be only one `Action` element per XACML request. A specific `Action` is identified by the `Attribute` child element. In the *Example XACML request* on page 154, the `Subject` wants read access the following `Resource`: `http://localhost:8280/services/echo/echoString` |
| **Environment** | A more complex request context may contain some attributes not associated with the `Subject`, `Resource`, or `Action`. These are placed in an optional `Environment` element after the `Action` element. |

When you click the **Add** button on each tab, the **XACML** dialog is displayed to enable you to add attributes. Complete the following fields on this dialog:

**Attribute ID**:
Enter a custom `AttributeId` or select one provided by the XACML specification from the list. For example, the XACML special identifiers defined for the `Subject` include the following:

```
urn:oasis:names:tc:xacml:1.0:subject:
authn-locality:dns-name
urn:oasis:names:tc:xacml:1.0:subject:
authn-locality:ip-address
urn:oasis:names:tc:xacml:1.0:subject:
authentication-method
urn:oasis:names:tc:xacml:1.0:subject:
authentication-time
urn:oasis:names:tc:xacml:1.0:subject:
key-info
urn:oasis:names:tc:xacml:1.0:subject:
request-time
```

```
urn:oasis:names:tc:xacml:1.0:subject:
session-start-time
urn:oasis:names:tc:xacml:1.0:subject:
subject-id
...
```

In the *Example XACML request* on page 154, the first attribute under the `Subject` element uses the `urn:oasis:names:tc:xacml:1.0:subject:subject-id` identifier. The next is a custom `department` attribute. This can be any custom attribute for example, `mail`, `givenName`, or `accessList`), which is identified by the XACML policy defined where this request is evaluated.

**Value(s)**:
Click the **Add** button to add an attribute value. Enter the value in the **Add** dialog, and click **OK**. You can add multiple values for a single attribute.

**Type**:
Select the type of data that the `AttributeValue` element should contain from the list. For example, the set of data types defined in XACML includes the following:

```
http://www.w3.org/2001/XMLSchema#string
http://www.w3.org/2001/XMLSchema#boolean
http://www.w3.org/2001/XMLSchema#integer
http://www.w3.org/2001/XMLSchema#double
http://www.w3.org/2001/XMLSchema#time
http://www.w3.org/2001/XMLSchema#date
http://www.w3.org/2001/XMLSchema#dateTime
http://www.w3.org/TR/2002/WD-xqueryoperators-
20020816#dayTimeDuration
http://www.w3.org/TR/2002/WD-xqueryoperators-
20020816#yearMonthDuration
http://www.w3.org/2001/XMLSchema#anyURI
http://www.w3.org/2001/XMLSchema#hexBinary
...
```

In the *Example XACML request* on page 154, the Attributes are of type `http://www.w3.org/2001/XMLSchema#string`.

**Issuer**:
Specify an optional issuer for the attribute. For example, this may be a Distinguished Name, or some other identifier agreed with the issuer.

## *AuthzDecisionQuery Settings*

This section enables you to configure settings for the Authorization Decision Query, which is sent in the XACML request to the PDP. Complete the following fields in this group:

**Decision based on external XACML attributes**:
If this is selected, the authorization decision must be made based only on the information contained in the XACML Authz Decision Query, and external XACML attributes must not be used. If this is unselected, the authorization decision can be made based on XACML attributes not contained in the XACML Authz Decision Query. This is unselected by default, which is equivalent to the following setting in the XACML Authz Decision Query:

```
<InputContextOnly value="false">
```

**Return Context**:
If this is selected, the PDP must include an `xacmlcontext:Request` instance in the `XACMLAuthzDecision` statement in the `XACMLAuthzDecision` response. The `xacmlcontext:Request` instance must include all attributes supplied by the PEP in the `xacml-samlp:XACMLAuthzDecisionQuery` used to make the authorization decision. If this is unselected, the PDP must not include an `xacmlcontext:Request` instance in the `XACMLAuthzDecision` statement in the `XACMLAuthzDecision` response. This is unselected by default, which is equivalent to the following setting in the XACML request:

```
<ReturnContext value="false">
```

**Combine Policies**:
If this is selected, the PDP must insert all policies passed in the `xacmlsamlp:XACMLAuthzDecisionQuery` into the set of policies or policy sets that define the PDP. If this is unselected, there must be no more than one `xacml:Policy` or `xacml:PolicySet` passed in the `xacml-samlp:XACMLAuthzDecisionQuery` . This is selected by default, which is equivalent to the following setting in the XACML request:

```
<CombinePolicies value="true">
```

## *XACML Message Security*

This section enables you to delegate message-level security to the configured custom security polices. Complete the following fields in this group:

- **XACML Request Security**:
  Click the browse button, select a policy in the **XACML request security policy** dialog, and click **OK**.
- **XACML Response Security**:
  Click the browse button, select a policy in the **XACML response security policy** dialog, and click **OK**.

## *XACML Response*

Select the **Required response decision** from the PDP that is required for this **XACML PEP** filter to pass. Defaults to `Permit`. Possible values are as follows:

- `Permit`
- `Deny`
- `Indeterminate`
- `NotApplicable`

# Routing settings

The **Routing** tab enables you to specify configuration settings for routing the XACML request to the PDP. You can specify a direct connection to the PDP using a URL. Alternatively, if the routing behavior is more complex, you can delegate to a custom routing policy, which takes care of the added complexity.

**Use the following URL**:

To route XACML requests to a URL, select this option, and enter the **URL**. You can also specify the URL as a selector so that the URL is built dynamically at runtime from the specified message attributes. For example, `${host}:${port}`, or `${http.destination.protocol}://${http.destination.host}:${http.destination.port}`. For more details on selectors, see "Select configuration values at runtime" in the *API Gateway Policy Developer Guide*.

You can configure SSL settings, credential profiles for authentication, and other settings for the direct connection using the tabs in the **Connection Details** group. For more details, see *Connect to URL* on page 380.

**Delegate routing to the following policy**:
To use a dedicated routing policy to send XACML requests to the PDP, select this option. Click the browse button next to the **Routing Policy** field. Select the policy to use to route XACML requests, and click **OK**.

# Advanced settings

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

## *SOAP Settings*

The available SOAP settings are as follows:

**SOAP version required**:
Specifies the SOAP version required when creating the XACML request message. The available options are as follows:

- `SOAP1_1`
- `SOAP1_2`
- `NONE`

Defaults to `SOAP1_1`.

**SOAP Operation**:
Specifies the SOAP operation name used in the XACML request message. Defaults to `XACMLAuthzDecisionQuery`.

**Prefix**:
Specifies the prefix name used in the XACML request message. Defaults to `xacml-samlp`.

**Namespace**:
Specifies the namespace used in the XACML request message. Defaults to `urn:oasis:xacml:2.0:saml:protocol:schema:os`.

**SOAP Action**:
You can specify an optional `SOAPAction` field used in the XACML request header to indicate the intent of the request message.

## *Advanced Settings*

The available advanced settings are as follows:

**Store and restore original message**:
Specifies whether to store the original client request before generating the XACML request, and then to restore the original client request after access is granted. This option is selected by default.

**Split subject attributes into individual elements**:
Specifies whether to split `Subject` attributes into individual elements in the XACML request. This option is not selected by default.

**Split resource attributes into individual elements**:
Specifies whether to split `Resource` attributes into individual elements in the XACML request. This option is not selected by default.

# CA SiteMinder filters 5

The following filters enable you to integrate with CA SiteMinder:



## Prerequisites

Integration with CA SiteMinder requires CA SiteMinder SDK version 12.52-sp02 or later. You must add the required third-party binaries to your API Gateway installation.

1. Ensure that any SiteMinder binaries you may have previously added to API Gateway have been deleted.
2. Install CA SiteMinder SDK.
3. Copy the following `jar` files from the `java` directory of the CA SDK :
   - `cryptoj.jar`
   - `smagentapi.jar`
   - `smjavasdk2.jar`
4. Add the files to the following directory on API Gateway:

   ```
   INSTALL_DIR/apigateway/<platform>/lib
   ```

5. Restart API Gateway.

## CA SiteMinder certificate authentication

CA SiteMinder can authenticate end users and authorize them to access protected web resources. When API Gateway retrieves an X.509 certificate from a message or during an SSL handshake, it can use the **Certificate Authentication** filter to authenticate to SiteMinder on behalf of the user using the certificate. SiteMinder decides whether the user should be authenticated, and API Gateway then enforces this decision.

Integration with CA SiteMinder requires adding the required third-party binaries to your API Gateway and Policy Studio installations. For more details, see *Prerequisites* on page 160.

# Configuration

Configure the following fields:

**Name**:
Enter an appropriate name for the filter to display in a policy.

**Agent Name**:
Click the browse button, and select a previously configured agent to connect to SiteMinder. This name *must* correspond with the name of an agent previously configured in the CA SiteMinder Policy Server. At runtime, API Gateway connects as this agent to a running instance of SiteMinder.

To add an agent, in the node tree, click **Environment Configuration > External Connections**, right-click the **SiteMinder/SOA Security Manager Connections**, and select **Add a SiteMinder Connection**. For more details on how to configure a SiteMinder connection, see "Configure SiteMinder/SOA Security Manager connections" in the *API Gateway Policy Developer Guide*.

**Resource**:
Enter the name of the protected resource for which the end user must be authenticated. You can enter a selector representing a message attribute that is expanded to a value at runtime. For example, by default API Gateway specifies the original path the end user requested as the resource:

```
${http.request.uri}
```

**Action**:
The user must be authenticated for a specific action on the protected resource. By default, API Gateway takes this action from the HTTP verb used in the incoming request. You can use the following selector to get the HTTP verb:

```
${http.request.verb}
```

Alternatively, any user-specified value can be entered. For more details on selectors, see "Select configuration values at runtime" in the *API Gateway Policy Developer Guide*.

**Single Sign-On Token**:
By default, when an end user has been authenticated for a given resource, SiteMinder generates a *single sign-on token* (a session cookie). API Gateway stores this cookie in a user-specified message attribute. API Gateway returns the cookie to the end user along with the response. The end user can then pass this cookie with future requests to API Gateway. When API Gateway receives such a request, it can validate the session cookie using the **Session Validation** filter. The end user stays authenticated for the entire lifetime of the session cookie. As long as the session cookie is valid, API Gateway does not need to re-authenticate the end user against SiteMinder for every request. This increases throughput and performance considerably.

Typically, API Gateway copies the cookie to the `attribute.lookup.list` message attribute using the **Copy / Modify Attributes** filter, before inserting the cookie into a SAML attribute statement using the **Insert SAML Attribute Assertion** filter. The attribute statement is then returned to the end user to be used use in subsequent requests.

**Put Token in Message Attribute**:
Enter the name of the message attribute where you wish to store the session cookie. By default, the cookie is stored in the `siteminder.session` attribute.

For more details how to integrate API Gateway with SiteMinder, see the *API Gateway Authentication and Authorization Integration Guide*.

# CA SiteMinder session validation

CA SiteMinder can authenticate end users and authorize them to access protected web resources. When API Gateway has authenticated successfully to SiteMinder on behalf of an end user, SiteMinder can issue a *single sign-on* token (a session cookie) and return it to API Gateway. API Gateway returns the cookie along with the response to the end user. API Gateway can also insert the cookie into a SAML attribute assertion for downstream web services to consume.

The client then includes the session cookie in subsequent requests to API Gateway. API Gateway can then use the **Session Validation** filter to ensure that the session cookie is still valid and hence that the user is still authenticated. This means that API Gateway does not have to authenticate every request to SiteMinder and unnecessary round-trips to SiteMinder can be avoided.

Integration with CA SiteMinder requires adding the required third-party binaries to your API Gateway and Policy Studio installations. For more details, see *Prerequisites* on page 160.

## Configuration

Configure the following fields:

**Name**:
Enter an appropriate name for the filter to display in a policy.

**Agent Name**:
Click the browse button, and select a previously configured agent to connect to SiteMinder. This name *must* correspond with the name of an agent previously configured in the CA SiteMinder Policy Server. At runtime, API Gateway connects as this agent to a running instance of SiteMinder.

To add an agent, in the node tree, click **Environment Configuration > External Connections**, right-click the **SiteMinder/SOA Security Manager Connections**, and select **Add a SiteMinder Connection**. For more details on how to configure a SiteMinder connection, see "Configure SiteMinder/SOA Security Manager connections" in the *API Gateway Policy Developer Guide*.

**Resource**:
Enter the name of the protected resource for which the end user must be authenticated. You can enter a selector representing a message attribute that is expanded to a value at runtime. For example, by default API Gateway specifies the original path the end user requested as the resource:

```
${http.request.uri}
```

**Action**:
The user must be authenticated for a specific action on the protected resource. By default, API Gateway takes this action from the HTTP verb used in the incoming request. You can use the following selector to get the HTTP verb:

```
${http.request.verb}
```

Alternatively, any user-specified value can be entered. For more details on selectors, see "Select configuration values at runtime" in the *API Gateway Policy Developer Guide*.

**Selector Expression to retrieve session**:
Enter the name of the message attribute that contains the session cookie SiteMinder generated. By default, the token is stored in the `siteminder.session` message attribute.

# CA SiteMinder logout

When API Gateway has authenticated successfully to CA SiteMinder on behalf of an end user, SiteMinder can issue a *single sign-on* token (a session cookie) and return it to API Gateway. API Gateway returns the cookie along with the response to the end user. The client includes the session cookie in subsequent requests to API Gateway. API Gateway can then use the **Session Validation** filter to ensure that the session cookie is still valid and hence that the user is still authenticated.

You can use the **Logout** filter to terminate a previously issued session cookie. After the cookie has been terminated, the client is no longer considered authenticated.

**Note** You must have already validated the session before calling the **Logout** filter in your policy. For more details, see *CA SiteMinder session validation* on page 162.

Integration with CA SiteMinder requires adding the required third-party binaries to your API Gateway and Policy Studio installations. For more details, see *Prerequisites* on page 160.

# CA SiteMinder authorization

CA SiteMinder can authenticate end users and authorize them to access protected web resources. API Gateway can use the **Authorization** filter to interact directly with SiteMinder, and ask SiteMinder to make authorization decisions on behalf of end users that have successfully authenticated to API Gateway. API Gateway then enforces the decisions made by SiteMinder.

**Note** You must configure authentication to CA Siteminder before you configure the **Authorization** filter. For example, you could use a HTTP Basic authentication filter configured with the CA Siteminder authentication repository, or a CA SiteMinder **Certificate Authentication** filter.

Integration with CA SiteMinder requires adding the required third-party binaries to your API Gateway and Policy Studio installations. For more details, see *Prerequisites* on page 160.

# Configuration

Configure the following fields on the **Authorization** filter:

**Name**:
Enter an appropriate name for the filter to display in a policy.

**Attributes**:
After the end user is successfully authorized, the attributes listed here are returned to API Gateway and stored in the `attribute.lookup.list` message attribute. API Gateway can use these attributes in subsequent filters in a policy to make decisions based on their values. Alternatively, API Gateway can insert the attributes into a SAML attribute assertion so that the target service can apply some business logic based on their values (for example, if role is CEO, escalate the request, and so on).

To specify an attribute to fetch from SiteMinder, select the **Retrieve attributes from CA SiteMinder**option, and click the **Add** button. If you select the **Retrieve attributes from CA SiteMinder** option, and do not specify attribute names to be retrieved, all attributes returned by SiteMinder are added to the `attribute.lookup.list` message attribute.

# Certificate filters 6

The following filters enable you to work with certificates:



# Introduction to certificate validation

## Overview

Whenever API Gateway receives an X.509 certificate, either as part of the SSL handshake or as part of the XML message itself, it is important to be able to determine whether that certificate is legitimate or not.

Certificates can be revoked by their issuers if it becomes apparent that the certificate is being used maliciously. Such certificates should never be trusted, and so it is very important that API Gateway can perform certificate validation.

API Gateway uses the following methods/protocols to validate certificates:

- **Online Certificate Status Protocol** (OCSP) – An automated certificate checking network protocol. API Gateway can query the OCSP responder for the status of a certificate. The responder returns whether the certificate is still trusted by the CA that issued it.
- **Certificate Revocation List** (CRL) – A signed list indicating a set of certificates that are no longer considered valid (that is, revoked certificates) by the certificate issuer. API Gateway can query a CRL to find out if a given certificate has been revoked. If the certificate is present in the CRL, it should not be trusted.

- **XML Key Management Services** (XKMS) – An XML-based protocol for (amongst other things) establishing the trustworthiness of a certificate over the Internet. API Gateway can query an XKMS responder to determine whether or not a given certificate can be trusted or not.

## Certificate validation filters

The API Gateway can check the validity of a client certificate using any of the following filters:

- *OCSP client* on page 178
- *Static CRL certificate validation* on page 168
- *Dynamic CRL certificate validation* on page 166
- *CRL LDAP validation* on page 167
- *XKMS certificate validation* on page 186

**Note** To validate a certificate using an OCSP request or CRL lookup, the issuing CA's certificate should be trusted by API Gateway.

For a CRL lookup, the CA's public key is needed to verify the signature on the CRL. For an OCSP request, the CA's public key must be submitted as part of the request. The issuing CA's public key is not always present in issued certificates, so it is necessary to retrieve it from the API Gateway certificate store instead.

# Dynamic CRL certificate validation

## Overview

This filter enables validation of certificates against a Certificate Revocation List (CRL) that has been published by a Certificate Authority (CA). The CRL is retrieved from the specified URL and is cached by the server for certificate validation. The filter automatically fetches a potentially updated CRL from this URL when the criteria specified in the **Automatic CRL Update Preferences** section are met.

**Note** Because the CRL is typically signed by the CA that owns it, the CA certificate that issued the CRL *must* first be imported into the certificate store. In addition, the **CRL (Dynamic)** filter requires the `certificates` message attribute to be set by a preceding filter.

## Example CRL-based validation policy

The topic on the **CRL (Static)** filter shows a typical CRL-based policy that first uses a **Find Certificate** filter to find the certificate, which is stored in a `certificate` message attribute. It

then uses a **Copy/Modify Attributes** filter to copy this `certificate` attribute to the `certificates` attribute by selecting its **Create list attribute** setting. This `certificates` attribute is then used by the CRL-based filter.

You can use the same approach with the **CRL (Dynamic)** filter instead of the **CRL (Static)** filter. For more details, see *Static CRL certificate validation* on page 168.

## Configuration

Configure the following fields on the **CRL (Dynamic)** window:

**Name**:
Enter an appropriate name for the filter to display in a policy.

**CRL Import URL**:
Enter the full URL of the CRL to use to validate the certificate, or browse to the CRL location.

**Automatic CRL Update Preferences**:
Typically, a CA publishes an updated CRL at regular intervals. You can configure the filter to dynamically pull the latest CRL published by the CA at specified intervals. Select one of the following update options:

- **Do not update**:
  The filter never attempts to automatically retrieve the latest CRL.
- **Update on "next update" date**:
  The CRL published by the CA contains a *Next Update* date, which indicates the next date on which the CA publishes the CRL. You can choose to dynamically retrieve the updated CRL on the Next Update date by selecting this option. This effectively synchronizes the server with the CA updates.
- **Update every number of days**:
  The filter retrieves the CRL every number of days specified.
- **Trigger update on cron expression**:
  You can enter a cron expression to determine when to perform the automatic update.

# CRL LDAP validation

## Overview

A Certificate Revocation List (CRL) is a signed list indicating a set of certificates that are no longer considered valid (revoked certificates) by the certificate issuer. API Gateway can query a CRL to find out if a given certificate has been revoked. If the certificate is present in the CRL, it should not be trusted.

To validate a certificate using a CRL lookup, the certificate's issuing CA certificate should be trusted by API Gateway. This is because for a CRL lookup, the CA public key is needed to verify the signature on the CRL. The issuing CA public key is not always included in the certificates that it issues, so it is necessary to retrieve it from API Gateway's certificate store instead. For more information on how to trust CA certificates, see "Manage X.509 certificates and keys" in the *API Gateway Policy Developer Guide*.

## Configuration

The **Name** and **URL** of all currently configured LDAP directories are displayed in the table on the **CRL Certificate Validation** window. API Gateway checks the CRL of all selected LDAP directories to validate the client certificate. The filter fails as soon as API Gateway determines that one of the CRLs has revoked the certificate.

To configure LDAP connection information, complete the following fields:

**Name**:
Enter an appropriate name for the filter to display in a policy.

**LDAP Connection**:
Click the button on the right, and select the LDAP directory to check its CRL. To use an existing LDAP directory, (for example, `Sample Active Directory Connection`), you can select it in the tree. To add an LDAP directory, right-click the **LDAP Connections** tree node, and select **Add an LDAP Connection**.

Alternatively, you can add LDAP connections under the **Environment Configuration** > **External Connections** node in the Policy Studio tree. For more details on how to configure LDAP connections, see "Configure LDAP directories" in the *API Gateway Policy Developer Guide*.

# Static CRL certificate validation

## Overview

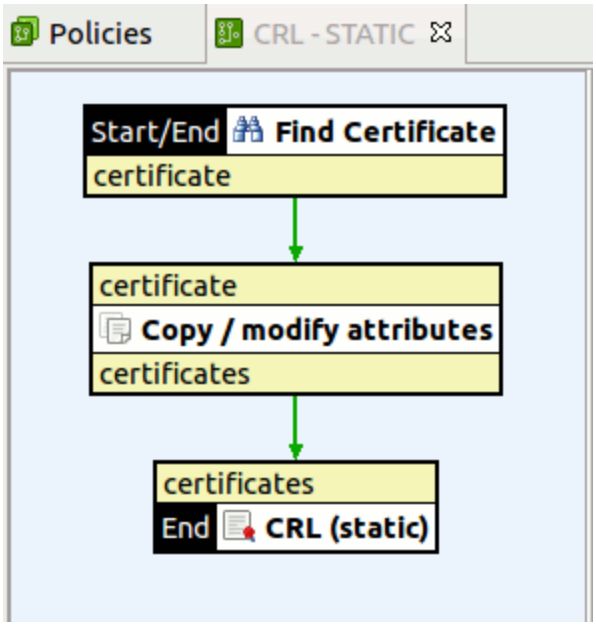
A Certificate Authority (CA) might wish to publish a Certificate Revocation List (CRL) to a file. In this case, API Gateway can load the revoked certificates from the file-based CRL and validate user certificates against it.

**Note** Because the CRL is typically signed by the CA that owns it, the CA certificate that issued the CRL *must* first be imported into the certificate store. In addition, the **CRL (Static)** filter requires the `certificates` message attribute to be set by a preceding filter.

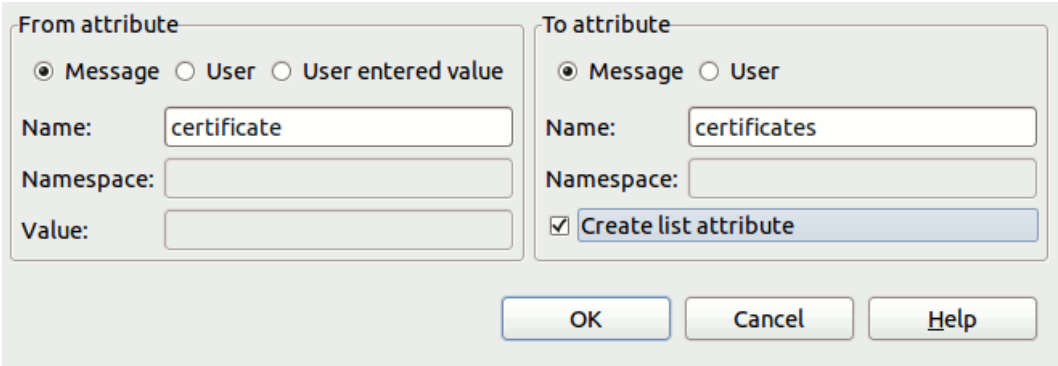
# Example CRL-based validation policy

Typically, a **Find Certificate** filter is first used to find the certificate, which is stored in a `certificate` message attribute. You can then use a **Copy / Modify Attributes** filter to copy the `certificate` attribute to the `certificates` attribute by selecting its **Create list attribute** setting.

The following example policy shows the filters used:




The following example shows the settings used in the **Copy / Modify Attributes** filter:




**Note** Typically, a CA publishes a new CRL, containing the most up-to-date list of revoked certificates at regular intervals. However, the **CRL (Static)** filter does not automatically update the CRL when it is loaded from a local file. If you need to automatically retrieve updated CRLs from a particular URL, you should use the **CRL (Dynamic)** filter instead. For more details, see *Dynamic CRL certificate validation* on page 166.

## Configuration

Enter a name for the filter to display in a policy in the **Name** field.

Click the **Load CRL** button to browse to the location of the CRL file. When the CRL has been loaded from the selected location, read-only information regarding revoked certificates and update dates is displayed in the other fields on the window.

# CRL responder

## Overview

This filter enables API Gateway to behave as Certificate Revocation List (CRL) responder, which returns CRLs to clients. This filter imports the CRL from a specified URL. You can also configure it to periodically retrieve the CRL from this URL to ensure that it always has the latest version.

## Configuration

Configure the following fields on the **CRL Responder** window:

**Name**:
Enter an appropriate name for the filter to display in a policy.

**CRL Import URL**:
Enter the full URL of the CRL that you want to return to clients. Alternatively, browse to the location of the CRL file by clicking the browse button on the right.

**Automatic CRL Update Preferences**:
Because keeping up-to-date with the latest list of revoked certificates is crucial in any *trust network* , it is important that you configure the filter to retrieve the latest version of the CRL on a regular basis. The following automatic update options are available:

- **Do not update**:
  The CRL is not automatically updated.
- **Update on "next update" date**:
  The CRL published by the CA contains a *Next Update* date, which indicates the next date on which the CA publishes the CRL. You can choose to dynamically retrieve the updated CRL on the Next Update date by selecting this option. This effectively synchronizes the server with the CA updates.
- **Update every number of days**:
  The CRL is updated after the specified number of days has elapsed (for example, every 3 days).
- **Trigger update on cron expression**:
  You can enter a cron expression to determine when to perform the automatic update.

# Certificate chain check

## Overview

It is a trivial task for a user to generate a structurally sound X.509 certificate, and use it to negotiate mutually authenticated connections to publicly available services. However, this scenario is a security nightmare for IT administrators. You cannot allow every user to generate their own certificate and use it on the Internet. For this reason, API Gateway can establish the authenticity of the client certificate by ensuring that the certificate originated from a trusted source. To do this, a server can perform a *certificate chain check* on the client certificate.

The main purpose of certificate chain validation is to ensure that a certificate has been issued by a trusted source. Typically, in a Public Key Infrastructure (PKI), a Certificate Authority (CA) is responsible for issuing and distributing certificates. This infrastructure is based on the premise of *transitive trust*—if everybody trusts the CA, everybody transitively trusts the certificates issued by that CA. If entities only trust certificates that have been issued by the CA, they can reject certificates that have been self-generated by clients.

When a CA issues a certificate, it digitally signs the certificate and inserts a copy of its own certificate into it. This is called a certificate chain. Whenever an application (such as API Gateway) receives a client certificate, it can extract the issuing CA certificate from it, and run a certificate chain check to determine whether it should trust the CA. If it trusts the CA, it also trusts the client certificate.

API Gateway maintains a repository of trusted CA certificates, which is known as the certificate store. To *trust* a specific CA, that CA certificate must be imported into the certificate store. For more details, see "Manage X.509 certificates and keys" in the *API Gateway Policy Developer Guide*.

## Configuration

You can configure the following settings on the **Certificate Chain Check** window:

**Name**:
Enter an appropriate name for this filter to display in a policy.

**Certificates Message Attribute**:
You can specify a message attribute that contains the certificate or certificates to check. The message attribute type can be an `X509Certificate` object, or an `ArrayList` of `X509Certificate` objects.

**Distinguished Name**:
This table lists the Distinguished Names of the certificates currently in the certificate store. Select the check box beside a CA to enable this filter to consider it as *trusted* when performing the certificate chain check. You can select multiple CAs in the table.

# Certificate validity

## Overview

The validity period of an X.509 certificate is encoded in the certificate. The **Certificate Validity** filter performs a simple check on a certificate to ensure that it has not expired.

By default, the **Certificate Validity** filter searches for the X.509 certificate in the `certificate` message attribute, which must be set by a predecessor filter in the policy (for example, by an **SSL Authentication** filter).

## Configuration

Configure the following fields on the **Certificate Validity** window:

**Name**:
Enter an appropriate name for the filter to display in a policy.

**Certificate Selector Expression**:
Enter the selector expression that specifies where to obtain the certificate (for example, from a message attribute). The filter checks the validity of the specified certificate. If no certificate is found, the filter returns an error. Defaults to `${certificate}`.

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" in the *API Gateway Policy Developer Guide*.

# Create thumbprint from certificate

## Overview

The **Create Thumbprint** filter can be used to create a human-readable thumbprint (or fingerprint) from the X.509 certificate that is stored in the `certificate` message attribute. The generated thumbprint is stored in the `certificate.thumbprint` attribute.

## Configuration

Configure the following fields on this filter:

**Name**:
Enter a name for this filter to display in a policy.

**Digest Algorithm**:
Select the digest algorithm to create the thumbprint of the certificate from the list.

# Extract certificate attributes

## Overview

You can use the **Extract Certificate Attributes** filter to extract the X.509 attributes from a certificate stored in a specified API Gateway message attribute.

Typically, this filter is used in conjunction with the **Find Certificate** filter, which is found in the **Certificates** category of message filters. In this case, the **Find Certificate** filter can locate a certificate from one of many possible sources (for example, the message itself, an HTTP header, or the API Gateway certificate store), and store it in a message attribute, which is usually the `certificate` attribute.

The **Extract Certificate Attributes** filter can then retrieve this certificate and extract the X.509 attributes from it. For example, you can then use a **Validate Message Attribute** filter to check the values of the attributes.

## Generated message attributes

The **Extract Certificate Attributes** filter extracts the X.509 certificate attributes and populates a number of API Gateway message attributes with their respective values. The following table lists the message attributes that are generated by this filter, and shows what each of these attributes contains after the filter has executed:

| Generated message attribute | Contains |
|---|---|
| `attribute.lookup.list` | This user attribute list contains an attribute for each Distinguished Name (DName) attribute for the subject (`cn`, `o`, `l`, and so on). The user attributes are named `cn`, `o`, and so on. |
| `attribute.subject.id` | The DName of the subject of the cert. |
| `attribute.subject.format` | Set to `X509DName`. |

| Generated message attribute | Contains |
|---|---|
| cert.basic.constraints | If the subject is a Certificate Authority (CA), and the BasicConstraints extension exists, this field gives the maximum number of CA certificates that may follow this certificate in a certification path. A value of zero indicates that only an end-entity certificate may follow in the path. This contains the value of pathLenConstraint if the BasicConstraints extension is present in the certificate and the subject of the certificate is a CA, otherwise its value is -1. If the subject of the certificate is a CA and pathLenConstraint does not appear, there is no limit to the allowed length of the certification path. |
| cert.extended.key.usage | A String representing the OBJECT IDENTIFIERs of the ExtKeyUsageSyntax field of the extended key usage extension (OID = 2.5.29.37). It indicates a purpose for which the certified public key may be used, in addition to, or instead of, the basic purposes indicated in the key usage extension field. |
| cert.hash.md5 | An MD5 hash of the certificate. |
| cert.hash.sha1 | A SHA1 hash of the certificate. |
| cert.issuer.alternative.name | An alternative name for the certificate issuer from the IssuerAltName extension (OID = 2.5.29.18). |
| cert.issuer.id | The DName of the issuer of the certificate. |
| cert.issuer.id.c | The c attribute of the issuer of the certificate, if it exists. |
| cert.issuer.id.cn | The cn attribute of the issuer of the certificate, if it exists. |
| cert.issuer.id.emailaddress | The email or emailaddress attribute of the issuer of the certificate, if it exists. |

| Generated message attribute | Contains |
|---|---|
| cert.issuer.id.l | The l attribute of the issuer of the certificate, if it exists. |
| cert.issuer.id.o | The o attribute of the issuer of the certificate, if it exists. |
| cert.issuer.id.ou | The ou attribute of the issuer of the certificate, if it exists. |
| cert.issuer.id.st | The st attribute of the issuer of the certificate, if it exists. |
| cert.key.usage.cRLSign | Set to true or false if the key can be used for crlSign. |
| cert.key.usage.dataEncipherment | Set to true or false if the key can be used for dataEncipherment. |
| cert.key.usage.decipherOnly | Set to true or false if the key can be used for decipherOnly. |
| cert.key.usage.digitalSignature | Set to true or false if the key can be used for digital signature. |
| cert.key.usage.encipherOnly | Set to true or false if the key can be used for encipherOnly. |
| cert.key.usage.keyAgreement | Set to true or false if the key can be used for keyAgreement. |
| cert.key.usage.keyCertSign | Set to true or false if the key can be used for keyCertSign. |
| cert.key.usage.keyEncipherment | Set to true or false if the key can be used for keyEncipherment. |
| cert.key.usage.nonRepudiation | Set to true or false if the key can be used for non-repudiation. |
| cert.not.after | Not after validity period date. |
| cert.not.before | Not before validity period date. |

| Generated message attribute | Contains |
|---|---|
| `cert.serial.number` | Certificate serial number. |
| `cert.signature.algorithm` | The signature algorithm for certificate signature. |
| `cert.subject.alternative.name` | An alternative name for the subject from the `SubjectAltName` extension (`OID = 2.5.29.17`). |
| `cert.subject.id` | The DName of the subject of the certificate. |
| `cert.subject.id.c` | The `c` attribute of the subject of the certificate, if it exists. |
| `cert.subject.id.cn` | The `cn` attribute of the subject of the certificate, if it exists. |
| `cert.subject.id.emailaddress` | The `email` or `emailaddress` attribute of the subject of the certificate, if it exists. |
| `cert.subject.id.l` | The `l` attribute of the subject of the certificate, if it exists. |
| `cert.subject.id.o` | The `o` attribute of the subject of the certificate, if it exists. |
| `cert.subject.id.ou` | The `ou` attribute of the subject of the certificate, if it exists. |
| `cert.subject.id.st` | The `st` attribute of the subject of the certificate, if it exists. |
| `cert.version` | The certificate version. |

# Configuration

Configure the following fields:

**Name**:
Enter a name for the filter to display in a policy.

**Certificate Attribute**:
The **Extract Certificate Attributes** filter extracts the attributes from the certificate contained in the message attribute selected or entered here. The selected attribute must contain a single certificate only.

**Include Distribution Points**:
If the certificate contains CRL Distribution Point X.509 extension attributes (which point to the location of the certificate issuer's CRL), you can also extract these and store them in message attributes by selecting this check box. The extracted distribution points are stored in message attributes with a `distributionpoint.` prefix.

# 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 several different locations. For example, this includes the certificate store, user store, selector expressions, HTTP headers, and message attachments.

By default, API Gateway 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.

## Configuration

Complete the following fields:

**Name**:
Enter an appropriate name for the filter to display in a policy.

**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. Defaults to `certificate`.

**Certificate Store**:
Click **Select**, and select a certificate from the certificate store.

**User**:
This field provides an alternative way to specify the user certificate. You can enter an explicitly named user in the API Gateway user store, or you can enter a selector expression that evaluates to a string that contains the name of a user in the user store. For example:

```
${authentication.subject.id}
```

**Tip** You can associate a certificate with this user using the **Signing Key** option in the user configuration. For more details, see "Manage API Gateway users" in the *API Gateway Policy Developer Guide*. In this way, the specified user name is used to retrieve the certificate associated with that user.

**Selector Expression**:
You can specify a selector expression by enclosing the message attribute name that contains the certificate in curly brackets, and prefixing this with `$` (for example, `${certificate}`). Using a selector is a more flexible way of locating certificates than specifying the user directly. For more details on selector expressions, see "Select configuration values at runtime" in the *API Gateway Policy Developer Guide*.

**Note** This selector expression must return an X.509 certificate. Selectors used in other fields return a string that is then used to look up the certificate (for example, in the certificate store, user store, HTTP header, or attachment).

**HTTP Header Name**:
Enter the name of the HTTP header that contains the certificate. Alternatively, you can enter a selector expression that evaluates to a string that contains the HTTP header name, and whose value is the certificate.

**Attachment Name**:
Enter the name of the attachment (`Content-Id`) that contains the certificate. Alternatively, you can enter a selector expression that evaluates to a string that contains the ID of the attachment that encapsulates the certificate.

**Certificate Alias**:
Enter the alias name of the certificate. Alternatively, you can enter a selector expression that evaluates to a string that contains the certificate alias in the certificate store.

**Tip** This certificate alias refers to a certificate in the API Gateway's trusted certificate store, and not to a certificate in the Java keystore. For more details, see "Manage X.509 certificates and keys" in the *API Gateway Policy Developer Guide*.

# OCSP client

You can use the Online Certificate Status Protocol (OCSP) to retrieve the revocation status of a certificate, as an alternative to retrieving Certificate Revocation Lists (CRLs).

The **OCSP Client** filter enables you to retrieve certificate revocation status from an OSCP responder, such as Axway Validation Authority. The input to this filter is the certificate to be checked. You must specify the message attribute that contains the certificate (`java.security.cert.X509Certificate`).

This filter returns the following outputs:

- `True` if the certificate status is `GOOD`
- `False` if the certificate status is `REVOKED` or `UNKNOWN` (or if an exception occurs)

This filter also outputs the following message attributes:

- `ocsp.response.certificate.status`: The status of the OCSP responder certificate as an `java.lang.Integer`. The possible values are:
  - `0` (`GOOD`)
  - `1` (`REVOKED`)
  - `2` (`UNKNOWN`)

  You can use this attribute if the filter return value is not detailed enough.
- `ocsp.response.signing.certificate`: The optional certificate included in the OCSP response (`java.security.cert.X509Certificate`) used to sign the response. You can use an additional OCSP filter to verify the status of this certificate.

# General settings

Configure the following general settings on the **OCSP Client** dialog:

**Name**:

Enter a suitable name for this OCSP client filter to display in a policy.

**OCSP Responder URL**:

Enter the URL of the OCSP responder.

# Message settings

Configure the following OCSP message settings on the **Settings** tab:

**The message attribute storing the certificate to validate**:

Enter the name of the attribute that contains the certificate to be checked (`java.security.cert.X509Certificate`). The default is `${certificate}`.

**The key to sign the request**:

Click the **Signing Key** button to open the list of certificates in the certificate store. You can then select the key to use to sign requests to the OCSP responder.

You can select a specific certificate from the certificate store in the dialog, or click **Create/Import** to create or import a certificate. Alternatively, you can specify a certificate to bind to at runtime using an environment variable selector (for example, `${env.serverCertificate}`). For more details on selectors, see "Select configuration values at runtime" in the *API Gateway Policy Developer Guide*.

**Validate response**:

Select the **Do not validate response** option to disable response validation. The response from the OCSP responder is not validated when this option is selected.

Select the **Validate response** option to enable response validation. Click one or more of the following options to specify how the response from the OCSP responder is validated:

- **Against the certificate contained in the response**:

  The response is validated against the certificate contained in the response. This option is selected by default.

- **Against the CA certificate of the certificate being validated**:

  The response is validated against the CA certificate of the certificate being validated. This option is selected by default.

- **Against the specified certificate**:

  Click **Signing Key** to choose a certificate from the certificate store or to specify a certificate to bind to at runtime.

You can select any combination of these options. If multiple options are selected, the filter continues as soon as the response is successfully validated against one of the selected options.

When the **Validate response** option is selected, you can also use the following time validation options:

- **Allowable time difference between this system and update time in the response**:

  Enter a value in milliseconds. You can use this field to allow for drift on server and client machines. The default value is 3000 (3 seconds).

- **The response is valid until**:

  Select the **Expiration date** check box to use the `nextUpdate` field in the OCSP response as the expiration date. This is the default.

  Alternatively, to override the expiration date in the response (for example, to set a longer expiration), deselect the **Expiration date** check box, enter a value in the text box, and select a time unit (days, hours, minutes, or seconds).

- **A response with no expiration date is valid for**:

  Enter a value in the text box and select a time unit (days, hours, minutes, or seconds). The default value is 6 hours.

  For more information on the time validation logic, see *Time validation logic* on page 181.

**Use nonce to prevent reply attack**:

Select this option to include a nonce in the request. This is a randomly generated number that is added to the message to help prevent reply attacks.

**Store results of certificate status in**:

Click the browse button to select the cache in which to store the certificate status result. The list of currently configured caches is displayed in the tree. To add a cache, right-click the **Caches** tree node, and select **Add Local Cache** or **Add Distributed Cache**. Alternatively, you can configure caches under the **Environment Configuration** > **Libraries** node in the Policy Studio tree. For more details, see the *API Gateway Policy Developer Guide*.

Storing the certificate status in the cache enables the certificate status to be retrieved without having to return to the OCSP responder.

## *Time validation logic*

When you enable response validation, the following time validation logic is applied in the following order:

1. The `thisUpdate` field in the response is checked against the value you entered in **Allowable time difference between this system and update time in the response**. The response is valid if the current time on this system does not differ from the `thisUpdate` field in the response by more than the value entered (or the default value of 3000 milliseconds). For example:

   ```
   thisUpdate + allowable time difference < current time on this system
   ```

2. If you selected the **Expiration date** check box in the **The response is valid until** field, the response is checked against the `nextUpdate` field in the response. The response is considered *valid until* the `nextUpdate` time in the response. For example:

   ```
   thisUpdate + allowable time difference < current time on this system <
   nextUpdate
   ```

3. If you did not select the **Expiration date** check box in the **The response is valid until** field, the response is checked against the expiration value you entered in the text box. The response is considered valid if the current time on this system is less than the sum of the `thisUpdate` time in the response and the value you entered. For example:

   ```
   thisUpdate + allowable time difference < current time on this system <
   thisUpdate + valid until
   ```

   **Note** The **The response is valid until** settings are only applicable when the `nextUpdate` field is present in the response and the **Expiration date** check box is not selected.

4. If no expiration date is available (that is, if the `nextUpdate` field is not present in the response), the response is checked against the value you entered in **A response with no expiration date is valid for**. The response is considered valid if the current time on this system is less than the sum of the `thisUpdate` time in the response and the value you entered (or the default value of 6 hours). For example:

   ```
   thisUpdate + allowable time difference < current time on this system <
   thisUpdate + valid for
   ```

   **Note** The **A response with no expiration date is valid for** settings are only applicable when the `nextUpdate` field is not present in the response.

# Routing settings

You can configure the settings for routing the OCSP request to the OCSP responder on the **Routing** tab.

You can configure SSL settings, credential profiles for authentication, and other settings for the connection using the **SSL**, **Authentication**, and **Settings** tabs. For more details, see *Connect to URL* on page 380.

# Advanced settings

On the **Advanced** tab, you can enable a specific policy to run after the message is created, or after the response is received.

Configure the following advanced settings:

**Run this policy after the message has been created**:

Click the browse button to select a policy to be run after the message has been created.

**Run this policy after a response has been received**:

Click the browse button to select a policy to be run after a response has been received.

**Record outbound transactions**:

Select this option to enable recording of outbound transactions on the **Traffic** tab in API Gateway Manager. This field is not selected by default. For more details, see the *API Gateway Administrator Guide*.

# Integration with Axway Validation Authority

When using the **OCSP client** with Axway Validation Authority (VA) as an OCSP responder, you can use the following trust models:

- **Direct trust**

  In this model, OCSP responses are signed with the OCSP signing certificate of the VA server. The signing certificate is not included in the OCSP response.

- **VA delegated trust**

  In this model, the signing certificate is included in the OCSP response. API Gateway might not have this certificate. If not, it must have the issuer (CA) certificate of the signing certificate.

You can import certificates into the API Gateway trusted certificate store under the **Environment Configuration** > **Certificates and Keys** node in the Policy Studio tree. For more details, see "Manage X.509 certificates and keys" in the *API Gateway Policy Developer Guide*.

For more information on Axway Validation Authority, see the Axway Validation Authority user documentation.

# Validate certificate store

## Overview

The **Validate Server's Certificate Store** filter checks API Gateway's certificate store for certificates that are due to expire before a specified number of days. This enables you to monitor the certificates that API Gateway is running with.

For example, you can configure a policy that includes a **Validate Server's Certificate Store** filter and an **Alert** filter, which sends an email alert when it finds certificates that are due to expire. You can also configure this policy to run at regular intervals using the policy execution scheduler provided with API Gateway.

## Configuration

Configure the following fields:

**Name**:
Enter an appropriate name for the filter to display in a policy.

**Days before expires**:
Enter the number of days before the certificates are due to expire.

**Check Server's Certificate Store**:
Select whether to check the certificates in API Gateway's certificate store. This is selected by default.

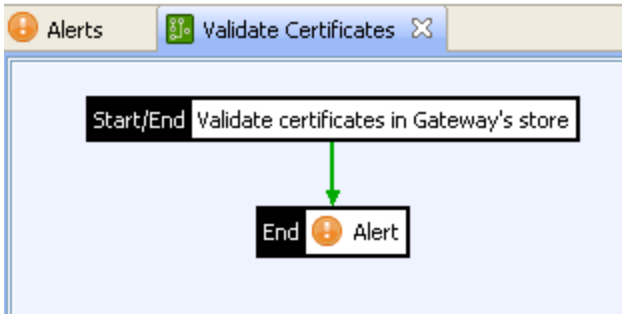
**Check Server's Java Keystore**:
Select whether to check the certificates in API Gateway's Java Keystore. This is not selected by default. When selected, you must enter the password for this keystore.

**Check Java Keystore**:
Select whether to check the certificates in the specified Java Keystore. This is not selected by default. When selected, you must configure the following fields:

- **Keystore Location**: Specify the path to this keystore (for example, `/home/oracle/osr-client.jks`).
- **Password**: Enter the password for this keystore.

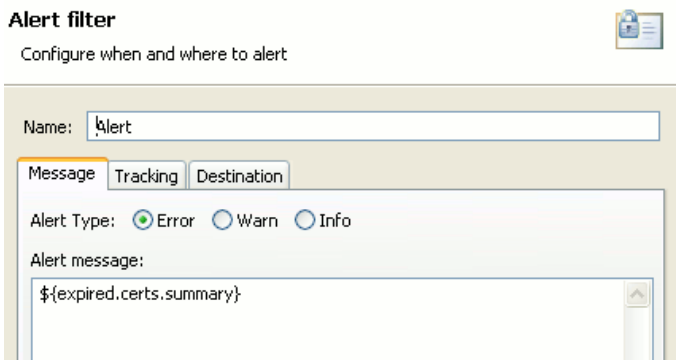
## Deployment example

The following example shows a **Validate Certificates** policy that includes a **Validate Server's Certificate Store** filter and an **Alert** filter. This policy sends an email alert when it finds certificates that are due to expire:




## Configure an email alert

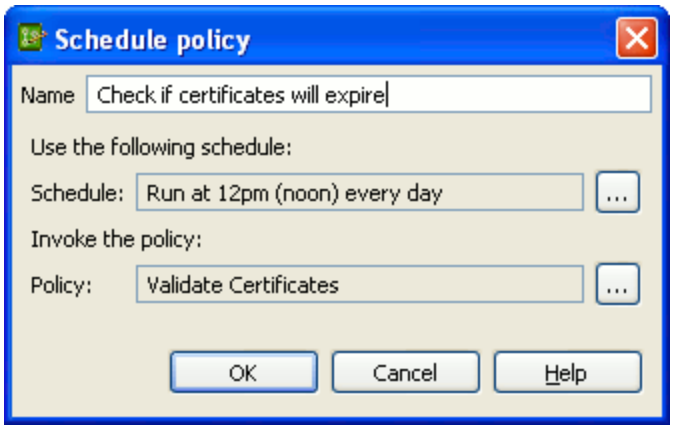
When this filter is successful, and finds certificates that are due to expire, it generates an `expired.certs.summary` attribute, which contains a summary of certificates due to expire. You can then use this attribute in the **Alert** filter to send an email alert to the API Gateway administrators, as shown in the following example:




You must also select a preconfigured email alert destination on the **Destination** tab (for example, **Email API Gateway Administrators**). For more details on configuring email alert destinations, see the *API Gateway Policy Developer Guide*.

## Configure a policy execution schedule

You can configure this policy to run at regular intervals (for example, once every day) using the policy scheduler provided with API Gateway. Under the **Environment Configuration** > **Listeners** node, right-click the API Gateway instance node, and select **Add policy execution scheduler**. The following example runs the policy at 12 noon every day:




For more details on policy execution scheduling, see the *API Gateway Policy Developer Guide*.

## *Example email alert*

An email alert is sent if any certificates that are due to expire are detected. The contents of the email are obtained from the `expired.certs.summary` message attribute. For example:

```
Axway API Gateway running on Roadrunner contains certificates that will expire in
730 days.
2 expired certificates in API Gateway certificate store:
1. Cert details:
Cert issued to: CN=CA
Cert issued by: CN=CA
SHA1 fingerprint: 72:04:35:7C:A1:B1:C2:F5:E2:86:75:C4:83:12:9C:70:A8:D6:21:8E
MD5 fingerprint: 82:23:6F:59:F2:8F:C3:95:56:87:70:B5:51:3F:53:05
Subject Key Identifier (SKI): dfABenFoM0r7iJ3E1ZqU7HmKiyY=
Expires on: 2012-04-20
2. Cert details:
Cert issued to: CN=John Doe
Cert issued by: CN=CA
SHA1 fingerprint: 83:32:EB:3F:9C:15:87:FB:81:E1:D5:AC:CC:35:C3:F8:21:BB:DF:CD
MD5 fingerprint: 48:02:F6:3F:B9:64:EB:DA:DF:CF:F9:82:AC:CC:13:AB
Subject Key Identifier (SKI): HabJNMjAsBAWp4AcCq8yZkTEJKQ=
Expires on: 2012-04-20
```

# XKMS certificate validation

## Overview

XML Key Management Specification (XKMS) is an XML-based protocol that enables you to establish the trustworthiness of a certificate over the Internet. API Gateway can query an XKMS responder to determine whether a given certificate can be trusted.

## Configuration

Configure the following fields:

**Name**:
Enter an appropriate name for this XKMS filter to display in a policy.

**XKMS Connection**:
Click the button on the right, and select an XKMS connection in the tree. To add an XKMS connection, right-click the **XKMS Connections** node, and select **Add an XKMS Connection**. Alternatively, you can configure an XKMS connection under the **Environment Configuration** > **External Connections** node in the Policy Studio tree. For more details, see the *API Gateway Policy Developer Guide*.

# Cache filters 7

The following filters enable you to work with caches. For more information on caching, see "Global caches" in the *API Gateway Policy Developer Guide*.



# Cache attribute

## Overview

The **Cache Attribute** filter allows you to configure what part of the message to cache. Typically, response messages are cached and so this filter is usually configured *after* the routing filters in a policy. In this case, the `content.body` attribute stores the response message body from the web service and so this message attribute should be selected in the **Attribute Name to Store** field.

For more information on how to configure this filter in a caching policy, see "Global caches" in the *API Gateway Policy Developer Guide*.

## Configuration

**Name**:
Enter a name for this filter to display in a policy.

**Select Cache to Use**:
Click the button on the right, and select the cache to store the attribute value. The list of currently configured caches is displayed in the tree. To add a cache, right-click the **Caches** tree node, and select **Add Local Cache** or **Add Distributed Cache**. Alternatively, you can configure caches under the **Environment Configuration** > **Libraries** node in the Policy Studio tree. For more details, see "Global caches" in the *API Gateway Policy Developer Guide*.

**Attribute Key**:
The value of the message attribute entered here acts as the key into the cache. In the context of a caching policy, it *must* be the same as the attribute specified in the **Attribute containing key** field on the **Is Cached?** filter.

**Attribute Name to Store**:
The value of the API Gateway message attribute entered here is cached in the cache specified in the **Cache to use** field above.

# Create key

## Overview

The **Create Key** filter is used to identify the part of the message that determines whether a message is unique. For example, you can use the request message body to determine uniqueness so that if two successive identical message bodies are received by the API Gateway, the response for the second request is taken from the cache.

You can also use other parts of the request to determine uniqueness (for example, HTTP headers, client IP address, client SSL certificate, and so on). This means that you can use the **Create Key** filter to create keys for a range of different caching scenarios (for example, caching a user's role, or caching a session for a user).

For more information on how to configure this filter in the context of a caching policy, see "Global caches" in the *API Gateway Policy Developer Guide*. This shows the order in which caching filters such as the **Create Key** filter are placed in an example caching policy.

## Configuration

**Name**:
Enter a suitable name for this filter.

**Attribute Name**:
Select or enter the name of the message attribute to use to determine whether an incoming request is unique or not. For example, if `http.request.clientcert` (the client SSL certificate) is selected, the API Gateway takes a cached response for successive requests in which the client SSL certificate is the same. Defaults to `content.body`.

**Output attribute name**:
Select or enter the name of the output message attribute to be used as the key for objects in the cache. Defaults to `message.key`. This attribute contains a hash of the request message, which can then be used as the key for objects in the cache.

# Check if attribute is cached

## Overview

The **Is Cached?** filter looks up a named cache to see if a specified message attribute has already been cached. A message attribute (usually `message.key`) is used as the key to search for in the cache. If the lookup succeeds, the retrieved value overrides a specified message attribute, which is usually the `content.body` attribute.

For example, if a response message for a particular request has already been cached, the response message overrides the request message body so that it can be returned to the client using the **Reflect** filter.

For more information on how to configure this filter in the context of a caching policy, see "Global caches" in the *API Gateway Policy Developer Guide*.

## Configuration

**Name**:
Enter a suitable name for this filter to display in a policy.

**Select Cache to Use**:
Click the button on the right, and select the cache to lookup to find the attribute specified in the **Attribute containing key** field below. The list of currently configured caches is displayed in the tree. To add a cache, right-click the **Caches** tree node, and select **Add Local Cache** or **Add Distributed Cache**. Alternatively, you can configure caches under the **Environment Configuration** > **Libraries** node in the Policy Studio tree. For more details, see "Global caches" in the *API Gateway Policy Developer Guide*.

**Attribute containing key**:
The message attribute entered here is used as the key to lookup in the cache. In the context of a caching policy, the attribute entered here *must* be the same as the attribute specified in the **Attribute key** field on the **Cache Attribute** filter.

**Overwrite Attribute Name if Found**:
Usually the `content.body` is selected here so that value retrieved from the cache (which is usually a response message) overrides the request `content.body` with the cached response, which can then be returned to the client using the **Reflect** filter.

# Remove cached attribute

## Overview

The **Remove Cached Attribute** filter allows you to delete a message attribute value that has been stored in a cache. Each cache is essentially a map of name-value pairs, where each value is keyed on a particular message attribute. For example, you can store a cache of request messages according to their message ID. In this case the message's `id` attribute is the key into the cache, which stores the value of the request message's `content.body` message attribute.

In this example, the **Remove Cached Attribute** filter can be used to remove a particular entry from the cache based on the runtime value of a particular message attribute. By specifying the `id` message attribute to remove, the API Gateway looks up the cache based on the value of the `id` message attribute. When it finds a matching message ID in the cache, it removes the corresponding entry from the cache.

The example described above might be useful in cases where a request message needs to be cached and stored until the request is fully processed and a response returned to the client. For example, if the request must be routed on to a back-end web service, but that web service is temporarily unavailable, you can configure the policy to resend the cached request instead of forcing the client to retry.

For more information on how to configure a caching policy, see "Global caches" in the *API Gateway Policy Developer Guide*.

## Configuration

**Name**:
Enter a name for this filter to display in a policy.

**Select Cache to Use**:
Click the button on the right, and select the cache that contains the cached values that have been keyed according to the message attribute specified below. The list of currently configured caches is displayed in the tree. To add a cache, right-click the **Caches** tree node, and select **Add Local Cache** or **Add Distributed Cache**. Alternatively, you can configure caches under the **Environment Configuration** > **Libraries** node in the Policy Studio tree. For more details, see "Global caches" in the *API Gateway Policy Developer Guide*.

**Attribute Key**:
Enter the message attribute that is used as the key into the cache in this field. At runtime, the API Gateway populates the value of this message attribute, which is then used to lookup the cache selected in the table above. If a match is found in the cache, the corresponding entry is deleted from the cache.

# Content filtering filters 8

The following filters enable you to filter messages based on their content:



# Scan with ClamAV anti-virus

## Overview

You can use the **ClamAV Anti-Virus** filter to check messages for viruses by connecting to a ClamAV daemon running on network. The ClamAV daemon inspects the message and if the daemon finds a virus, it returns a corresponding response to the API Gateway, which can then block the message, if necessary.

## Configuration

Complete the following fields to configure the **ClamAV Anti-Virus** filter:

**Name**:
Enter an appropriate name for this filter to display in a policy.

**ClamAV Daemon Host**:
Enter the host name of the machine on which the ClamAV daemon is running.

**ClamAV Daemon Port Number**:
Enter the port on which the ClamAV daemon is listening.

# 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 message using Multipurpose Internet Mail Extensions (MIME). In this way, both XML and non-XML data, including binary data, can be encapsulated in a SOAP message.

API Gateway can accept or block multipart messages with certain MIME 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** window lists the content types that are allowed or denied by this filter.

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

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

**Deny Content Types**:
To *block* multipart messages containing most content types, but 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. To block all multipart messages, you do not need to select any of the MIME types in the list.
2. To allow messages with parts of a certain MIME type, select the check box next to those types. Multipart messages with parts of the MIME types selected here are allowed. All other MIME types are denied.

MIME types can be added by clicking the **MIME Registered Types** button.

## Configure MIME types

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

Click the **Add** button to add a new MIME 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 Type** dialog.

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

# Content validation

## Overview

The API Gateway 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 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 in a **Content Validation** filter:


- *Manual XPath configuration* on page 193
- *XPath wizard* on page 194


## Manual XPath configuration

To manually configure an XPath expression:

1. On the Content Validation dialog, click the **Add** button next to the **XPath Expression** field. Alternatively, you can select a previously configured XPath expression from the list.
2. In the XPath Expression dialog, enter a name for the expression in the **Name** field, and enter the expression in the **XPath Expression** field.
3. To resolve any prefixes within the XPath expression, enter the namespace mappings in the table.

For some example XPath expressions, see "Configure XPath expressions" in the *API Gateway Policy Developer Guide*.

## XPath wizard

The XPath wizard assists you with creating valid XPath expressions. For more information, see "Configure XPath expressions" in the *API Gateway Policy Developer Guide*.

# Send to ICAP

## Overview

You can use an **ICAP** filter to send a message to a preconfigured ICAP server for content adaptation. For example, this includes specific operations such as virus scanning, content filtering, ad insertion, and language translation.

## Configuration

Configure the following settings:

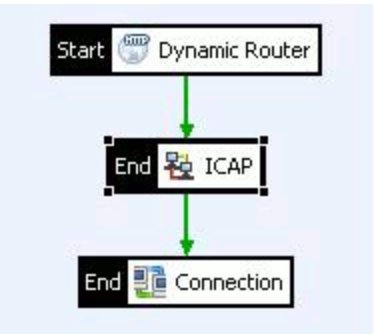
**Name**:
Enter an appropriate name for the filter to display in a policy.

**ICAP Server**:
Click the button next to this field, and select a preconfigured ICAP server in the tree. To add an ICAP server, right-click the **ICAP Servers** tree node, and select **Add an ICAP Server**. Alternatively, you can configure ICAP servers under the **Environment Configuration** > **External Connections** node in the Policy Studio tree. For more details on configuring ICAP servers, see the *API Gateway Policy Developer Guide*.

## Example policies

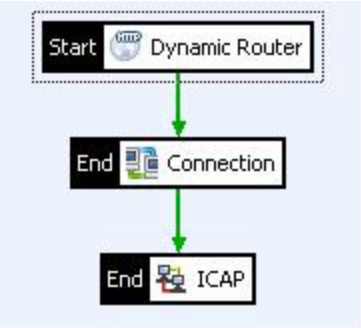
This section shows some example use cases of the **ICAP** filter configured in policies.

**Request Modification Mode**
The following policy shows an ICAP filter used in Request Modification (REQMOD) mode:



This example policy is essentially an internet proxy but with all incoming messages being sent to an ICAP server for virus-checking before being sent to the destination. All ICAP server-bound messages in this instance are REQMOD requests.

**Response Modification Mode**
The following policy illustrates an ICAP Filter used in Response Modification (RESPMOD) mode:




This example policy also is an internet proxy but with all responses being sent to an ICAP server for virus-checking after being sent to the destination and before being sent back to the client. All ICAP server-bound messages in this instance are RESPMOD requests.

# JSON schema validation

The API Gateway can check that JavaScript Object Notation (JSON) messages conform to the format expected by a web service by validating requests against a specified JSON schema. A JSON schema precisely defines the items that constitute an instance of an incoming JSON message. It also specifies their data types to ensure that only appropriate data is allowed through to the web service.

For example, the following simple JSON schema requires that all requests sent to a particular web service use this format:

```
{
  "description":"A geographical coordinate",
  "type":"object",
  "properties":{
    "latitude":{"type":"number"},
    "longitude":{"type":"number"}
  }
}
```

If a **JSON Schema Validation** filter is configured with this JSON schema, and the API Gateway receives an incorrectly formed message, the API Gateway rejects that message.

For example, the following message would pass because it specifies the coordinates correctly as numbers:

```
{
```

```
  "latitude":55.22,
  "longitude":117.22
}
```

However, the following message would fail because it specifies the coordinates incorrectly as strings:

```
{
  "latitude":"55.22",
  "longitude":"117.22"
}
```

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

API Gateway supports draft version 3 and draft version 4 of the JSON Schema specification. For more details on JSON schemas, see http://www.json-schema.org.

# Configuration

Configure the following settings:

**Name**:
Enter an appropriate name for this filter to display in a policy.

**Select which JSON schema to validate messages with**:
Select one of the following options:

- **Use JSON schema file**

  Click the button on the right, and select a JSON schema to validate incoming messages from the tree in the dialog. To add a schema, right-click the **JSON Schemas** node, and select **Add Schema** to load the schema from a `.json` file. Alternatively, you can configure schemas under the **Resources** > **JSON Schemas** node in the Policy Studio tree. By default, the API Gateway provides the example JSON schemas available from http://www.json-schema.org.

  Select the **Use v4 draft validator** to validate against draft version 4 of the JSON Schema specification, or select **Use v3 draft validator** to validate against draft version 3 of the JSON Schema specification. The default is v4.

- **Use this class**

  Enter the Java class name used to specify the JSON schema (for example, `com.vordel.samples.GeoLocationTest`), and enter the name of message attribute to store the created object (for example, `my.geo.location`).This option enables you to take incoming JSON message data and deserialize it into a Java object.

### *Example using a Java class*

For example, to use a Java class for the geographical schema used in the previous section, perform the following steps:

1. Create the annotated Java class as follows:

```
package com.vordel.samples;
import java.lang.String;
import javax.xml.bind.annotation.XmlAttribute;
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement
public class GeoLocationTest {
    public GeoLocationTest() { }
    public int latitude;
    public int longitude;
}
```

2. Place this class in a JAR file, and put it in the API Gateway `ext/lib` directory.
3. In the **JSON Schema Validation** filter window, enter `com.vordel.samples.GeoLocationTest` in the **Use this class** field.
4. Enter `my.geo.location` in the **Store created object in message attribute** field.

At runtime when the JSON message arrives, the API Gateway takes the JSON data and tries to instantiate a new `GeoLocationTest` object. If this succeeds, the **JSON Schema Validation** filter passes. However, if the object cannot be instantiated, the filter fails because the incoming data does not conform to the JSON schema specified by the annotated class.

## Generate a JSON schema using Jython

The API Gateway also provides a Jython script to enable you to generate a JSON schema based on a specified Java annotated class. This script is available in the following directory of your API Gateway installation:

```
INSTALL_DIR/apigateway/samples/scripts/json/schemagenerator.py
```

Given an annotated Java `.class` file, you can generate a schema from this and output a `.json` schema file. This schema can then be imported into the API Gateway schema library and used subsequently for future validations.

For example, using the Java class from the previous section, you can generate the schema as follows:

```
sh run.sh json/schemagenerator.py com.vordel.samples.GeoLocationTest
MyLocationSchema.json
```

This script requires that you specify the location of your JAR file. You can do this by setting the `JYTHONPATH` environment variable, for example:

```
export JYTHONPATH=/home/user/mylocation.jar
```

Alternatively, if you have compiled your classes to `/home/user/classes`, specify the following:

```
export JYTHONPATH=/home/user/classes
```

For more details on using Jython, see http://www.jython.org/.

## Exceptions

The **JSON Schema Validation** filter aborts with a `CircuitAbortException` if:

- Content body of the payload is not in valid JSON format
- **Use this class** option is selected to validate schema and the specified Java class is not found
- **Use JSON schema file** option is selected and the payload is not in valid JSON format

# Scan with McAfee anti-virus

The **McAfee Anti-Virus** filter scans incoming HTTP requests and their attachments for viruses and exploits. For example, if a virus is detected in a MIME attachment or in the XML message body, the API Gateway can reject the entire message and return a SOAP Fault to the client. In addition, this filter supports cleaning of messages from infections such as viruses and exploits. It also provides scan type presets for different detection levels, and reports overall message status after scanning.

## Prerequisites

McAfee virus scanner integration requires the McAfee Scan Engine and a valid McAfee license. The required third-party binaries are included in your API Gateway installation.

## Configuration

To configure the **McAfee Anti-Virus** filter, perform the following steps:

1. Enter a name that indicates the role of this filter in the **Name** field.
2. Select a **Scan type** from the list. The available options are as follows:
    - **Custom**

      Enables you to set the **Custom options** described in *Custom options* on page 199. This provides backwards compatibility with earlier API Gateway versions.

- **Normal**

  Processes the entire message detecting exploits and viruses in the message headers, macros, multi-file archives, executables, MIME-encoded/UU-encoded/XX-encoded/BinHex and TNEF/IMC format files. Performs heuristic analysis to find new viruses and potentially unwanted programs. This is the default scan type. For more information, see *Custom options for normal or fast scans* on page 200.

- **Fast**

  Detects infections in the top level of each message part, such as exploits that use headers and multiple bodies. The detection is less precise, but the performance is better if the top-level object is infected. For more information, see *Custom options for normal or fast scans* on page 200.

- **Multi-pass**

  Combines the **Normal** and **Fast** scan types. The **Fast** scan (pass 1) runs first on the whole message with no cleaning. The scanner stops if it finds an infected object, and if the clean type is set to **No cleaning**, the scanner reports the infection, or otherwise deletes the message. If pass 1 does not detect any virus or exploit, the **Normal** scan (pass 2) runs with the specified clean type and provides more precise detection.

3. Select a **Clean type** from the list. The available options are as follows:

   - **No cleaning**

     Fails if any infection is detected. This is the default clean type.

   - **Always remove infected parts**

     Removes the infected message part, and does not try to repair it.

   - **Attempt to repair infected parts**

     Attempts to repair the found infection (if repairable), otherwise deletes the infected message part.

   **Note** When existing policies are upgraded to the current API Gateway version, the **McAfee Anti-Virus** filter scan type is set to **Custom** and the clean type is set to **No cleaning** for backwards compatibility.

## *Custom options*

When you configure a custom scan type, the following **Custom options** are available:

**Decompress Archives**:

This instructs the filter to scan each file in an archive for viruses. Types of archived files include the ZIP, JAR, TAR, ARJ, LHA, PKARC, PKZIP, RAR, WinACE, BZip, and Zcompress formats.

**Decompress Executables**:

Executables are sometimes compressed to decrease overall message size. In such cases, any embedded viruses are also compressed and may be missed by conventional scans. If this option is selected, the filter decompresses the executable before scanning it for viruses.

**Fail Any Macros**:

A *macro* is a series of commands that can be invoked in a single command or keystroke. While calling the macro can appear to be harmless, the initiated command sequence may be harmful. Macros are usually configured to run automatically when the host document is opened. When this option is selected, the API Gateway fails if any macro is detected in a compound document (whether it matches a virus signature or not). An appropriate SOAP Fault is returned to the client.

**Heuristic Program Analysis**:

A heuristic virus detection algorithm runs a series of probing tests on a file in an attempt to solicit virus-like behavior from it. Based on the results of these tests, the algorithm can then make an educated guess on whether the file represents a potential threat or not. For example, programs that attempt to modify or delete files, invoke email clients, or replicate themselves all display virus-like behavior and so may be treated as viruses by the scanner.

The major advantage of this type of analysis is that new viruses can be detected. With the signature detection method, the scanner attempts to find a fixed number of known virus signatures in a file. Because the number of known signatures is fixed, new or unknown viruses can not be detected. If this option is selected, the filter runs heuristic analysis on executables only.

**Heuristic Macro Analysis**:

When this option is enabled, the filter runs heuristic detection analysis on macros contained in any body parts of the message. If any viruses are detected, the message is blocked. If this option is selected, the API Gateway searches for virus signatures in the respective body parts of a MIME message. However, it can only search for *known* viruses using this method.

**Note** Macros embedded in MIME parts are also scanned for virus signatures.

**Scan Embedded Scripts**:

The API Gateway can scan MIME parts, such as HTML documents, for embedded scripts. If this option is selected, the filter scans for embedded scripts.

**Scan for Test Files**:

When this option is selected, the API Gateway fails if it encounters an anti-virus test file (for example, `eicar.com`). This is a convenient way to check that the anti-virus filter successfully detects known viruses.

## *Custom options for normal or fast scans*

The custom options are disabled when you select the normal or fast scan types. The custom options used for these options (where applicable) are as follows:

| | Normal scan type | Fast scan type |
|---|---|---|
| Decompress Archives | On | Off |
| Decompress Executables | On | N/A (instead, ignore certain compressed executables) |

| | Normal scan type | Fast scan type |
|---|---|---|
| Fail Any Macros | N/A (instead, all macros are scanned) | N/A (instead, ignore macros in compound documents) |
| Heuristic Program Analysis | On | Off |
| Heuristic Macro Analysis | On | Off |
| Scan Embedded Scripts | On | Off |
| Scan for Test Files | Off | Off |

# Message status

When the scan is complete, the **McAfee Anti-Virus** filter reports the overall message status in the `mcafee.status` message attribute, which is generated by the filter. This reflects the overall status of the scan for all message parts, and includes one of the following values:

| | |
|---|---|
| `NOVIRUS` | No virus or exploit detected in the message. |
| `INFECTED` | Infection detected in the message. |
| `REPAIRED` | Message repaired. |
| `REMOVED` | Some or all message parts successfully removed. |
| `REPAIRED, REMOVED` | Some message parts successfully repaired and some others removed. |

# Load McAfee updates

When the **McAfee Anti-Virus** filter has been loaded, it searches for virus definitions in the following directory:

```
INSTALL_DIR/apigateway/conf/plugin/mcafee/datv2
```

When these have been loaded, it periodically checks for the presence of a directory named as follows:

```
INSTALL_DIR/apigateway/conf/plugin/mcafee/datv2.new
```

If the `datv2.new` directory is found, the scanner is stopped and the `datv2` directory is renamed to `datv2.0`. If a `datv2.0` directory already exists from a previous rollover, a `datv2.1` directory is created instead, and so on, until the limit of 50 is reached (`datv2.50`). When the limit of 50 is reached you must remove the old files. This means that the server never deletes the old files, and rolls them out of the way.

When the engine is stopped and restarted, any messages that require scanning are suspended until the restart completes. In addition, an initiated reload is suspended until all currently active scans are completed.

Like all file system scanning approaches, there is an inherent ordering problem. If you create the `datv2.new` directory before copying the files into the directory, the scanner may pick up the new directory before it is ready to be used. For example, you might experience problems if you enter the following commands from the `INSTALL_DIR/apigateway/conf/plugin/mcafee` directory:

```
mkdir datv2.new
cp /var/tmp/mcafee/newfiles/*.* datv2.new
```

You can use the following commands to prevent this problem:

```
mkdir datv2.tmp
cp /var/tmp/mcafee/newfiles/*.* datv2.tmp
mv datv2.tmp datv2.new
```

These create a temporary folder, copy the files into this folder, and rename the temporary folder to `datv2.new`. In this way, the scanner is guaranteed to pick up the virus definition files when it detects the new directory.

# Message size filtering

It is sometimes useful to filter incoming messages based not only on the internal content of the message but also on external characteristics of the message such as size. You can use the **Message Size** filter to configure the API Gateway to reject messages that are greater or less than a specified size.

This filter only rejects messages where the actual content size, after any truncation by the network layer, exceeds the limit configured in the filter. The network layer of API Gateway reads at most the value of the Content-Length header of any incoming request, before transmitting it to the application layer (**Message Size** filter in this case). This means that API Gateway is never vulnerable to a "size control by-pass" attack, with or without this filter activated, because the network layer truncation is always active.

For example, if you configure a **Message Size** filter with a limit of 200 bytes, and send a message with a Content-Length equal to 100 bytes, but with an actual content of 300 bytes, the filter does not reject the message. This is because API Gateway reads only 100 bytes from the message (the Content-Length) and therefore when the message arrives at the Message Size filter it does not exceed the limit of 200 bytes.

**Note** You should not use the **Message Size** filter on HTTP `GET` requests.

## Configuration

To configure the API Gateway to block messages of a certain size, complete the following fields:

**At least**:
Enter the size (in bytes) of the smallest message that should be processed. The default size is `-1`, which indicates that there is no minimum message size. You can also use a selector, for example, `${env.MIN.MESSAGE.SIZE}` to set the message size.

**At most**:
Enter the size (in bytes) of the largest message that should be processed. The default size is `-1`, which indicates that there is no maximum message size. You can also use a selector, for example, `${env.MAX.MESSAGE.SIZE}` to set the message size.

**Use in Size Calculation**:
Select one of the following options to specify the portion of the message that is to be used when calculating the size of the message:

- **Root body only**: The API Gateway calculates the size of the message body excluding all other MIME parts (attachments).
- **Attachments only**: The API Gateway only calculates the size of all attachments to the message. This excludes the size of the root body payload from its calculation. The **Message Size** filter still works even when there are no attachments.
- **Root body and attachments**: The API Gateway includes the root body together with all other MIME parts when it calculates the size of the message.

**Note** The message size measured by the API Gateway does *not* include HTTP headers.

# Schema validation

## Overview

API Gateway 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 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.

# Select the schema

To configure the XML schema to validate messages against, click the **Schema to use** tab. You can select the schema from the WSDL for the current web service, and specific schemas from the global cache of WSDL and XML schema documents. Both validation mechanisms can be selected at the same time. Configure the following options:

**Use Schema from WSDL of web service**:
Select this option to dynamically use the appropriate SOAP operation schema from the current web service context. When this option is selected, this filter has an additional required message attribute named `webservice.context`, which must be provided. This enables you to share this filter to perform validation across multiple web services.

**Select which XML Schema to validate message with**:
Select this option to use schemas from the global cache. This is the default option. Click the browse button, and select a schema from the tree view. You can select multiple schemas under the **XML Schema Document Bundles** and the **WSDL Document Bundles** nodes. Click the check box next to the schema document to select the schema. You can also select a particular version of a schema by clicking the check box next to the version.

To add a new schema, right-click the **XML Schema Document Bundles** > **User-defined Catalog** node, and select **Add Schema**. Alternatively, you can add schemas under the **Resources** node in the Policy Studio tree. For more details, see "Manage WSDL and XML schema documents" in the *API Gateway Policy Developer Guide*.

**Tip** If you have a WSDL file that contains an XML schema, you can use this schema to validate messages by importing the WSDL file into the Web service repository. The **Import WSDL** wizard automatically adds any XML schemas contained in the WSDL to the global cache under the **Resources > WSDL Document Bundles** node. For more details, see "Configure policies from WSDL files" in the *API Gateway Policy Developer Guide*.

# Select 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 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 "Configure XPath expressions" in the *API Gateway Policy Developer Guide*.

# 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 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 (for example, using the **Extract MTOM Content** filter) 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 Tester
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 XML-binary Optimized Packaging (XOP) specification.

# Report 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 *Set message* on page 264) 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:

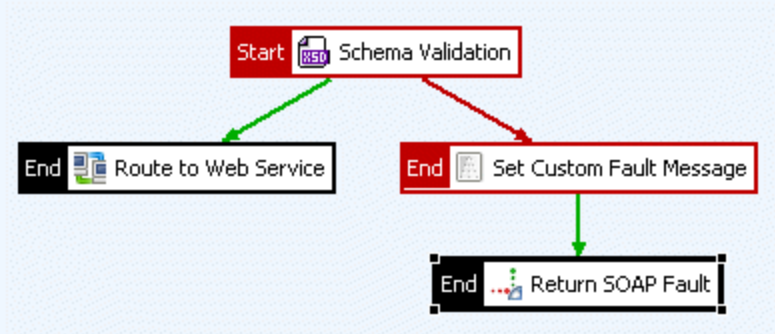
```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
<env:Body>
  <env:Fault>
    <env:Code>
      <env:Value>env:Receiver</env:Value>
        <env:Subcode>
        <env:Value xmlns:fault="http://www.axway.com/soapfaults">
          fault:MessageBlocked
        </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.axway.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. You can then return the SOAP Fault to the client using a **Reflect** filter (for more information, see *Reflect message* on page 448). 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.axway.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.axway.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.




# Scan with Sophos anti-virus

## Overview

The **Sophos Anti-Virus** filter uses the Sophos Anti-Virus Interface (SAVI) to screen messages for viruses. You can configure the behavior of the Sophos library using configuration options available in the **Sophos Anti-Virus** filter.

**Note** Because the API Gateway does not ship with any Sophos binaries, the API Gateway must be installed on the same machine as the Sophos AV distribution. On Linux, before starting the API Gateway, ensure that the Sophos AV `lib` directory is on your `LD_LIBRARY_PATH`.

## Prerequisites

Integration with Sophos requires Sophos SAV Interface version 4.8. You must add the required third-party binaries to your API Gateway and Policy Studio installations.

**Add third-party binaries to API Gateway**

To add third-party binaries to API Gateway, perform the following steps:

1. Add the binary files as follows:
    - Add `.jar` files to the `INSTALL_DIR/apigateway/ext/lib` directory.
    - Add `.so` files to the `INSTALL_DIR/apigateway/<platform>/lib` directory.
2. Restart API Gateway.

**Add third-party binaries to Policy Studio**

To add third-party binaries to Policy Studio, perform the following steps:

1. Select **Window > Preferences > Runtime Dependencies** in the Policy Studio main menu.
2. Click **Add** to select a JAR file to add to the list of dependencies.

3. Click **Apply** when finished. A copy of the JAR file is added to the `plugins` directory in your Policy Studio installation.
4. Click **OK**.
5. Restart Policy Studio with the `-clean` option. For example:

```
> cd INSTALL_DIR/policystudio/
> policystudio -clean
```

# Sophos configuration settings

All SAVI configuration options take the form of a name-value pair. Each name is unique and its corresponding value controls specific behavior in the Sophos anti-virus library (for example, decompress `.zip` files to examine their content). You can specify these SAVI configuration settings in the **Sophos configuration settings** section:

**Name**:
The **Sophos Anti-Virus** filter ships with two sets of default configuration settings: one for UNIX-based platforms, and the other for Windows platforms. Select the Linux configuration settings for your target platform from the list.

You can create a new set of configuration options by clicking the **Add** button, and adding the name-value pairs to the table provided. For convenience, you can base a new configuration set on a previously existing one, including the default Linux set. In this way, you can create a new configuration set that *inherits* from the default set, and then adds more configuration options.

To add a new configuration name-value pair, click the **Add** button under the table, and configure the following fields in the dialog:

**Name**:
Enter a name for the SAVI configuration option. This name must be available in the version of the SAVI library that is used by the API Gateway. See your SAVI documentation for a complete reference on available options.

**Value**:
Enter an appropriate value for the SAVI configuration option entered above. See your SAVI documentation for more information on acceptable values for this configuration option.

**Type**:
Select the appropriate type of this configuration option from the list. See your SAVI documentation for more information on the type of the value for this configuration option.

# Threatening content

## Overview

The **Threatening Content** filter can run a series of regular expressions that identify different attack signatures against request messages to check if they contain threatening content. Each expression identifies a particular attack signature, which can run against different parts of the request, including the request body, HTTP headers, and the request query string. In addition, you can configure the MIME types on which the **Threatening Content** filter operates.

The threatening content regular expressions are stored in the global **Black list** library, which is displayed under the **Environment Configuration** > **Libraries** node in the Policy Studio tree. By default, this library contains regular expressions to identify SQL syntax to guard against SQL injection attacks, DOCTYPE DTD references to avoid against DTD expansion attacks, Java exception stack trace information to prevent call stack information getting returned to the client, and expressions to identify other types of attack signature.

## Scanning settings

To configure the **Scanning Details** tab, complete the following:

**Additional message parts to scan**:
This section configures what parts of the incoming request are scanned for threatening content. By default, the **Threatening Content** filter acts on the request body. However, it can also scan the HTTP headers and the request query string for threatening content. Select the appropriate check boxes to indicate what additional parts of the request message to scan.

**Blacklist**:
The table lists all the regular expressions that have been added to the global **Black list** library. These regular expressions are used to identify threatening content. For example, there are regular expressions to match SQL syntax, ASCII control characters, and XML processing instructions, all of which can be used to attack a web service. For more information on how to configure global **Black list** library, see the *API Gateway Policy Developer Guide*.

Select the regular expressions to run against incoming requests using the check boxes in the table. You can add new expressions using the **Add** button. When adding new regular expressions on the **Add Regular Expression** dialog, the expressions are added to the global **Black list** library.

You can edit or remove existing regular expressions by selecting the expression in the tree, and selecting the **Edit** or **Delete** button.

## MIME type settings

The **MIME Types** tab lists the MIME types to be scanned for incoming messages. By default, all text- and XML-related types are scanned for threatening content. However, you can select any type from the list.

Similar to the way in which the **Black list** regular expressions are global, so too are the MIME types. You can add these globally by selecting the **Environment Configuration** > **Server Settings** node in the Policy Studio tree, and clicking the **General > MIME** option. For more information, see the *API Gateway Administrator Guide*.

You can add new types by selecting the **Add** button and entering a type name and corresponding extension on the **Configure MIME Type** dialog. You can enter a list of extensions by separating them with spaces. You can edit or delete existing types by selecting the **Edit** and **Delete** buttons.

## Regular expression format

This filter uses the regular expression syntax specified by `java.util.regex.Pattern`. For more details, go to:

http://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html

# Throttling

The **Throttling** filter enables you to limit the number of requests that pass through an API Gateway in a specified time period. This enables you to enforce a specified message quota or *rate limit* on a client application, and to protect a back-end service from message flooding.

You can configure this filter, for example, to allow only a specified number of messages from a specified client over a configured time period through to a virtualized API. If the number of messages exceeds the specified limit, the filter fails for the excess messages.

When the configured constraints are breached, the API Gateway behavior is determined by the filter that is next in the policy failure path from the **Throttling** filter. Typically, an **Alert**, **Trace**, or **Log** filter is configured as the successor filter in the failure path.

Some example use cases for the **Throttling** filter include:

- Protect a back-end service that can handle a maximum of only 20 messages per client per second
- Enforce a specific message rate limit or quota where a customer has purchased a maximum of 100 messages per client per hour only

**Note** API Gateway version 7.6.0 introduced the **Smooth Rate Limiting** algorithm for improved handling of high traffic levels. The existing algorithm has been renamed as **Floating Time Window**, and is still provided for backwards compatibility with previous API Gateway versions. Existing configuration from previous API Gateway versions is preserved during upgrade and executes as before.

# About the rate limit algorithm options

You can select the following options from the rate limit algorithm drop-down list:

- **Smooth Rate Limiting**: This algorithm smooths out the traffic by dividing per second limits into regular millisecond intervals. For example, a setting of 500 requests per second results in 1 request being accepted every 2 milliseconds. It provides most protection to back-end servers, and is especially suitable for back-end server throttling in elastic environments, but can also be used in on-premise deployments.

  The Smooth Rate Limiting algorithm distributes rate limits among running API Gateways evenly (round robin) or dynamically (based on past traffic) to match your load balancing strategy. It also keeps track of the number of running API Gateways and dynamically updates the limits for each API Gateway when there is a change in the number of running API Gateways.

  The smooth rate limits are stored in an Apache Cassandra database. If you want to use this algorithm, you must first configure an Apache Cassandra connection. For more details, see "Cassandra Settings" in the *API Gateway Administrator Guide*
- **Floating Time Window**: This algorithm is provided for backwards compatibility with previous API Gateway versions. It does not include any traffic smoothing, and is suitable for lower traffic levels, over longer time intervals (for example, 10 transactions per minute or 100 transactions per hour). This means that if a rate limit is set to 100 requests per minute, all 100 can arrive in the first 10 seconds, and will be served. But any requests in next 50 seconds will be rejected. Floating time window is the default algorithm. Its rate limits are stored in a local or distributed cache.

## *Smooth rate limiting algorithm settings*

The following settings are available when you select **Smooth Rate Limiting** algorithm:

**Name**:

Enter a name for this filter to display in a policy.

**Allow**:

Enter the number of messages to allow in a one second time interval.

**Strategy**:

Select a rate limiting strategy.

- **Evenly distributed**: Select this option to evenly distribute the configured rate limit across all API Gateway instances. This is the default option, and is most suitable when your API Gateway load is handled by a round-robin load balancer.

  For example, if you configure a rate limit of 100 transactions per second, and have 5 API Gateway instances, 20 transactions are allocated to each API Gateway instance. During the specified interval, 1 second, if an API Gateway instance receives more than 20 transactions, the excess transactions will be lost. In addition, if an API Gateway instance is added or removed, a recalculation of the distribution is performed.

- **Dynamically distributed**: Select this option to dynamically distribute the configured rate limit across all API Gateway instances based on the recent load experienced by each node. This is more suitable when load distribution is based on geography.

  On startup, the rate limit is distributed equally among the active API Gateways. Over time, the rate limit distribution reflects the load experienced by each API Gateway in the previous refresh interval. In addition, if the number of API Gateways has increased (scaled out) or decreased (scaled in), the rate limit for each API Gateway is recalculated accordingly.

**Burst size**:

Enter the number of requests in the configured time period to allow as a burst to handle spikes in incoming message traffic. The burst size defines how many requests that a client can make in excess of the specified rate limit at the millisecond level. Defaults to `0` requests.

For example, a configured rate limit of 500 transactions per second means 1 request per 2 milliseconds, so 2 milliseconds is the burst zone. If the burst size is 0, and 2 requests are received in the 2 millisecond zone, one request is processed, and all other requests are rejected. However, if you set the burst size to 10, the following occurs:

- First 2 ms: 11 requests received—1 is allowed as per rate limit, and another 10 are allowed per burst size. Now the burst count is full at 10 (no more space).
- Second 2 ms: 2 requests received—1 is allowed per rate limit, but the second is rejected because the burst queue is full.
- Third 2 ms: No requests received—1 is reduced from the burst queue, and it has 9 requests.
- Fourth 2 ms: 2 requests received—1 is allowed per rate limit, and 1 is added to the burst queue of 9, so now it is full.
- Fifth 2 ms: No requests received—burst queue decreased to 9, and so on.

API Gateway keeps count of how many requests in excess of the configured rate limit have been sent in a bursts queue.

**Note** The smooth rate limiting algorithm uses Apache Cassandra for rate limit storage. You must configure the Cassandra settings under the **Server Settings > Cassandra > Throttling** node in Policy Studio before using the smooth rate limiting. For more details, see "Cassandra Settings" in the *API Gateway Administrator Guide*.

## *Floating time window algorithm settings*

The following settings are available when you select **Floating Time Window** algorithm:

**Name**:

Enter a name for this filter to display in a policy.

**Allow**:

Enter the number of messages to allow in the time interval specified in the adjoining **Messages every** field. If the API Gateway receives more than the specified number of messages during the specified time interval, the filter fails. Otherwise, the filter passes.

**Messages every**:

Enter the allowed time interval. If the API Gateway receives more than the number of messages in the **Allow** field in the time interval specified, the filter fails. The time interval depends on the units selected (**seconds**, **minutes**, **hours**, **days**, or **weeks**). Defaults to **seconds**.

**Note** The specified time period starts when a message is received. When the time period is over, the message count is reset, and the counter starts again when another message is received.

**Store rate limits in**:

Select a cache for rate limit storage. You can configure a local cache or a distributed cache:

- *Local cache*: Use when you have a single API Gateway deployed. The **Throttling** filter uses a preconfigured local cache called **Local maximum messages** by default. To configure a different local cache, click the button on the right.
- *Distributed cache*: Use when you have multiple API Gateways deployed for load balancing purposes, and you want to maintain a single count of all messages processed by all API Gateway instances.

To add a new cache or modify an existing cache, see "Global caches" in the *API Gateway Policy Developer Guide*.

**Rate limit based on**:

Select this setting to configure API Gateway to keep track of request messages based on a specified key value (for example, to track rates based on client IP address). All rate limits are stored in a cache, and a key is required to look up the cache. The key can be any value that identifies the message sender. For example, this includes the authenticated subject, the client IP address, or even a combination of API method name and IP address. If you do not specify a key, the cache contains a single entry for the filter, and the associated rate limit is used each time the filter is invoked.

For example, to track rate limits based on the client IP address, use the following default setting:

```
${http.request.clientaddr.getAddress()}
```

The value entered can also be a combination of a fixed string value and an API Gateway message attribute selector. For example, use the following value to keep track of the number of times an API named `StockQuote` is requested by an authenticated subject:

```
StockQuote-${authentication.subject.id}
```

## *When do days / weeks start?*

**A day starts on the following hour**:

You must configure this field if you select `Day` from the list when configuring the **messages every** field. For example, if you select `Day`, and enter `00:00` in this field, this means that only the specified number of messages can be received in a one day period starting from midnight tonight until midnight the next day.

**A week starts on the following day**:

You must configure this field if you select `Week` from the list when configuring the **messages every** field. For example, if you select `Week` and `00:00`, and enter `Sunday` in this field, this means that the time period starts next Sunday at midnight, and lasts for one week exactly. The time period is reset on midnight of the next Sunday.

# Rate limit HTTP headers settings

The following settings apply when the **Smooth Rate Limiting** or **Floating Time Window** algorithms are selected:

**Include remaining limit in HTTP response headers**:

Specifies whether to include the following `X-Rate-Limit` headers in the HTTP response message:

| | |
|---|---|
| `X-Rate-Limit-Limit` | Rate limit ceiling for the given request (for example, `100` messages). |
| `X-Rate-Limit-Remaining` | Number of requests left for the time window (for example, `45` messages). |
| `X-Rate-Limit-Reset` | Remaining time window before the rate limit resets in UTC epoch seconds (for example, `1353517297`). |

**Begin rate limit HTTP Headers with**:

Specifies the string used to begin the name of rate limit HTTP headers. Defaults to `X-Rate-Limit-` . For example, if you specify a value of `My-Corp-Quota-` , the following HTTP headers are inserted into the response message:

- `My-Corp-Quota-Limit`
- `My-Corp-Quota-Remaining`
- `My-Corp-Quota-Reset`

# Multiple throttling filters

Using multiple **Throttling** filters depends on whether you have selected the **Smooth Rate Limiting** or **Floating Time Window** algorithm.

## *Smooth rate limiting*

When **Smooth Rate Limiting** is selected, you must use only one **Throttling** filter for each back-end to be protected. For example, if you configure two filters for the same back-end, the final rate limit is an aggregate of the limits set in both filters.

### *Floating time window*

When the **Floating Time Window** algorithm is selected, to use two or more **Throttling** filters to maintain separate message counts, you must use a different rate limit value for each filter, or use different caches for each filter.

### *Use a different rate limit per filter*

With this approach, you can use a unique **Rate limit based on** value in each **Throttling** filter. The easiest way to do this is to prepend the `${http.request.clientaddr.getAddress()}` selector value with the filter name, for example:

```
Acme Corp Quota Filter ${http.request.clientaddr.getAddress()}
```

This ensures that each filter maintains its own separate message count in the selected cache.

### *Use a unique cache per filter*

Alternatively, you can use a unique cache to store the message count of each **Throttling** filter. With this solution, you must configure a separate cache for each **Throttling** filter that you have configured throughout *all* policies running on the API Gateway.

# HTTP header validation

## Overview

The API Gateway 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** window:

- **Enter Regular Expression**:
  HTTP header values can be checked using regular expressions. You can select regular expressions from the global library (**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 library (**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 **Environment Configuration** > **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 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" in the *API Gateway Policy Developer Guide*.
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 user name 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.

**Note** 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** check box. 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="axway.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 `axway.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** check box 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.

**Note** If any regular expressions are configured in *Configure HTTP header regular expressions* on page 218, 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 prepopulated 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, go to:

http://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html

# Query string validation

## Overview

The API Gateway can check the request *query string* to ensure that only properly configured name and value pairs appear. *Regular expressions* are used to test the attribute values. This enables you to make decisions on what to do with the message (for example, if the query sting value is `X`, route to service `X`).

You can configure the following sections on the **Validate Query String** window:

- **Enter Regular Expression**:
  Query string values can be checked using regular expressions. You can select regular expressions from the global library (**White list**) or enter them manually. For example, if you know that a query string 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 library (**Black list**) or to run against all query string names and values. 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 **Environment Configuration** > **Libraries** node in the Policy Studio tree.

## Request query string

The request query string is the portion of the URL that comes after the `?` character, and contains the request parameters. It is typically used for HTTP `GET` requests in which form data is submitted as name-value pairs on the URL. This contrasts with the HTTP `POST` method where the data is submitted in the body of the request. The following example shows a request URL that contains a query string:

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

In this example, the query string is `first=john&last=smith`. Query strings consist of attribute name-value pairs, and each name-value pair is separated by the `&` character.

The **Query String Validation** filter can also operate on the form parameters submitted in an HTTP Form `POST`. Instead of encoding the request parameters in the query string, the client uses the `application/x-www-form-urlencoded` content-type, and submits the parameters in the HTTP `POST` body, for example:

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

```
Host: localhost:8095
Content-Length: 21
SOAPAction: HelloService
Content-Type: application/x-www-form-urlencoded
first=john&last=smith
```

If the API Gateway receives an HTTP request body such as this, the **Query String Validation** filter can validate the form parameters.

# Configure query string attribute regular expressions

The **Enter Regular Expression** table displays the list of configured query string names together with the white list of regular expressions that restrict their values. For this filter to run successfully, *all* required attributes must be present in the request, and *all* must have the correct value.

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

If the **Allow unspecified names** check box is selected, additional unnamed query string attributes are not filtered by the API Gateway. For example, this is useful if you are interested in filtering the content of only a small number of query string attributes but the request may contain many attributes. In such cases, you only need to filter those few attributes, and by selecting this check box, the API Gateway ignores all other query string attributes.

## *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 request query string attributes. To configure a regular expression, perform the following steps:

1. Enter the name of the query string attribute in the **Name** field.
2. Select whether this request parameter is **Optional** or **Required** using the appropriate radio button. If it is **Required**, the parameter name *must* be present in the request. If the parameter 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 query string 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).
   You can use selectors representing the values of message attributes to compare the value of the query string attribute 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 query string attribute value that you want to check.

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 query string attribute value that is run against the regular expression. The extracted substring can also be Base64 decoded if necessary. The following is an example of a URL containing a query string. The value of the `password` attribute is Base64 encoded, and must be extracted from the query string and decoded before it is run against the regular expression.

```
http://axway.com/services?username=user&password=dXNlcg0K&dept=eng
```

You can extract the encoded value of the `password=` attribute value by specifying the string that occurs directly before the substring you want to extract, together with the string that occurs directly after the substring. Enter `password=` in the **Start substring** field, and `&` in the **End substring** field.

**Note** You must select the start and end substrings to ensure that the exact substring is extracted. For example, in this example, `password=` (including the equals sign) should be entered in the **Start substring** field, and not `password` (without the equals sign).

By specifying the correct substrings, you are left with the Base64-encoded attribute value (`dXNlcg0K`). However, you still need to Base64 decode it before you can run a regular expression on it. Make sure to select the **Base64 decode** check box. The Base64-decoded password value is `user`. This is the value that you need to run the regular expression against.

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

# Configure threatening content regular expressions

The regular expressions entered in this section guard against the possibility of a query string attribute 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* attribute values in the query string. If the expression matches *any* of the values, the filter fails.

If any regular expressions are configured in *Configure query string attribute regular expressions* on page 222, 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 prepopulated with regular expressions that guard against common attack signatures. These include a 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 character).

Enter or select an appropriate regular expression to restrict the value of the specified query string. 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, go to:

http://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html

# Validate REST request

The **Validate REST Request** filter enables you to validate the following aspects of a REST request:

- The HTTP method used in the request
- Each of the path parameters against a set of restrictive conditions called a *request parameter restriction*
- Each of the query string parameters against a request parameter restriction

For example, a request parameter restriction enables you to specify the expected data type of a named parameter, a regular expression for the parameter value, the minimum and maximum length of a string parameter, the minimum and maximum value of a numeric parameter, and so on.

This filter is found in the **Content Filtering** category in Policy Studio. For details on how to create a REST request, see *Create REST request* on page 242.

# General settings

Complete the following general settings:

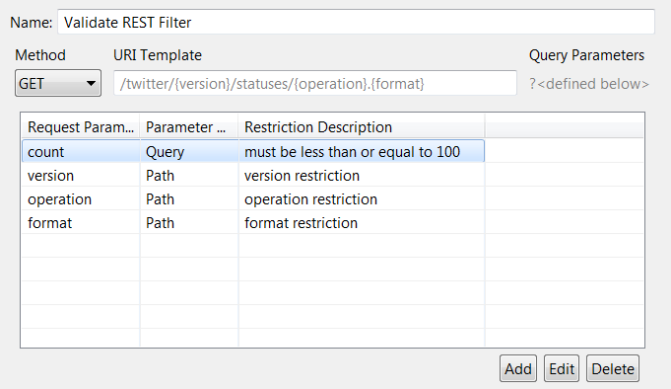
**Name**:
Enter an appropriate name for the filter to display in a policy.

**Method**:
Enter or select the HTTP method of the incoming message (for example, `POST`, `GET`, `DELETE`, and so on). The HTTP method of the incoming request must match the method specified here.

**URI Template**:
Enter the URI template of the inbound path. You can include parameters in the path (for example, `/twitter/{version}/statuses/{operation}.{format}`). For more information, see *URI path templates* on page 228. Any path parameters are automatically added to the table of parameter restrictions below. The path of the incoming request must match the path specified here, and the path parameters must meet the restrictions defined in the parameter restrictions table.

**Query Parameters**:
You can define query string parameters and restrictions in the parameter restrictions table below. The query string parameters must meet the restrictions defined in the parameter restrictions table. For more information, see *Add REST request parameter restrictions* on page 226.

The following figure shows the parameter restrictions table for the path `/twitter/{version}/statuses/{operation}.{format}` and the query string parameter `count`:




**Fail if unspecified query string parameters found**:
Select this option to have this filter fail if a request parameter is found on the incoming query string that has not been specified in the parameter restrictions table. This option is selected by default. You can use this option to guard against processing a query string containing a potentially malicious request parameter (for example, `/uri?number=2&badParam=System.exit(1);`).

**Extract valid parameters into individual message attributes**:
Select this option to store valid parameters in message attributes. This option is not selected by default.

**On failure save invalid parameter name as a message attribute**:
Select this option to save the invalid parameter name as a message attribute if the filter fails. This option is not selected by default.

**Apply path matching against decoded url**:
If this option is selected, API Gateway decodes a URL in the REST request and matches the decoded URL against a path parameter using the message attribute `http.request.path`. This is the default behavior.

If the path parameter contains URL encoded characters, the decoded URL that API Gateway uses (for example, `http://localhost:8080/myapi/myoperation/first/test`) does not match the path parameter with URL encoding (`http://localhost:8080/myapi/myoperation/first%2Ftest`). To achieve the match, deselect this option. Instead of using the decoded URL, API Gateway then uses the message attribute `http.request.rawURI.path` to match the encoded URL against the path parameter.

# Add REST request parameter restrictions

Click the **Add** button to configure restrictions on the values of query string parameters. Click the **Edit** button to configure restrictions on the values of path parameters that have been automatically added to the table. You can configure the following settings in the **REST Request Parameter Restriction** dialog:

**Request Parameter Name**:
Enter the name of the parameter to validate (for example, `customer_name`). This field is prefilled for path parameters. This is displayed in the table of parameter restrictions.

**Description**:
Enter a description of the restriction (for example, `Name parameter must be a string no longer than 10 characters`).This field is prefilled for path parameters. This is displayed in the table of parameter restrictions.

**Data Type**:
Enter or select the data type of the parameter (for example, `string` or `integer`). The default type is `string`.

**Fail if request parameter not found**:
Select this option if the specified request parameter must be present in the request. The filter fails if the parameter is not found. This option is not selected by default for query string parameters, but is selected by default for path parameters.

Complete the following fields on the **Request Parameter Restrictions** tab:

- **Min Length**

  Specifies the minimum number of characters or list items allowed (for example, `0`). The default value of `-1` means that this restriction is ignored.

- **Max Length**

Specifies the exact number of characters or list items allowed (for example, `10`). The default value of `-1` means that this restriction is ignored.

- **Regular Expression**

  Specifies the exact sequence of characters that are permitted using a regular expression (for example, `[a-zA-Z\s]*`).

  **Note** Do not enter the `^` character at the beginning of the expression and the `$` character at the end of the expression. This filter uses the XML Schema Pattern Facet regular expression language, which implicitly anchors all regular expressions at the head and tail.

- **Enumeration of Allowed Values**

  Specifies a list of permitted values. Click **Add** to enter an item in the list, and Click **OK**. Repeat as necessary to add multiple values.

Complete the following fields on the **Advanced Restrictions** tab:

- **Greater than**

  Specifies that the value entered in the **Minimum Value** field represents an exclusive lower bound (the value must be greater than this).

- **Greater than or Equal to**

  Specifies that the value entered in the **Minimum Value** field represents an inclusive lower bound (the value must be greater than or equal to this).

- **Minimum Value**

  Specifies the lower bounds for numeric values (for example, the value must be greater than 20).

- **Less than**

  Specifies that the value entered in the **Maximum Value** field represents an exclusive lower bound (the value must be less than this).

- **Less than or Equal to**

  Specifies that the value entered in the **Maximum Value** field represents an inclusive lower bound (the value must be less than or equal to this).

- **Maximum Value**

  Specifies the upper bounds for numeric values (for example, the value must be less than or equal to 30).

- **Max Total Digits for Number**

  Specifies the maximum number of digits allowed for a numeric data type. The default value of `-1` means that this restriction is ignored.

- **Max Digits in Fraction Part of Number**

  Specifies the exact number of digits allowed in the fraction part of a numeric type. For example, the number 1.23 has two fraction digits (two numbers after the decimal point). The default value of `-1` means that this restriction is ignored.

- **Whitespace**

Specifies how white space is handled (for example, line feeds, tabs, spaces, and carriage returns). You can select one of the following values:

- `Preserve` means the XML processor preserves (does not remove) any white space characters.
- `Replace` means the XML processor replaces any white space characters (line feeds, tabs, and carriage returns) with spaces.
- `Remove` means the XML processor removes any white space characters (line feeds, tabs, spaces, carriage returns are replaced with spaces, leading and trailing spaces are removed, and multiple spaces are reduced to a single space).

## URI path templates

Paths are specified using a formatted Jersey `@Path` annotation string. The `@Path` annotation's value is a relative URI path (for example, `/twitter`). The Jersey `@Path` annotation enables you to parametrize the path specified in the incoming request, by embedding variables in the URIs.

URI path templates are URIs with variables embedded within the URI syntax. These variables are substituted at runtime in order for a resource to respond to a request based on the substituted URI. Variables are denoted by curly braces (for example, `/twitter/{version}`).

The following is an example URI template:

```
/twitter/{version}/statuses/{operation}.{format}
```

It contains the variables `version`, `operation`, and `format`. At runtime, this URI template might resolve to:

```
/twitter/1.1/statuses/retweets_of_me.json
```

For more information on Jersey `@Path` annotations, go to https://jersey.java.net/.

# 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, 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 **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" in the *API Gateway Policy Developer Guide*.

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

- **Enter Regular Expression**:
  You can configure selectors that are checked against a regular expression from the global library (**White list**), 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 library (**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 **Environment Configuration** > **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 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 extracted substring can also be Base64 decoded if necessary.

# Configure 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.

**Note** If any regular expressions are configured in *Configure selector-based regular expressions* on page 229, 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 prepopulated 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**.

# Validate timestamp

## Overview

You can use the **Validate Timestamp** filter to validate a timestamp that has been stored in a message attribute by a previous filter in a policy. For example, you can extract the value of a `wsu:Created` element from a WS-Security token and store it in a created attribute using the **Retrieve from Message** filter in the **Attributes** category. You can then use the **Validate Timestamp** filter to ensure that the created timestamp is not *after* the current time.

Similarly, you can use the **Retrieve from Message** filter to extract the value of the `wsu:Expires` element and store it in a timestamp message attribute. You can then use the **Validate Timestamp** filter to check that the timestamp is not *before* the current time.

You can configure the drift time to resolve discrepancies between clock times on the machine that generated the timestamp, and the machine running API Gateway. If you are validating a `Created` timestamp (**Timestamp must be in the past** is selected), the time must be after the `Created` time minus the drift time. Alternatively, if you are validating an `Expires` timestamp (**Timestamp must be in the future** is selected), the time now must be before the `Expires` time plus the drift time.

**Note** To validate the timestamp stored in a WS-Security Username Token or SAML assertion, you can use the **WS-Security Username Token Authentication**, **SAML Authentication**, **SAML Authorization**, or **SAML Attribute** filters. Furthermore, there is an implicit timestamp validation check performed by the **Extract WSS Header** filter, which you can use to validate a WS-Utility Timestamp that appears in a WSS `Header` block.

The **Validate Timestamp** filter does not require an entire WS-Utility Timestamp element. Instead, this filter only requires a single date-formatted string.

## Configuration

Complete the following fields to configure the API Gateway to validate a timestamp that has been stored in a message attribute:

**Name**:
Enter a name for the filter to display in a policy.

**Selector Expression to Retrieve Timestamp**:
Enter the name of the selector expression that contains the value of the timestamp. Defaults to `${timestamp}`. The specified selector is expanded at runtime to the corresponding message attribute value. For more details, see "Select configuration values at runtime" in the *API Gateway Policy Developer Guide*.

**Note** You must configure a predecessor of this filter to extract the timestamp from the message and store it in the specified attribute (for example, the **Retrieve from Message** filter in the **Attributes** category.

**Format of Timestamp**:
Enter the format of the timestamp that is contained in the specified message attribute. The default date/time format is `yyyy-MM-dd'T'HH:mm:ss.SSS'Z'`, which can be altered if necessary. For more information on how to use this format, see the Javadoc for the `java.text.SimpleDateFormat` class.

**Timezone**:
Select the time zone to use to interpret the time stored in the message attribute selected above. The default option is GMT.

**Drift (ms)**:
Specify the drift time in milliseconds when determining whether the current time falls within a certain time interval. The drift time can be used to account for differences in the clock times of the machine running the API Gateway and the machine on which the timestamp was generated.

**Timestamp must be in the past**:
The time in the timestamp must be *before* the time at which the server validates the timestamp. This is used for validating a timestamp that represents a `Created` time (the created time must be before the validation time).

**Timestamp must be in the future**:
The time in the timestamp must be *after* the time at which the server validates the timestamp. This is used for validating a timestamp that represents an `Expires` time (the expiry time must be some time in the future relative to the validation time).

# Verify the WS-Policy security header layout

## Overview

Web services can use the WS-Policy specification to advertise the security requirements that clients must adhere to in order to successfully connect and send messages to the service. For example, a typical WS-Policy would mandate that the SOAP request body be signed and encrypted (using XML Signature and XML Encryption) and that a signed WS-Utility Timestamp must be present in a WS-Security header.

To guarantee that the security tokens used to *protect* the message are added to the request in the most efficient and interoperable manner, WS-Policy uses the `<wsp:Layout>` assertion. The semantics of this assertion are implemented by the **WS-Security Policy Layout** filter and are outlined in the configuration details in the next section.

## Configuration

To check a SOAP message for a particular WS-Policy layout, complete the following fields:

**Name**:
Enter an intuitive name for this filter to display in a policy (for example, `Check SOAPRequest for Lax Layout`).

**Actor**:
Enter the name of the SOAP Actor/Role where the security tokens are present.

**Select Required Layout Type**:
Select the required layout from the following WS-Policy options:

- **Strict**:
  Select this option to check that a SOAP message adheres to the WS-Policy strict layout rules. For more information, see the WS-Policy specification.
- **Lax**:
  Select this option if you want to ensure that the security tokens in the SOAP header have been inserted according to the Lax WS-Policy layout rules. The WS-Policy Lax rules are effectively identical to those stipulated by the SOAP Message Security specification.
- **LaxTimestampFirst**:
  This layout option ensures that the WS-Policy Lax rules have been followed, but also checks to make sure that the WS-Utility Timestamp is the first security token in the WS-Security header.
- **LaxTimestampLast**:
  This option ensures that the WS-Utility Timestamp is the last security token in the WS-Security header and that all other Lax layout rules have been followed.

**WS-Security Version**:
The layout rules for WS-Security versions 1.0 and 1.1 are slightly different. Select the version of the layout rules to apply to SOAP requests. For details on the differences between these versions, see the WS-Security specifications (WSS 1.0 and WSS 1.1).

# XML complexity

## Overview

Parsing XML documents is a notoriously processor-intensive activity. This can be exploited by hackers by sending large and complex XML messages to web services in a type of denial-of-service attack attempting to overload them. The **XML Complexity** filter can protect against such attacks by performing the following checks on an incoming XML message:

- Checking the total number of nodes contained in the XML message.
- Ensuring that the message does not contain deeply nested levels of XML nodes.
- Making sure that elements in the XML message can only contain a specified maximum number of child elements.
- Making sure that each element can have a maximum number of attributes.

By performing these checks, the API Gateway can protect back-end web services from having to process large and potentially complex XML messages.

You can also use the **XML Complexity** filter on the web service response to prevent a dictionary attack. For example, if the web service is a phone book service, a `name=*` parameter could return all entries.

# Configuration

The **XML Complexity** filter should be configured as the first filter in the policy that processes the XML body. This enables this filter to block any excessively large or complex XML message *before* any other filters attempt to process the XML.

To configure the **XML Complexity** filter, complete the following fields:

**Name**:
Enter an appropriate name for this filter to display in a policy.

**Maximum Total Number of Nodes**:
Specify the maximum number of nodes to allow in an XML message.

**Note** This number does not include text nodes or comments. You can use *Message size filtering* on page 202 to stop large text nodes or comments.

**Maximum Number of Levels of Descendant Nodes**:
Enter the maximum number of descendant nodes that an element is allowed to have. Again, this number does not include text nodes or comments.

**Maximum Number of Child Nodes per Node**:
Enter the maximum number of child nodes that an element in an XML message is allowed to have.

**Maximum Number of Attributes per Node**:
Enter the maximum number of attributes that an element is allowed to have.

# Conversion filters 9

The following filters enable you to perform various conversions:



# Add HTTP header

## Overview

The API Gateway 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, until it is processed by the back-end system.

Each message being processed by the API Gateway 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" in the *API Gateway Policy Developer Guide*.

# Configuration

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

**Name**:
Enter an appropriate name for the filter to display in a policy.

**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 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:

```
${message_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. Use this option for HTTP body entity headers, which provide metadata about the message body. For example, this includes headers such as the following:

```
Content-Language
Content-Length
Content-Location
Content-MD5
Content-Range
Content-Type
Expires
Last-Modified
```

```
Extension header
```

**Add header to HTTP headers attribute**:
Select this option to add the HTTP header to the `http.headers` message attribute. Use this option for general HTTP headers, which apply to both request and response messages. For example, this includes headers such as `SOAPAction`:

```
Server:
Connection: close
X-CorrelationID: Id-9e38c653c24c000000000009593813a 0
Host: localhost:8083
SOAPAction: "http://example.com/api/ConvertUnits"
User-Agent: Gateway
Content-Type: text/xml; charset="utf-8"
MyHeader: FOO

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
   <soap:Body>
      <ConvertUnits xmlns="http://example.com/api/">
         <EnterUnit>100</EnterUnit>
         <FromUnits>K</FromUnits>
         <ToUnits>M</ToUnits>
      </ConvertUnits>
    </soap:Body>
</soap:Envelope>
```

# Add XML node

## Overview

You can use the **Add XML Node** filter to add an XML element, attribute, text, comment, or CDATA section node to an XML message. The new node is inserted into the location specified by an XPath expression or a SOAP actor/role. XPath is a query language that enables you to select nodes in an XML document. A SOAP actor/role provides a way of identifying a particular WS-Security block in a message.

## Configuration

You can configure the following settings.

## Where to insert new nodes

You can insert the new node into a location specified by an XPath expression or into a WS-Security header for the specified SOAP actor or role. Select one of the following options:

**Insert using XPath**:
Select or enter an XPath expression to specify where to insert the new node. If this expression returns more than one node, the first one is used. If the expression returns no nodes, the filter returns false. You can add, edit, or delete XPath expressions using the buttons provided.

Select one of the following options to determine where the new node is placed relative to the nodes returned by the XPath expression.

- **Append**:
  The new node is appended as a child node of the node returned by the XPath expression. If there are already child nodes of the node returned by the XPath expression, the new node is added as the last child node.
- **Before**:
  The new node is inserted as a sibling node before the node returned by the XPath expression.
- **Replace**:
  The node pointed to by the XPath expression is completely replaced by the new node.

**Insert into WS-Security element for SOAP Actor/Role**:
Select or enter the name of the SOAP actor/role that specifies the WS-Security element into which the XML node is inserted. A SOAP actor/role provides a way of distinguishing a particular WS-Security block from others that may be present in the message. For example, this setting is useful if there is no SOAP header or WS-Security element in the message because these are created when this option is specified.

## Node source

Select one of the following options for the source of the new node:

- **Create a new node**:
  If this option is selected, a new node is created and inserted into the location pointed to by the XPath expression configured above.
- **Insert previously removed nodes**:
  You can configure a **Remove XML Node** filter to remove XML nodes from the message and store them in the `deleted.node.list` message attribute. You can use the **Add XML Node** filter to reinsert these nodes back into a different location inside the message, effectively moving the deleted nodes in the message. To use this option, select the **Save deleted nodes** option on a **Remove XML Node** filter that is configured to run before the **Add XML Node** filter in the policy.
- **Message attribute**:
  If this option is selected, a new node is created from the contents of the selected message attribute. The expected type of the message attribute is Node of List of Nodes.

## New node details

Configure the following node details:

**Node Type**:
Select the type of the new node from the list.

**Note** When choosing a node type, consider the following:

- You can only append a node to a **Node Type** of `Element` or `Text`.
- If you append to a `Text` node, you must append another `Text` node.
- If you add a new `Attribute` node using the **Replace** option, it must replace an existing `Attribute` node.

**Node Content**:
This field contains the node to be inserted into the message. The **Node Content** field must contain valid XML when the **Node Type** is set to `Element`. You can also enter selector expressions, which are populated at runtime.

**Note** To insert an XML node containing a *namespace*, you must define the namespace before using it in the filter, or the insertion will fail due to invalid XML. For more details, see Knowledge Base article 178251 on Axway Support at https://support.axway.com.

## Attribute node details

The following attribute-related fields are only enabled when you select `Attribute` from the **Node Type** list:

**Attribute Name**:
Enter the name of the attribute in this field.

**Attribute Namespace URL**:
Enter the namespace of the attribute in this field.

**Attribute Namespace Prefix**:
Specify the prefix to use to map the element to the namespace entered above in this field. The new attribute is prefixed with this value.

# Examples

The following are some examples of using the **Add XML Node** filter to replace and add attributes and elements.

## Replace an attribute value

To replace an attribute value, perform the following steps:

1. In the **Configure where to insert the new nodes** section, select **Insert using XPath**.
2. Select a value from the list (for example, `SOAP Header "mustUnderstand" attribute`).
3. Select the **Replace** option.
4. In the **Configure new node details** section, select `Attribute` from the a **Node Type** list.
5. Enter the **Node Content** in the text box (for example, in this case, `1` or `0`).
6. In the **Attribute Details** section, you must enter the **Attribute Name**.

## *Add an attribute*

To add an attribute to an element, perform the following steps:

1. In the **Configure where to insert the new nodes** section, select **Insert using XPath**.
2. Select a value from the drop-down list (for example, `SOAP Header Element`).
3. Select the **Append** option.
4. In the **Configure new node details** section, select `Attribute` from the a **Node Type** list.
5. Enter the **Node Content** in the text box (for example, in this case `1` or `0`).
6. In the **Attribute Details** section, you must enter the **Attribute Name**.

## *Add an element*

To add an element, perform the following steps:

1. In the **Configure where to insert the new nodes** section, select **Insert using XPath**.
2. Select a value from the drop-down list (for example, `SOAP Header Element`).
3. Select the **Append** option.
4. In the **Configure new node details** section, select `Element` from the a **Node Type** list.
5. Enter the **Node Content** in the text box (for example, in this case, the contents of the SOAP header).

## *Replace an element*

To replace element A with new element B, perform the following steps:

1. In the **Configure where to insert the new nodes** section, select **Insert using XPath**.
2. Select a value from the drop-down list (for example, `SOAP Method Element`).
3. Select the **Replace** option.
4. In the **Configure new node details** section, select `Element` from the a **Node Type** list.

5. Enter the **Node Content** in the text box (for example, in this case, the contents of the SOAP method).

# Convert multipart or compound body type message

## Overview

The **Convert Multipart/compound body type** filter is typically used to compress a MIME message before transferring a message using FTP to an FTP server. You could create a simple policy containing a **Convert Multipart/compound body type** filter as a predecessor of an **FTP Upload** filter to achieve this functionality.

The **Convert Multipart/compound body type** filter outputs a `content.body` message attribute, which represents the last part processed. For example, in a three part multipart message, the value of the `content.body` attribute is the third and last part in the series.

## Configuration

Configure the following fields for this filter:

**Name**:
Enter a suitable name for this filter to display in a policy.

**Multipart content type**:
Enter the MIME content type to compress the multipart message to. For example, to compress a multipart message to a ZIP file, enter `multipart/x-zip` in this field.

# Create cookie

## Overview

An HTTP cookie is data sent by a server in an HTTP response to a client. The client can then return an updated cookie value in subsequent requests to the server. For example, this enables the server to store user preferences, manage sessions, track browsing habits, and so on.

The **Create Cookie** filter is used to create a `Set-Cookie` header or a `Cookie` header. The `Set-Cookie` header is used when the server instructs the client to store a cookie. The `Cookie` header is used when a client sends a cookie to a server. This filter adds the appropriate HTTP cookie header to the message header, and saves the cookie as a message attribute.

For more details, see *Get cookie* on page 37.

## Configuration

Configure the following fields:

**Filter Name**:
Enter an appropriate name to display for this filter to display in a policy.

**HTTP Header Type**:
Select the HTTP cookie header type to create: **Set-Cookie (Server)** or **Cookie (Client)**. When this is set to **Set-Cookie (Server)**, all attributes from the **Cookie Details** list are used. When this is set to **Cookie (Client)**, only the **Cookie Value** attribute is used.

### *Cookie details*

You can configure the following settings for the cookie in the **Cookie Details** section:

**Cookie Name**:
Enter the name of the cookie.

**Cookie Value**:
Enter the value of the cookie.

**Domain**:
Enter the domain name for this cookie.

**Path**:
Enter the path on the server to which the browser returns this cookie.

**Max-Age**:
Enter the maximum age of the cookie in days, hours, minutes, and seconds.

**Secure**:
Select this option to restrict sending this cookie to a secure protocol. This setting is not selected by default, which means that it can be sent using any protocol.

**HTTPOnly**:
Select this option to restrict the browser to use cookies over HTTP only. This setting is not selected by default.

# Create REST request

## Overview

Representational State Transfer (REST) is a client-server architectural style used to represent the state of application resources in distributed systems. Typically, servers expose resources using a URI, and clients access these resources using HTTP verbs such as HTTP GET, HTTP POST, HTTP DELETE, and so on.

The **Create REST Request** filter enables you to create HTTP requests to RESTful web services. You can also configure the query string parameters that are sent with the REST request. For example, for an HTTP GET request, the parameters are URL-encoded and appended to the request URI as follows:

```
GET /translate_a/t?client=t&sl=en&tl=ga&text=Hello
```

For an HTTP POST request, the parameters are URL-encoded and added to the request body as follows:

```
POST /webservices/tempconvert.asmx/CelsiusToFahrenheit
Host:ww.w3schools.com
Accept-charset:en
Celsius=200
```

This filter is found in the **Conversion** category in Policy Studio. For details on how to extract REST request attributes from a message, see *Extract REST request attributes* on page 33. For details on how to validate a REST request, see *Validate REST request* on page 224.

# Configuration

Complete the following fields:

**Name**:
Enter an appropriate name for the filter to display in a policy.

**HTTP Method**:
Enter or select an HTTP method from the list (for example, `POST`, `GET`, `DELETE`, and so on).

**REST Request Parameters**:
You can add query string parameters to the REST request. These are simple name-value pairs (for example, `Name=Joe Bloggs`). To add query string parameters, click the **Add** button, and enter the name-value pair in the **Configure REST Request Parameters** dialog. Repeat to add multiple parameters.

This filter generates the `http.querystring` and `http.raw.querystring` message attributes to store the query string. For example, you can then append contents of the `http.raw.querystring` message attribute to a **Connect to URL** or **Rewrite URL** filter using a message attribute selector (for example, `${http.raw.querystring}`). For more details on selectors, see "Select configuration values at runtime" in the *API Gateway Policy Developer Guide*.

**Add attributes stored in attribute lookup list to REST request**:
If you have populated the `attribute.lookup.list` message attribute using a previous filter in a policy, you can select this setting to include these message attributes in the serialized query string that is written to the request.

# Transform with data map

Data maps enable you to define how to map XML and JSON messages to other XML and JSON message formats. You can use the **Execute Data Map** filter to execute a data map as part of a policy. This filter is available in the **Conversion** category in Policy Studio

## Configuration

Configure the following fields:

**Name**:

Enter a name that reflects the role of this filter in the policy.

**Data Map**:

Click the browse button to view the list of available data maps and select the required data map.

You can create data maps under the **Resources > Data Maps** node in the Policy Studio tree. For more information, see "Manage data maps" in the *API Gateway Policy Developer Guide*.

**Default Encoding**:

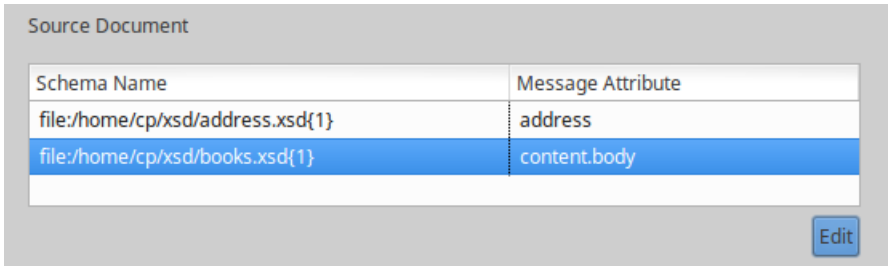
Enter the default encoding for the data map. The default value is `UTF-8`.

**Source Document**:

This section is automatically populated with the input schemas when you select a data map. The ordering of the schemas in the filter matches the ordering in the data map.

You must specify the message attribute to map to the inputs of each schema. Click **Edit** to edit the message attributes.

- If you select a data map with a single input schema, the message attribute defaults to `content.body`.
- If you select a data map with multiple input schemas, you must specify the message attribute to map to the inputs of each schema. For example:




**External Parameters**:

This section is automatically populated when you select a data map. Click **Edit** to edit the expression for each parameter. For more details on parameters, see the *API Gateway Visual Mapper User Guide*.

## Example policy

For an example of how to use a data map in a policy, see "Manage data maps" in the *API Gateway Policy Developer Guide*.

# Extract MTOM content

## Overview

Message Transmission Optimization Mechanism (MTOM) provides a way to send binary data to web services within 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 in the SOAP message using the `<xop:Include>` element.

The following MTOM message 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 the API Gateway receives this request, the **Extract MTOM Content** filter can be used to extract the Base64-encoded content from the `<image>` element, replace it with an `<xop:Include>` element, which 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 Tester
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
```

**Note** Note the following in the resulting MTOM message:

- The Base64-encoded contents of the `<image>` element have been replaced by the `<xop:Include>` element.
- The `<xop:Include>` element points to a MIME part using the `href` attribute.
- The value of the `href` attribute corresponds to the value of the `Content-ID` HTTP header of the MIME part that contains the binary octets of the actual image file.

# Configuration

Complete the following fields for the **Extract MTOM Content** filter:

**Name**:
Enter an appropriate name for the filter to display in a policy.

**XPath Location**:
Use an XPath expression to locate the encoded data elements. For example, in the sample SOAP request message above, you would configure an XPath expression to point to the `<image>` element. For more information, see "Configure XPath expressions" in the *API Gateway Policy Developer Guide*.

# Insert MTOM attachment

## Overview

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 in the SOAP message using the `<xop:Include>` element.

The following MTOM message contains a binary image encapsulated in a MIME part:

**Note** The MIME part that contains the binary image is referenced in the SOAP request body using the `<xop:Include>` element. The `href` attribute of this element refers to the `Content-ID` HTTP header of the MIME part.

```
POST /services/uploadImages HTTP/1.1
Host: API Tester
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 the API Gateway receives this request, the **Insert MTOM Attachment** filter can be used to read the binary data in the MIME parts pointed to by the `<xop:Include>` elements embedded in the SOAP request. The binary data is then Base64-encoded and inserted into the message in place of the `<xop:Include>` elements. The resulting message is as follows:

```
<?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>
```

## Configuration

Complete the following fields for the **Insert MTOM Attachment** filter:

**Name**:
Enter a name for the filter.

**XPath Location**:
Use an XPath expression to point to the location of the `<xop:Include>` element that refers to the binary attachment. The specified XPath expression can point to multiple `<xop:Include>` elements if necessary. For example, an XPath expression of `//xop:Include` returns all `<xop:Include>` elements in the SOAP envelope. For more information, see "Configure XPath expressions" in the *API Gateway Policy Developer Guide*.

**Remove attachments once they have been included in the message:**
Select this option to remove the MIME parts that contain the actual binary content from the message after they have been inserted into the message.

# Add node to JSON document

## Overview

You can use the **JSON Add Node** filter to add a node to a JavaScript Object Notation (JSON) document. The new node is inserted into the location specified by a JSON Path expression. JSON Path is a query language that enables you to select nodes in a JSON document.

For more details on JSON Path, see http://code.google.com/p/jsonpath.

# Configuration

You can configure the following settings:

**Name**:
Enter a suitable name that reflects the role of this filter in the policy.

**JSON Path Expression**:
Enter the JSON Path expression used to add the node to the JSON document (for example, `$.store`). Policy Studio warns you if you enter an unsupported JSON Path expression.

**Note** If this expression returns more than one node, the first node is used. If the expression returns no nodes, the filter returns false.

**Node Source**:
In the **Content** area, enter the JSON node to be inserted into the message. For example, the following node source represents a new car:

```
{
  "make":"Ford",
  "airbags":true,
  "doors":4,
  "price":1111.00
}
```

Select one of the following options for the source of the new node:

- **Add as a new item to an array**:
  If you select this option, the new JSON node is added as an item in an array.
- **Add as a new item with field name**:
  If you select this option, the new JSON node is added as a field specified in the **Field Name** field (for example, `car`).
- **Insert previously removed nodes**:
  You can configure a **JSON Remove Node** filter to remove JSON nodes from the message and store them in the `deleted.json.node.list` message attribute. You can then use the **JSON Add Node** filter to reinsert these nodes in a different location in the message, effectively moving the deleted nodes in the message. When selecting this option, you must also select **Save deleted nodes to be reinserted to new location** in the **Remove JSON Node** filter, which runs before the **Add JSON Node** filter in the policy. For more details, see *Remove node from JSON document* on page 253.

**What to do with any existing siblings in the container**:
Select one of the following options to determine where the new node is placed relative to the nodes returned by the JSON Path expression:

- **Append**:
  The new node is appended as a child node of the node returned by the JSON Path expression. If there are already child nodes of the node returned by the JSON expression, the new node is

added as the last child node.

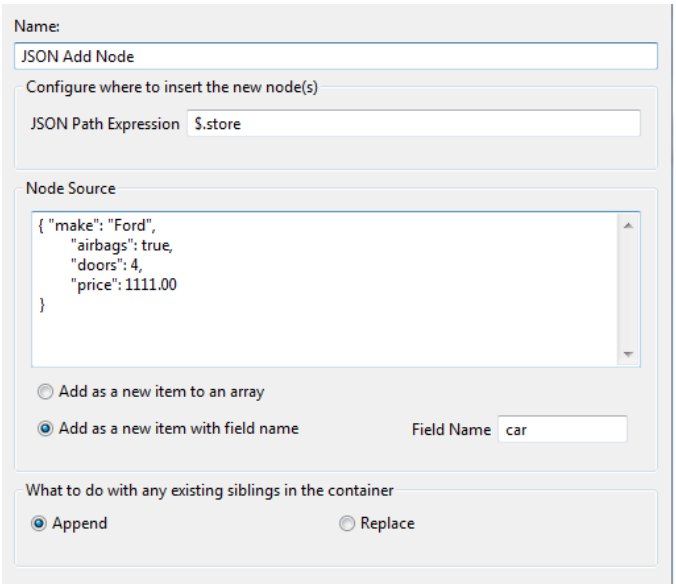
- **Replace**:
  The node pointed to by the JSON expression is completely replaced by the new node.

# Examples

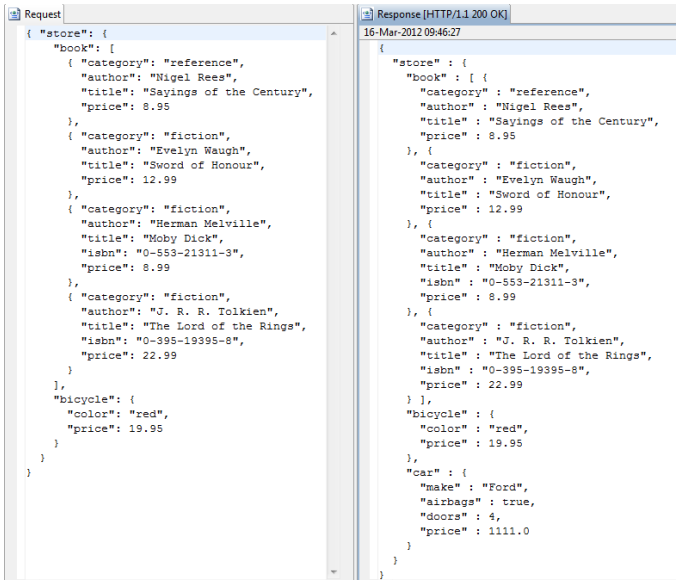
The following are some examples of using the **JSON Add Node** filter to add and replace JSON nodes.

## *Add a JSON node*

The following example shows the settings required to add a car node to the store:

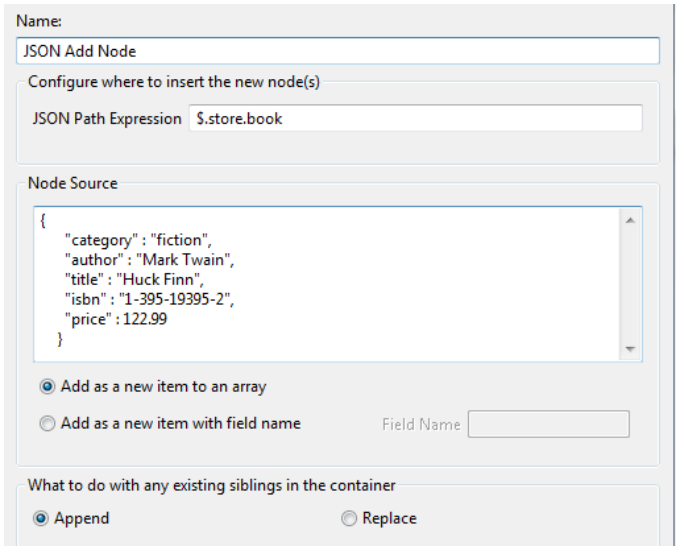


The following example shows the corresponding request and response message in Axway API Tester:

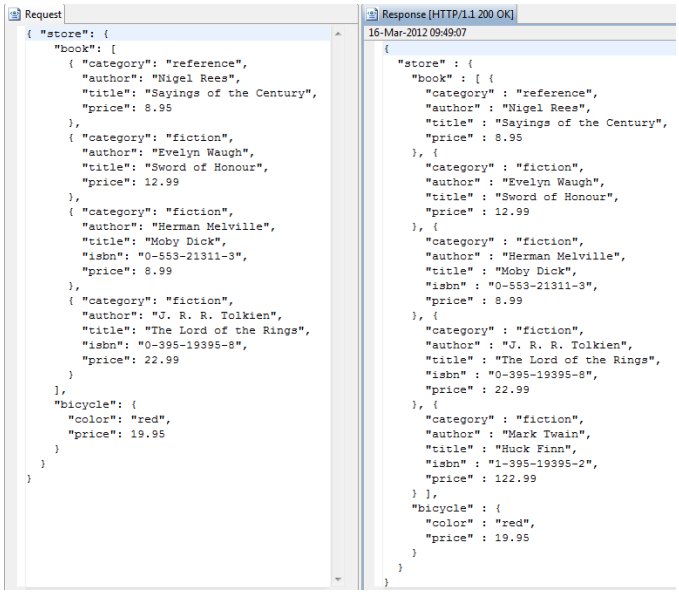



## *Add an item to an array*

The following example shows the settings required to add a book to an array:

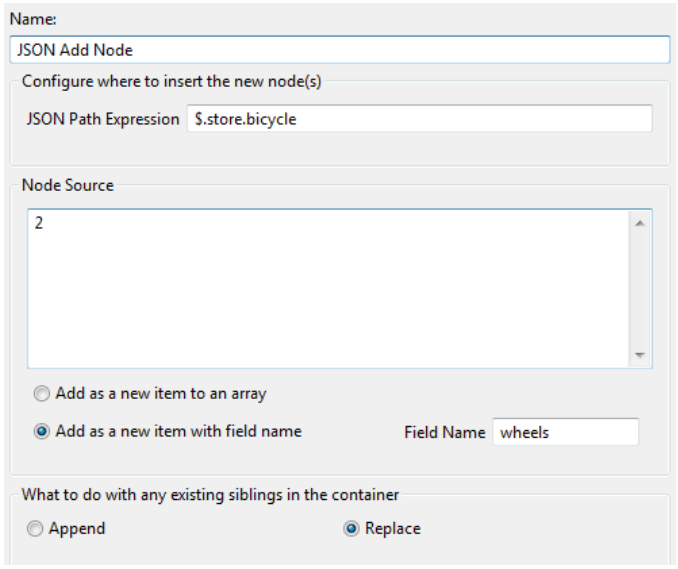


The following example shows the corresponding request and response message in API Tester:

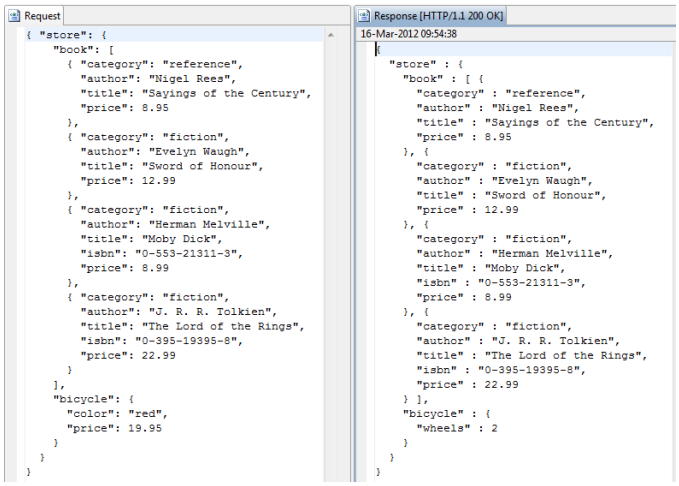



## *Add a field replacing others*

The following example shows the settings required to add a field to the bicycle, removing any other fields that may exist:



The following example shows the corresponding request and response message in API Tester:




## Exceptions

The **JSON Add Node** filter aborts with a `CircuitAbortException` if:

- Content body of the payload is not in valid JSON format
- New node to be added (content in **Node Source**) is not in valid JSON format

# Remove node from JSON document

## Overview

You can use the **JSON Remove Node** filter to remove a JSON node from a JSON message. You can specify the node to remove using a JSON Path expression. The JSON Path query language enables you to select nodes in a JSON document.

For more details on JSON Path, see http://code.google.com/p/jsonpath.

## Configuration

To configure this filter, specify the following fields:

**Name**:
Enter a suitable name that reflects the role of the filter in the policy.

**JSON Path Expression**:
Enter a JSON Path expression to specify the node to remove (for example, `$.store.bicycle`). Policy Studio warns you if you enter an unsupported JSON Path expression.

**Note** If the specified expression returns more than one node, all returned nodes are removed.

**Fail if no nodes returned from JSON Path**:
When this option is selected, and the JSON Path expression returns no nodes, the filter returns false. If this option is *not* selected, and the JSON Path returns no nodes, the filter returns true, and no nodes are removed. This option is not selected by default.

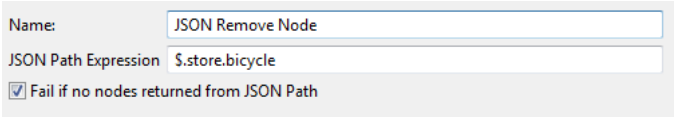
**Save deleted nodes to be reinserted to new location**:
Select this option if you want to move JSON nodes from one location in the message to another. The deleted nodes are stored in the `deleted.json.node.list` message attribute. You can then use the **JSON Add Node** filter to insert the deleted nodes into a different location in the message. For more details, see *Add node to JSON document* on page 248.

# Examples

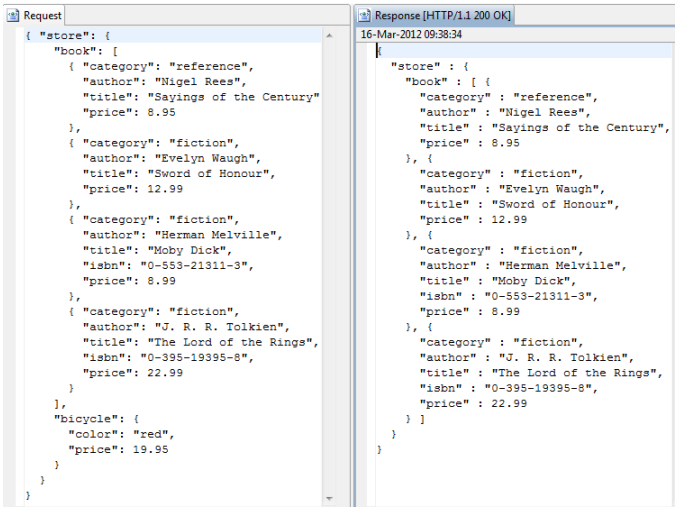
The following are some examples of using the **JSON Remove Node** filter.

## *Remove a node*

The following example shows removing a bicycle from the store:

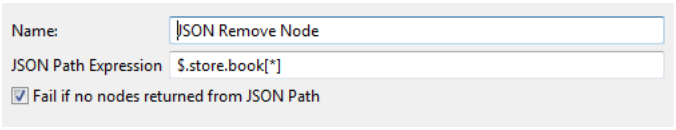



The following example shows the corresponding request and response message in Axway API Tester:

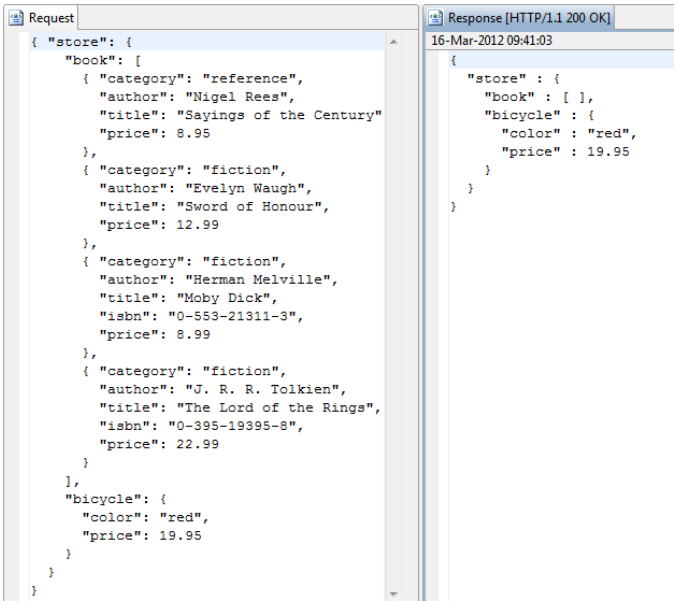


### *Remove all items in an array*

The following example shows removing all books in an array:




The following example shows the corresponding request and response message in API Tester:




## Exceptions

The **JSON Remove Node** filter aborts with a `CircuitAbortException` if:the content body of the payload is not in valid JSON format.

# Convert JSON to XML

## Overview

You can use the **JSON to XML** filter to convert a JavaScript Object Notation (JSON) document to an XML document.

For details on the mapping conventions used, go to:

https://github.com/beckchr/staxon/wiki/Mapping-Convention

# Configuration

To configure the **JSON to XML** filter, specify the following fields:

**Name**:
Enter a suitable name that reflects the role of the filter in the policy.

**Virtual root element**:
If the incoming JSON document has multiple root elements, enter a virtual root element to be added to the output XML document. This is required because multiple root elements are not valid in XML. Otherwise, the XML parser fails. For more details, see *Examples* on page 256.

**Insert processing instructions into the output XML representing JSON array boundaries**:
Select this option to enable round-trip conversion back to JSON. This inserts the necessary processing instructions into the output XML. This option is not selected by default. For more details, see *Examples* on page 256.

**Note** This option is recommended if you wish to convert back to the original JSON array structures. This information would be lost during the translation back to XML.

For more details, see *Convert XML to JSON* on page 266.

**Convert JSON object names to valid XML element names**:
Select this option to convert your JSON object names to XML element names. This option is not selected by default.

**Note** You should ensure that your JSON object names are also valid XML element names. If this is not possible, this option analyzes each object name and automatically performs the conversion. This has a performance overhead and is not recommended if you wish to convert back to the original JSON.

# Examples

This section shows examples of using **JSON to XML** filter options.

## *Multiple root elements*

For example, the following incoming JSON message has multiple root elements:

```
{
  "firstName": "John",
  "lastName": "Smith",
  "age": 25,
  "address":
  {
```

```
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021"
  },
  "phoneNumber":
  [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "fax",
      "number": "646 555-4567"
    }
  ]
}
```

If you enter `customer` in the **Virtual root element** field, this results in the following output XML:

```
<?xml version="1.0" encoding="utf-8"?>
<customer>
  <firstName>John</firstName>
  <lastName>Smith</lastName>
  <age>25</age>
  <address>
    <streetAddress>21 2nd Street</streetAddress>
    <city>New York</city>
    <state>NY</state>
    <postalCode>10021</postalCode>
  </address>
  <phoneNumber>
    <type>home</type>
    <number>212 555-1234</number>
  </phoneNumber>
  <phoneNumber>
    <type>fax</type>
    <number>646 555-4567</number>
  </phoneNumber>
</customer>
```

## *Insert processing instructions into the output XML*

For example, take the following incoming JSON message:

```
{
"customer" :
{
  "first-name" : "Jane",
  "last-name" : "Doe",
  "address" :
  {
    "street" : "123 A Street"
  },
  "phone-number" :
  [
    {
      "@type" : "work",
      "$" : "555-1111"
    },
    {
      "@type" : "cell",
      "$" : "555-2222"
    }
  ]
}
}
```

When the **Insert processing instructions into the output XML representing JSON array boundaries** option is selected, the output XML is as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<customer>
  <first-name>Jane</first-name>
  <last-name>Doe</last-name>
  <address>
    <street>123 A Street</street>
  </address>
<?xml-multiple phone-number?>
  <phone-number type="work">555-1111</phone-number>
  <phone-number type="cell">555-2222</phone-number>
</customer>
```

# Exceptions

The **JSON to XML** filter aborts with a `CircuitAbortException` if:

- Content body of the payload is not in valid JSON format
- **Convert JSON object names to valid XML element names** option is selected, and there is a problem with the conversion process

# Load contents of a file

## Overview

The **Load File** filter enables you to load the contents of the specified file, and set them as message content to be processed. When the contents of the file are loaded, they can be passed to the core message pipeline for processing by the appropriate message filters. For example, you might use the **Load File** filter in cases where an external application drops XML or JSON files on to the file system to be validated, modified, and potentially routed on over HTTP, JMS, or stored to a directory where the application can access them again.

For example, this sort of protocol mediation can be useful when integrating legacy systems. Instead of making drastic changes to the legacy system by adding an HTTP engine, the API Gateway can load files from the file system, and route them on over HTTP to another back-end system. The added benefit is that messages are exposed to the full range of message processing filters available in the API Gateway. This ensures that only properly validated messages are routed on to the target system.

## Configuration

Configure the following fields.

### *Input settings*

**File**:
Enter the name of the file to load, or browse to the file in the file system. This setting is required.

### *Processing settings*

The fields in this section determine what processing is performed on the input files, and where files are placed before and after processing.

**Processing Directory**:
Enter or browse to the directory to which the input file is copied prior to processing. This field is optional. If this is not specified, the input file remains in the current input directory.

**Response Directory**:
Enter or browse to the directory to which the response file is copied. This field is optional. If this is not specified, the response file is not written to disk.

**Processing Policy**:
Select the policy executed on the input file. For example, the policy could perform message validation, routing, virus checking, or XSLT transformation. This field is optional.

**File Type**:
Specifies how the input file is interpreted. Select one of the following options:

- **Raw**:
  Assumes a content-type of `application/octet-stream`. This is the default.
- **Treat as HTTP Message (including headers)**:
  Assumes the inbound file contains an HTTP request (optionally with HTTP headers).
- **Infer content-type from extension**:
  Performs a lookup on configured MIME types to determine the content-type of the file based on its extension.
- **Use Content-type**:
  Enables you to specify a content-type in the text box.

### *On completion settings*

You can specify what to do when the file processing has completed. Select one of the following options:

- **Do Nothing**:
  The input file remains in the input directory or in the **Processing Directory**. This is the default.
- **Delete Input File**:
  The input file is deleted from the input directory or the **Processing Directory**.
- **Move Input File**:
  The input file is moved (archived) to the directory specified in the **To Directory** field. You can also specify an optional **File Prefix** or **File Suffix** for the archived file.

# Remove HTTP header

## Overview

The API Gateway can strip a named HTTP header from the message as it passes through a policy. This is especially useful in cases where end user credentials are passed to the API Gateway in an HTTP header. After processing the credentials, you can use the **Remove HTTP Header** filter to strip the header from the message to ensure that it is not forwarded on to the destination web service.

## Configuration

To configure the **Remove HTTP Header** filter, perform the following steps:

1. Enter an appropriate name for this filter in the **Name** field.
2. Specify the name of the HTTP header to remove in the **HTTP Header Name** field.
3. Select **Fail if header is not present** to configure the API Gateway to abort the filter if the message does not contain the named HTTP header. Headers can be added to the message using the **Add HTTP Header** filter (see *Add HTTP header* on page 235).

# Remove XML node

## Overview

You can use the **Remove XML Node** filter to remove an XML element, attribute, text, or comment node from an XML message. You can specify the node to remove using an XPath expression. The XPath query language enables you to select nodes in an XML document.

## Configuration

To configure this filter, specify the following fields:

**Name**:
Enter a suitable name that reflects the role of the filter in the policy. For example, if the purpose of this filter is to remove an `<ID>` element from the message, it would be appropriate to name this filter **Remove ID Element**.

**XPath Location**:
Specify an XPath expression to indicate the node to remove. When the expression is configured correctly, you can remove an element, attribute, text, or comment node. If this expression returns more than one node, all returned nodes are removed.

You can select XPath expressions from the list, and edit or add expressions by clicking the relevant button. The following are some example expressions:

| Name | XPath Expression | Prefix | URI |
|---|---|---|---|
| The First WSSE Security element | `//wsse:Security[1]` | `wsse` | `http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd` |

| Name | XPath Expression | Prefix | URI |
|---|---|---|---|
| Text Nodes in SOAP Body | `/soap:Envelope/soap:Body/text()` | `soap` | `http://schemas.xmlsoap.org/soap/envelope/` |

**Fail if no nodes returned from XPath**:
If this option is selected, and the XPath expression returns no nodes, the filter returns false. If this option is *not* selected, and the XPath returns no nodes, the filter returns true, and no nodes are removed.

**Save deleted nodes to be reinserted to new location**:
You can use this option in cases where you want to move XML nodes from one location in the message to another. By selecting this option, the deleted nodes are stored in the `deleted.node.list` message attribute. You can then use the **Add XML Node** filter to insert the deleted nodes back into a different location in the message. For more details, see *Add XML node on page 237*.

## Exceptions

The **Remove XML Node** filter aborts with a `CircuitAbortException` if the content body of the payload is not in valid JSON format.

# 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 to display in a policy.

# Restore message

## Overview

You can use the **Restore Message** filter to restore message content at runtime using a specified selector expression. You can restore the contents of a request message or a response message, depending on where the filter is placed in the policy.

For example, you could use this filter to restore original message content if you needed to manipulate the message for authentication or authorization. Typically, this filter is used with the **Store Message** filter, which is first used to store the original message content. For more details, see *Store message* on page 265.

## Configuration

**Name**:
Enter a suitable name for this filter to display in a policy.

**Selector Expression to retrieve message**:
Enter the selector expression used to restore the message content. Defaults to `${store.content.body}`. For more details on selector expressions, see "Select configuration values at runtime" in the *API Gateway Policy Developer Guide*.

# 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. By default, all messages are routed onwards using the HTTP verb that the API Gateway 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 to display in a policy.

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

# 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.

You can also use the **Set Message** filter to customize SOAP faults that are returned to clients in the case of a failure or exception in the policy. For a detailed explanation of how to use this filter to customize SOAP faults, see *SOAP fault handling* on page 311.

## Configuration

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

1. Enter a name for this filter to display in a policy 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 *Example of using selectors in the message body* on page 264.

   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="axway.com">
      <ticker>ORM.L</ticker>
    </getQuote>
  </soap:Body>
</soap:Envelope>
```

Assuming the user authenticated successfully to the API Gateway, 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>axway</Username>
  </soap:Header>
  <soap:Body>
    <getQuote xmlns="axway.com">
      <ticker>ORM.L</ticker>
    </getQuote>
  </soap:Body>
</soap:Envelope>
```

For more details on selectors, see "Select configuration values at runtime" in the *API Gateway Policy Developer Guide*.

# Store message

## Overview

You can use the **Store Message** filter to store message content in a specified message attribute. You can store the contents of a request message or a response message, depending on where the filter is placed in the policy.

For example, you could use this filter to store the original message content for reuse later if you need to manipulate the message for authentication or authorization. Typically, this filter is used with the **Restore Message** filter, which is then used to restore the original message content. For more details, see *Restore message* on page 263.

## Configuration

**Name**:
Enter a suitable name for this filter to display in a policy.

**Attribute to store message**:
Enter the name of the message attribute used to store the message content. Defaults to `store.content.body`.

# Convert XML to JSON

## Overview

You can use the **XML to JSON** filter to convert an XML document to a JavaScript Object Notation (JSON) document.

For details on the mapping conventions used, go to:

https://github.com/beckchr/staxon/wiki/Mapping-Convention

## Configuration

To configure the **XML to JSON** filter, specify the following fields:

**Name**:
Enter a suitable name to reflect the role of this filter in a policy.

**Automatically insert JSON array boundaries**:
Select this option to attempt to automatically reconstruct JSON arrays from the incoming XML document. This option is selected by default.

**Convert number/boolean/null elements to primitives**:
Select this option to convert number, boolean, or null elements in the incoming XML document to JSON primitive types. This option is selected by default.

When this option is selected, the filter converts an XML number element to a JSON primitive. Otherwise, an XML number element is converted to a JSON text node. For example:

Incoming XML:

```
<number>123.4</number>
```

JSON output with this option selected:

```
"number" :12.4
```

JSON output with this option not selected:

```
"number" :"12.4"
```

Similarly, XML boolean or null elements are converted to JSON primitives if this option is selected.

**Convert namespace declarations**:
Select this option to convert namespace declarations in the incoming XML and add them to the resulting JSON. This option is not selected by default, and any namespace declarations are removed from the resulting JSON.

**Use the following XPath to convert**:
Select an XPath expression to specify which elements of the incoming XML to convert. The options are:

- All elements inside SOAP body (SOAP 1.1 or SOAP 1.2)
- All elements inside SOAP body (SOAP 1.1)
- The entire message

**Note** If the incoming XML document includes the `<?xml multiple>` processing instruction, the JSON array is reconstructed regardless of this option setting. If the XML document does not contain `<?xml multiple>`, and this option is selected, the filter makes an attempt at guessing what should be part of the array by examining the element names.

For more details and examples of `<?xml multiple>` processing instructions, see *Convert JSON to XML* on page 255.

## Exceptions

The **XML to JSON** filter aborts with a `CircuitAbortException` if the JSON parser has a problem parsing the stream converted from XML to JSON.

# Transform with XSLT

## Overview

Extensible Stylesheet Language Transformations (XSLT) is a declarative, XML-based language used to transform XML documents into other XML documents. An XSL stylesheet is used to transform an XML document into another document type. The stylesheet defines how elements in the XML source document should appear in the resulting XML document.

The API Gateway can convert XML data to other data formats using XSL files. For example, an incoming XML message adhering to a specific XML schema can be converted to an XML message adhering to a different schema before it is sent to the destination web service. You can use the **XSLT Transformation** filter to convert the contents of a message using an XSLT stylesheet.

This type of conversion is especially valuable in the web services arena, where a web service might receive SOAP requests from various types of clients, such as browsers, applications, and mobile phones. Each client might send up a different type of SOAP request to the web service. Using stylesheets, the API Gateway can then convert each type of request to the same format. The requests can then be processed in the same fashion.

**Note** The **XSLT Transformation** filter supports XSLT version 1.0 by default.

# Configuration

Configure the following fields for the **XSLT Transformation** filter:

**Name**:
Enter a suitable name to reflect the role of this filter in a policy.

## *Stylesheet location settings*

On the **Stylesheet Location** tab, select an XSL stylesheet from the **Stylesheet Location** list, which is populated with the contents of the Stylesheets library.

To import a new stylesheet into the library, click the **View/Import** button, and click **Add** on the dialog that appears. Alternatively, you can add stylesheets under the **Resources > Stylesheets** node in the Policy Studio tree.

You can also modify existing stylesheets in the **XSLT Contents** text area of the dialog. Click the **Update** button to update them in the API Gateway configuration.

## *Stylesheet parameter settings*

You can pass parameters to an XSL stylesheet using specified values in `<xsl:param>` elements. These values are then used in the templates defined throughout the stylesheet.

Using the **XSLT Transformation** filter, you can pass the values of message attributes to the configured stylesheet. For example, you can take the value of the `authentication.subject.id` message attribute, pass it to the configured XSL stylesheet, and then output this value to the result produced by the conversion.

To use this feature, select the **Use Message Attributes as Stylesheet Parameters** check box, and click **Add** to specify the message attribute to pass to the stylesheet.

The following example from an XSL stylesheet that uses parameters shows how to configure this:

```
<xsl:param name="authentication.subject.id"/>
<xsl:param name="authentication.issuer.id"/>
```

To pass the corresponding message attribute values to the stylesheet, you must add the `authentication.subject.id` and `authentication.issuer.id` message attributes to the **Message Attributes to use** table.

**Note** The name of the specified parameter must be a valid API Gateway message attribute name, and there *must* be an equivalent parameter name in the stylesheet.

## *Advanced settings*

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

**Provider class name**:
Enter the fully qualified name of the XSLT provider class of the XSLT library to be used. This class *must* be added to the API Gateway's classpath. If this field is left blank, the default provider is used.

**Tip** The simplest way to add a provider class to the API Gateway's classpath is to drop the required JAR file into the `INSTALL_DIR/apigateway/ext/lib` directory, where `INSTALL_DIR` refers to the root of your API Gateway installation.

**Result will be XML**:
You can convert an incoming XML message to other data formats. Select this option if the result of the XSLT conversion is always XML. If not, the content-type of the result document depends on the output method of the XSLT stylesheet. For example, if the stylesheet specifies an output method of HTML (`<xsl:output method="html">`), this field should be left blank so that the API Gateway can forward on the HTML output document to the target web service.

**Do not change the content type header**:
You can select whether to change the HTTP `Content-Type` header in this XSLT transformation. This setting is selected by default, so the content type is preserved.

# Encryption filters 10

The following filters enable API Gateway to encrypt and decrypt messages:



# Generate key

## Overview

The **Generate Key** filter enables you to generate an asymmetric key pair, or a symmetric key. The generated keys are placed in message attributes, which are then available for consumption by other filters.

A typical use case for this filter is in conjunction with the **Security Token Service Client** filter. For example, you wish to request a SAML token with a symmetric proof-of-possession key from an STS. You need to provide the key material to the STS as a binary secret, which is the private key of an asymmetric key pair. You can use an asymmetric private key generated on-the-fly instead of from the Certificate Store with an associated certificate.

You must configure the **Generate Key** filter in a **Security Token Service Client** filter policy that runs before the WS-Trust request is created. You can then configure the **Security Token Service Client** filter to consume the generated asymmetric private key. For more details, see *STS client authentication* on page 111.

**Note** An asymmetric key pair generated by the **Generate Key** filter can also be used by the **Security Token Service Client** filter when a proof-of-possession key of type `PublicKey` is requested. The generated public key can be used as the `UseKey` in the request to the STS.

## Configuration

Complete the following fields to configure this filter:

**Name**:
Enter an appropriate name for the filter to display in a policy.

**Key Type**:
Select the key type from the drop-down list. Defaults to `RSA Asymmetric Key Pair`. You can also select `Symmetric Key`, which is based on Hash-based Message Authentication Code - Secure Hash Algorithm (HMAC-SHA1).

**Key Size**:
Enter the key size in bits. Defaults to `2048` bits.

# JWT decrypt filter

## Overview

You can use the **JWT Decrypt** filter to decrypt encrypted JWTs.

Upon successful decryption, the filter removes all metadata, such as headers and encryption-specific information of the incoming encrypted JWT, and outputs the originally encrypted payload.

For example, when you decrypt the following JWE Payload:

```
eyJhbGciOiJSU0EtT0FFUCIsImVuYyI6IkEyNTZHQ00ifQ.
OKOawDo13gRp2ojaHV7LFpZcgV7T6DVZKTyKOMTYUmKoTCVJRgckCL9kiMT03JGe
ipsEdY3mx_etLbbWSrFr05kLzcSr4qKAq7YN7e9jwQRb23nfa6c9d-StnImGyFDb
Sv04uVuxIp5Zms1gNxKKK2Da14B8S4rzVRltdYwam_lDp5XnZAYpQdb76FdIKLaV
mqgfwX7XWRxv2322i-vDxRfqNzo_tETKzpVLzfiwQyeyPGLBIO56YJ7eObdv0je8
1860ppamavo35UgoRdbYaBcoh9QcfylQr66oc6vFWXRcZ_ZT2LawVCWTIy3brGPi
6UklfCpIMfIjf7iGdXKHzg.
48V1_ALb6US04U3b.
5eym8TW_c8SuK0ltJ3rpYIzOeDQz7TALvtu6UG9oMo4vpzs9tX_EFShS8iB7j6ji
SdiwkIr3ajwQzaBtQD_A.
XFBoMYUZodetZdvTiFvSkQ
```

The output is:

```
The true sign of intelligence is not knowledge but imagination.
```

**Note** The JWT Decrypt filter automatically detects whether the input JWT is encrypted with symmetric or asymmetric key and automatically uses the corresponding settings. For example, you can configure decryption with symmetric key and certificate; however, the filter uses the former or latter depending on the type of JWE it receives as input.

# General settings

Configure the following field on the **JWT Decrypt** window:

**Name**:

Enter an appropriate name for the filter to display in a policy.

**Token location**:

Enter the selector expression to obtain the JWT to be decrypted.

# Decryption using key selection

Optionally, configure the following fields in the **Key selection** section:

**X509 certificate**:

Select the certificate from the certificate store that is used to decrypt the payload.

**Selector expression**:

Alternatively, enter a selector expression to retrieve the alias of the certificate in the certificate store.

# Shared key selection details

Optionally, configure the following fields in the **Shared key selection** section:

**None**:

Select if you do not want to decrypt tokens that are encrypted with shared keys.

**Shared key**:

Enter the shared key that is used to encrypt the payload. The key should be given as a base64-encoded byte array.

**Selector expression**:

Alternatively, enter a selector expression to obtain the shared key. The value returned by the selector should contain:

- Byte array (possibly produced by a different filter)
- Base64-encoded byte array

# JWT encrypt filter

## Overview

You can use the **JWT Encrypt** filter to encrypt arbitrary content (for example, a JWT claims set). The result of the encryption is called JSON Web Encryption (JWE).

JWE represents encrypted content using JSON-based data structures. The JWE cryptographic mechanisms encrypt and provide integrity protection for an arbitrary payload.

A JWE represents the following logical values:

- JOSE Header
- JWE Encrypted Key
- JWE Initialization Vector
- JWE Ciphertext
- JWE Authentication Tag

The encrypted content is outputted in JWE Compact Serialization format, which is produced by base64-encoding the logical values and concatenates them with a period (`'.'`) in between:

```
BASE64URL(UTF8(JWE Protected Header)) ,.'

BASE64URL(JWE Encrypted Key) ,.'

BASE64URL(JWE Initialization Vector) ,.'

BASE64URL(JWE Ciphertext) ,.'

BASE64URL(JWE Authentication Tag)
```

For example, when you use the RSAES-OAEP algorithm and AES GCM encryption method to encrypt the following content (the JWE Payload):

```
The true sign of intelligence is not knowledge but imagination.
```

The following string is returned:

```
eyJhbGciOiJSU0EtT0FFUCIsImVuYyI6IkEyNTZHQ00ifQ.
OKOawDo13gRp2ojaHV7LFpZcgV7T6DVZKTyKOMTYUmKoTCVJRgckCL9kiMT03JGe
ipsEdY3mx_etLbbWSrFr05kLzcSr4qKAq7YN7e9jwQRb23nfa6c9d-StnImGyFDb
Sv04uVuxIp5Zms1gNxKKK2Da14B8S4rzVRltdYwam_lDp5XnZAYpQdb76FdIKLaV
mqgfwX7XWRxv2322i-vDxRfqNzo_tETKzpVLzfiwQyeyPGLBIO56YJ7eObdv0je8
1860ppamavo35UgoRdbYaBcoh9QcfylQr66oc6vFWXRcZ_ZT2LawVCWTIy3brGPi
6UklfCpIMfIjf7iGdXKHzg.
48V1_ALb6US04U3b.
```

```
5eym8TW_c8SuK0ltJ3rpYIzOeDQz7TALvtu6UG9oMo4vpzs9tX_EFShS8iB7j6ji
SdiwkIr3ajwQzaBtQD_A.
XFBoMYUZodetZdvTiFvSkQ
```

# General settings

Configure the following field on the **JWT Encrypt** window:

**Name**:

Enter an appropriate name for the filter to display in a policy.

# Encryption details

Configure the following fields **Encryption details** section:

**Token location**:

Enter a selector expression to retrieve the payload to be signed. The content can be JWT claims, encrypted token, or you can enter a different option.

**Key type**:

Select whether to encrypt using a asymmetric key, symmetric key, or JSON Web Key.

# Asymmetric key details

If you have selected the asymmetric key type, configure the following fields in the **Asymmetric** section:

**X509 certificate**:

Select the certificate from the certificate store that is used to encrypt the payload.

**Selector expression**:

Alternatively, enter a message attribute containing the alias of the certificate in the certificate store.

**Algorithm**:

Select the algorithm to use for encryption.

**Encryption method**:

Select the method to use for encryption.

# Symmetric key details

If you have selected the symmetric key type, configure the following fields on the **Symmetric** section:

**Shared key**:

Enter the shared key that should be used to encrypt the payload. The key should be given as a base64-encoded byte array and must must have a specific length depending on the algorithm selected for encryption:

| Algorithm | Key length |
|---|---|
| AES key wrap with default initial value of 128 bit key (A128KW) | 16 bytes (128 bits) |
| AES key wrap with default initial value of 192 bit key (A192KW) | 24 bytes (192 bits) |
| AES key wrap with default initial value of 256 bit key (A256KW) | 32 bytes (256 bits) |
| Key wrap with AES GCM of 128-bit key (A128GCMKW) | 16 bytes (128 bits) |
| Key wrap with AES GCM of 192-bit key (A192GCMKW) | 24 bytes (192 bits) |
| Key wrap with AES GCM of 256-bit key (A256GCMKW) | 32 bytes (256 bits) |
| Direct use of a shared symmetric key (dir) | 32 bytes (256 bits) |

**Selector expression**:

Alternatively, enter a selector expression to retrieve the shared key. The value returned by the selector should contain:

- Byte array (possibly produced by a different filter)
- Base64-encoded byte array

**Algorithm**:

Select the algorithm used to encrypt.

**Encryption method**:

Select the method used tp encrypt.

# JWK details

If you have selected the JWK key type, complete the following fields in the **JWK details** section:

**JSON Web key**:

Enter a selector expression to retrieve the JSON Web Key used to encrypt the payload. JWKs can be retrieved with the Connect to URL filter.

**Algorithm**:

Select the algorithm to use for encryption.

- To use the algorithm specified in the JWK, select **Always use JWK algorithm**. If no algorithm is specified within the JWK, the filter will fail.
- To always use the algorithm specified in the JWK (if one is defined), but default to the algorithm specified in the **Algorithm** field (if no algorithm is specified in the JWK), select **Use JWK algorithm if available, but default to the selected algorithm below if not**.
- To always use a specific algorithm (regardless if an algorithm is specified in the JWK), select **Always use the algorithm selected below**.

**Note** The encryption method should always be specified.

**Encryption method**:

Select the method to use for encryption.

## FIPS restrictions

The following algorithms are not supported when API Gateway is in FIPS mode:

- Key wrap with AES GCM using 128-bit key (A128GCMKW)
- Key wrap with AES GCM using 192-bit key (A192GCMKW)
- Key wrap with AES GCM using 256-bit key (A256GCMKW)

The following encryption methods are not supported when API Gateway is in FIPS mode:

- AES GCM using 128-bit key (A128GCM)
- AES GCM using 192-bit key (A192GCM)
- AES GCM using 256-bit key (A256GCM)

# PGP decrypt and verify

## Overview

You can use the **PGP Decrypt and Verify** filter to decrypt a message encrypted with Pretty Good Privacy (PGP). This filter decrypts an incoming message using the specified PGP private key, and creates a new message body using the specified content type. The decrypted message can be processed by API Gateway, and then encrypted again using the **PGP Encrypt and Sign** filter.

An example use case for this filter would be when files are sent to API Gateway over Secure Shell File Transfer Protocol (SFTP) in PGP-encrypted format. API Gateway can use the **PGP Decrypt and Verify** filter to decrypt the message, and then use threat detection filters to perform virus scanning. The clean files can be PGP-encrypted again using the **PGP Encrypt and Sign** filter before being sent over SFTP to their target destination. For more details, see *PGP encrypt and sign* on page 279.

You can also use the **PGP Decrypt and Verify** filter to verify signed messages passing through the API Gateway pipeline. Signed messages received by API Gateway can be verified by validating the signature using the public PGP key of the message signer.

**Note** PGP decryption and verification require two different keys: your own private key for decryption, and the sender's public key for verification.

# Configuration

Complete the following fields to configure this filter:

**Name**:
Enter an appropriate name for this filter to display in a policy.

**Decrypt**:
Select whether to use this filter to PGP decrypt an incoming message with a private key.

**PGP Private Key to be retrieved from one of the following locations**:
If you selected the **Decrypt** option, select the location of the private key from one of the following options:

- **PGP Key Pair list**:
  Click the browse button on the right, and select a PGP key pair configured in the certificate store. If no PGP key pairs have already been configured, right-click **PGP Key Pairs**, and select **Add PGP Key**. For details on configuring PGP key pairs, see "Manage X.509 certificates and keys" in the *API Gateway Policy Developer Guide*.
- **Alias**:
  Enter the alias name used to look up the PGP key in the certificate store (for example, `My PGP Test Key`). Alternatively, you can enter a selector expression with the name of a message attribute that contains the alias. The value of the selector is expanded at runtime (for example, `${my.pgp.test.key.alias}`).
- **Message attribute**:
  Enter a selector expression with the name of the message attribute that contains the key. The value of the selector is expanded at runtime (for example, `${my.pgp.test.private.key}`).

For more details on selectors, see "Select configuration values at runtime" in the *API Gateway Policy Developer Guide*.

**Verify**:
Select whether to use this filter to verify an incoming signed message with the public key used to sign the message.

**Verification Key Location (Public Key)**:
If you selected the **Verify** option, select the location of the public key from one of the following options:

- **PGP Key Pair list**:
  Click the browse button on the right, and select a PGP key pair configured in the certificate store. If no PGP key pairs have already been configured, right-click **PGP Key Pairs**, and select **Add PGP Key**. For details on configuring PGP key pairs, see "Manage X.509 certificates and keys" in the *API Gateway Policy Developer Guide*.
- **Alias**:
  Enter the alias name used to look up the PGP key in the certificate store (for example, `My PGP Test Key`). Alternatively, you can enter a selector expression with the name of a message attribute that contains the alias. The value of the selector is expanded at runtime (for example, `${my.pgp.test.key.alias}`).
- **Message attribute**:
  Enter a selector expression with the name of the message attribute that contains the key. The value of the selector is expanded at runtime (for example, `${my.pgp.test.public.key}`).

For more details on selectors, see "Select configuration values at runtime" in the *API Gateway Policy Developer Guide*.

**Signing Method**:
If you selected to verify but not decrypt the incoming message, select a signing method from one of the following options:

- **Compressed**:
  Verifies a compressed signature. Because the message is contained in the signature, this signature is used in place of the message. This is the default.
- **Clear signed**:
  In a clear signed message, the message is intact with a signature attached beneath the clear message text. Verifying this message verifies the sender and the message integrity.
- **Detached signature (MIME)**:
  Verifies a multipart MIME document where the message is in clear text and the signature is attached as a MIME part.

**Decrypt and Verify Method**:
If you selected to decrypt and verify the incoming message, select the decrypt and verify method from one of the following options:

- **Decrypt and Verify in One Pass**:
  Decrypts and verifies the message in a single pass. This is the default. API Gateway decrypts the message while reading the data packet, and continues on sequentially when it reaches the signature packet.
- **Decrypt and Verify in Two Passes**:
  Decrypts the message in the first pass, and then verifies the signature in the second pass. Use this option when the message has been encrypted and signed in two passes.

**Content type**:
Enter the `Content-Type` of the unencrypted message data. Defaults to `application/octet-stream`.

# PGP encrypt and sign

## Overview

You can use the **PGP Encrypt and Sign** filter to generate a Pretty Good Privacy (PGP)-encrypted message. This filter enables you to configure the PGP public key used when encrypting the message body (the `content.body` attribute). You can also configure advanced options such as whether the message outputs ASCII armor, or whether it uses a symmetrically encrypted integrity protected data packet to protect against modification attacks.

**PGP Encrypt and Sign** filter does not encrypt any files attached to your message.

For example, using the default options, the **PGP Encrypt and Sign** filter creates a PGP-encrypted message such as the following, and overwrites the `content.body` attribute with the encrypted version:

```
-----BEGIN PGP MESSAGE-----
Version:BCPG v1.46

hQIOA3ePizxHLIA8EAgAmVNAgJO7TXI9vWCJHZS27r4FIfZIYWNc0+MiQ3H+LZrW
29Wageetg5N7cFAbRpG28iKYSE5O0uFMThuWuhnMZ/GtRwMogiRsNyBY0Cq0LKaG
7oIbkjWE1BHdWXQLWW44zYl8ekTWJ4ZPNCemTtHyULB9QwuWx5b6QfAyh1jvFrSN
ub9mQzU8caY7xQrVgWii1tBFOzTcGw6/Vb7AtMZfwGGjqmzYLT5pLozWUKB0gZe1
/7wpWDsHsn+53lrRXdoqwvAhY2AnOLPyrrVsykXS38YtIh9N5D+uCCvgmICej9Ok
iieh1hgGnuzw3VdQ6n3TS0t/Xk3shB95I3IkXU1l4ggAjPSnf9qEuN2u8dsRooNR
J6nYWs/OGwBSj0/MtssoRAcEVYu93tITUXqybduq8CATHGD8at4WRiTLOcndgJTp
0aUU+aOi3l3SsnrlpPSKIu18K9AqFCE+lafdXKlqU2OaGMBbsU22Vy6SkDgSXuGg
EOG0KYHRdrAntHJiO3qrJRTd8BDrPc5PZWUwDfCUWuQRMJiJVp0bxrK8Qzz/Ni7T
XgRSL1cYzAQRsQAbne69On+5n+NWO1Qcx9SnSimBtPOQXJfff+a+Wb45ABj5TdYr
4PCd2OJ0uOapSWfiSA5mZ1sB9hEAR0FidXs1iAploun1qggNYZK94BGGXblnTCzR
ccnARKqWmWanr1VVnp6fs9WI6I3zkiCGTJlQMNa0UKZdEPe1wJb7NHgJFMKrwN1X
3rSVyUWiovnYYMlDGBHG/RWGsTSd0LT7VugtIByefCI2G7WevgLUJbOq+U/0Sh6A
82oMNuWbXbDTp3pfZae/SHqOyEdDp5zsGqZ/F4M7CQFx63XCIBsFA6JRj6GdYqYf
dewuej3WJtRDdHmikjb3o7Utl8fFhKjA9GdEZueG9ls+XcAx21iBT656HRof8wio
oSca8ui3SYbhZ+0uzwImDJ0054P3Xr24+iwI4vlKjiQNY23GjXsVa2rQn6VHT60o
CYo08tDYBH4gyetLAqczCVyh6sff9SqX=qkB0

-----END PGP MESSAGE-----
```

For an example use case, see *PGP decrypt and verify* on page 276.

You can also use the **PGP Encrypt and Sign** filter to digitally sign messages passing through the API Gateway pipeline. Messages signed by API Gateway can be verified by the recipient by validating the signature using the public PGP key of the signer. Signed messages received by API Gateway can be verified in the same way.

**Note** PGP encryption and signing require two different keys: a public key for encryption and your own private key for signing.

# General settings

Complete the following field on the **PGP Encrypt and Sign** window:

**Name**:
Enter an appropriate name for the filter to display in a policy.

# Encrypt and sign settings

Complete the following fields on the **Encrypt and Sign** tab:

**Encrypt**:
Select whether to use this filter to PGP encrypt an outgoing message with a public key.

**Encryption Key location (Public Key)**:
If you selected the **Encrypt** option, select the location of the public key from one of the following options:

- **PGP Key Pair list**:
  Click the browse button on the right, and select a PGP key pair configured in the certificate store. If no PGP key pairs have already been configured, right-click **PGP Key Pairs**, and select **Add PGP Key**. For details on configuring PGP key pairs, see "Manage X.509 certificates and keys" in the *API Gateway Policy Developer Guide*.
- **Alias**:
  Enter the alias name used to look up the PGP key in the certificate store (for example, `My PGP Test Key`). Alternatively, you can enter a selector expression with the name of a message attribute that contains the alias. The value of the selector is expanded at runtime (for example, `${my.pgp.test.key.alias}`).
- **Message attribute**:
  Enter a selector expression with the name of the message attribute that contains the key. The value of the selector is expanded at runtime (for example, `${my.pgp.test.public.key}`).

For more details on selectors, see "Select configuration values at runtime" in the *API Gateway Policy Developer Guide*.

**Sign**:
Select whether to use this filter to sign an outgoing message with a private key.

**Signing Key location (Private Key)**:
If you selected the **Sign** option, select the location of the private key from one of the following options:

- **PGP Key Pair list**:
  Click the browse button on the right, and select a PGP key pair configured in the certificate store. If no PGP key pairs have already been configured, right-click **PGP Key Pairs**, and select **Add PGP Key**. For details on configuring PGP key pairs, see "Manage X.509 certificates and keys" in the *API Gateway Policy Developer Guide*.
- **Alias**:
  Enter the alias name used to look up the PGP key in the certificate store (for example, `My PGP Test Key`). Alternatively, you can enter a selector expression with the name of a message attribute that contains the alias. The value of the selector is expanded at runtime (for example, `${my.pgp.test.key.alias}`).
- **Message attribute**:
  Enter a selector expression with the name of the message attribute that contains the key. The value of the selector is expanded at runtime (for example, `${my.pgp.test.private.key}`).

For more details on selectors, see "Select configuration values at runtime" in the *API Gateway Policy Developer Guide*.

**Signing Method**:
If you selected to sign but not encrypt the outgoing message, select the signing method from one of the following options:

- **Compressed**:
  Compresses the message and creates a hash of the contents before signing. Because the message is contained within the signature this signature can be used in place of the message. The typical use of this method produces a signature in printable ASCII form (ASCII Armor). This option can be turned off to produce a binary signature.
- **Clear signed**:
  Clear signing a message leaves the message intact and adds the signature beneath the clear message text. This provides for optional verification of the message signature and contents. The output has the content type `application/pgp-signature`. It is not possible to clear sign binary objects.
- **Detached signature (MIME)**:
  Creates a multipart MIME document where the message remains in clear text and the signature is attached as a MIME part.

**Encrypt and Sign Method**:
If you selected to encrypt and sign the outgoing message, select the encrypt and sign method from one of the following options:

- **Encrypt and Sign in One Pass**:
  Encrypts and signs the message in a single pass. This is the default setting. This enables the receiver to decrypt and verify in a single pass.

- **Encrypt and Sign in Two Passes**:
  Signs the message in a first pass, and then encrypts it in a second pass. Because the message is signed before it is encrypted, this means that the message must be decrypted first before the signature can be verified.

## Advanced settings

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

**ASCII Armor Output**:
Select whether to output the binary message data as ASCII Armor. ASCII Armor is a special text format used by PGP to convert binary data into printable ASCII text. ASCII Armored data is especially suitable for use in email messages, and is also known as Radix-64 encoding. This option is selected by default.

**Symmetric Encrypted Integrity Protected Data Packet**:
Select whether the message uses a Symmetrically Encrypted Integrity Protected Data packet. This is a variant of the Symmetrically Encrypted Data packet, and is used detect modifications to the encrypted data. This option is selected by default.

**Symmetric Key Algorithm**:
Select a symmetric-key algorithm to use to encrypt the data. The default is `CAST5`.

**Hash Algorithm**:
Select a hash algorithm to use to protect against modification. The default is `SHA1`.

**Compression Algorithm**:
Select a compression algorithm to use to compress the data. The default is `ZIP`.

# SMIME decryption

## Overview

The **SMIME Decryption** filter can be used to decrypt an encrypted Secure/Multipurpose Internet Mail Extensions (SMIME) message.

See also *SMIME encryption* on page 283.

## Configuration

Complete the following fields to configure this filter:

**Name**:
Enter a name for the filter to display in a policy.

**Use Certificate to Decrypt**:
Check the box next to the certificate that you want to use to decrypt the encrypted PKCS#7 message with. The private key associated with this certificate is used to actually decrypt the message.

# SMIME encryption

## Overview

You can use the **SMIME Encryption** filter to generate an encrypted Secure/Multipurpose Internet Mail Extensions (SMIME) message. This filter enables you to configure the certificates of the recipients of the encrypted message. You can also configure advanced options such as ciphers and Base64 encoding.

See also *SMIME decryption* on page 282.

## General settings

Complete the following field:

**Name**:
Enter an appropriate name for the filter to display in a policy.

## Recipient settings

The **Recipients** tab enables you to configure the certificates of the recipients of the encrypted SMIME message. Select one of the following options:

**Use the following certificates**:
This is the default option. Select the certificates of the recipients of the encrypted message. The public keys associated with these certificates are used to encrypt the data so that it can only be decrypted using the associated private keys.

**Certificate in attribute**:
Alternatively, enter the message attribute that contains the certificate of the recipients of the encrypted message. Defaults to the `certificate` message attribute.

## Advanced settings

The **Advanced** tab includes the following settings:

**Cipher**:
Enter the cipher that you want to use to encrypt the message data. Defaults to the `DES-EDE3-CBC` cipher.

**Content-Type**:
Enter the `Content-Type` of the message data. Defaults to `application/pkcs7-mime`.

**Base64 encode**:
Select whether to Base64 encode the message data. This option is not selected by default.

# 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.

See also *XML encryption* on page 290.

## Configuration

Enter an appropriate name for the filter to display in a policy.

## 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* on page 285. When finished, an **XML-Decryption Settings** filter is created along with an **XML-Decryption** filter.

# XML decryption settings

## Overview

The API Gateway 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.

**Note** The output of a successfully executed decryption filter is the original unencrypted message. Depending on whether the **Remove Encrypted Key used in decryption** has been enabled, all information relating to the encryption key can be removed from the message. For more details, see *Options* on page 289.

See also *XML decryption* on page 284.

## 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 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 Axway:

```
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
  <s:Body>
    <getCompanyInfo xmlns="www.axway.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 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 using the following approaches:

- Decrypt all encrypted nodes
- Use XPath to select encrypted nodes

Configure one of the following settings:

**Decrypt All**:
The API Gateway 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 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 *XML encryption wizard* on page 304, 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 Axway 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 Axway 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 Axway 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" in the *API Gateway Policy Developer Guide*.

**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 look up 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 settings 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 Encrypted Key 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.

**Note** 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 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.

See also *XML encryption* on page 290.

## Configuration

Enter a suitable name for the filter to display in a policy.

## 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 *XML encryption wizard* on page 304.

# XML encryption settings

The API Gateway 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.

See also *XML encryption* on page 290.

## 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.

### *Elements*

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

```
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
  <s:Body>
    <getCompanyInfo xmlns="www.axway.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.

## *Asymmetric and symmetric cryptography*

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.

**XML-Encryption Settings** filter requires a preceding filter to populate the message attribute with the symmetric key. To add the preceding filter, right-click **XML-Encryption Settings** filter, select **Create Predecessor**, and select the filter to populate the message attribute.

The following filters generate the `symmetric.key` message attribute:

- HTTP Header
- SMIME Verify
- XML Signature Verification
- XML Signature Generation
- Insert SAML Authorization Assertion
- SSL
- XML Decryption

The most typical example is to sign the message with a symmetric key using an **XML Signature Generation** filter before encrypting it with **XML Encryption Filter** using the settings from the **XML Encryption Settings** filter. You can configure the **XML Signature Generation** filter to generate a symmetric key to sign the message symmetrically and automatically populate the `symmetric.key` message attribute with the generated key.

Now that you understand the structure and mechanics of XML Encryption, you can configure the API Gateway 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:

**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" in the *API Gateway Policy Developer Guide*.

**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 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 using 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 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 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:X509Data>
      <dsig:X509SubjectName>CN=Sample,C=IE...</dsig:X509SubjectName>
      <dsig:X509Certificate>MIIEZDCCA0yg...RNp9aKD1fEQgJ</dsig:X509Certificate>
    </dsig:X509Data>
  </dsig:KeyInfo>
</enc:EncryptedData>
```

**Put Certificate in an Attachment**:
The API Gateway 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
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 `<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 list.

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

**Note** When using the Kerberos Token Profile standard, and the API Gateway 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 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, 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 (*axway_1* and *axway_2*). The encryption key has been encrypted twice: once for *axway_1* using its public key, and a second time for *axway_2* using its public key:

**Note** 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 axway_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>axway_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 axway_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>axway_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 is to 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 fora 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 recipient (Recipient2), you would specify certificate_recip2 as the location of the encryption key.

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

# What to encrypt settings

The **What to Encrypt** tab is used to identify parts of the message that must be encrypted. Each encrypted part is 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. For more details on how to use these node selectors, see *Locate XML nodes* on page 441.

**Note** There is a difference between encrypting the element and encrypting its content. When encrypting the element, the entire element is replaced by the `<EncryptedData>` block. This is not recommended if you wish to encrypt the SOAP Body. If this element is removed from the SOAP message, the message may not be 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 does 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 **Encrypt Node Content**.

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 *XML encryption wizard* on page 304.

# XML encryption wizard

## Overview

The following filters are involved in encrypting a message using XML encryption:

| Filter | Role | Topic |
|---|---|---|
| Find Certificate | Specifies the certificate that contains the public key to use in the encryption. The data is encrypted such that it can only be decrypted with the corresponding private key. | *Find certificate* on page 177 |
| XML-Encryption Settings | Specifies the recipient of the encrypted data, what data to encrypt, what algorithms to use, and other such options that affect the way the data is encrypted. | *XML encryption settings* on page 291 |
| XML-Encryption | Performs the actual encryption using the certificate selected in the **Find Certificate** filter, and the options set in the **XML-Encryption Settings** filter. | *XML encryption* on page 290 |

While these filters can be configured independently of each other, it makes sense to configure them all at the same time because they must play a role in the policy that XML-encrypts messages. You can do this using the **XML Encryption Wizard**.

The wizard is available by right-clicking the name of the policy in the tree view of the Policy Studio, and selecting the **XML Encryption Settings** menu option. The next section describes how to configure the settings on this dialog.

# 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. 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**.

When the wizard is completed, the information configured on this screen results in the auto-generation of a **Find Certificate** filter. This filter is automatically configured to use the selected certificate from the Certificate Store. For more details, see *Find certificate* on page 177.

After clicking **Next** on the first screen of the wizard, the configuration options for the **XML-EncryptionSettings** filter are displayed. For more details, see *XML encryption settings* on page 291.

When you have completed all the steps in the wizard, a policy is created that comprises a **Find Certificate**, **XML-Encryption Settings**, and **XML-Encryption** filter. You can insert other filters into this policy as required. However, the order of the encryption filters must be maintained as follows:

- Find Certificate
- XML-Encryption Settings
- XML-Encryption

# Fault handling filters 11

The following filters enable you to handle faults and errors:



# Generic error handling

In cases where a transaction fails, API Gateway can use a *generic error* to convey error information to the client based on the message type (SOAP or JSON). By default, API Gateway returns a very basic error when a message filter fails. You can add the **Generic Error** filter to a policy to return more meaningful error information based on the message type.

When the **Generic Error** filter is configured, API Gateway examines the incoming message and attempts to infer the type of message to be returned.

For example, for an incoming SOAP message, it sends an appropriate SOAP response (SOAP 1.1 or 1.2) using the SOAP fault processor. For an incoming JSON message, it sends an appropriate JSON response. If the inference process fails, API Gateway sends a SOAP message by default.

For security reasons, it is good practice to return as little information as possible to the client. However, for diagnostic reasons, it is useful to return as much information to the client as possible. Using the **Generic Error** filter, administrators have the flexibility to configure just how much information to return to clients, depending on their individual requirements.

## General settings

Configure the following settings on the **General** tab:

**Name**:
Enter an appropriate name for this filter to display in a policy.

**HTTP Response Code Status**:
Enter the HTTP response code status for this **Generic Error** filter. This ensures that a meaningful response is sent to the client in the case of an error occurring in a configured policy. Defaults to `500` (`Internal Server Error`). For a complete list of status codes, see the HTTP Specification.

### *Generic error contents*

Configure the following options in the **Generic Error Contents** section:

**Show detailed explanation of error**:
Returns a detailed explanation of the generic error in the error message. This makes it possible to suppress the reason for the exception in a tightly locked down system (the reason is displayed as `message blocked` in the generic error). Defaults to the value of the `${circuit.failure.reason}` message attribute selector.

**Show filter execution path**:
Returns a generic error containing the list of filters run on the message before the error occurred. For each filter listed in the generic error, the status is given (`pass` or `fail`).

**Show stack trace**:
Return the Java stack trace for the error to the client. This option should only be enabled under instructions from Axway Support.

**Show current message attributes**:
Return the message attributes present at the time the generic error was generated to the client. For example, for an incoming SOAP message, each message attribute forms the content of a `<fault:attribute>` element.

**Caution** For security reasons, do not use **Show filter execution path**, **Show stack trace**, and **Show current message attributes** in a production environment.

**Use Stylesheet**:
Select this option to transform the error message returned by the **Generic error** filter by applying an XSLT stylesheet. Click the browse button next to the **Stylesheet** field. Select an existing stylesheet from the list, or right-click the **Stylesheets** node to add a new stylesheet.

Because XSLT stylesheets accept XML as input, API Gateway implicitly transforms the incoming message into XML. Next, it retrieves the selected XSLT stylesheet and applies the transformation to the message, and sends the response in the format specified.

# Create customized generic errors

Use the following approaches to create customized generic errors:

## *Use the Generic Error filter*

On the **Advanced** tab, you can customize the generation of a response message when the request cannot be inferred as a SOAP or JSON request.

**Apply Response Rules**: When this setting is selected, you can select one of the following rules:

- **Delegate to Policy**: Select a policy to generate the response message. For example, the policy could use a **JSON Error** filter to generate a JSON response.
- **Response Content**: Set the response body content and headers directly in the respective text areas. You can use the API Gateway selector syntax to evaluate and expand request details at runtime. The values specified on this tab are used in the outbound request to the URL.
    - **Body**: Enter the content of the incoming request message body (body headers and body content). Defaults to `${content.body}`.

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>
```

- **Headers**: Enter the HTTP headers associated with the incoming request message. Defaults to `${http.headers}`.

### *Use the Set Message filter*

You can also use the **Set Message** filter to create customized generic errors. The **Set Message** filter can change the contents of the message body to any arbitrary content. When an exception occurs in a policy, you can use this filter to customize the body of the generic error.

For details on how to use the **Set Message** filter to generate customized faults and return them to the client, see the example in *SOAP fault handling* on page 311. You can use the same approach to generate customized generic errors.

# JSON error handling

## Overview

In cases where a JavaScript Object Notation (JSON) transaction fails, the API Gateway can use a *JSON error* to convey error information to the client. By default, the API Gateway returns a very basic fault to the client when a message filter fails.

You can add the **JSON Error** filter to a policy to return more meaningful error information to the client. For example, the following message extract shows the format of a JSON error raised when a JSON Schema Validation filter fails:

```
{
    "reasons":[
       {
          "language":"en",
          "message":"JSON Schema Validation filter failed"
       }
    ],
    "details":{
       "msgId":"Id-f5aab7304f6c754804f70000",
       "exception message":"JSON Schema Validation filter failed",
        ...
    }
}
```

**Note** For security reasons, it is good practice to return as little information as possible to the client. However, for diagnostic reasons, it is useful to return as much information to the client as possible. Using the **JSON Error** filter, administrators have the flexibility to configure just how much information to return to clients, depending on their individual requirements.

For more details on JSON schema validation, see *JSON schema validation* on page 195. For more details on JSON, see http://www.json.org/index.html.

# General settings

Configure the following general settings:

**Name**:
Enter an appropriate name for this filter to display in a policy.

**HTTP Response Code Status**:
Enter the HTTP response code status for this JSON error filter. This ensures that a meaningful response is sent to the client in the case of an error occurring in a configured policy. Defaults to `500` (`Internal Server Error`). For a complete list of status codes, see the HTTP Specification.

# JSON error contents

The following configuration options are available in the **JSON Error Contents** section:

**Show detailed explanation of error**:
Select this option to return a detailed explanation of the JSON error in the error message. This makes it possible to suppress the reason for the exception in a tightly locked down system. By default, the reason is displayed as `message blocked` in the JSON error. This option displays the value of the `${circuit.failure.reason}` message attribute selector.

**Show filter execution path**:
Select this option to return the list of filters run on the message before the error occurred. For each filter listed in the JSON Error, the status is output (`Pass` or `Fail`). The following message extract shows a *filter execution path* returned in a JSON error:

```
"path" :{
   "policy" :"test_policy",
   "filters" :[ {
      "name" :"True Filter",
      "status" :"Pass"
   }, {
    "name" :"JSON Schema Validation",
    "status" :"Fail",
   "filterMessage" :"Filter failed"
   }, {
    "name" :"Generic Error",
```

```
    "status" :"Fail",
    "filterMessage" :"Filter failed"
   } ]
},
```

**Show stack trace**:
Select this option to return the Java stack trace for the error to the client. This option should only be enabled under instructions from Axway Support.

**Show current message attributes**:
Select this option to return the message attributes present when the JSON error is generated to the client. The value of each message attribute is output as shown in the following example:

```
"attributes":[
   {
    "name":"circuit.exception",
    "value":"com.vordel.circuit.CircuitAbortException:JSON Schema Validation filter
failed"
   },
   {
    "name":"circuit.failure.reason",
    "value":"JSON Schema Validation filter failed"
   },
   {
    "name":"content.body",
    "value":"com.vordel.mime.JSONBody@185afba1"
   },
   {
    "name":"failure.reason",
    "value":"JSON Schema Validation filter failed"
   },
   {
    "name":"http.client",
    "value":"com.vordel.dwe.http.ServerTransaction@7d3e1384"
   },
   {
    "name":"http.headers",
    "value":"com.vordel.mime.HeaderSet@76737f58"},
   {
    "name":"http.response.info",
    "value":"ERROR"
   },
   {
    "name":"http.response.status",
    "value":"500"
   },
   {
    "name":"id",
    "value":"Id-f5aab7304f6c754804f70000"
   },
   {
```

```
    "name":"json.errors",
    "value":"org.codehaus.jackson.JsonParseException: Unexpected character
     ('\"' (code 34)):was expecting comma to separate OBJECT entries\n at
     [Source:com.vordel.dwe.InputStream@592c34b; line:3, column:25]"
    },
...
]
```

**Caution** For security reasons, **Show filter execution path**, **Show stack trace**, and **Show current message attributes** should not be used in a production environment.

## Create customized JSON errors

You can use the following approaches to create customized JSON errors:

### *Use the Generic Error filter*

Instead of using the **JSON Error** filter, you can use the **Generic Error** filter to transform the JSON error message returned by applying an XSLT stylesheet. The **Generic Error** filter examines the incoming message and infers the type of message to be returned (for example, JSON or SOAP). You can use the **Advanced** tab to customize the generation of a response message if the request cannot be inferred as a SOAP or JSON request.

For more details, see *Generic error handling* on page 306.

### *Use the Set Message filter*

You can create customized JSON errors using the **Set Message** filter with the **JSON Error** filter. The **Set Message** filter can change the contents of the message body to any arbitrary content. When an exception occurs in a policy, you can use this filter to customize the body of the JSON error.

For details on how to use the **Set Message** filter to generate customized faults and return them to the client, see the example in *SOAP fault handling* on page 311. You can use the same approach to generate customized JSON errors.

# SOAP fault handling

## Overview

In cases where a typical SOAP transaction fails, a *SOAP fault* can be used to convey error information to the SOAP client. The following message shows the format of a SOAP fault:

```
<?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>Receiver</env:Value>
            <env:Subcode>
                <env:Value>policy failed</env:Value>
            </env:Subcode>
         </env:Code>
         <env:Detail xmlns:axwayfault="axway.com/soapfaults"
axwayfault:type="exception" type="exception"/>
      </env:Fault>
   </env:Body>
</env:Envelope>
```

By default, the API Gateway returns a very basic SOAP fault to the client when a message filter fails. You can add the **SOAP Fault** filter to a policy to return more complicated error information to the client.

For security reasons, it is good practice to return as little information as possible to the client. However, for diagnostic reasons, it is useful to return as much information to the client as possible. Using the **SOAP Fault** filter, administrators have the flexibility to configure just how much information to return to clients, depending on their individual requirements.

# SOAP fault format settings

The following configuration options are available in the **SOAP Fault Format** section:

**SOAP Version**:
Select the appropriate SOAP version. You can send either a SOAP Fault 1.1 or 1.2 response to the client.

**Fault Namespace**:
Select the default namespace to use in SOAP faults, or enter a new one if necessary.

**Indent SOAP Fault**:
If this option is selected, an XSL stylesheet is run over the SOAP fault to indent nested XML elements. The indented SOAP fault is returned to the client.

# SOAP fault content settings

The following configuration options are available in the **SOAP Fault Contents** section:

**Show Detailed Explanation of Fault**:
Select this option to return a detailed explanation of the SOAP fault in the fault message. This makes it possible to suppress the reason for the exception in a tightly locked down system (the reason is displayed as `message blocked` in the SOAP fault).

**Show Filter Execution Path**:
Select this option to return a SOAP fault containing the list of filters run on the message before the error occurred. For each filter listed in the SOAP fault, the status is given (`pass` or `fail`). The following message shows a *filter execution path* returned in a SOAP fault:

```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
   <env:Header></env:Header>
   <env:Body>
      <env:Fault>
         <env:Code>
            <env:Value>Receiver</env:Value>
            <env:Subcode>
                <env:Value>policy failed</env:Value>
            </env:Subcode>
         </env:Code>
         <env:Detail xmlns:axwayfault="http://www.axway.com/soapfaults"
             axwayfault:type="exception" type="exception">
             <axwayfault:path>
                <axwayfault:visit node="HTTP Parser"
status="Pass"></axwayfault:visit>
                <axwayfault:visit node="/services" status="Fail"></axwayfault:visit>
                <axwayfault:visit node="/status" status="Fail"></axwayfault:visit>
             </axwayfault:path>
         </env:Detail>
       </env:Fault>
   </env:Body>
</env:Envelope>
```

**Show Stack Trace**:
Select this option to return the Java stack trace for the error to the client. This option should only be enabled under instructions from Axway Support.

**Show Current Message Attributes**:
Select this option to return the message attributes present at the time the SOAP fault was generated to the client. Each message attribute forms the content of a `<fault:attribute>` element, as shown in the following example:

```
<fault:attributes>
   <fault:attribute name="circuit.failure.reason" value="null">
   <fault:attribute name="circuit.lastProcessor" value="HTTP Digest">
   <fault:attribute name="http.request.clientaddr" value="/127.0.0.1:4147">
   <fault:attribute name="http.response.status" value="401">
   <fault:attribute name="http.request.uri" value="/authn">
   <fault:attribute name="http.request.verb" value="POST">
   <fault:attribute name="http.response.info" value="Authentication Required">
   <fault:attribute name="circuit.name" value="Digest AuthN">
</fault:attributes>
```

# Create Customized SOAP faults

You can use the following approaches to create customized SOAP faults:

## Use the Generic Error filter

Instead of using the **SOAP Fault** filter, you can use the **Generic Error** filter to transform the SOAP fault message returned by applying an XSLT stylesheet. The **Generic Error** filter examines the incoming message and infers the type of message to be returned (for example, JSON or SOAP). You can use the **Advanced** tab to customize the generation of a response message if the request cannot be inferred as a SOAP or JSON request.

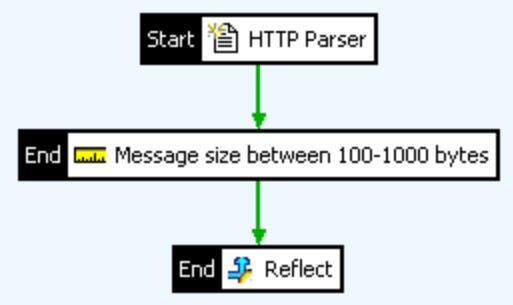
For more details, see *Generic error handling* on page 306.

## Use the Set Message filter

You can create customized SOAP faults using the **Set Message** filter with the **SOAP Fault** filter. The **Set Message** filter can change the contents of the message body to any arbitrary content. When an exception occurs in a policy, you can use this filter to customize the body of the SOAP fault. The following example demonstrates how to generate customized SOAP faults and return them to the client.

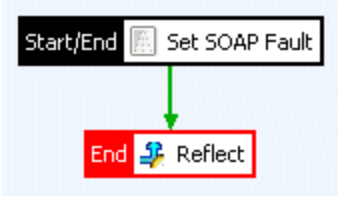
### Step 1: Create the top-level policy

This example first creates a very simple policy called **Main Policy**. This policy ensures the size of incoming messages is between 100 and 1000 bytes. Messages in this range are echoed back to the client.



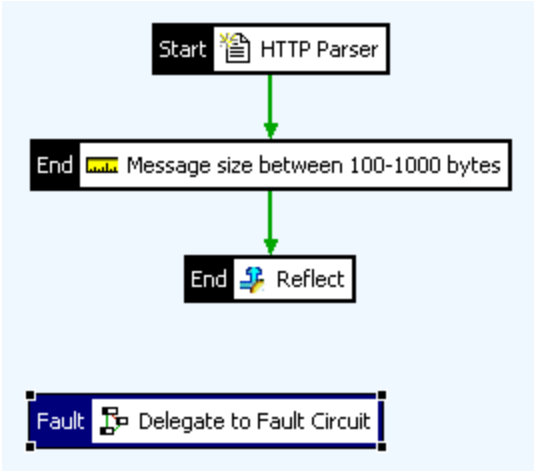

### Step 2: Create the fault policy

Next, create a second policy called **Fault Circuit**. This policy uses the **Set Message** filter to customize the body of the SOAP fault. When configuring this filter, enter the contents of the customized SOAP fault to return to clients in the text area provided.




### *Step 3: Create a shortcut to the fault policy*

Add a **Policy Shortcut** filter to the **Main Policy** and configure it to refer to the **Fault Circuit**. Do *not* connect this filter to the policy. Instead, right-click the filter, and select **Set as Fault Handler**. The **Main Policy** is displayed as follows:




### *How this example works*

Assume a 2000-byte message is received by the API Gateway and is passed to the **Main Policy** for processing. The message is parsed by the **HTTP Parser** filter, and the size of the message is checked by the **Message Size** filter. Because the message is greater than the size constraints set by this filter, and because there is no failure path configured for this filter, an exception is thrown.

When an exception is thrown in a policy, it is handled by the designated *fault handler*, if one is present. In the **Main Policy**, a **Policy Shortcut** filter is set as the fault handler. This filter delegates to the **Fault Circuit**, meaning that when an exception occurs, the **Main Policy** invokes (or delegates to) the **Fault Circuit**.

The **Fault Circuit** consists of two filters, which play the following roles:

1. **Set Message**:
   This filter is used to set the body of the message to the contents of the customized SOAP fault.
2. **Reflect**:
   When the SOAP fault has been set to the message body, it is returned to the client using the **Reflect** filter.

# Integrity filters 12

The following filters enable you to ensure the integrity of messages:



# JWT Sign

## Overview

You can use the **JWT Sign** filter to sign arbitrary content (for example, a JWT claims set). The result is called JSON Web Signature (JWS).

JWS represents digitally signed or MACed content using JSON data structures and base64url encoding.

A JWS represents the following logical values:

- JOSE Header
- JWS Payload
- JWS Signature

The signed content is outputted in JWS Compact Serialization format, which is produced by base64 encoding the logical values and concatenating them with periods ( `.` ) in between. For example:

```
{"iss":"joe",
„exp":1300819380,
„http://example.com/is_root":true}
```

When the JOSE Header, JWS Payload, and JWS Signature is combined as follows:

```
BASE64URL(UTF8(JWS Protected Header)) '.'

BASE64URL(JWS Payload) '.'

BASE64URL(JWS Signature)
```

The following string is returned:

```
eyJhbGciOiJSUzI1NiJ9
.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFt
cGxlLmNvbS9pc19yb290Ijp0cnVlfQ
.
cC4hiUPoj9Eetdgtv3hF80EGrhuB__dzERat0XF9g2VtQgr9PJbu3XOiZj5RZmh7
AAuHIm4Bh-0Qc_lF5YKt_O8W2Fp5jujGbds9uJdbF9CUAr7t1dnZcAcQjbKBYNX4
BAynRFdiuB—f_nZLgrnbyTyWzO75vRK5h6xBArLIARNPvkSjtQBMHlb1L07Qe7K
0GarZRmB_eSN9383LcOLn6_dO—xi12jzDwusC-eOkHWEsqtFZESc6BfI7noOPqv
hJ1phCnvWh6IeYI2w9QOYEUipUTI8np6LbgGY9Fs98rqVt5AXLIhWkWywlVmtVrB
p0igcN_IoypGlUPQGe77Rw
```

# General settings

Configure the following settings on the **JWT Sign** window:

**Name**:

Enter an appropriate name for the filter to display in a policy.

# Signing details

Configure the following fields in the **Signing details** section:

**Token location**:

Enter the selector expression to obtain the payload to be signed. The content can be JWT claims, encrypted token, or you can enter a different option.

**Key type**:

Select whether to sign with a private (asymmetric) key or HMAC (symmetric key).

# Asymmetric key details

If you selected the asymmetric key type, configure the following fields in the **Asymmetric** section:

**Signing key**:

Select the private key from the certificate store that is used to sign the payload.

**Selector expression**:

Alternatively, enter a selector expression to get the alias of the private key in the certificate store.

**Algorithm**:

Select the algorithm used to sign.

## Symmetric key details

If you selected the symmetric key type, complete the following fields **Symmetric** section:

**Shared key**:

Enter the shared key used to sign the payload. The key should be given as a base64-encoded byte array and must use the following minimum lengths depending on the selected algorithm used to sign:

| Algorithm | Minimum key length |
|---|---|
| HMAC using SHA-256 (HS256) | 32 bytes (256 bits) |
| HMAC using SHA-384 (HS384) | 48 bytes (384 bits) |
| HMAC using SHA-512 (HS512) | 64 bytes (512 bits) |

**Selector expression**:

Alternatively, enter a selector expression to obtain the shared key. The value returned from the selector should contain:

- Byte array (possibly produced by a different filter)
- Base64-encoded byte array

**Algorithm**:

Select the algorithm used to sign.

# JWT Verify

## Overview

You can use the **JWT Verify** filter to verify a signed JSON Web Token (JWT) with the token payload. Upon successful verification, the **JWT Verify** filter removes the headers and signature of the incoming signed JWT and outputs the original JWT payload. For example, when you verify the following signed JWT payload input:

```
eyJhbGciOiJSUzI1NiJ9
.
eyJpc3MiOiJqb2UiLA0KICJleHAiOjEzMDA4MTkzODAsDQogImh0dHA6Ly9leGFt
cGxlLmNvbS9pc19yb290Ijp0cnVlfQ
.
cC4hiUPoj9Eetdgtv3hF80EGrhuB__dzERat0XF9g2VtQgr9PJbu3XOiZj5RZmh7
AAuHIm4Bh-0Qc_lF5YKt_O8W2Fp5jujGbds9uJdbF9CUAr7t1dnZcAcQjbKBYNX4
```

```
BAynRFdiuB-f_nZLgrnbyTyWzO75vRK5h6xBArLIARNPvkSjtQBMHlb1L07Qe7K
0GarZRmB_eSN9383LcOLn6_dO-xi12jzDwusC-eOkHWEsqtFZESc6BfI7noOPqv
hJ1phCnvWh6IeYI2w9QOYEUipUTI8np6LbgGY9Fs98rqVt5AXLIhWkWywlVmtVrB
p0igcN_IoypGlUPQGe77Rw
```

The resulting payload output is:

```
{"iss":"joe",
"exp":1300819380,
"http://example.com/is_root":true}
```

**Note** The **JWT Verify** filter automatically detects whether the input JWT is signed with hash-based message authentication code (HMAC) or asymmetric key and uses the corresponding settings as appropriate. For example, you can configure verification with HMAC or certificate, depending on the type of JWT received as input.

# General settings

Configure the following settings on the **JWT Verify** dialog:

**Name**:
Enter an appropriate name for the filter to display in a policy.

**Token location**:
Enter the selector expression to retrieve the JWT to be verified. This must contain the value of token in the format of `HEADER.PAYLOAD.SIGNATURE`, but without the `Bearer` prefix. You can use a filter such as *Retrieve from HTTP header* on page 54 in your policy to get the token from any header. For example: `${http.headers["Authorization"].substring(7)}`

# Verify using RSA/EC public key (asymmetric)

You can configure the following optional settings in the **Verify using RSA/EC public key** section:

**X509 certificate**:
Select the certificate that is used to verify the payload from the API Gateway certificate store.

**Note** Asymmetric keys are associated with the x509 certificate, but for verification, you only need the public key, which is encoded in the certificate. Alternatively, you can use a JSON Web Key (JWK) with a **Connect to URL** filter to download the key from a known source. For more details, see *JWK from external source* on page 320.

**Selector expression**:
Alternatively, enter a selector expression to retrieve the alias of the certificate from the API Gateway certificate store.

# Verify using symmetric key

You can configure the following optional settings in the **Verify using symmetric key** section:

**None**:
Select if you do not want to verify tokens signed with HMAC.

**Shared key**:
Enter the shared key that was used to sign the payload. The key should be provided as a base64-encoded byte array.

**Selector expression**:
Alternatively, enter a selector expression to obtain the shared key. The value returned by the selector should contain:

- Byte array (possibly produced by a different filter)
- Base64-encoded byte array

# JWK from external source

You can configure the following optional setting in the **JWK from external source** section:

**JSON web key**:
You can verify signed tokens using a selector expression containing the value of a `JSON Web Key (JWK)`. The return type of the selector expression must be of type String. For more details on converting a JSON contained in the body to a string value, see "Access configuration values dynamically at runtime" in the *API Gateway Developer Guide*.

# Additional JWT verification steps

The **JWT Verify** filter verifies the JWT signature with the token payload only. The following additional verification steps are also typically required:

- Make sure that the certificate used to generate the signature is valid (for example, check that it is not blacklisted or expired). You can use the API Gateway CRL and OCSP filters in your policy for this step (for example, see the *Dynamic CRL certificate validation* on page 166 and *OCSP client* on page 178 filters).
- Validate the JWT token claims. For example, this includes the following checks:
  - `aud`: Audience—check that the token has been created for the correct user.
  - `iss`: Issuer—check that the token was issued by a trusted token provider.
  - `exp`: Expiry time—check that the token has not already expired.

# Sign SMIME message

## Overview

You can use the **SMIME Sign** filter to digitally sign a multipart Secure/Multipurpose Internet Mail Extensions (SMIME) message when it passes through the API Gateway core pipeline. The recipient of the message can then verify the integrity of the SMIME message by validating the Public Key Cryptography Standards (PKCS) #7 signature.

See also *Verify SMIME message* on page 321.

## Configuration

Complete the following fields:

**Name**:
Enter an appropriate name for the filter to display in a policy.

**Sign Using Key**:
Select the certificate that contains the public key associated with the private signing key to be used to sign the message.

**Create Detached Signature in Attachment**:
Specifies whether to create a detached digital signature in the message attachment. This is selected by default. For example, this is useful when the software reading the message does not understand the PKCS#7 binary structure, because it can still display the signed content, but without verifying the signature.

If this is not selected, the message content is embedded with the PKCS#7 binary signature. This means that user agents that do not understand PKCS#7 cannot display the signed content. Intermediate systems between the sender and final recipient can modify the text content slightly (for example, line wrapping, whitespace, or text encoding). This might cause the message to fail signature validation due to changes in the signed text that are not malicious, nor necessarily affecting the meaning of the text.

# Verify SMIME message

## Overview

You can use the **SMIME Verify** filter to check the integrity of a Secure/Multipurpose Internet Mail Extensions (SMIME) message. This filter enables you to verify the Public Key Cryptography Standards (PKCS) #7 signature over the message.

You can select the certificates that contain the public keys to be used to verify the signature. Alternatively, you can specify a message attribute that contains the certificate with the public key to be used.

See also *Sign SMIME message* on page 321.

## Configuration

Complete the following fields:

**Name**:
Enter an appropriate name for the filter to display in a policy.

**Certificates from the following list**:
Select the certificates that contain the public keys to be used to verify the signature. This is the default option.

**Certificate in attribute**:
Alternatively, enter the message attribute that specifies the certificate that contains the public key to be used to verify the signature. Defaults to `${certificate}`.

**Remove Outer Envelope if Verification is Successful**:
Select this option to remove the PKCS#7 signature and all its associated data from the message if it verifies successfully.

# XML signature generation

The API Gateway 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.

You can configure the **XML Signature Generation** filter to generate a symmetric key to sign the message symmetrically, and automatically populate the `symmetric.key` message attribute with the generated, so that a successive filter, for example, the **XML-Encryption Settings** filter, can use that symmetric key. For more details, see *XML encryption settings* on page 291.

For more details on XML signature validating the integrity of the message, see *XML signature verification* on page 345.

## 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 X.509 certificates and keys" in the *API Gateway Policy Developer Guide*.

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, or environment variable). For more details, see "Select configuration values at runtime" in the *API Gateway Policy Developer Guide*.

## 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 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 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, or environment variable). For more details, see "Select configuration values at runtime" in the *API Gateway Policy Developer Guide*.

**Include Encrypted Symmetric Key in Message**:
The symmetric key is typically encrypted for the recipient and included in the message. However, the initiator and recipient of the transaction may 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 to encrypt the symmetric key with a public key from the certificate store. Click **Signing Key** and 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 can 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 setting 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, or environment variable). For more details, see "Select configuration values at runtime" in the *API Gateway Policy Developer Guide*.

**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, instead of 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 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, and 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 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 selecting the appropriate setting.

```
<dsig:Signature xmlns:dsig="http://www.w3.org/2000/09/xmldsig#" id="Sample">
  ...
  <dsig:KeyInfo>
    <dsig:KeyName>test@axway.com</dsig:KeyName>
  </dsig:KeyInfo>
</dsig:Signature>
```

**Put Certificate in an Attachment**:
The API Gateway 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 contained in an attachment, and not in 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 in 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 sent, the certificate attachment is 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 refers to the `Content-Id` of the attachment:

```
POST /adoWebSvc.asmx HTTP/1.0
Content-Length: 3790
User-Agent: API Gateway
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>` (in the `<dsig:KeyInfo>`) can contain a `<wsse:Embedded>` security token. Alternatively, the `<wsse:SecurityTokenReference>`, (in 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.

**Note** When using the Kerberos Token Profile standard and the API Gateway is acting as the initiator of a secure transaction, it can use Kerberos session keys to sign 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 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 is 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. For details on the settings on these tabs, see the following sections:

- *Node locations* on page 336
- *XPaths* on page 336
- *XPath predicates* on page 337
- *Message attribute* on page 338

**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 is 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://www.axway.com">
      <vs:Name>API Gateway</vs:Name>
      <vs:Version>7.6.2</vs:Version>
    </vs:getProductInfo>
  </s:Body>
  </s:Envelope>
```

In this 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="7.6.2">
      <name>API Gateway</name>
      <company>axway</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="7.6.2">
      <name>API Gateway</name>
      <company>Axway</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=7.6.2>
      <name>API Gateway</name>
      <company>Axway</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=7.6.2>
      <name>API Gateway</name>
      <company>Axway</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>
```

## Node locations

Node locations are perhaps the simplest way to configure the message content that must be signed. The table on this tab is prepopulated 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.

## XPaths

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 as follows:

```
<?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.axway.com">
      <name>SOA Product</name>
      <company>Company</company>
      <description>Web services Security</description>
    </product>
  </soap:Body>
</soap:Envelope>
```

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/ |

## *XPath predicates*

Select this option 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 `Sign SOAP Body` predicate is selected:

```
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
  <s:Body>
    <vs:getProductInfo xmlns:vs="http://www.axway.com">
      <vs:Name>API Gateway</vs:Name>
      <vs:Version>7.6.2</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://www.axway.com">
      <vs:Name>API Gateway</vs:Name>
      <vs:Version>7.6.2</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 **Locate XML Nodes** filter (see *Locate XML nodes* on page 441) 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 Selector Expression** check box, and enter the name of the attribute that contains the nodes in the field provided. The default value is `${node.list}`.

## Where to place signature settings

**Append Signature to Root or SOAP Header**:
If the message is a SOAP message, the signature is inserted into the SOAP `Header` element when this radio button is selected. The XML signature is 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>
  <dsig:Signature xmlns:dsig="http://www.w3.org/2000/09/..." id="Sample">
...
  </dsig:Signature>
 </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 is 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 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
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 uses 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 are then 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 is the initiator as outlined in the scenario above. The API Gateway 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 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 are 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 breaks.

**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.

**Include Transforms**:
Select this option to include a `<Transforms>` element in the `<Reference>` block to explain the transformation chain applied before hashing the content.

**Note** This option must be selected when **Create Enveloped Signature** is selected.

# XML signature verification

## Overview

In addition to validating XML signatures for authentication purposes, the API Gateway 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. Therefore by validating the XML signature on a message, the API Gateway can guarantee the *integrity* of the message.

See also *XML signature generation* on page 322.

## 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 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 on signature location options, see the *API Gateway Policy Developer Guide*.

**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" in the *API Gateway Policy Developer Guide*.
- **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 **Environment Configuration** > **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 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 settings* on page 329.

**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 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 acts as the recipient in the transaction, select the **Recipient** option. In this case, the API Gateway returns the `SignatureConfirmation` elements in the response to the initiator.

**Default Derived Key Label**:
If the API Gateway 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 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 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, or environment variable).

For more details, see "Select configuration values at runtime" in the *API Gateway Policy Developer Guide*.

**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.

# Monitoring filters 13

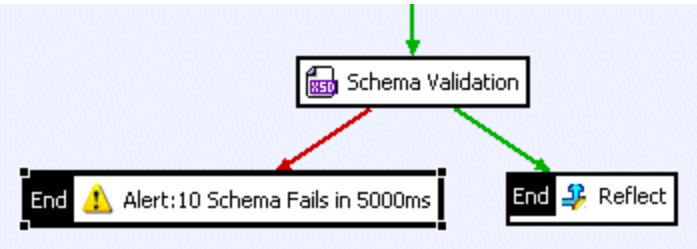
The following filters enable you to configure alerts, logging, and integration with Axway Sentinel:



# Alert

You can use the **Alert** filter to send system alerts to different alert destinations, for example, when another filter fails, or in API firewalling when ModSecurity detects a threat. For more details on system alerts and alert destinations, see "Configure system alerts" in the *API Gateway Policy Developer Guide*.

Typically, an **Alert** filter is placed on the failure path of another filter in the policy. In the following example, an alert is configured if a schema validation fails 10 times within a 5000 millisecond period for a specified web service, and the **Alert** filter is placed on the failure path from the **Schema Validation** filter:




When editing policies, you can drag and drop the **Alert** filter onto the policy canvas from the **Monitoring** category.

## General settings

Configure the following settings on the **Alert** filter window:

- **Name**: Enter a descriptive name for the filter to display in a policy, especially more complicated policies might have several alert filters so good names make it easier to tell them apart.

- **Alert Type**: Select the severity level of the alert (**Error**, **Warn**, or **Info**). This option is only relevant for alert destinations that support severity levels, such as the Windows Event Log.

# Notifications settings

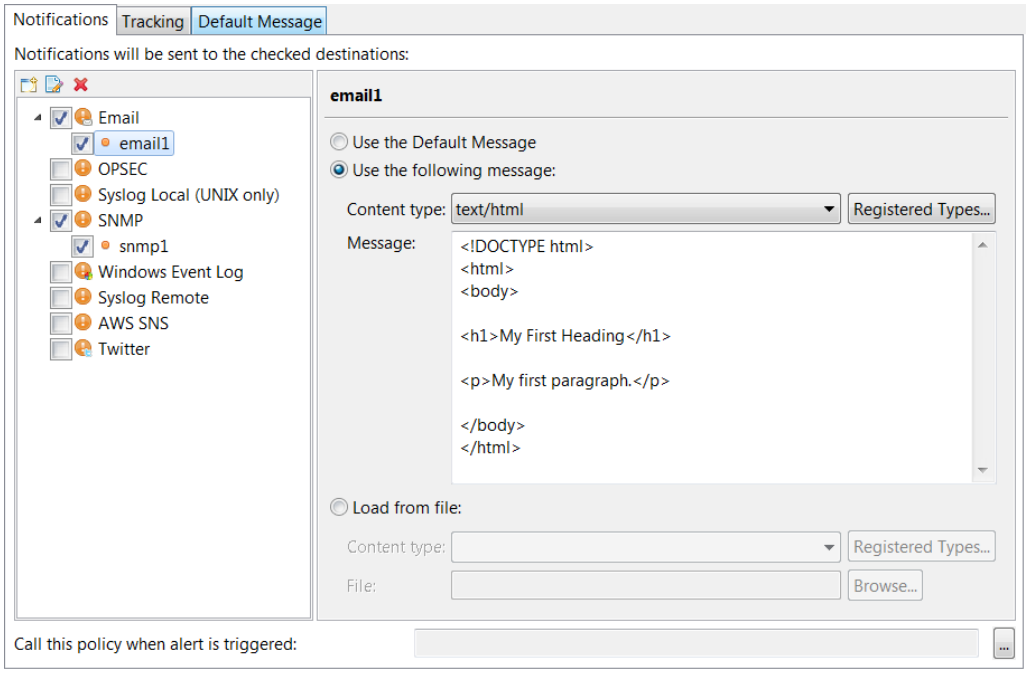
You can configure the alert destination on the **Notifications** tab. The alert destinations you have already configured are nested under the respective alert destination type on the left of the window. You can also use the icons above the alert destination listing to create, edit, or delete alert destinations directly from this window. For more details, see "Configure alert destinations" in the *API Gateway Policy Developer Guide*.

To select an alert destination, follow these steps:

1. Select an alert destination or all alert destinations of a given alert destination type (for example, all **Email** destinations).
2. Set the message that is sent to the alert destination:
    - Select **Use the following message**, set the content type, and enter a custom message in the text box.
    - Select **Use the Default Message**, and go to the **Default Message** tab to write the message you want to send.
    - Select **Load from file**, set the content type, and browse to the file you want to use (this option is available for email destinations only).

      To register new MIME content types, click the **Registered Types** button.

The following figure shows an example of an email alert destination with a `text/html` custom message.



# Tracking settings

Use the **Tracking** tab to configure how often alerts are sent. Configure the following:

- **Accumulated number of messages**: Enter how many times this filter can be invoked before the alert is sent. The default value is `1`.
- **In time period (secs)**: Enter the time period over which the accumulated number of messages can occur before an alert is triggered. The default is `60` seconds.
- **Track per client**: Select this option to record the accumulated number of messages in the specified time period for each client. This option is selected by default.

# Default message

Enter the default message used for alerts in **Message to send** on the **Default Message** tab.

**Note** An alert message is not required for alerts sent to an OPSEC firewall.

You can include message attributes using selectors that API Gateway looks up and expands at runtime. For example, instead of sending a generic alert stating `Authentication Failed`, you can use a message attribute to include the ID of the user whose authentication failed:

```
Authentication failure for user:${authentication.subject.id}.

${alert.number.failures} authentication failures have occurred in
${alert.time.period} seconds.

${alert.number.failures} exceptions have occurred in policy ${circuit.name}.

The last exception was ${circuit.exception} with path ${circuit.path}.
```

You can also use the default message to pass information to other systems. For example, the ModSecurity threat reports in API firewalling are stored in the message attribute `modsec.error.message` in addition to being written to the trace files. You can configure an alert policy to capture the detected threats, and use an Alert filter with a selector for `modsec.error.message` in the default message to send the threat report to third-party monitoring systems.

For more information on selectors, see "Select configuration values at runtime" in the *API Gateway Policy Developer Guide*.

# Log message payload

## Overview

The **Log Message Payload** filter is used to log the message payload at any point in the policy. The message payload includes the HTTP headers and MIME attachments.

By placing the **Log Message Payload** filter at various key locations in the policy, a complete audit trail of the message can be achieved. For example, by placing the filter after each filter in the policy, the complete history of the message can be logged. This is especially useful in cases where the message has been altered by the API Gateway (for example, by signing or encrypting the message, inserting security tokens, or by converting the message to another grammar using XSLT).

Log messages can be stored in several locations, including a database, a file, or the system console. For more details on configuring logging destinations, see the *API Gateway Administrator Guide*.

See also *Set transaction log level and log message* on page 361.

## Configuration

Enter an appropriate name for the **Log Message Payload** filter in the **Name** field. It is good practice to use descriptive names for these filters. For example, **Log message before signing message** and **Log message after signing** would be useful names to give to two **Log Message Payload** filters that are placed before and after a **Sign Message** filter.

By default, the **Log Message Payload** filter writes entries to the log file in the following format:

```
${timestamp} ${id} ${filterName} ${payload}
```

However, you can alter the format of the logging output using the values entered in the **Format** field. You can use selectors to output logging information that is specific to the request. You can specify the following properties:

- **`level`**:
  The log level (i.e. fatal, fail, success).
- **`id`**:
  The unique transaction ID assigned to the message.
- **`ip`**:
  The IP address of the client that sent the request.
- **`timestamp`**:
  The time that the message was processed in user-readable form.
- **`filterName`**:
  The name of the filter that generated the log message.

- **`filterType`**:
  The type of the filter that logged the message.
- **`text`**:
  The text of the log message that was configured in the filter itself.
- **`payload`**:
  The complete contents of the HTTP request, including HTTP headers, body, and attachments.

# Send cycle link event to Sentinel

You can use the **Sentinel Cycle Link** filter to send cycle link events to Axway Sentinel. Sentinel uses *cycle links* to link processing cycles sequentially. A *processing cycle* is a group of related tracked events (identified by the same cycle ID).

Every cycle link must specify a parent processing cycle and a child processing cycle. A processing cycle is identified by a cycle ID, and the tracked object name and version are used to identify the tracked events within the processing cycle.

This filter can be used to link related events from different products. For example, if B2Bi and API Gateway are both sending events to the same Sentinel server, each product sends the events with different cycle IDs. You can link the events from B2Bi with the events from API Gateway by sending a cycle link event to the Sentinel server. This links the two cycle IDs in Sentinel.

See also *Send event to Sentinel* on page 353.

## General settings

Configure the following settings on the **Sentinel Cycle Link** window:

**Name** :
Enter a suitable name for the filter.

**Sentinel Server**:
Click the browse button to select a Sentinel server connection.

The **Parent Settings** section enables you to specify the cycle ID for the parent processing cycle. You also need to specify the tracked object name and version to identify the relevant tracked events.

**Parent Cycle ID**:
Enter the cycle ID of the parent processing cycle. This should be the cycle ID of the upstream product (for example, B2Bi). For example, `${http.headers["X-TRACKING-CYCLEID"]}` retrieves the parent cycle ID from the HTTP request headers.

**Use the following tracked object**:
Select this option and click the browse button to select a Sentinel tracked object to use. If no tracked objects are already defined, right-click **Sentinel Tracked Objects** in the dialog and select **Add a tracked object**. Enter a **Name** and a **Version** for the tracked object. The values entered must correspond to the **Public name** and **Version** of the tracked object in Sentinel.

**Or use the following from the message**:
Select this option to retrieve the tracked object name and tracked object version from a message received from an upstream product (for example, B2Bi). If the upstream product has inserted tracking information inside some HTTP headers, you can use selectors to retrieve these from the message.

For example, `${http.headers["X-TRACKEDOBJECT-NAME"]}` retrieves the tracked object name from the HTTP headers, and `${http.headers["X-TRACKEDOBJECT-IDENTITY"]}` retrieves the tracked object version from the HTTP headers.

The **Child Settings** section enables you to specify the cycle ID for the child processing cycle. You also need to specify the tracked object name and version to identify the relevant tracked events.

**Child Cycle ID**:
Enter the cycle ID of the child processing cycle. This should be the cycle ID of API Gateway. For example, enter `${id}` to specify the API Gateway transaction ID.

**Use the following tracked object**:
Select this option and click the browse button to select a Sentinel tracked object to use. If no tracked objects are already defined, right-click **Sentinel Tracked Objects** in the dialog and select **Add a tracked object**. Enter a **Name** and a **Version** for the tracked object. The values entered must correspond to the **Public name** and **Version** of the tracked object in Sentinel.

**Or use the following from the message**:
Select this option to retrieve the tracked object name and tracked object version from a message received from an upstream product (for example, B2Bi). If the upstream product has inserted tracking information inside some HTTP headers, you can use selectors to retrieve these from the message.

For example, `${http.headers["X-TRACKEDOBJECT-NAME"]}` retrieves the tracked object name from the HTTP headers, and `${http.headers["X-TRACKEDOBJECT-IDENTITY"]}` retrieves the tracked object version from the HTTP headers.

## Further information

For more details, see the *API Gateway Sentinel Interoperability Guide*.

# Send event to Sentinel

You can use the **Sentinel Event** filter to send tracked events to Axway Sentinel. Sentinel uses *tracked objects* to identify events. Every tracked object contains a unique name, version number, and a list of attributes.

Every tracked event must specify a tracked object, and this tracked object must already be defined in Sentinel. A tracked event can also contain attributes, and the attributes must already be defined as tracked object attributes in Sentinel.

See also *Send cycle link event to Sentinel* on page 352.

# General settings

Configure the following settings on the **Sentinel Event** window:

**Name**:
Enter a suitable name for the filter to display in a policy.

## *Settings tab*

Configure the following:

**Sentinel Server**:
Click the browse button to select a Sentinel server connection.

The **Tracked object** section enables you to specify the tracked object to use in the Sentinel event.

**Note** Tracked objects must exist in your Sentinel database before you can start using Sentinel to monitor your applications and track their activities. For more information on defining tracked objects in Sentinel, see the *Sentinel Configuration Guide* available on https://support.axway.com.

**Use the following tracked object**:
Select this option and click the browse button to select a Sentinel tracked object to use. If no tracked objects are already defined, right-click **Sentinel Tracked Objects** in the dialog and select **Add a tracked object**.

Enter a **Name** and a **Version** for the tracked object. The values entered must correspond to the **Public name** and **Version** of the tracked object in Sentinel.

**Or use the following from the message**:
Select this option to retrieve the tracked object name and tracked object version from a message received from an upstream product (for example, B2Bi). If the upstream product has inserted tracking information inside some HTTP headers, you can use selectors to retrieve these from the message.

For example, `${http.headers["X-TRACKEDOBJECT-NAME"]}` retrieves the tracked object name from the HTTP headers, and `${http.headers["X-TRACKEDOBJECT-IDENTITY"]}` retrieves the tracked object version from the HTTP headers.

The **Event will contain the following** section enables you to specify the tracked object attributes to use in the Sentinel event.

**Note** The named event attributes specified in this section must be contained within the tracked object definition in Sentinel.

**Include Cycle ID**:
Select this option to include the cycle ID in the event, and enter a value. For example, enter `${id}` to use the API Gateway transaction ID as the cycle ID. This value is used to populate the `CycleId` attribute of the tracked object in Sentinel.

**Include policy name in event named**:
Select this option to include the name of the policy in the event, in an attribute with the specified name. You can use any name for the attribute, as long as the attribute name exists in the tracked object definition in Sentinel.

**Include filter name in event named**:
Select this option to include the name of the filter in the event, in an attribute with the specified name. You can use any name for the attribute, as long as the attribute name exists in the tracked object definition in Sentinel.

You can also add other attributes to be included in the event by populating entries in the table. Click the **Add** button to add an attribute. Enter a **Name** and a **Value** for the attribute. For example, to populate an attribute named `UserName` with the authenticated user, you would enter `UserName` for the **Name** and `${authentication.subject.id}` for the **Value**.

**Send as update**:
Select this option to send the event as an update.

### *Tracking tab*

Configure the following:

**Add Cycle ID in header named**:
Select this option and enter a value, to include the cycle ID in the HTTP header.

**Add Tracked Object name in header named**:
Select this option and enter a value, to include the tracked object name in the HTTP header.

**Add Tracked Object version in header named**:
Select this option and enter a value, to include the tracked object version in the HTTP header.

## Further information

For more details, see the *API Gateway Sentinel Interoperability Guide*.

# Service level agreement

## Overview

A service level agreement (SLA) is an agreement put in place between a web services host and a client of that web service in order to guarantee a certain minimum quality of service. It is common to see SLAs in place to ensure that a minimum number of messages result in a communications failure and that responses are received within an acceptable time frame. In cases where the conditions of the SLA are breached, it is crucial that an alert can be sent to the appropriate party.

When one of the specified thresholds is breached, an alert is sent to a configured alert destination to ensure that interested parties are notified promptly of the SLA breach. For more details on configuring system alerts, see the *API Gateway Policy Developer Guide*.

API Gateway satisfies these requirements by allowing SLAs to be configured at the policy level. You can configure SLAs to monitor the following types of problems:

- Response times
- HTTP status codes returned from the web service
- Communication failures

The SLA monitoring performed by API Gateway is *statistical*. Because of this, a single message (or even a small number of messages) is not considered a sufficient sample to cause an alert to be triggered. The monitoring engine actually uses an *exponential decay* algorithm to determine whether an SLA is failing or not. This algorithm is best explained with an example.

## *Example SLA*

Assume the *poll rate* is set to 3 seconds (3000 ms), the *data age* is set to 6 seconds (6000 ms), and you have a web service with an average processing time of 100 ms. A single client sending a stream of requests through API Gateway can generate about 10 requests per second, given the web service's 100 ms response time.

At every 3 seconds poll period you have data from a previous 30 samples to consider the average response times of. However, rather than simply using the response time of the *last* 3 seconds worth of data, historical data is "smoothed" into the current estimate of the failing percentage. The new data is combined with the existing data such that it will take approximately the data age time for a sample to disappear from the average.

Therefore the closer the data age is to the sampling rate, the less significant historical data becomes, and the more significant the "last" sample becomes.

To generate an alert, you must also have enough significant samples at each poll period to consider the date to be statistically valid. For example, if a single request arrives over a period of 1 hour it might not be fair to say that "less than 20%" of all received requests have failed the response time requirements. For this reason, statistical analysis provides a more realistic SLA monitoring mechanism than a solution based purely on absolute metrics.

# Response time requirements

You can monitor the response times of web services protected by the **SLA Filter**. This filter provides different ways of measuring response times:

| Response Time Measurement | Description |
|---|---|
| **receive-request-start** | The time that the API Gateway receives the first byte of the request from the client. |
| **receive-request-end** | The time that the API Gateway receives the last byte of the request from the client. |
| **send-request-start** | The time that the API Gateway sends the first byte of the request to the web service. |
| **send-request-end** | The time that the API Gateway sends the last byte of the request to the web service |
| **receive-response-start** | The time that the API Gateway receives the first byte of the response from the web service |
| **receive-response-end** | The time that the API Gateway receives the last byte of the response from the web service. |
| **send-response-start** | The time that the API Gateway sends the first byte of the response to the client. |
| **send-response-end** | The time that the API Gateway sends the last byte of the response to the client. |

API Gateway measures each of the 8 time values. They are available for processing after the policy has completed for a single request. These 8 options are available for the following reasons:

- API Gateway might start to send the first byte to the web service before the last byte is received from the client (send-request-start <receive-request-end). This occurs if the invoked policy does not require the full message to be read into memory.
- API Gateway might start to send the response to the client before the complete response has been received from the web service (send-response-start < receive-response-end). This occurs when invoked policy does not require the full message to be read into memory.
- It is possible that the web service might start to send the response before it has received the complete request. However, API Gateway does not start to read the response until it has sent the complete request. This means that the following is always true: send-request-end < receive-response-start.
- The time value for send-response-end depends upon the client application. This value is larger if the client is slow to read the response.

To add a response time requirement for an SLA, click **Add**.

To configure the start time and end time for the response time measurement, click **Add**. On the **Settings** tab, specify the percentage of response times that must be below a specified time interval (in milliseconds) in the fields provided. The purpose of these options is to allow for situations where a very small number of unusually slow requests might cause an SLA to trigger unnecessarily. By using percentages, such requests do not distort the statistics collected by API Gateway.

Click the **Message Text** tab to configure the messages to appear in the alert message when the SLA is breached and also when the SLA is cleared, that is, when the breached conditions are no longer in breach of the SLA.

Finally, click the **Advanced** tab to configure timing information. Select a **Start Timing Point** from the 8 times listed in the table above. API Gateway *starts* measuring the response time from this time. Then select an **End Timing Point** from the 8 times listed in the table above. API Gateway *stops* measuring the response time from this time.

# HTTP status requirements

HTTP status codes might be received from a web service. API Gateway can be configured to monitor these and generate alerts based on the number of occurrences of certain types of status code response. HTTP status codes are three digit codes that can be grouped into standard status classes, with the first digit indicating the status class.

The status classes are as follows:

| HTTP Status Code Class | Description |
|---|---|
| **1xx** | These status codes indicate a provisional response. |
| **2xx** | These status codes indicate that the client's request was successfully received, understood, and accepted. |
| **3xx** | These status codes indicate that further action needs to be taken by the user agent in order to fulfill the request. |
| **4xx** | These status codes are intended for cases in which the client seems to have erred. For example 401, means that authentication has failed. |
| **5xx** | These status codes are intended for cases where the server has encountered an unexpected condition that prevented it from fulfilling the request. For example, 500 is used to transmit SOAP faults. |

API Gateway might monitor a class (that is, range) of status codes, or it might monitor specific status codes. For example, it is possible to configure the following HTTP status code requirements:

- At least 97% of the requests must yield HTTP status codes between 200 and 299
- At most 2% of requests can yield HTTP status codes between 400 and 499
- At most 0% of requests can yield HTTP status code 500

Click **Add** in the **HTTP Status Code Requirements** section.

Select an existing status code or class of status codes from the **HTTP Status Code** field. To add a new code or range of codes, click **Add**.

Enter a name for the new code or range of codes in the **Name** field of the **Configure HTTP Status Code** dialog. Enter the *first* HTTP status code in the range of status codes that you want to monitor in the **Start Status** field. Then enter the *last* HTTP status code in the range of status codes that you want to monitor in the **End Status** field.

To monitor just one specific status code, enter the same code in the **Start Status** and **End Status** fields.

Click **OK** when you are satisfied with the selected range of status codes to return to the previous dialog. The remaining 2 fields allow the administrator to specify the minimum or maximum percentage of received HTTP status codes that fall into the configured range before an alert is triggered.

Again, the use of percentages here is to allow for situations where a very small number of requests return the status codes within the "forbidden" range. By using percentages, such requests do not distort the statistics collected by API Gateway.

Click the **Message Text** tab to configure the messages to appear in the alert message when the SLA is breached and also when the SLA is cleared, (when the breached conditions are no longer in breach of the SLA).

# Communications failure requirements

API Gateway is deemed to have experienced a *communications failure* when it fails to connect to the web service, fails to send the request, or fails to receive the response.

The requirements for communications failures can be expressed as follows:

- No more than 4% of requests can result in communications failures.

Enter the percentage of allowable communications failures in the field provided. An alert is configured if the percentage of communicates failures rises above this level.

Click the **Message Text** tab to configure the messages to appear in the alert message when the SLA is breached and also when the SLA is cleared (when the breached conditions are no longer in breach of the SLA).

# Select alerting system

If an alert is triggered, it must be sent to an alerting destination. API Gateway can send alerts to the following destinations:

- Windows Event Log
- Email Recipient
- SNMP Network Management System
- Local Syslog
- Remote Syslog
- CheckPoint FireWall-1 (OPSEC)
- Twitter

The **Select Alerting System** table at the bottom of the window displays all available alerting destinations that have been configured. You can click **Add** to configure an alert destination. For more details, see the *API Gateway Policy Developer Guide*.

Select one or more alerting systems in the table. An alert is sent to each selected system in the event of a violation of the performance requirements. Alert clearances will be generated when the violation no longer exists.

# Set service context

The **Set Service Context** filter configures service-level monitoring details. For example, you can use the fields on this filter window to configure whether API Gateway stores service usage and service usage per client details. You can also set the name of the service displayed in the web-based API Gateway Manager monitoring tool and the API Gateway Analytics reporting tool.

## General settings

**Name**:
Enter an appropriate name for the filter to be displayed in a policy.

**Service Name**:
Enter an appropriate name for this service to be displayed in the web-based API Gateway Manager tool.

**Monitoring Options**:
The fields in this group enable you to configure whether API Gateway displays usage metrics data for this web service:

- **Enable monitoring**:
  Select this option to enable monitoring for this web service in the **Monitoring** view in API Gateway Manager, and in API Gateway Analytics.
- **Which attribute is used to identify the client**:
  Enter the message attribute to use to identify authenticated clients. The default is `authentication.subject.id`, which stores the identifier of the authenticated user (for example, the user name or user's X.509 Distinguished Name).

- **Composite context**:
  This setting enables you to select a service context as a composite context in which multiple service contexts are monitored during the processing of a message. This setting is not selected by default.

  For example, API Gateway receives a message, and sends it to `serviceA` first, and then to `serviceB` . Monitoring is performed separately for each service by default. However, you can set a composite service context before `serviceA` and `serviceB` that includes both services. This composite service passes if both services complete successfully, and monitoring is also performed on the composite service context.

For more details on monitoring and reporting, see:

- *API Gateway Administrator Guide*
- *API Gateway Analytics User Guide*

# Set transaction log level and log message

## Overview

By default, logging is configured for a service with logging level of failure. You can also configure each filter in a policy to log its own message depending on whether it succeeds, fails, and/or throws an exception. Log messages can be stored in several locations, including a database, a file, or the system console.

Logging levels apply to the following cases:

- A filter succeeds if it returns a true result after carrying out its processing. For example, if an LDAP directory returns an `authorized` result to an authorization filter, the filter succeeds.
- A filter fails if it returns a false result after performing its processing. For example, an authorization filter returns false if an LDAP directory returns a `not authorized` result to the filter.
- A filter aborts when it cannot make the decision it is configured to make. For example, if an LDAP-based authorization filter cannot connect to the LDAP directory, it aborts because it can neither authorize nor refuse access. This is regarded as a *fatal* error.

For more details on configuring logging destinations, see the *API Gateway Administrator Guide*.

See also *Log message payload* on page 351.

## Configuration

You can access the **Transaction Log Level and Message** window by clicking **Next** on the main window of any filter. This window includes the following fields:

**Logging Level**:
Configure one of the following options:

| | |
|---|---|
| **Use Service Level Settings** | This option is selected by default. Logging is configured for the Web service with logging level of **Failure**. |
| **Override Logging Level for this Filter** | Alternatively, select this option to configure log messages for this filter when it succeeds, fails, and/or aborts. Select **Success**, **Failure**, and/or **Fatal** to configure this filter to log at the respective levels. |

**Log Messages**:
Default log message values are provided at each level for all filters. When you select the checkbox for a particular level, the default log message for that level is used. You can specify an alternative log message by entering the message in the text field provided.

All filters *require* and *generate* message attributes, while some *consume* attributes. In some cases, it may be useful to log the value of these attributes. For example, instead of an authentication filter logging a generic `Authentication Failed` message, you can use the value of the `authentication.subject.id` attribute to log the ID of the user that could not be authenticated.

Use the following format to enter a message attribute selector in a log message:

```
${name_of_attribute}
```

At runtime, the API Gateway expands these selectors to the value of the message attribute. For example, to make sure the ID of a non-authenticated user is logged in the message, enter something like the following in the text field for the **Failure** case:

```
The user '${authentication.subject.id}' could not be authenticated.
```

Then if a user with ID `MadCap:variable name="api_gateway_variables.lc_company"/>` cannot be authenticated by the API Gateway (a failure case), the following message is logged:

```
The user 'axway' could not be authenticated.
```

For more details on selectors, see "Select configuration values at runtime" in the *API Gateway Policy Developer Guide*.

**Transaction Logging Behavior**:
This setting is relevant only in cases where you have configured the API Gateway to log audit trail messages to a database. For more details, see the *API Gateway Administrator Guide*.

You can select **Abort policy processing on database log error** if you have configured the API Gateway to write log messages to a database, but that database is not available at runtime. If you have selected this setting, and the database is not available, the filter aborts, which in turn causes the policy to abort. In this case, the Fault Handler for the policy is invoked.

**Filter Category**:
The category selected here identifies the category of filters to which this filter belongs. The default selection should be appropriate in most cases.

# Oracle Access Manager filters 14

The following filters enable you to integrate with Oracle Access Manager:



# Oracle Access Manager authorization

## Overview

The **Authorization** filter enables you to authorize an authenticated user for a particular resource against Oracle Access Manager (OAM). The user must first have been authenticated to OAM using HTTP basic or digest authentication (see *HTTP Basic authentication* on page 82 or *HTTP digest authentication* on page 85). After successful authentication, OAM issues a Single Sign On (SSO) token, which can then be used instead of the user name and password.

See also *Oracle Access Manager certificate authentication* on page 366.

## General settings

Configure the following general fields:

**Name**:
Enter a descriptive name for this filter to display in a policy.

**Attribute Containing SSO Token**:
Enter the name of the message attribute that contains the user's SSO token. This attribute is populated when authenticating to Oracle Access Manager using *HTTP Basic authentication* on page 82 or *HTTP digest authentication* on page 85. By default, the SSO token is stored in the `oracle.sso.token` message attribute.

# Request settings

Configure the following fields to authorize a user for a particular resource against Oracle Access Manager:

**Resource Type**:
Enter the resource type for which you are requesting access (for example, `http` for access to a web-based URL).

**Resource Name**:
Enter the name of the resource for which the user is requesting access. The default is `//hostname${http.request.uri}`, which contains the original path requested by the client.

**Operation**:
In most access management products, it is common to authorize users fora limited set of actions on the requested resource. For example, users with management roles may be able to write (HTTP POST) to a certain web service, but users with more junior roles might only have read access (HTTP GET) to the same service.

You can use this field to specify the operation to grant the user access to on the specified resource. By default, this field is set to the `http.request.verb` message attribute, which contains the HTTP verb used by the client to send the message to the API Gateway (for example, POST).

**Include Query String**:
Select whether the OAM server uses the HTTP query string parameters to determine the policy that protects this resource. This setting is optional if the configured policies do not rely on the query string parameters. This setting is not configured by default.

# OAM Access SDK settings

Configure the following fields for the OAM Access SDK:

**OAM ASDK Directory**:
Enter the path to your OAM Access SDK directory. For more details on the OAM Access SDK, see your Oracle Access Manager documentation.

**OAM ASDK Compatibility Mode**:
Select the Oracle Access Manager server version to which this filter connects (`OAM 10g` or `OAM 11g`). Defaults to `OAM 11g`.

# Oracle Access Manager certificate authentication

## Overview

The **Log in with certificate** filter enables authentication to Oracle Access Manager (OAM) using an X.509 certificate presented by the client. After successful authentication, OAM issues a Single Sign On (SSO) token, which can then be used by the client for subsequent calls to the virtualized service.

See also *Oracle Access Manager authorization* on page 364.

## General settings

Configure the following general settings:

**Name**:
Enter an appropriate name for this filter to display in a policy.

**Attribute containing X509 certificate**:
Enter the name of the message attribute that contains the user's X.509 certificate. By default, this is stored in the `certificate` message attribute.

**Attribute to contain SSO token id**:
Enter the name of the message attribute to contain the user's SSO token. By default, the SSO token is stored in the `oracle.sso.token` message attribute.

## Resource settings

Configure the following resource settings:

**Resource Type**:
Enter the type of the resource for which you are requesting access. For example, when seeking access to a web-based URL, enter `http`.

**Resource Name**:
Enter the name of the resource for which the user is requesting access. By default, this field is set to `//hostname${http.request.uri}`, which contains the original path requested by the client.

**Operation**:
In most access management products, it is common to authorize users fora limited set of actions on the requested resource. For example, users with management roles may be able to write (HTTP POST) to a certain web service, but users with more junior roles might only have read access (HTTP GET) to the same service.

You can use this field to specify the operation to grant the user access to on the specified resource. By default, this field is set to the `http.request.verb` message attribute, which contains the HTTP verb used by the client to send the message to the API Gateway (for example, POST).

**Include query string**:
Select whether the query string parameters are used by the OAM server to determine the policy that protects this resource. This setting is optional if the policies configured do not rely on the query string parameters.

# Session settings

Configure the following session settings:

**Location**:
If the client location must be passed to OAM for it to make its decision, you can enter a valid DNS name or IP address to specify this location.

**Parameters**:
You can add optional additional parameters to be used in the authentication decision. The available optional parameters include the following:

| | |
|---|---|
| `ip` | IP address, in dotted decimal notation, of the client accessing the resource. |
| `operation` | Operation attempted on the resource (for HTTP resources, one of `GET`, `POST`, `PUT`, `HEAD`, `DELETE`, `TRACE`, `OPTIONS`, `CONNECT`, or `OTHER`). |
| `resource` | The requested resource identifier (for HTTP resources, the full URL). |
| `targethost` | The host (`host:port`) to which resource request is sent. |

**Note** One or more of these optional parameters might be required by certain authentication schemes, modules, or plug-ins configured in the OAM server. To determine which parameters to add, see your OAM server documentation.

# OAM Access SDK settings

Configure the following fields for the OAM Access SDK:

**OAM ASDK Directory**:
Enter the path to your OAM Access SDK directory. For more details on the OAM Access SDK, see your Oracle Access Manager documentation.

**OAM ASDK Compatibility Mode**:
Select the Oracle Access Manager server version to which this filter connects (`OAM 10g` or `OAM 11g`). Defaults to `OAM 11g`.

# Oracle Access Manager SSO session logout

## Overview

The **Log out session** filter enables you to log out a session from Oracle Access Manager by invalidating the SSO token that is associated with this session.

See also *Oracle Access Manager certificate authentication* on page 366.

## Configuration

Configure the following fields to explicitly log out (invalidate)an SSO token from Oracle Access Manager:

**Name**:
Enter a descriptive name for this filter to display in a policy.

**Attribute Containing SSO Token ID**:
Enter the name of the message attribute that contains the SSO token to invalidate. This attribute is populated when authenticating to Oracle Access Manager using *HTTP Basic authentication* on page 82 or *HTTP digest authentication* on page 85. By default, the SSO token is stored in the `oracle.sso.token` message attribute.

**OAM ASDK Directory**:
Enter the path to your OAM Access SDK directory. For more details on the OAM Access SDK, see your Oracle Access Manager documentation.

**OAM ASDK Compatibility Mode**:
Select the Oracle Access Manager server version to which this filter connects (`OAM 10g` or `OAM 11g`). Defaults to `OAM 11g`.

# Oracle Access Manager SSO token validation

## Overview

The **SSO Token Validation** filter enables you to check an Oracle Access Manager Single Sign On (SSO) token to ensure that it is still valid. The SSO token is issued by Oracle Access Manager (OAM) after the API Gateway authenticates to it on behalf of an end user using *HTTP Basic authentication* on page 82 or *HTTP digest authentication* on page 85. After successfully authenticating to OAM, the SSO token is stored in the `oracle.sso.token` message attribute.

Oracle Access Manager SSO enables a client to send up its user name and password once, and then receive an SSO token (for example, in a cookie or in the XML payload). The client can then send up the SSO token instead of the user name and password.

See also *Oracle Access Manager certificate authentication* on page 366.

# Configuration

Configure the following fields to validate an SSO token issued by Oracle Access Manager:

**Name**:
Enter a descriptive name for the filter to display in a policy.

**Attribute Containing SSO Token ID**:
Enter the name of the message attribute that contains the SSO token to validate. This attribute is populated when authenticating to Oracle Access Manager using *HTTP Basic authentication* on page 82 or *HTTP digest authentication* on page 85. By default, the SSO token is stored in the `oracle.sso.token` message attribute.

**OAM ASDK Directory**:
Enter the path to your OAM Access SDK directory. For more details on the OAM Access SDK, see your Oracle Access Manager documentation.

**OAM ASDK Compatibility Mode**:
Select the Oracle Access Manager server version to which this filter connects (`OAM 10g` or `OAM 11g`). Defaults to `OAM 11g`.

# Oracle Entitlements Server filters 15

The following filters enable you to integrate with Oracle Entitlements Server:



# Oracle Entitlements Server 10g authorization

## Overview

This filter enables you to authorize an authenticated user for a particular resource against Oracle Entitlements Server (OES) 10g. The user must first have been authenticated to OES 10g (for example, using *HTTP Basic authentication* on page 82 or *HTTP digest authentication* on page 85).

This filter enables you to configure API Gateway to delegate authorization to OES 10g. You can configure API Gateway to authorize an authenticated user for a particular resource against OES 10g. Credentials used for authentication can be extracted from the HTTP Basic header, WS-Security Username Token, or the message payload. After successful authentication, the API Gateway can authorize the user to access a resource using OES 10g.

See also *Get roles from Oracle Entitlements Server 10g* on page 372.

## Configuration

Configure the following fields:

### *Settings*

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

**Resource**:
Enter the URL for the target resource (for example, web service). Alternatively, if this policy is reused for multiple services, enter a URL using message attribute selectors, which are expanded at runtime to the value of the specified attribute. For example:

```
${http.destination.protocol}://${http.destination.host}:${http.destination.port}${http.request.uri}
```

**Resource Naming Authority**:
Enter `apigatewayResource` to match the Naming Authority Definition loaded in the OES 10g settings. For more details, see "Oracle Security Service Module settings (10g)" in the *API Gateway Policy Developer Guide*.

**Action**:
Enter the HTTP verb (for example, `POST`, `GET`, `DELETE`, and so on). Alternatively, if this policy is reused for multiple services, enter a message attribute selector, which is expanded at runtime to the value of the specified attribute (for example, `${http.request.verb}`). For more details on selectors, see "Select configuration values at runtime" in the *API Gateway Policy Developer Guide*.

**Action Naming Authority**:
Enter `apigatewayAction` to match the Naming Authority Definition loaded in the OES 10g settings. For more details, see "Oracle Security Service Module settings (10g)" in the *API Gateway Policy Developer Guide*.

**How access request is processed**:
Select one of the following options:

- `ONCE`

  Specifies that the authorization query is only asked once for a resource and action.

- `POST`

  Specifies that the authorization query is asked after a resource is acquired, but before it has been processed or presented.

- `PRIOR`

  Specifies that the authorization query is asked before a resource is acquired.

## *Application Context*

Configure the following field on the **Application Context** tab:

**Application's Current Context**:
Click **Add** to specify optional Application Contexts as name-value pairs. Enter a **Name** and **Value** in the **Properties** dialog. Repeat to specify multiple properties.

# Get roles from Oracle Entitlements Server 10g

## Overview

This filter enables you to get the set of roles that are assigned to an identity for a specific resource (for example, web service) and a specific action (for example, HTTP POST) from Oracle Entitlements Server (OES) 10g.

See also *Oracle Entitlements Server 10g authorization* on page 370.

## Configuration

Configure the following fields:

### *Settings*

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

**Resource**:
Enter the URL of the target resource (for example, web service). Alternatively, if this policy is reused for multiple services, enter a URL using message attribute selectors, which are expanded at runtime to the value of the specified attribute. For example:

```
${http.destination.protocol}://${http.destination.host}:${http.destination.port}${http.request.uri}
```

**Resource Naming Authority**:
Enter `apigatewayResource` to match the Naming Authority Definition loaded in the OES 10g settings. For more details, see "Oracle Security Service Module settings (10g)" in the *API Gateway Policy Developer Guide*.

**Action**:
Enter the HTTP verb (for example, `POST`, `GET`, `DELETE`, and so on). Alternatively, if this policy is reused for multiple services, enter a message attribute selector, which is expanded at runtime to the value of the specified attribute (for example, `${http.request.verb}`). For more details on selectors, see "Select configuration values at runtime" in the *API Gateway Policy Developer Guide*.

**Action Naming Authority**:
Enter `apigatewayAction` to match the Naming Authority Definition loaded in the OES 10g settings. For more details, see "Oracle Security Service Module settings (10g)" in the *API Gateway Policy Developer Guide*.

### *Application Context*

Configure the following field on the **Application Context** tab:

**Application's Current Context**:
Click **Add** to specify optional Application Contexts as name-value pairs. Enter a **Name** and **Value** in the **Properties** dialog. Repeat to specify multiple properties.

# Oracle Entitlements Server 11g authorization

## Overview

This filter enables you to authorize an authenticated user for a particular resource against Oracle Entitlements Server (OES) 11g. The user must first have been authenticated to OES 11g (for example, using *HTTP Basic authentication* on page 82 or *HTTP digest authentication* on page 85).

This filter enables you to configure API Gateway to delegate authorization to OES 11g. You can configure API Gateway to authorize an authenticated user for a particular resource against OES 11g. Credentials used for authentication can be extracted from the HTTP Basic header, WS-Security Username Token, or the message payload. After successful authentication, the API Gateway can authorize the user to access a resource using OES 11g.

## Configuration

Configure the following fields:

**Name**:
Enter an appropriate descriptive name for this filter.

**Resource**:
Enter the URL for the target resource to be authorized (for example, web service). Alternatively, if this policy is reused for multiple services, enter a URL using selectors, which are expanded at runtime to the value of the specified attributes. For example:

```
${http.destination.protocol}://${http.destination.host}:${http.destination.port}${http.request.uri}
```

**Action**:
Enter the HTTP verb (for example, `POST`, `GET`, `DELETE`, and so on). Alternatively, if this policy is reused for multiple services, enter a selector, which is expanded at runtime to the value of the specified attribute (for example, `${http.request.verb}`). For more details on selectors, see "Select configuration values at runtime" in the *API Gateway Policy Developer Guide*.

**Environment/Context attributes**:
Click **Add** to specify optional Application Contexts as name-value pairs. Enter a **Name** and **Value** in the **Properties** dialog. Repeat to specify multiple properties.

# Resolver filters 16

The following filters enable you to identify incoming XML messages:



# Relative path resolver

## Overview

The **Relative Path** filter enables you to identify an incoming XML message based on the relative path on which the message is received.

The following example shows how to find the relative path of an incoming message. Consider the following SOAP message:

```
POST /services/helloService HTTP/1.1
Host:localhost:8095
Content-Length:196
SOAPAction:HelloService
Accept-Language:en-US
UserAgent:API Gateway
Content-Type:text/XML; utf-8

<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
   <soap:Header></soap:Header>
   <soap:Body>
       <getHello xmlns="http://www.axway"/>.com/"/>
   </soap:Body>
</soap:Envelope>
```

The relative path for this message is as follows:

```
/services/helloService
```

### *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

## Configuration

To configure the **Relative Path** filter:

1. Enter an appropriate name for the filter to display in a policy in the **Name** field.
2. Enter a regular expression to match the value of the relative path on which messages are received in the **Relative Path** field.

   For example, enter `^/services/helloService$` to exactly match a path with a value of `/services/helloService`. Incoming messages received on a matching relative path value are passed on to the next filter on the success path in the policy.

## Further information

For more details, see the following filters:

- *SOAP action resolver* on page 376
- *Operation name resolver* on page 377

# SOAP action resolver

## Overview

The **SOAP Action Resolver** filter enables you to identify an incoming XML message based on the `SOAPAction` HTTP header in the message. The **SOAP Action Resolver** filter applies to SOAP 1.1 and SOAP 1.2.

The following example illustrates how to locate the `SOAPAction` header in an incoming message. Consider the following SOAP message:

```
POST /services/helloService
HTTP/1.1Host:localhost:8095
Content-Length:196
SOAPAction:HelloService
Accept-Language:en-US
UserAgent:API Gateway
Content-Type:text/XML; utf-8
```

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
   <soap:Header>
   </soap:Header>
   <soap:Body>
      <getHello xmlns="http://www.axway/>.com/"/>
   </soap:Body>
</soap:Envelope>
```

The SOAP Action for this message is `HelloService`.

### *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

## Configuration

To configure the **SOAP Action Resolver** filter:

1. Enter an appropriate name for the filter to display in policy in the **Name** field.
2. Enter a regular expression to match the value of the `SOAPAction` HTTP header in the **SOAP Action** field.

   For example, enter `^getQuote$` to exactly match a `SOAPAction` header with a value of `getQuote`. Incoming messages with a matching `SOAPAction` value are passed on to the next filter on the success path in the policy.

## Further information

For more details, see the following filters:

- *Operation name resolver* on page 377
- *Relative path resolver* on page 375

# Operation name resolver

## Overview

The **Operation Name** filter enables you to identify an incoming XML message based on the SOAP operation in the message.

The following example shows how to find the SOAP operation of an incoming message. Consider the following SOAP message:

```
POST /services/timeservice
HTTP/1.0Host:localhost:8095
Content-Length:374
SOAPAction:TimeService
Accept-Language:en-US
UserAgent:API Gateway
Content-Type:text/XML; utf-8

<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
   <soap:Body>
      <ns1:getTime xmlns:ns1="Some-URI">
         <ns1:city>Dublin</ns1:city>
      </ns1:getTime>
   </soap:Body>
</soap:Envelope>
```

The SOAP operation for this message and its namespace are as follows:

- SOAP operation:`getTime`
- SOAP operation namespace:`urn:timeservice`

The SOAP operation is the first child element of the SOAP `<soap:Body>` element.

# Configuration

To configure the **Operation Name** filter:

1. Enter an appropriate name for the filter to display in a policy in the **Name** field.
2. Enter the name of the SOAP operation in the **Operation** field. Incoming messages with an operation name matching the value entered here are passed on to the next success filter in the policy.
3. Enter the namespace to which the SOAP operation belongs in the **Namespace** field.

# Further information

For more details, see the following filters:

- *SOAP action resolver* on page 376
- *Relative path resolver* on page 375

# Routing filters 17

The following filters enable you to route messages to external services:



# Call internal service

## Overview

The **Call internal service** filter is a special filter that passes messages to an internal servlet application or static content provider that has been deployed at the API Gateway. The appropriate application is selected based on the relative path on which the request message is received.

This filter is used by Management Services that are configured to listen on the Management Interface on port 8090. For more information on how the **Call internal service** filter is used by these services, see "Management services" in the *API Gateway Policy Developer Guide*.

## Configuration

You can configure the following fields on the filter window:

**Name**:
Enter an appropriate name for this filter to display in a policy.

**Additional HTTP Headers to Send to Internal Service**:
Click the **Add** button to configure additional HTTP headers to send to the internal application. Specify the following fields on the **HTTP Header** dialog:

- **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, API Gateway expands these selectors to the current value of the corresponding message attribute. For example, the `${id}` selector is replaced by the value of the current message ID. For more details on selectors, see "Select configuration values at runtime" in the *API Gateway Policy Developer Guide*.

# Connect to URL

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 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

The **Connect to URL** filter enables the API Gateway to act as the endpoint to the client connection (and not as a proxy), and to hide the deployment hierarchy of protected web services from clients. In other words, the API Gateway performs *service virtualization*. For an introduction to routing scenarios and the filters in the **Routing** category, see "Get started with routing configuration" in the *API Gateway Policy Developer Guide*.

## General settings

Configure the following general settings:

**Name**:
Enter an appropriate name for the filter to display in a policy.

**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" in the *API Gateway Policy Developer Guide*. 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 selector syntax to evaluate and expand request details at runtime. For more details, see "Select configuration values at runtime" in the *API Gateway Policy Developer Guide*. 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}`.

**Note** 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 select the SSL/TLS protocol options and specify the ciphers that API Gateway supports on the **Advanced (SSL)** tab.

## Trusted certificates

When API Gateway 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 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 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.

## Advanced

You can configure the following settings on the **Advanced (SSL)** tab.

**SSL Protocol Options**:

You can configure the following SSL protocol options:

| Option | Description |
|---|---|
| Do not use the SSL v2 protocol | Specifies not to use SSL v2 for outbound connections to avoid any weaknesses in this protocol. This option is selected by default. |
| Do not use the SSL v3 protocol | Specifies not to use SSL v3 for outbound connections to avoid any weaknesses in this protocol. This option is selected by default. |
| Do not use the TLS v1 protocol | Specifies not to use TLS v1 for outbound connections to avoid any weaknesses in this protocol. This option is selected by default. |

| Option | Description |
|---|---|
| Do not use the TLS v1.1 protocol | Specifies not to use TLS v1.1 for outbound connections to avoid any weaknesses in this protocol. This option is selected by default. |
| Do not use the TLS v1.2 protocol | Specifies not to use TLS v1.2 for outbound connections. This option is not selected by default. |

**Ciphers**:

You can specify the ciphers that the server supports in this field. The API Gateway 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 used to encrypt the data as it is sent over the secure channel. For more information on the syntax of this setting, see the OpenSSL documentation.

**Note** The default cipher string of `FIPS:!SSLv3:!aNULL` performs the following:

- Enables FIPS-compatible cipher suites only
- Explicitly blocks cipher suites that require SSLv3 or lower
- Forces the use of TLSv1.2 only
- Forbids unauthenticated cipher suites

# 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 **Environment Configuration** > **External Connections** node in the Policy Studio tree. For more details on configuring client credentials, see "Configure client credentials" in the *API Gateway Policy Developer Guide*.

# Additional settings

The **Settings** tab enables you to configure the following additional settings:

- **Retry**
- **Failure**
- **Proxy**
- **Redirect**
- **Headers**

- **Response Body**
- **Connection**

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`.

**Note** When **Retry** setting is enabled, the **Retry Count** value is used as the default number of redirects to follow in the **Redirect** settings.

**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" in the *API Gateway Policy Developer Guide*.

**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 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 must connect to the destination web service through an HTTP proxy. In this case, the API Gateway 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 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. To add a proxy server, right-click the **Proxy Servers** tree node, and select **Add a Proxy Server**. Alternatively, you can configure proxy servers under **Environment Configuration** > **External Connections** in the Policy Studio tree. For more details, see the *API Gateway Policy Developer Guide*.

**Note** API Gateway does not support persistent SSL connections to back-end servers using proxy tunneling. The API Gateway connection caching mechanism is not designed for proxy tunnel connections.

**Transparent Proxy (present client's IP address to server)**:
Enables the API Gateway 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. For more details, see "Configure a transparent proxy" in the *API Gateway Policy Developer Guide*.

## *Redirect settings*

To specify the redirect settings for this filter, complete the following fields:

**Follow Redirects**:
Specifies whether the API Gateway follows HTTP redirects, and connects to the redirect URL specified in the HTTP response. This setting is enabled by default, and the default number of redirects to follow is 2.

**Note** When the **Retry Count** property of the **Retry** settings is enabled, its value is used as the default number of redirects to follow.

## *Header settings*

To specify the header settings for this filter, complete the following fields:

**Forward spurious received Content headers**:
Specifies whether the API Gateway 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 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 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 to always forward on the original `Host` header that it received from the client. Alternatively, to configure the API Gateway to include the address and port number of the destination web service in the `Host` header, select the **Generate new Host header** radio button.

## *Response body settings*

**Load response body and release connection**:

Select this option to load the response message body into API Gateway memory before exiting the filter. Loading and parsing durations are included in the corresponding traffic monitoring leg duration. The HTTP connection is released after the response body is parsed.

## *Connection settings*

**Release previously opened connection**:

Select this option to release any connection opened by a previous call to a connection filter (where the filter is for the same message and has this option selected). This avoids leaving connections open when a connection filter is called several times during policy execution for a message. If this option is not selected, the default behavior is to release resources only after policy execution.

**Note** This option replaces the system property `<VMArg name="-DConnectToUrlFilter.removePreviousConnections=true"/>` that was added in 7.5.3 SP3 to enable releasing previously opened connections. If you were using this system property, you must select this option on each connection filter requiring this behavior, as the system property no longer exists.

# Connection

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.

Depending on how the API Gateway is perceived by the client, different combinations of routing filters can be used. For an introduction to using the various filters in the **Routing** category, see "Get started with routing configuration" in the *API Gateway Policy Developer Guide*.

## SSL settings

You can configure SSL settings, such as trusted certificates, client certificates, SSL/TLS protocols, and ciphers on the **SSL** tab. For details on the fields on this tab, see *SSL settings* on page 381.

## Authentication settings

You can select credential profiles to use for authentication on the **Authentication** tab. For details on the fields on this tab, see *Authentication settings* on page 383.

## Additional settings

The **Settings** tab allows you to configure the following additional settings:

- **Retry**
- **Failure**
- **Proxy**
- **Redirect**
- **Headers**
- **Response Body**
- **Connection**

By default, these sections are collapsed. Click a section to expand it. For details on the fields on this tab, see *Additional settings* on page 383.

# Dynamic router

## Overview

API Gateway can act as a proxy for clients of the secured web service. When a client uses a proxy, it includes the fully qualified URL of the destination in the request line of the HTTP request. It sends this request to the configured proxy, which then forwards the request to the host specified in the URL. The relative path used in the original request is preserved by the proxy on the outbound connection.

The following is an example of an HTTP request line that was made through a proxy, where `WEB_SERVICE_HOST` is the name or IP address of the machine hosting the destination web service:

```
POST http://WEB_SERVICE_HOST:80/myService HTTP/1.0
```

When API Gateway acts as a proxy for clients, it can receive requests like the one above. The **Dynamic Router** filter can route the request on to the URL specified in the request line (`http://WEB_SERVICE_HOST:80/myService`).

Depending on how API Gateway is perceived by the client, different combinations of routing filters can be used. For an introduction to using the various filters in the **Routing** category, see "Get started with routing configuration" in the *API Gateway Policy Developer Guide*.

# Extract path parameters

## Overview

The **Extract Path Parameters** filter enables API Gateway to parse the contents of a specified HTTP path into message attributes. This means that you can define HTTP path parameters, and then extract their values at runtime using selectors. For example, this is useful when passing in parameters to REST-based requests. For more details on selectors, see "Select configuration values at runtime" in the *API Gateway Policy Developer Guide*.

## Configuration

Complete the following settings:

**Name**:
Enter a descriptive name for this filter to display in a policy.

**URI Template**:
Enter the URI template for the path to be parametrized. This is a formatted Jersey `@Path` annotation string, which enables you to parametrize the path specified in the incoming `http.request.path` message attribute. The following is an example URI template entry:

```
/twitter/{version}/statuses/{operation}.{format}
```

**Path Parameters**:
The **Path Parameters** table enables you to map the path parameters specified in the **URI Template** to user-defined message attributes. These attributes can then be used by other filters downstream in the policy. Click **Add** to configure a path parameter, and specify the following in the dialog:

- **Path Parameter**:

  Enter the name of the path parameter (for example, `version`).

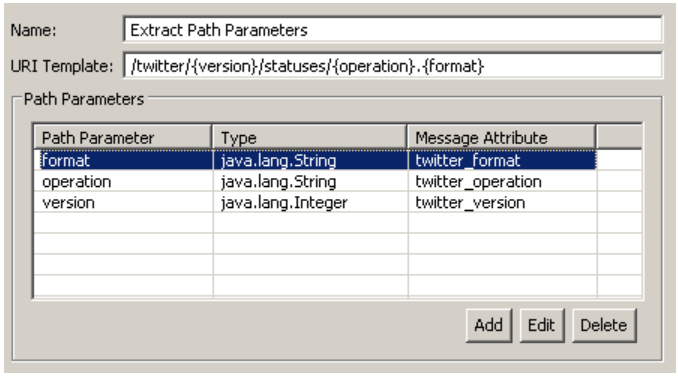
- **Type**:

  Enter the type of the path parameter (for example, `java.lang.String`).

- **Message Attribute**:

Enter the name of the message attribute that stores the parameter value (for example, `twitter_version`).

The following figure shows the example path parameters:




# Required input and generated output

The incoming `http.request.path` message attribute is required as input to this filter.

This filter generates the message attributes for the parameters that you specify in the **Path Parameters** table. For example, in the previous figure, the following attributes are generated:

- `twitter_format`
- `twitter_operation`
- `twitter_version`

# Possible outcomes

The possible outcomes of this filter are as follows:

- `True` if the specified **URI Template** is successfully parsed.
- `False` if an error occurs during **URI Template** parsing.
- `CircuitAbortException` if an exception occurs during **URI Template** parsing.

# File upload

## Overview

You can use the **File Upload** filter to upload processed messages as files to a file transfer server. This enables you to upload the contents of the `content.body` message attribute as a file. The **File Upload** filter supports the following protocols:

- **FTP**: File Transfer Protocol
- **FTPS**: FTP over Secure Sockets Layer (SSL)
- **SFTP**: Secure Shell (SSH) File Transfer Protocol

Configuring a **File Upload** filter can be useful when integrating with Business-to-Business (B2B) partner destinations or with legacy systems. For example, instead of making drastic changes to either system, API Gateway can make files available for upload to the other system. The added benefit is that the files are exposed to the full complement of API Gateway message processing filters. This ensures that only properly validated files are uploaded to the target system.

The **File Upload** filter is available from the **Routing** category of filters in Policy Studio. This topic describes how to configure the fields on the **File Upload** filter dialog.

See also *File download* on page 394.

## General settings

Configure the following general settings:

**Name**:
Enter a descriptive name for this filter to display in a policy.

**Host**:
Enter the name of the host machine on which the file transfer server is running.

**Port**:
Enter the port number to connect to the file transfer server. Defaults to `21`.

**Username**:
Enter the user name to connect to the file transfer server.

**Password**:
Specify the password for this user.

## File details

Configure the following fields in the **File details** section:

**Filename**:
The message body (in the `content.body` message attribute) is stored using this file name on the destination file transfer server. The default value of `${id}.out` enables you to use the unique identifier associated with each message processed by API Gateway. When this value is specified, messages are stored in individual files on the file transfer server according to their unique message identifier.

**Directory**:
Specify the directory where the file is stored on the destination file transfer server.

**Use temporary file name during upload**:
This option specifies whether to use a temporary file name of `${id}.part` when the file is uploading to the file transfer server. When the file has uploaded, it then uses the file name specified in this filter (for example, the default `${id}.out` file name). This prevents an incomplete file from being uploaded. This option is selected by default.

**Note** You must deselect this option if the file transfer server is API Gateway. For example, this option applies when the API Gateway uploads to a file transfer server, and then another server (possibly API Gateway) polls the file transfer server for new files to process. The poller server is configured to consume `*.xml` files and ignores the temporary file. When the upload is complete, the file is renamed and the poller sees the new file to process.

# Connection type

The fields configured in the **Connection Type** section determine the type of file transfer connection. Select the FTP connection type from the following options:

- FTP - File Transfer Protocol
- FTPS - FTP over SSL
- SFTP - SSH File Transfer Protocol

## *FTP and FTPS connections*

The following general settings apply to FTP and FTPS connections:

**Passive transfer mode**:
Select this option to prevent problems caused by opening outgoing ports in the firewall relative to the file transfer server (for example, when using *active* FTP connections). This is selected by default.

**Note** To use passive transfer mode, you must perform the steps described in "Configure passive transfer mode" in the *API Gateway Policy Developer Guide*.

**File Type**:
Select **ASCII** mode for sending text-based data, or **Binary** mode for sending binary data over the file transfer connection. Defaults to **ASCII** mode.

## FTPS connections

The following security settings apply to FTPS connections only:

**SSL Protocol**:
Enter the SSL protocol used (for example, `SSL` or `TLS`). Defaults to `SSL`.

**Implicit**:
When this option is selected, security is automatically enabled as soon as the **File Upload** client makes a connection to the remote file transfer service. No clear text is passed between the client and server at any time. In this case, a specific port is used for secure connections (`990`). This option is not selected by default.

**Explicit**:
When this option is selected, the remote file transfer service must explicitly request security from the **File Upload** client, and negotiate the required security. If the file transfer service does not request security, the client can allow the file transfer service to continue insecure or refuse or limit the connection. This option is selected by default.

**Trusted Certificates**:
To connect to a remote file server over SSL, you must trust that server's SSL certificate. When you have imported this certificate into the Certificate Store, you can select it on the **Trusted Certificates** tab.

**Client Certificates**:
If the remote file server requires the **File Upload** client to present an SSL certificate to it during the SSL handshake for mutual authentication, you must select this certificate from the list on the **Client Certificates** tab. This certificate must have a private key associated with it that is also stored in the Certificate Store.

## SFTP connections

The following security settings apply to SFTP connections only:

**Present following key for authentication**:
Click the button on the right, and select a previously configured key to be used for authentication from the tree. To add a key, right-click the **Key Pairs** node, and select **Add**. Alternatively, you can import key pairs under the **Environment Configuration** > **Certificates and Keys** node in the Policy Studio tree. For more details, see "Manage X.509 certificates and keys" in the *API Gateway Policy Developer Guide*.

**SFTP host must present key with the following finger print**:
Enter the fingerprint of the public key that the SFTP host must present (for example, `43:51:43:a1:b5:fc:8b:b7:0a:3a:a9:b1:0f:66:73:a8`).

# File download

## Overview

You can use the **File download** filter to download files from a file transfer server and store their contents in the `content.body` message attribute. The **File download** filter supports the following protocols:

- **FTP**: File Transfer Protocol
- **FTPS**: FTP over Secure Sockets Layer (SSL)
- **SFTP**: Secure Shell (SSH) File Transfer Protocol

Configuring a **File download** filter can be useful when integrating with Business-to-Business (B2B) partner destinations or with legacy systems. For example, instead of making drastic changes to either system, API Gateway can download files from the other system. The added benefit is that the files are exposed to the full compliment of API Gateway message processing filters. This ensures that only properly validated files are downloaded from the target system. The **File download** filter is available from the **Routing** category of filters in Policy Studio.

See also *File upload* on page 391.

## General settings

Configure the following general settings:

**Name**:
Enter a descriptive name for this filter to display in a policy.

**Host**:
Enter the name of the host machine on which the file transfer server is running.

**Port**:
Enter the port number to connect to the file transfer server. Defaults to `21`.

**Username**:
Enter the user name to connect to the file transfer server.

**Password**:
Specify the password for this user.

## File details

Configure the following fields in the **File details** section:

**Filename**:
Specifies the file name to download from the file transfer server. The default value is `filename.xml`. You can enter a different file name or use a message attribute selector, which is expanded at runtime (for example, `${authentication.subject.id}`).

When downloading a file from the file transfer server, API Gateway uses a temporary file name of `filename.part`. When the file has been downloaded, it then uses the file name specified in this filter (for example, the default `filename.xml`). This prevents an incomplete file from being downloaded.

**Directory**:
Specify the directory where the file is stored.

# Connection type

The fields configured in the **Connection Type** section determine the type of file transfer connection. Select the FTP connection type from the following options:

- FTP - File Transfer Protocol
- FTPS - FTP over SSL
- SFTP - SSH File Transfer Protocol

## *FTP and FTPS connections*

The following general settings apply to FTP and FTPS connections:

**Passive transfer mode**:
Select this option to prevent problems caused by opening outgoing ports in the firewall relative to the file transfer server (for example, when using *active* FTP connections). This is selected by default.

**Note** To use passive transfer mode, you must perform the steps described in "Configure passive transfer mode" in the *API Gateway Policy Developer Guide*.

**File Type**:
Select **ASCII** mode for sending text-based data, or **Binary** mode for sending binary data over the file transfer connection. Defaults to **ASCII** mode.

## *FTPS connections*

The following security settings apply to FTPS connections only:

**SSL Protocol**:
Enter the SSL protocol used (for example, `SSL` or `TLS`). Defaults to `SSL`.

**Implicit**:
When this option is selected, security is automatically enabled as soon as the **File Download** client

makes a connection to the remote file transfer service. No clear text is passed between the client and server at any time. In this case, a specific port is used for secure connections (`990`). This option is not selected by default.

**Explicit**:
When this option is selected, the remote file transfer service must explicitly request security from the **File Download** client, and negotiate the required security. If the file transfer service does not request security, the client can allow the file transfer service to continue insecure or refuse or limit the connection. This option is selected by default.

**Trusted Certificates**:
To connect to a remote file server over SSL, you must trust that server's SSL certificate. When you have imported this certificate into the Certificate Store, you can select it on the **Trusted Certificates** tab.

**Client Certificates**:
If the remote file server requires the **File Download** client to present an SSL certificate to it during the SSL handshake for mutual authentication, you must select this certificate from the list on the **Client Certificates** tab. This certificate must have a private key associated with it that is also stored in the Certificate Store.

### *SFTP connections*

The following security settings apply to SFTP connections only:

**Present following key for authentication**:
Click the button on the right, and select a previously configured key to be used for authentication from the tree. To add a key, right-click the **Key Pairs** node, and select **Add**. Alternatively, you can import key pairs under the **Environment Configuration** > **Certificates and Keys** node in the Policy Studio tree. For more details, see "Manage X.509 certificates and keys" in the *API Gateway Policy Developer Guide*.

**SFTP host must present key with the following finger print**:
Enter the fingerprint of the public key that the SFTP host must present (for example, `43:51:43:a1:b5:fc:8b:b7:0a:3a:a9:b1:0f:66:73:a8`).

# HTTP redirect

## Overview

You can use the **HTTP Redirect** filter to enable API Gateway to send an HTTP redirect message. For example, you can send an HTTP redirect to force a client to enter user credentials on an HTML login page if no HTTP cookie already exists. Alternatively, you can send an HTTP redirect if a web page has moved to a new URL address.

See also *HTTP status code* on page 397.

## Configuration

Complete the following settings:

**Name**:
Enter a descriptive name for this filter to display in a policy.

**HTTP response code status**:
Enter the HTTP response code status to use in the HTTP redirect message. Defaults to `301`, which means that the requested resource has been assigned a new permanent URI, and any future references to this resource should use the returned redirect URL.

**Redirect URL**:
Enter the URL address to which the message is redirected.

**Content-Type**:
Enter the `Content-Type` of the HTTP redirect message (for example, `text/xml`).

**Message Body**:
Enter the message body text to send in the HTTP redirect message.

# HTTP status code

## Overview

This filter sets the HTTP status code on response messages. This enables you 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 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
```

See also *HTTP redirect* on page 396.

## Configuration

**Name**:
Enter an appropriate name for this filter to display in a policy.

**HTTP response code status**:
Enter the status code returned to the client. For a complete list of status codes, see the HTTP specification.

# Insert WS-Addressing information

## Overview

The WS-Addressing specification defines a transport-independent standard for including addressing information in SOAP messages. API Gateway can generate WS-Addressing information based on a configured endpoint in a policy, and then insert this information into SOAP messages.

See also *Read WS-Addressing information* on page 401.

## Configuration

Complete the following fields to configure API Gateway to insert WS-Addressing information into the SOAP message header.

**Name**:
Enter an appropriate name for the filter to display in a policy.

**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.

# Read from JMS

The **Read from JMS** filter enables you to configure a JMS messaging system from which the API Gateway reads messages. You can configure various settings for the JMS message source, message type, and processing options.

API Gateway 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 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 installation, and `PLATFORM` is the platform on which API Gateway is installed (`Linux.x86_64`). If the provider implementation is platform-independent, copy the JAR files to `INSTALL_DIR/ext/lib`.

## Message source

The **Message source** settings enable you to configure the following:

**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**. Alternatively, you can configure JMS services under the **Environment Configuration** > **External Connections** node in the Policy Studio tree. For more details, see *Configure messaging services* on page 1.

**Source type**:
Select one of the following from the list:

- **Queue**
- **Topic**
- **JNDI lookup**

Defaults to **Queue**.

**Source Name**:
Enter the name of the JMS queue, JMS topic, or JNDI lookup to specify where you want read the messages from.

**Note** The source name to use depends on the configured **Source type**. For example, for IBM WebSphere MQ, this name may need to use a format of `queue://`. For details on source name requirements, please see the user documentation for your third-party JMS provider tool.

**Selector**:
Enter an SQL 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 instance only receives messages whose headers and properties match the selector. For example, the following expression gets the selected message from the queue:

```
JMSCorrelationID='${params.query.id}'
```

**Read timeout (ms)**:
Enter the timeout after which the **Read from JMS** filter fails. The accepted range of values is 1–20000 ms. Defaults to `1000` ms.

# JMS consumer type

The **JMS consumer type** settings enable you to configure the following:

**Durable subscription**:
Create or use a durable topic subscription to consume messages from the server. This option is only available for **Topic** and **JNDI lookup** source types.

**Note** This is only available with a **Topic** source and the JMS service used must have a client ID configured. If a **JNDI lookup** source is configured, the name must not point to a topic.

**Topic subscriber name**:
Enter the JMS subscriber name used to identify the durable subscription.

# Message processing

The **JMS consumer type** settings enable you to configure the following:

**Extraction Method**:
Specify how to extract the data from the JMS message from the list:

- Insert the JMS message directly into the attribute named below (this is the default)
- Populate the attribute below with the value inferred from message type to Java

**Attribute Name**:
The name of the API Gateway message attribute that holds the data extracted from the JMS message. Defaults to the `jms.message` message attribute.

**Policy**:
Select the appropriate policy to run on the JMS message after it has been consumed by the API Gateway.

**Send Response to Configured Destination**:
Specifies whether the API Gateway sends a reply to the response queue named in the incoming message (in the `ReplyTo` header). This option is selected by default. Deselecting this option means that the API Gateway never sends a reply to the response queue named in the `ReplyTo` header.

# Read WS-Addressing information

## Overview

The WS-Addressing specification defines a transport-independent standard for including addressing information in SOAP messages. API Gateway can read WS-Addressing information contained in a SOAP message and subsequently use this information to route the message to its intended destination.

See also *Insert WS-Addressing information* on page 398.

## Configuration

Complete the following fields to configure API Gateway to read WS-Addressing information contained in a SOAP message.

**Name**:
Enter an appropriate name for the filter to display in a policy.

**Address location**:
Specify the name of the element in the WS-Addressing block that contains the address of the destination server to which the API Gateway routes the message. For more information on configuring XPath expressions, see "Configure XPath expressions" in the *API Gateway Policy Developer Guide*.

By default, XPath expressions are available to extract the destination server from the `From`, `To`, `ReplyTo`, and `FaultTo` elements. Click the **Add** button to add a new XPath expression to extract the address from a different location.

**Remove enclosing WS-Addressing element**:
If this option is selected, the WS-Addressing element returned by the XPath expression configured above is removed from the SOAP header when it has been consumed.

# 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 *Static router* on page 409.

Depending on how API Gateway is perceived by the client, different combinations of routing filters can be used. For an introduction to using the various filters in the **Routing** category, see "Get started with routing configuration" in the *API Gateway Policy Developer Guide*.

## Configuration

Configure the following fields on the **Rewrite URL** filter configuration window:

**Name**:
Enter an appropriate name for the filter to display in a policy.

**URL**:
Enter the relative path of the web service in the **URL** field. API Gateway 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 filter to display in a policy.

**SMTP Server Settings**:
Click the browse button and select a preconfigured SMTP server in the tree. To add an SMTP server, right-click the **SMTP Servers** node, and select **Add an SMTP Server**. Alternatively, you can configure SMTP servers under the **Environment Configuration** > **External Connections** node in the Policy Studio tree. For more details on configuring SMTP servers, see the *API Gateway Policy Developer Guide*.

## 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 on selectors, see "Select configuration values at runtime" in the *API Gateway Policy Developer Guide*.

# Save to file

## Overview

The **Save to File** filter enables you to write the current message contents to a file. For example, you can save the message contents to a file in a directory where it can be accessed by an external application. This can be used to quarantine messages to the file system for offline examination.

This filter can also be useful when integrating legacy systems. Instead of making drastic changes to the legacy system by adding an HTTP engine, API Gateway can save the message contents to the file system, and route them on over HTTP to another back-end system.

## Configuration

To configure the **Save to File** filter, specify the following fields:

**Name**:
Name of the filter to be displayed in a policy.

**File name**:
Enter the name of the file that the 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" in the *API Gateway Policy Developer Guide*.

**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 instance.

**Maximum number of files in directory**:
Enter the maximum number of files that can be saved in the directory. Defaults to `500`. When this limit is reached, the oldest file is removed.

**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 request line in the file. This is not included by default.

# Send to JMS

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 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 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 installation, and `PLATFORM` is the platform on which API Gateway is installed (`Linux.x86_64`). 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**. Alternatively, you can configure JMS services under the **Environment Configuration** > **External Connections** node in the Policy Studio tree. For more details, see *Configure messaging services* on page 1.

**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.

**Note** The destination name to use depends on the configured **Destination type**. For example, for IBM WebSphere MQ, this name may need to use a format of `queue://`. For details on destination name requirements, please see the user documentation for your third-party JMS provider tool.

**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 (`${id}`). However, you can use a proprietary correlation system, perhaps using MIME message IDs instead of API Gateway message IDs.

**Correlation ID**:
Enter an identifier for the message that API Gateway uses to correlate response messages with the corresponding request messages. Usually, if `${id}` is specified in the **Message ID** field, it is also used here to correlate request messages with their correct response messages.

**Message Type**:
This 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 recommendation. Alternatively, to serialize a list of name-value pairs to the JMS message, choose the option to create a `MapMessage`.

Select one of the 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 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 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 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 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 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** 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 messages 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 uses asynchronous or synchronous communication when talking to the messaging system. For example, to use asynchronous communication, 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 waits on a response message from a queue or topic from the messaging system. API Gateway 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 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 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 starts up. Only API Gateway can read from the temporary queue, but any application can write to it. API Gateway 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 waits to receive a response:

- **Wait with timeout (ms)**:
  API Gateway waits a specific time period to receive a response before it times out. If API Gateway times out waiting for a response, the **Send to JMS** filter fails. Enter the timeout value in milliseconds. The default value of `10000` means that API Gateway 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 an SQL 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 instance only receives messages whose headers and properties match the selector. For example:

  ```
  JMSCorrelationID='${params.query.id}'
  ```

**Note** 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 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 and infers the JMS response message type based on the object type stored in the message.

# Static router

## Overview

API Gateway 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 *Rewrite URL* on page 402.

Depending on how API Gateway is perceived by the client, different combinations of routing filters can be used. For an introduction to using the various filters in the **Routing** category, see "Get started with routing configuration" in the *API Gateway Policy Developer Guide*.

## Configuration

Configure the following fields 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 should send the message to the remote machine over plain HTTP.

**HTTPS** :
Select this option if API Gateway should send the message to the remote machine over a secure channel using SSL. You can use a **Connection** filter to configure API Gateway to mutually authenticate to the remote system.

# Route to TIBCO Rendezvous

## Overview

TIBCO Rendezvous is a low latency messaging product for real-time high throughput data distribution applications. It facilitates the exchange of data between applications over the network. A TIBCO Rendezvous *daemon* runs on each participating node on the network. All data sent to and read by each application passes through the daemon. API Gateway uses the TIBCO Rendezvous API to communicate with a TIBCO Rendezvous daemon running locally (by default) to send messages to other TIBCO Rendezvous programs.

You can configure the **TIBCO Rendezvous** filter to route messages (using a TIBCO Rendezvous daemon) to other TIBCO Rendezvous programs. This filter is found in the Routing category of filters.

## Configuration

Configure the following fields to route messages to other TIBCO Rendezvous programs:

**Name**:
Enter an appropriate name for this filter.

**TIBCO Rendezvous Daemon to Use**:
Click the button on the right, and select a previously configured TIBCO Rendezvous Daemon from the tree. API Gateway sends messages to the specified TIBCO **Rendezvous Subject** on this daemon. To add a TIBCO Rendezvous Daemon, right-click the **TIBCO Rendezvous Daemons** tree node, and select **Add a TIBCO Rendezvous Daemon**. For more details, see the *API Gateway Policy Developer Guide*.

**Rendezvous Subject**:
The message is sent with the subject entered here meaning that all other TIBCO daemons on the network that have subscribed to this subject name will receive the message. The subject name comprises a series of elements, including wild cards (for example, `*`), separated by dot characters, for example:

- `news.sport.soccer`
- `news.sport.*`
- `FINANCE.ACCOUNT.SALES`

For more information on the subject name syntax, see the TIBCO Rendezvous documentation.

**Add Fields**
Click the **Add** button to add details about a particular field to add to the message. On the **Message Field Definition** dialog, complete the following fields:

**Field Name**:
Enter the name of the field to send in the message.

**Type**:
Select the data type of the value specified in either of the following fields:

**Set value to the following constant value**:
You can explicitly set this value by entering it here.

**Set value to the object found in the following attribute**:
To dynamically populate the field value using the contents of a message attribute, select the attribute from the list. At runtime, the contents of the message attribute are placed into the message that is sent to TIBCO Rendezvous.

# Wait for response packets

## Overview

*Packet sniffers* are a type of passive service. Rather than opening up a TCP port and *actively* listening for requests, the packet sniffer *passively* reads data packets off the network interface. The sniffer assembles these packets into complete messages that can then be passed into an associated policy.

Because the packet sniffer operates passively (does not listen on a TCP port) and transparently to the client, it is most useful for monitoring and managing web services. For example, you can deploy the sniffer on a machine running a web server acting as a container for web services.

Assuming that the web server is listening on TCP port 80 for traffic, the packet sniffer can be configured to read all packets destined for port 80 (or any other port, if necessary). The packets can then be marshaled into complete HTTP/SOAP messages by the sniffer and passed into a policy that, for example, logs the message to a database.

## Packet sniffer configuration

Because packet sniffers are mainly used as passive monitoring agents, they are usually created in their own service group. For example, to create a new group, right-click the API Gateway instance under **Environment Configuration** > **Listeners** in the Policy Studio tree, and select **Add Service Group**. Enter `Packet Sniffer Group` in the dialog.

You can then add a relative path service to this group by right-clicking the `Packet Sniffer Group`, and selecting **Add Relative Path**. Enter a path in the field provided, and select the policy to dispatch messages to when the packet sniffer detects a request for this path (after it assembles the packets). For example, if the relative path is configured as `/a`, and the packet sniffer assembles packets into a request for this path, the request is dispatched to the policy selected in the relative path service.

Finally, to add the packet sniffer, right-click the `Packet Sniffer Group` node, and select **Packet Sniffer** > **Add**, and complete the following fields:

**Device to Monitor**:
Enter the name of the network interface that the packet sniffer monitors. The default is `any` (valid on Linux only). On UNIX, network interfaces are usually identified by names like `eth0` or `eth1` . On Windows, names are more complicated (for example, `\Device\NPF_{00B756E0-518A-4144 ... }`).

**Filter**:
You can configure the packet sniffer to only intercept certain types of packets. For example, it can ignore all UDP packets, only intercept packets destined for port 80 on the network interface, ignore packets from a certain IP address, listen for all packets on the network, and so on.

The packet sniffer uses the libpcap library filter language to achieve this. This language has a complicated but powerful syntax that enables you to *filter* what packets are intercepted, and what packets are ignored. As a general rule, the syntax consists of one or more expressions combined with conjunctions, such as `and`, `or`, and `not`.

The following table lists a few examples of common filters and explains what they filter:

| Filter Expression | Description |
|---|---|
| `port 80` | Captures only traffic for the HTTP port (port 80). |
| `host 192.168.0.1` | Captures traffic to and from IP address 192.168.0.1. |
| `tcp` | Captures only TCP traffic. |
| `host 192.168.0.1 and port 80` | Captures traffic to and from port 80 on IP address 192.168.0.1. |
| `tcp portrange 8080-8090` | Captures all TCP traffic destined for ports from 8080 through to 8090. |
| `tcp port 8080 and not src host 192.168.0.1` | Captures all TCP traffic destined for port 8080 but not from IP address 192.168.0.1. |

The default filter of `tcp` captures all TCP packets arriving on the network interface. For more details on how to configure filter expressions, see http://www.tcpdump.org/tcpdump_man.html.

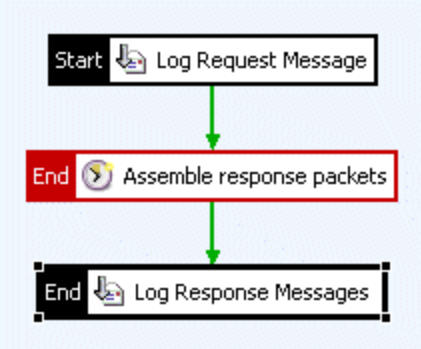
**Promiscuous Mode**:
When listening in promiscuous mode, the packet sniffer captures all packets on the same Ethernet network, regardless of whether the packets are addressed to the network interface that the sniffer is monitoring.

# Response packet sniffing

API Gateway can capture both incoming and outgoing packets when it is listening passively (not opening any ports) on the network interface. For example, a web service is deployed in a web server that listens on port 80. API Gateway can be installed on the same machine as the web server. It is configured *not* to open any ports and to use a packet sniffer to capture all packets destined for TCP port 80.

When packets arrive on the network interface that are destined for this port, they are assembled by the packet sniffer into HTTP messages and passed into the configured policy. Typically, this policy logs the message to an audit trail, and so usually consists of just a **Log Message** filter.

To also log response messages passively, as is typically required for a complete audit trail, you can use the **Wait for Response Packets** filter to correlate response packets with their corresponding requests. The **Wait for Response Packets** filter assembles the response messages into HTTP messages and can then log them again using the **Log Message Payload** filter. The following policy logs both request and response messages captured transparently by the packet sniffer:




You can see from the policy that the first logging filter logs the *request* message. By this stage, the packet sniffer has assembled the request packets into a complete HTTP request, and this is what is passed to the **Log Request Message** filter. The **Assemble response packets** filter is a **Wait for Response Packets** filter that assembles response packets into complete HTTP response messages and passes them to the **Log Response Message** filter, which logs the complete response message.

For more details on the **Log Message Payload** filter, see *Log message payload* on page 351.

# Security service filters 18

The following filters enable you to configure XML encryption and signature:



# Encrypt and decrypt web services

## Overview

The **Encrypt Web Service** filter allows the API Gateway to act as an XML *encrypting* web service, where clients can send up XML blocks to the API Gateway that are required to be encrypted. The API Gateway encrypts the XML data, replacing it with `<EncryptedData>` blocks in the message. The encrypted content is then returned to the client.

Similarly, the **Decrypt Web Service** filter allows the API Gateway to act as an XML *decrypting* web service, where clients can send up `<EncryptedData>` blocks to the API Gateway, which decrypts them and returns the plain-text data back to the client.

By deploying the API Gateway as a centralized encryption and decryption service, clients application can abstract out the security layer from their core business logic. This simplifies the logic of the client applications and makes the task of managing and configuring the security aspect a lot simpler because it is centralized.

Furthermore, the API Gateway's XML and cryptographic acceleration capabilities ensure that the process of encrypting and decrypting XML messages—a task that involves some very CPU-intensive operations—is performed at optimum speed.

See also *DSS signature generation* on page 415

## Configuration

To configure both the **Encrypt Web Service** and **Decrypt Web Service** filters, enter a descriptive name for the filter to display in a policy.

The settings for the filters are configured in the **XML-Encryption Settings** and **XML-Decryption Settings** filters in the Encryption category. For more details how to configure these filters, see *XML encryption settings* on page 291 and *XML decryption settings* on page 285.

# DSS signature generation

## Overview

The **Sign Web Service** filter enables the API Gateway to generate XML signatures as a service according to the OASIS Digital Signature Services (DSS) specification. The DSS specification describes how a client can send a message containing an XML signature to a DSS signature web service that can sign the relevant parts of the message, and return the resulting XML signature to the client.

The advantage of this approach is that the signature generation code is abstracted from the logic of the web service and does not have to be coded into the web service. Furthermore, a centralized DSS server provides a single implementation point for all XML signature related services, which can then be accessed by all services running in your API Gateway domain. This represents a much more manageable solution than one in which the security layer is coded into each service.

See also *DSS signature verification* on page 416

## Configuration

Complete the following fields:.

**Name**:
Enter a descriptive name for the filter to display in a policy.

**Signing Key**:
Click to select a private key from the certificate store. This key is used to perform the signing operation.

# STS web service

## Overview

The **STS Web Service** filter can be used to expose a Security Token Service (STS), allowing clients to obtain security tokens for use in your system.

See also *STS client authentication* on page 111.

## Configuration

Complete the following field:

**Name**:
Enter a descriptive name for the filter to display in a policy.

# DSS signature verification

## Overview

The **Verify Sig Web Service** filter enables the API Gateway to verify XML signatures as a service according to the OASIS Digital Signature Services (DSS) specification. The DSS specification describes how a client can send a message containing an XML signature to a DSS signature verification web service that can verify the signature and return the result of the verification to the client.

The advantage of this approach is that the signature verification code is abstracted from the logic of the web service and does not have to be coded into the web service. Furthermore, a centralized DSS server provides a single implementation point for all XML signature related services, which can then be accessed by all services running your system. This represents a much more manageable solution that one in which the security layer is coded into each web service.

See also *DSS signature generation* on page 415

## Configuration

Complete the following fields to configure the **Verify Sig Web Service** filter.

**Name**:
Enter a descriptive name for the filter to display in a policy.

**Find Signing Key**:
The public key to be used to verify the signature can be retrieved from one of the following locations:

- **Via KeyInfo in Message**:
  The verification certificate can be located using the `<KeyInfo>` block in the XML signature. For example, the certificate could be contained in a `<BinarySecurityToken>` element in a WSSE security header. The `<KeyInfo>` section of the XML signature can then reference this `BinarySecurityToken`. The API Gateway can automatically resolve this reference to locate the certificate that contains the public key necessary to perform the signature verification.
- **Via Selector Expression**:
  The certificate used to verify the signature can be extracted from the message attribute specified in the selector expression (for example, `${certificate}`). The certificate must

have been placed into the specified attribute by a predecessor of the **Verify Sig Web Service** filter. For more details on selector expressions, see "Select configuration values at runtime" in the *API Gateway Policy Developer Guide*.

- **Via Certificate in LDAP**:
  The certificate used to verify the signature can be retrieved from an LDAP directory. Click the button next to this field, and select a previously configured LDAP directory in the tree. To add an LDAP directory, right-click the **LDAP Connections** tree node, and select **Add an LDAP Connection** . Alternatively, you can configure LDAP connections under the **External Connections** node in the Policy Studio tree. For more details, see "Configure LDAP directories" in the *API Gateway Policy Developer Guide*.
- **Via Certificate in Store**:
  Finally, the verification certificate can be selected from the certificate store. Click the **Select** button to view the certificate that has been added to the store. Select the verification certificate by selecting the check box next to it in the table.

# Sun Access Manager filters 19

The following filters enable you to integrate with Sun Access Manager:



# Sun Access Manager authorization

**Note** This feature has been deprecated and will be removed in a future release. See *Oracle Access Manager filters* on page 364.

Sun Access Manager is an identity management product that provides authentication, authorization, and single sign-on (SSO) capabilities. API Gateway uses an access manager policy agent embedded in the **Authorization** filter to authorize previously authenticated users for a particular resource.

## Configuration

Complete the following fields to authorize an authenticated user against Sun Access Manager:

**Name**:
Enter a descriptive name for this filter.

**Service Name**:
Enter the name of the configured access manager service to which the **Resource** specified below belongs.

**Resource**:
Enter the name of the resource for which the request must be authorized. You can configure access rules for protected resources with Sun Access Manager. For more details, see the access manager documentation.

**Note** The value entered here can include one or more message attribute selectors. At runtime, each selector is replaced by the value of the corresponding API Gateway message attribute.

For example, the `${http.request.uri}` selector is replaced by the relative path on which the request was received, while the `${http.request.verb}` selector is replaced by the HTTP verb used in the request to the API Gateway. To view a list of available message attribute selectors, enter

the $ character in this field. You can then select one or more appropriate string-based selectors from the list to specify the resource name. For more details on selectors, see "Select configuration values at runtime" in the *API Gateway Policy Developer Guide*.

For example, the default configuration entry for this field is:

```
http://www.yourserverhostnamehere.com${http.request.uri}
```

At runtime, each of the selectors is replaced by the corresponding value to produce a **Resource** such as the following:

```
http://HOST_MACHINE:8080/services/getHello
```

`HOST_MACHINE` represents the IP address or host name of the machine running API Gateway.

**Action**:
Sun Access Manager can grant or restrict access to a protected resource based on the action in the request. For example, an administrator user can `POST` to a web page resource, but a normal user can only view the resource using `GET` requests. Therefore, for web page resources, the **Action** specified typically corresponds to an HTTP verb. However, other applications might have custom operations associated with them that can also be specified.

By default, the **Action** field is configured to use the `${http.request.verb}` selector, which is replaced at runtime by the HTTP verb (for example, `POST`) used by the client to send the request to API Gateway.

# Sun Access Manager SSO session logout

**Note** This feature has been deprecated and will be removed in a future release. See *Oracle Access Manager filters* on page 364.

Sun Access Manager is an identity management product that provides authentication, authorization, and single sign-on (SSO) capabilities. When a user authenticates to Sun Access Manager, they are granted an SSO token, which they can use to obtain access to other services without having to reauthenticate at each service. Each service employs an access manager policy agent to validate the SSO token to determine whether to grant the user access to the requested resource.

API Gateway can act as an access manager policy agent in this manner through the use of the **Log out session** filter. This filter can be used to invalidate an SSO token that has been obtained from Sun Access Manager. After a token has been invalidated, the user is effectively logged out of all services that are protected by the Sun Access Manager server. The user is no longer able to use the token and so must reauthenticate to access manager to obtain an SSO token again.

## Configuration

Complete the following fields to configure this filter:

**Name**:
Enter an appropriate name for the filter.

**Selector expression to retrieve SSO token ID**:
Specify the name of API Gateway message attribute that contains the SSO token to log out from Sun Access Manager. Typically this token is obtained by authenticating to Sun Access Manager. In this case, the token is stored by default in the `sun.sso.token` message attribute.

Alternatively, if the SSO token is inserted into an HTTP header (for example, as a cookie), you can use the **Retrieve from HTTP Header** filter (see *Retrieve from HTTP header* on page 54) to extract the token from the HTTP header, and store it in the specified message attribute (for example, the `sun.sso.token` attribute).

Similarly, the SSO token might have been stored in the message after an authentication event to Sun Access Manager (for example, in a SAML authentication or authorization statement). In such cases, API Gateway can retrieve the SSO token using the **Retrieve from Message** filter (see *Retrieve attribute from message* on page 62), and then store it in a message attribute.

# Retrieve attributes from Sun Access Manager

**Note** This feature has been deprecated and will be removed in a future release. See *Oracle Access Manager filters* on page 364.

Sun Access Manager is an identity management product that provides authentication, authorization, and single sign-on (SSO) capabilities. When a user authenticates to Sun Access Manager they are granted an SSO token, which they can use to obtain access to other services without having to reauthenticate at each service.

The token contains details about the authentication or authorization event, including the token lifetime, the resource for which the user was authorized, and attributes relating to the authenticated user, amongst others. An access manager policy agent deployed at the web services endpoint can retrieve the user attributes directly from the SSO token without the need to connect back to the access manager server.

API Gateway can be configured to act as such a policy agent through the use of the **Retrieve Attributes** filter. The filter is parametrized by the message attribute that contains the SSO token, from which the user attributes can be retrieved.

The retrieved attributes are stored in the `attributes.lookup.list` message attribute, which is a map of user attributes to their values. The keys correspond to the names of the attributes as they are stored in access manager. For example, if the user has `role`, `email`, and `department` attributes configured in access manager, the `attributes.lookup.list` message attribute contains entries for each of these attributes, together with their retrieved values.

To enable you to easily access individual attributes, the retrieved user attributes are also stored in separate message attributes. These message attributes are dynamically named according to the following convention:

```
user.[attribute_name].[index]
```

Each message attribute is prefixed with `user.`, followed by the name of the user attribute as configured in access manager (for example, `role`). The `[index]` part of the name is used in cases where the user attribute has multiple values, such as multiple roles or phone numbers.

For example, assuming that the `email`, (multivalued) `role`, and `department` attributes have been configured for the user, the following message attributes would be created for each of these attributes respectively:

| Message attribute | User attribute | Value |
|---|---|---|
| `user.email.1` | `email` | `eng_manager@company.org` |
| `user.role.1` | `role` | Engineer |
| `user.role.2` | `role` | Manager |
| `user.role.3` | `role` | Administrator |
| `user.department.1` | `department` | Engineering |

The table above illustrates how a multivalued user attribute such as `role` is processed so that each value is stored in a separate indexed message attribute, for example, `user.role.1`, `user.role.2`, and `user.role.3`.

# Configuration

The following fields must be configured for this filter:

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

**Selector expression to retrieve SSO token ID**:
The attributes are retrieved from the Sun Access Manager SSO token, which is stored by default in the `sun.sso.token` message attribute after a user authenticates to access manager.

In cases where the user has authenticated to Sun Access Manager independently of the API Gateway but has, for example, injected the SSO token into a SAML authentication or authorization assertion, the **Retrieve from Message** filter (see *Retrieve attribute from message* on page 62) can be used to extract the token from the assertion and store it in a message attribute. This message attribute must be specified in this field.

Similarly, if the SSO token was inserted into an HTTP header, the **Retrieve from HTTP Header** filter (see *Retrieve from HTTP header* on page 54) can be used to extract the token from the HTTP header and store it in a message attribute.

**Retrieve the following attributes from Sun Access Manager**:
In cases where only a certain named subset of the total number of available attributes need to be

retrieved for a given user, you can list the names of the attributes to retrieve in this table. The filter only retrieves these named attributes. If no attributes are explicitly named in this section, the filter retrieves all available attributes for the user.

Click the **Add** button to add a named attribute to the table. Simply enter a name for the attribute in the field provided. Existing named attributes can be edited and deleted by clicking the **Edit** and **Remove** buttons.

The retrieved attributes are added to the `attribute.lookup.list` message attribute. The contents of the `attribute.lookup.list` attribute can be used, for example, to create a SAML attribute assertion for the user. If a list of named attributes is configured in this section, only these attributes appear in the SAML attribute assertion.

For example, if you configure this filter to only retrieve the `EMPLOYEENUMBER` and `SN` attributes from Sun Access Manager (despite the fact that are several more attributes configured for this user), you can generate the following SAML attribute assertion:

```
<saml:AttributeStatement>
  <saml:Subject>
    <saml:NameIdentifier Format="urn:oasis:names:tc:SAML:1.1:nameid-
format:unspecified">
      qauser
    </saml:NameIdentifier>
  </saml:Subject>
  <saml:Attribute AttributeName="EMPLOYEENUMBER"
AttributeNamespace="urn:vordel:attribute:1.0">
    <saml:AttributeValue>12345</saml:AttributeValue>
  </saml:Attribute>
  <saml:Attribute AttributeName="SN" AttributeNamespace="urn:vordel:attribute:1.0">
    <saml:AttributeValue>User</saml:AttributeValue>
  </saml:Attribute>
</saml:AttributeStatement>
```

# Sun Access Manager SSO token validation

**Note** This feature has been deprecated and will be removed in a future release. See *Oracle Access Manager filters* on page 364.

Sun Access Manager is an identity management product that provides authentication, authorization, and single sing-on (SSO) capabilities. When a user authenticates to Sun Access Manager they are granted an SSO token, which they can use to obtain access to other services without having to reauthenticate at each service. Each service employs an access manager policy agent to simply validate the SSO token to determine whether or not to grant the user access to the requested resource.

API Gateway can act as an access manager policy agent in this manner through the use of the **SSO Token Validation** filter. This filter validates the SSO token that has already been stored in a specified message attribute.

**Note** This filter validates the SSO token only. It does not reauthenticate the user to Sun Access Manager.

## Configuration

Complete the following fields to validate the SSO token:

**Name**:
Enter a name for the filter.

**Selector expression to retrieve SSO token ID**:
Specify the name of the message attribute that contains the Sun Access Manager SSO token. Typically this token is obtained by authenticating to Sun Access Manager. In this case, the token is stored by default in the `sun.sso.token` message attribute.

Alternatively, the SSO token might have been stored within the message after an authentication event to Sun Access Manager, for example, in a SAML authentication or authorization statement. In such cases, API Gateway can retrieve the SSO token using the **Retrieve from Message** filter (see *Retrieve attribute from message* on page 62) and store it in a message attribute, for example, the `sun.sso.token` attribute. This attribute can then be specified here.

Similarly, if the SSO token was inserted into an HTTP header, the **Retrieve from HTTP Header** filter (see *Retrieve from HTTP header* on page 54) can be used to extract the token from the HTTP header and store it in a message attribute.

# Sun Access Manager certificate authentication

**Note** This feature has been deprecated and will be removed in a future release. See *Oracle Access Manager filters* on page 364.

Sun Access Manager is an identity management product that provides authentication, authorization, and single sign-on (SSO) capabilities. API Gateway uses X.509 certificates to authenticate to Sun Access Manager. When users have been successfully authenticated, they receive an SSO token, which they can use to obtain access to resources that are protected by the access manager. The **X.509 Certificate Authentication** is filter configures how API Gateway uses X.509 certificates to authenticate to Sun Access Manager.

See also *Sun Access Manager authorization* on page 418.

## Configuration

Complete the following fields.

### *Sun Access Manager settings*

Complete the following fields to configure Sun Access Manager login details:

**Login module name**:
Enter the name of the access manager *module instance* that is responsible for logging in users. The module entered must be registered with the **Realm** specified below.

**Realm**:
Specify the realm that the access manager *module instance* specified above belongs to.

## *SSO settings*

Complete the following fields to configure SSO details:

**Create SSO Token**:
Select this check box if Sun Access Manager should create an SSO (Single Sign-On) token when a user has successfully authenticated. The user can then use this SSO token to gain access to other resources that are protected by Access Manager.

**Store SSO Token in attribute named**:
Enter the name of the message attribute in which to store the SSO token. Most of API Gateway's access manager integration filters require the name of this attribute to retrieve the SSO token and process it.

**Add SSO Token to user attributes**:
If this option is selected, the SSO token that is retrieved after successfully authenticating to Sun Access Manager is stored in the `attribute.lookup.list` message attribute, which contains a list of user attributes.

**Note** Do not select this option if you plan to insert a SAML attribute statement into the message (using one of the SAML injection filters) later in the policy. If this option is unselected, the SSO token is inserted into the SAML attribute statement, which might not always be desirable. For this reason, the decision is left to you on whether to add the SSO token to the `attribute.lookup.list` of user attributes depending on your specific requirements.

# Trust filters 20

The following filters are available in the Trust category in Policy Studio:



# Consume WS-Trust message

## Overview

You can configure the API Gateway to consume various types of WS-Trust messages. For example, `RequestSecurityToken` (RST), `RequestSecurityTokenResponse` (RSTR), and `RequestSecurityTokenResponseCollection` (RSTRC).

For more details on WS-Trust messages and their semantics and format, see the WS-Trust specification. For details on creating WS-Trust messages, see *Create WS-Trust message* on page 428.

## Configuration

Configure the fields in the following sections.

### *Message types*

The API Gateway can consume the following types of WS-Trust messages.Select the appropriate message type based on your requirements:

- **RST:RequestSecurityToken**:
  The RST message contains a request for a single token to be issued by the Security Token Service (STS).
- **RSTR:RequestSecurityTokenResponse**:
  The RSTR message is sent in response to an RST message from a token requestor. It contains the token issued by the STS.
- **RSTRC:RequestSecurityTokenResponseCollection**:
  The RSTRC message contains an RSTR (containing a single issued token) for each RST that was received in an RSTC message.

## Message consumption settings

The configuration options available on the **Message Consumption** tab enable you to extract various parts of the WS-Trust message and store them in message attributes for use in subsequent filters.

**Extract Token**:
Extracts a `<RequestedSecurityToken>` from theWS-Trust message and stores it in a message attribute. Select the expected value of the `<TokenType>` element in the `<RequestSecurityToken>` block. The default URI is `http://schemas.xmlsoap.org/ws/2005/02/sc/sct`.

**Extract BinaryExchange**:
Extracts a `<BinaryExchange>` token from the message and stores it in a message attribute. Select the `ValueType` of the token from the list.

**Extract Entropy**:
The client can provide its own key material (entropy) that the token issuer may use when generating the token. The issuer can use this entropy as the key itself, it can derive another key from this entropy, or it can choose to ignore the entropy provided by the client altogether in favor of generating its own entropy.

**Extract RequestedProofToken**:
Select this option to extract a `<RequestedProofToken>` from the WS-Trust message and store it in a message attribute for later use. You must select the type of the token (`encryptedKey` or `computedKey`) from the list.

**Extract CancelTarget**:
You can select this option to extract a `<CancelTarget>` block from the WS-Trust message and store it in a message attribute.

**Extract RequestedTokenCancelled**:
You can select this option to extract a `<RequestedTokenCancelled>` block from the WS-Trust message and store it in a message attribute.

**Match Context ID** :
Select this option to correlate the response message from the STS with a specific request message. The `Context` attribute on the `RequestSecurityTokenResponse` message is compared to the value of the `ws.trust.context.id` message attribute, which contains the context ID of the current token request.

**Extract Lifetime**:
Select this option to remove the `<Lifetime>` elements from the WS-Trust token.

**Extract Authenticator**:
Select this option to extract the `<Authenticator>` from the WS-Trust token and store it in a message attribute.

## Advanced settings

The following fields can be configured on the **Advanced** tab:**WS-Trust Namespace**:
Enter the WS-Trust namespace that you expect all WS-Trust elements to be bound to in tokens that are consumed by this filter. The default namespace is `http://schemas.xmlsoap.org/ws/2005/02/trust`.

**Cache Security Context Session Key**:
Click the browse button, and select the cache to store the security context session key. The session key (the value of the `security.context.session.key` attribute), is cached using the value of the `security.context.token.unattached.id` message attribute as the key into the cache.

You can select a cache from the list of currently configured caches in the tree. To add a cache, right-click the **Caches** tree node, and select **Add Local Cache** or **Add Distributed Cache**. Alternatively, you can configure caches under the **Libraries** node in the Policy Studio tree. For more details, see "Global caches" in the *API Gateway Policy Developer Guide*.

**Lifetime of ComputedKey**:
The settings in this section enable you to add a time stamp to the extracted `computedKey` using the values specified in the `<Lifetime>` element. This section is enabledonly after selecting the **Extract RequestedProofToken** check box above, selecting the `computedKey` option from the associated list, and finally by selecting **Extract Lifetime**. Configure the following fields in this section:

- **Add Lifetime to ComputedKey**:
  Adds the `<Lifetime>` details to the`security.context.session.key` message attribute.This enables you to check the validity of the key every time it is used against the details in the `<Lifetime>` element.
- **Format of Timestamp**:
  Specify the format of the time stamp using the Java date and time pattern settings.
- **Timezone**:
  Select the appropriate time zone from the list.
- **Drift**:
  To allow for differences in the clock times on the machine on which the WS-Trust token was generated and the machine running the API Gateway, enter a drift time. This allows for differences in the clock times on these machines and is used when validating the time stamp on the `computedKey`.

**Verify Authenticator Using**:
You can verify the authenticator using either the **Generated** or **Consumed** message. In either case select the appropriate type of WS-Trust message from the available options.

# Create WS-Trust message

## Overview

You can configure the API Gateway to create various types of WS-Trust messages. The API Gateway can act both as a WS-Trust client when generating a`RequestSecurityToken` (RST) message. It can also act as a WS-Trust service, or Security Token Service (STS), when generating the following messages:

- `RequestSecurityTokenResponse` (RSTR)
- `RequestSecurityTokenResponseCollection` (RSTRC)

A token requestor generates an RST message and sends it to the STS, which generates the required token and returns it in an RSTR message. If several tokens are required, the requestor can send up multiple RST messages in a single `RequestSecurityTokenCollection` (RSTC) request. The STS generates an RSTR for each RST in the RSTC message and returns them all in batch mode in an RSTRC message.

For more details on the various types of WS-Trust messages and their semantics and format, see the WS-Trust specification. For details on consuming WS-Trust messages, see *Consume WS-Trust message* on page 425.

## Configuration

Configure the fields in the following sections.

### *Message types*

The **Create WS-Trust** filter can create the following types of WS-Trust message. Select the appropriate message type based on your requirements:

- **RST:RequestSecurityToken**:
  The RST message contains a request for a single token to be issued by the STS.
- **RSTR:RequestSecurityTokenResponse**:
  The RSTR message is sent in response to an RST message from a token requestor. It contains the token issued by the STS.
- **RSTRC:RequestSecurityTokenResponseCollection**:
  The RSTRC message contains an RSTR (containing a single issued token) for each RST that was received in an RSTC message.

## General message creation settings

The settings on this tab specify characteristics of the WS-Trust message. The following fields are available:

**Insert Token Type**:
Select the type of token requested from the list. The type of token selected here is returned in the response from the STS. By default, the Security Token Context type is used, which is identified by the URI `http://schemas.xmlsoap.org/ws/2005/02/sc/sct`.

**Binary Exchange**:
You can use a `<BinaryExchange>` when negotiating a secure channel that involves the transfer of binary blobs as part of another security negotiation protocol (for example, SPNEGO). The contents of the blob are always Base64-encoded to ensure safe transmission.

Select the **Binary Exchange** option to use a negotiation-type protocol for the exchange of keys, such as SPNEGO. The URI selected in the **Value Type** field identifies the type of the negotiation in which the blob is used. The URI is placed in the `ValueType` attribute of the `<BinaryExchange>` element.

**Entropy**:
The client can provide its own key material (entropy) that the token issuer may use when generating the token. The issuer can use this entropy as the key itself, it can derive another key from this entropy, or it can choose to ignore the entropy provided by the client altogether in favor of generating its own entropy.

Select this option to generate some entropy, which is included in the `<wst:entropy>` element of the `<wst:RequestSecurityToken>` block.

**Insert Key Size**:
The client can request the key size (in number of bits) required in a`<RequestSecurityToken>` request. However, the WS-Trust token issuer does not have to use the requested key size. It is merely intended as an indication of the strength of security required.The default request key size is 256 bits.

**Insert Lifetime**:
Select this option to insert a `<Lifetime>` element into the WS-Trust message. Use the associated fields to specify when the message expires. The lifetime of the WS-Trust message is expressed in terms of `<Created>` and `<Expires>` elements.

**Lifetime Format**:
The specified date and time pattern string determines the format of the`<Created>` and `<Expires>` elements. The default format is `yyyy-MM-dd'T'HH:mm:ss.SSS'Z'` ,which can be altered if necessary. For more details on how to use this format, see the Javadoc for the `java.text.SimpleDateFormat` Java class in the Java API specification.

**Insert RequestedTokenCancelled**:
Select this option to insert a `<RequestedTokenCancelled>` element into the generated WS-Trust message.

## RST creation settings

The following configuration fields specify the way in which a WS-Trust RST message is created:

**Insert Request Type**:
You can create two types of RST message. Select one of the following request types from the list:

- **Issue**: RST message used to request the STS to *issue* a token for the requestor
- **Cancel**: RST message used to cancel a specific token

**Insert Key Type**:
Select this option to insert the key type into the RST WS-Trust message.

**Insert Computed Key Algorithm**:
Select this option to insert the computed key algorithm into the message.

**Insert Endpoint Reference**:
Select this option and enter a suitable endpoint to include an endpoint reference in the RST message.

## RSTR creation settings

The following configuration fields determine the way in which a WS-Trust RSTR message is created:

**Insert RequestedProofToken**:
Select this check box to insert a `<RequestedProofToken>` element into the generated WS-Trust message. The type of this token can be set to either `computedKey` or `encryptedKey` using the associated list.

**Insert Authenticator**:
Select this option to insert an authenticator into the RSTR message.

## Advanced settings

This section enables you to configure certain advanced aspects of the SOAP message that is sent to the WS-Trust service.

**WS-Trust Namespace**:
Enter the WS-Trust namespace to bind all WS-Trust elements to in this field. The default namespace is `http://schemas.xmlsoap.org/ws/2005/02/trust`.

**WS-Addressing Namespace**:
Select the WS-Addressing namespace version to use in all created WS-Trust messages.

**WS-Policy Namespace**:
Select the appropriate WS-Policy namespace from the list. The selected version selected can affect the ordering of tokens that are inserted into the WS-Security header of the SOAP message.

**SOAP Version**:
Select the SOAP version to use when creating the WS-Trust message.

**Overwrite SOAP Method**:
Select this option if the WS-Trust token should overwrite the SOAP method in the request. In this case, the token appears as a direct child of the SOAP Body element. Use this option to preserve the contents of the SOAP Header, if present.

**Overwrite SOAP Envelope**:
Select this option if the generated WS-Trust message should form the entire contents of the message. In other words, the generated WS-Trust message replaces the original SOAP request.

**Content-Type**:
Specify the HTTP content-type of the WS-Trust message. For example, for Microsoft Windows Communication Foundation (WCF), you should use `application/soap+xml`.

**Generate Authenticator Using**:
You can verify the authenticator using the **Generated** or **Consumed** message. In either case, select the appropriate type of WS-Trust message from the available options.

# Utility filters 21

The following filters are available in the Utility category in Policy Studio:



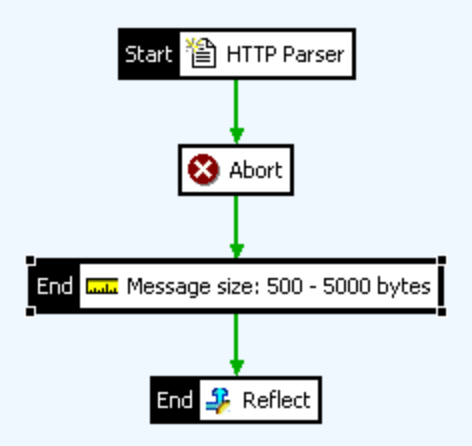
**Note** For information on the **Scripting Language** filter see "Scripting language filter" in the *API Gateway Policy Developer Guide*.

# Abort policy

## Overview

You can use the **Abort** filter to force a policy to throw an exception. You can use it to test the behavior of the policy when an exception occurs.

For example, to quickly test how the policy behaves when a **Message Size** filter throws an exception, you can place an **Abort** filter before it in the policy. The following policy diagram illustrates this:




## Configuration

Enter a meaningful name for the filter to display in a policy.

See also *False filter* on page 438.

# Check group membership

## Overview

The **Check Group Membership** filter checks whether the specified API Gateway user is a member of the specified API Gateway user group. The user and the group are both stored in the API Gateway user store. For more details, see "Manage API Gateway users" in the *API Gateway Policy Developer Guide*.

## Configuration

Configure the following required fields:

**Name**:
Enter an appropriate name for this filter to display in a policy.

**User**:
Enter the user name configured in the API Gateway user store. You can specify this value as a string or as a selector that expands to the value of the specified message attribute at runtime. Defaults to `${authentication.subject.id}`.

**Group**:
Enter the user group name configured in the API Gateway user store (for example, `engineering` or `sales` ). You can specify this value as a string or as selector that expands to the value of the specified message attribute at runtime (for example, `${groupName}`).

**Note** The message attribute specified in the selector must exist on the message whiteboard prior to calling the filter. For more details on selectors, see "Select configuration values at runtime" in the *API Gateway Policy Developer Guide*.

## Possible results

The possible paths through this filter are as follows:

| Result | Description |
|---|---|
| `True` | The specified user is a member of the specified group. |
| `False` | The specified user is not a member of the specified group. |
| `CircuitAbort` | An exception occurred while executing the filter. |

# Copy or modify attributes

## Overview

The **Copy/Modify Attributes** filter copies the values of message or user attributes to other message or user attributes. You can also set the value of a message or user attribute to a user-specified value.

See also *Remove attribute* on page 449.

# Configuration

The table lists the configured attribute-copying rules. To add a new rule, click **Add** and enter values in the dialog to copy a message or user attribute to a different message or user attribute. The **From attribute** represents the source attribute, while the **To attribute** represents the destination attribute.

## *From attribute settings*

The attribute value can be copied from one of the following sources:

- **Message**:
  Select this option to copy the value of a message attribute. Enter the name of the source attribute in the **Name** field.
- **User**:
  Select this option to copy a user attribute stored in the `attribute.lookup.list`. Enter the name (and namespace if the attribute was extracted from a SAML attribute assertion) of the user attribute in the **Name** and **Namespace** fields.

  If there are multiple values stored in the `attribute.lookup.list` for the attribute entered in the **Name** field, only the first value is copied.
- **User entered value**:
  Select this option to copy a user-specified value to an attribute. Enter the new attribute value in the **Value** field. You can enter a selector to represent the value of a message attribute instead of entering a specific value directly (for example, `${authentication.subject.id}`). In this case the value of the `authentication.subject.id` attribute is copied to the named attribute.

## *To attribute settings*

The message can be copied to one of the following types of attributes:

- **Message**:
  The attribute can be copied to any message attribute. Enter the name of the attribute in the **Name** field.
- **User**:
  Select this option if the attribute or value should be copied to a user attribute stored in the `attribute.lookup.list`. Specify the name and namespace (if necessary) of this attribute in the **Name** and **Namespace** fields.

  If there are multiple values stored in the `attribute.lookup.list` for the attribute entered in the **Name** field of the **From attribute** section, the attribute value is copied to the first occurrence of the attribute name in list.

Select **Create list attribute** if the new attribute can contain several items.

# Evaluate selector

## Overview

The **Evaluate Selector** filter enables you to evaluate the contents of a specified selector expression, and return a boolean result. A selector is a special syntax that enables API Gateway configuration settings to be evaluated and expanded at runtime.

This filter enables you to evaluate a specified selector expression and make a decision in a policy based on whether the expression value fails or passes. For example, you could use the following expression to check if the user belongs to a particular group that allows the user to access a particular resource:

```
${user[0].memberOf.contains("CN=Group Policy Creator
Owners,CN=Users,DC=acmeqa,DC=com")}
```

This expression checks if the `memberOf` attribute retrieved for the first `user` contains the specified value (in this case, membership of a particular group). If the expression matches, the filter passes.

Alternatively, you could use the following selector expression to check if the user email address is valid:

```
${user[0].mail.contains("admin@qa.acme.com")}
```

This expression checks if the `mail` attribute retrieved for the first `user` contains the specified value (in this case, a particular email address). If the expression matches, the filter passes.

For more details on selectors, see "Select configuration values at runtime" in the *API Gateway Policy Developer Guide*.

## Configuration

Configure the following settings:

**Name**:
Enter a descriptive name for this filter to display in a policy.

**Expression**:
Enter the selector expression to be evaluated. Defaults to the following selector expression:

```
${1 + 1 == 2}
```

# Execute external process

## Overview

This filter enables you to execute an external process from a policy. It can execute any external process (for example, start an SSH session to connect to another machine, run a script, or send an SMS message).

## Configuration

Complete the following fields:

**Name**:
Enter an appropriate name for the filter to display in a policy.

### *Command settings*

The **Command** tab includes the following fields:

| | |
|---|---|
| **Command to execute** | Specify the full path to the command to execute (for example, `c:\cygwin\bin\mkdir.exe`). |
| **Arguments** | Click **Add** to add arguments to your command. Specify an argument in the **Value** field (for example, `dir1`), and click **OK**. Repeat these steps to add multiple arguments (for example, `dir2` and `dir3`). |
| **Working directory** | Specify the directory to run the command from. You can specify this using a selector that is expanded to the specified value at runtime. Defaults to `${environment.VINSTDIR}`, where `VINSTDIR` is the location of a running API Gateway instance. For more details, see "Select configuration values at runtime" in the *API Gateway Policy Developer Guide*. |
| **Expected exit code** | Specify the expected exit code for the process when it has finished. Defaults to `0`. |
| **Kill if running longer than (ms)** | Specify the number of milliseconds after which the running process is killed. Defaults to `60000`. |

### *Advanced settings*

The **Advanced** tab includes the following fields:

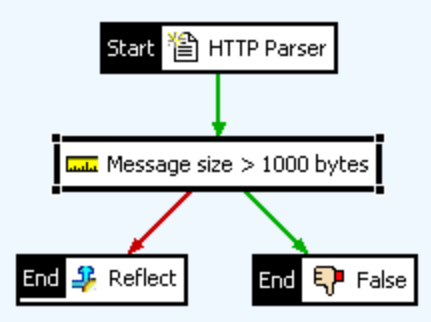
| | |
|---|---|
| **Environment variables to set** | Click **Add** to add environment variables. In the dialog, specify an **Environment variable name** (for example, `JAVA_HOME`) and a **Value** (for example, `c:\jdk1.6.0_18`), and click **OK**. Repeat to add multiple variables. |
| **Block till process finished** | Select whether to block until the process is finished in the check box. This is enabled by default. |

# 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.




See also *True filter* on page 456.

## Configuration

Enter a descriptive name for this filter to display in a policy.

# HTTP parser

## Overview

The **HTTP Parser** filter parses the HTTP message headers and body. As such, it acts as a barrier in the policy to guarantee that the entire content has been received before any other filters are invoked. It can be used, for example, to wait for an entire message from the back-end service before the gateway begins to reply to the caller. It requires the `content.body` attribute.

The **HTTP Parser** filter forces the server to do *store-and-forward* routing instead of the default *cut-through* routing, where the request is only parsed on-demand. For example, you can use this filter as a simple test to ensure that the message is XML.

See also *HTTP header authentication* on page 86.

## Configuration

Enter a descriptive name for this filter to display in a policy.

# Insert BST

## Overview

You can use the **Insert BST** filter to insert a Binary Security Token (BST) into a message. A BST is a security token that is in binary form, and therefore not necessarily human readable. For example, an X.509 certificate is a binary security token. Inserting a BST into a message is normally performed as a side effect of signing or encrypting a message. However, there are also some scenarios where you might insert a certificate into a message in a BST without signing or encrypting the message.

For example, you can use the **Insert BST** filter when the API Gateway is acting as a client to a Security Token Service (STS) that issues security tokens (for example, to create `OnBehalfOf` tokens). For more details, see the topic on *STS client authentication* on page 111. Finally, you can also use the **Insert BST** filter to generate XML nodes without inserting them into the message. In this case, the **WS-Security Actor** is set to blank.

## Configuration

You can configure the following settings on the filter dialog:

**Name**:
Enter an appropriate name for this filter to display in a policy.

**WS-Security Actor**:
Select or enter the WS-Security element in which to place the BST. Defaults to `Current actor / role only`. To use the **Insert BST** filter to generate XML nodes without inserting them into the message, you must ensure that this field is set to blank.

**Message Attribute**:
Select or enter the message attribute that contains the BST. The message attribute type can be `byte[]`, `String`, `X509Certificate`, or `X509Certificate[]`.

**Value Type**:
Select the BST value type, or enter a custom type. Example value types include the following:

- `http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509v3`
- `http://docs.oasis-open.org/wss/oasis-wss-kerberos-token-profile-1.1#GSS_Kerberosv5_AP_REQ`
- `http://xmlns.oracle.com/am/2010/11/token/session-propagation`

**Base64 Encode**:
Select this option to Base64-encode the data. This option applies only when the data in the message attribute is not already Base64 encoded. In some cases, the input might already be Base64 encoded, so you should deselect this setting in these cases.

# Invoke policy per message body

## Overview

In cases where API Gateway receives a multipart related MIME message, you can use the **Invoke Policy per Message Body** filter to pass each body part to a specified policy for processing.

For example, if other XML documents are attached to an XML message (using the SOAP with Attachments specification), you can pass each of these documents to an appropriate policy where they can be processed by the full complement of message filters.

See also *Convert multipart or compound body type message* on page 241.

## Configuration

Complete the following fields:

**Name**:
Enter a name for the filter to display in a policy.

**Policy Shortcut**:
Select the policy to invoke for each MIME body part in the message. Each body part is passed to the selected policy in turn. The filter fails if the selected policy fails for *any* of the passed body parts.

**Maximum level to unzip**:
In cases where a MIME body part is a MIME message itself (which might, in turn, contain more multipart messages), this setting determines how many levels of enveloped MIME messages to attempt to unzip. A default value of 2 levels ensures that the server does not attempt to unwrap unnecessarily *deep* MIME messages.

If one of the body parts is actually an archive file (for example, tar or zip), this setting determines the maximum depth of files to unzip in cases where the archive file contains other archive files, which might contain others, and so on.

# Locate XML nodes

## Overview

You can use the **Locate XML Nodes** filter to select a number of nodes from an XML message. The selected nodes are stored in a message attribute, which is typically used by a signature or XML encryption filter later in a policy.

The primary use of the **Locate XML Nodes** filter is when a series of policies is auto-generated by importing a Web Services Description Language (WSDL) file that contains WS-Policy assertions. For example, because there might be many different WS-Policy assertions that describe elements in the message that must be signed, you can use the **Locate XML Nodes** filter to build up the node list of elements. Eventually, this node list is passed into the **Sign Message** filter (using a message attribute) so that a single signature can be created that covers all the relevant parts.

However, you can also use this filter in similar cases where the message content that must be signed depends on content of the message. For example, a given policy runs a number of XPath expressions on a message where each XPath expression checks for a particular element. If that element is found, it can be marked as an element to be signed or encrypted by selecting that element in the **Locate XML Nodes** filter. This means that only a single signature or XML encryption filter must be configured, with each path feeding back into this filter and passing in the message attribute that contains the nodes set for each specific case.

## Configuration

As explained earlier, nodes can be selected using any combination of node locations, XPaths, or message attributes. The following sections explain how to use each different mechanism and how to store the selected nodes in a message attribute.

## Node locations

The simplest way to select nodes is using the preconfigured elements listed in the table on the **Node Locations** tab. The table is pre-populated with elements that are typically found in secured SOAP messages, including the SOAP Body, WSSE Security Header, WS-Addressing headers, SAML Assertions, WS UsernameToken, and so on.

The elements selected here are found by traversing the SOAP message as a DOM and finding the element name with the correct namespace and with the selected index position (for example, the first `Signature` element from the `http://www.w3.org/2000/09/xmldsig#` namespace).

You can select the check box in the **Name** column of the table to select the corresponding node. You can select any number of node locations in this manner.

To locate an element that is not already present in the table, click the **Add** button below the table to add a new node location. In the **Locate XML Nodes** dialog, enter the name of the element, its namespace, and its position in the message using the **Element Name**, **Namespace**, and **Index** fields.

To select this node for encryption purposes, select an appropriate **Encryption Type** . For example, WS-Security policy mandates that when encrypting the SOAP Body that only its contents are encrypted and not the SOAP Body element itself. This means that the`<xenc:EncryptedData>` is inserted as a direct child of the SOAP Body element. In this case, you should select the **Encrypt Node Content** option.

However, in most other cases, it is typically the entire node that gets encrypted. For example, when encrypting a `<wsse:UsernameToken>`, the entire node should be encrypted. In this case, the `<EncryptedData>` element replaces the `<UsernameToken>` element. To encrypt the entire node in this manner, select the **Encrypt Node** option.

## XPath expressions

To select nodes that exist under a more complicated element hierarchy, it might be necessary to use an XPath expression to locate the required nodes. The **XPaths** table is pre-populated with a number of XPath expressions to locate SOAP elements and common security elements, including SAML Assertions and SAMLP Responses.

To select an existing XPath expression, you can select the check box next to the **Name** of the appropriate XPath expression. You can select any number of XPath expressions in this manner.

To add a new XPath expression, click the **Add** button. You must enter a name for the XPath expression in the **Name** field and enter the XPath expression in the **XPath Expression** field. For more information on configuring this dialog, see "Configure XPath expressions" in the *API Gateway Policy Developer Guide*.

To select this node for encryption purposes, you must select an appropriate **Encryption Type**. For example, WS-Security policy mandates that when encrypting the SOAP Body that only its contents are encrypted and not the SOAP Body element itself. This means that the`<xenc:EncryptedData>` is inserted as a direct child of the SOAP Body element. In this case, you should select the **Encrypt Node Content** option.

However, in most other cases, it is typically the entire node that gets encrypted. For example, when encrypting a `<wsse:UsernameToken>`, the entire node should be encrypted. In this case, the `<EncryptedData>` element replaces the `<UsernameToken>` element. To encrypt the entire node in this manner, select the **Encrypt Node** option.

## *Message attribute*

Finally, you can also retrieve nodes that have been previously stored in a named message attribute. In such cases, another filter extracts nodes from the message and stores them in a named message attribute (for example, `node.list`). The **Locate XML Nodes** filter can then extract these nodes and store them in the message attribute configured in the **Message Attribute Name** field.

**Extract nodes from Selector Expression** enables you to specify whether to extract nodes from a specified selector expression (for example, `${node.list}`). This setting is not selected by default. 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" in the *API Gateway Policy Developer Guide*.

## *Message attribute in which to place list of nodes*

At runtime, the **Locate XML Nodes** filter locates and extracts the selected nodes from the message. It then stores them in the specified message attribute. For example, to sign the selected nodes, it would make sense to store the nodes in a message attribute called `sign.nodeList`, which would then be specified in the **Sign Message** filter. Alternatively, to encrypt the selected nodes, you could store the nodes in the `encrypt.nodeList` message attribute, which would then be specified in the **XML Encryption Properties** filter. The **Message Attribute Name** setting defaults to the `node.list` attribute.

Finally, you must specify whether the selected nodes should **Overwrite** any nodes that might already exist in the specified attribute, or if they should **Append** to any existing nodes. You can also decide to **Reset** the contents of the message attribute. Select the appropriate option depending on your requirements.

# Management services RBAC

## Overview

Role-Based Access Control (RBAC) is used to protect access to the API Gateway management services. For example, management services are invoked when a user accesses the server using Policy Studio or API Gateway Manager (`https://localhost:8090/`). For more information on RBAC, see the *API Gateway Administrator Guide*.

The **Management Services RBAC** filter can be used to perform the following tasks:

- Read the user roles from the configured message attribute (for example, `authentication.subject.role`).
- Determine which management service URI is currently being invoked.
- Return true if one of the roles has access to the management service currently being invoked, as defined in the `acl.json` file.
- Otherwise, return false.

**Caution** This filter is for management services use only. The **Management Services** HTTP services group should only be modified under strict supervision from Axway Support. For more details, see the "Management services" in the *API Gateway Policy Developer Guide*.

## Configuration

Configure the following settings:

**Name**:
Enter an appropriate name for this filter to display in a policy.

**Role Attribute**:
Select or enter the message attribute that contains the user roles.

# Pause processing

## Overview

The **Pause** filter is mainly used for testing purposes. This filter causes a policy to sleep for a specified amount of time.

See also *Reflect message* on page 448.

## Configuration

Configure the following settings:

**Name**:
Enter an appropriate name for the filter to display in a policy.

**Pause for**:
When the filter is executed in a policy, it sleeps for the time specified in this field. Defaults to `10000` milliseconds.

# Policy shortcut

## Overview

The **Policy Shortcut** filter enables you to reuse the functionality of one policy in another policy. For example, you could create a policy called **Security Tokens** that inserts various security tokens into the message. You can then create a policy that calls this policy using a **Policy Shortcut** filter.

In this way, you can adopt a design pattern of building up reusable pieces of functionality in separate policies, and then bringing them together when required using a **Policy Shortcut** filter. For example, you can create modular reusable policies to perform specific tasks, such as authentication, content-filtering, or logging, and call them as required using a **Policy Shortcut** filter.

For details on how to create a sequence of policy shortcuts in a single policy, see *Policy shortcut chain* on page 446.

## Configuration

Complete the following fields:

**Name**:
Enter an appropriate name for the filter to display in a policy.

**Policy Shortcut**:
Select the policy that to reuse from the tree. You can search for a specific policy by entering its name in the text box, and the policy tree is filtered automatically. The policy in which this **Policy Shortcut** filter is configured calls the selected policy when it is executed.

**Tip** Alternatively, to speed up policy shortcut configuration, you can drag a policy from the tree on the left of the Policy Studio and drop it on to the policy canvas on the right. This automatically configures a policy shortcut to the selected policy.

# Policy shortcut chain

## Overview

The **Policy Shortcut Chain** filter enables you to run a series of configured policies in sequence without needing to wire up a policy containing several **Policy Shortcut** filters. This enables you to adopt a design pattern of creating modular reusable policies to perform specific tasks, such as authentication, content-filtering, or logging. You can then link these policies together into a single, coherent sequence using this filter.

Each policy in the **Policy Shortcut Chain** is evaluated in succession. The evaluation proceeds as each policy in the chain passes, until finally the filter exits with a pass status. If a policy in the chain fails, the entire **Policy Shortcut Chain** filter also fails at that point.

See also *Policy shortcut* on page 445.

## General settings

Complete the following general setting:

**Name**:
Enter a meaningful name for the filter to display in a policy. For example, the name might reflect the business logic of the policies that are chained together in this filter.

## Add a policy shortcut

Click the **Add** button to display the **Policy Shortcut Editor** dialog, which enables you to add a policy shortcut to the chain. Complete the following settings in this dialog:

**Shortcut Label**:
Enter an appropriate name for this policy shortcut.

**Evaluate this shortcut when executing the chain**:
Select whether to evaluate this policy shortcut when executing a policy shortcut chain. When this option is selected, the policy shortcut has an **Active** status in the table view of the policy shortcut chain. This option is selected by default.

**Choose a specific policy to execute**:
Select this option to choose a specific policy to execute. This option is selected by default.

**Policy**:
Click the browse button next to the **Policy** field, and select a policy to reuse from the tree (for example, **Health Check**). You can search for a specific policy by entering its name in the text box, and the policy tree is filtered automatically. The policy in which this **Policy Shortcut Chain** filter is configured calls the selected policy when it is executed.

**Choose a policy to execute by label**:
Select this option to choose a policy to execute based on a specific policy label. For example, this enables you to use the same policy on all requests or responses, and also enables you to update the assigned policy without needing to rewire any existing policies. For more details on global policies, see the *API Gateway Policy Developer Guide*.

**Policy Label**:
Click the browse button next to the **Policy Label** field, and select a policy label to reuse from the tree (for example, **API Gateway request policy (Health Check)**). The policy in which this **Policy Shortcut Chain** filter is configured calls the selected policy label when it is executed.

Click **OK** when finished. You can click **Add** and repeat as necessary to add more policy shortcuts to the chain. You can alter the sequence in which the policies are executed by selecting a policy in the table and clicking the **Up** and **Down** buttons on the right. The policies are executed in the order in which they are listed in the table.

## Edit a policy shortcut

Select an existing policy shortcut, and click the **Edit** button to display the **Policy Shortcut Editor** dialog. Complete the following settings in this dialog:

**Shortcut Label**:
Enter an appropriate name for this policy shortcut.

**Evaluate this shortcut when executing the chain**:
Select whether to evaluate this policy shortcut when executing a policy shortcut chain. When this option is selected, the policy shortcut has an **Active** status in the table view of the policy shortcut chain.

**Policy** or **Policy Label**:
Click the browse button next to the **Policy** or **Policy Label** field (depending on whether you chose a specific policy or a policy label when creating the policy shortcut). Select a policy or policy label to reuse from the tree (for example, **Health Check** or **API Gateway request policy (Health Check)**). The policy in which this **Policy Shortcut Chain** filter is configured calls the selected policy or policy label when it is executed.

# Quote of the day

## Overview

The **Quote of the day** filter is a useful test utility for returning a simple SOAP response to a client. The API Gateway wraps the quote in a SOAP response, which can then be returned to the client.

See also *Reflect message* on page 448.

## Configuration

Simply enter the quote in the **Quotes** text area. This quote can be returned in a SOAP response to the client by setting the **Reflect** filter to be the successor of this filter in the policy.

The **Quote of the day** filter can also load a file containing a list of quotes at runtime. In this case, a random quote from the file is returned to the client in the SOAP response. Each quote should be delimited by a `%` character on a new line. This is analogous to the *BSD fortune format*. The format of this file is shown in the following example:

```
Most powerful is he who has himself in his own power.
%
All science is either physics or stamp collecting.
%
A cynic is a man who knows the price of everything and the value of nothing.
%
Intellectuals solve problems; geniuses prevent them.
%
If you can't explain it simply, you don't understand it well enough.
```

You can also enter the quotes in this format into the **Quotes** text area to achieve the same goal.

The following example shows a SOAP response returned by the API Gateway to a client who requested the **Quote of the day** service:

```
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
   <s:Header/>
   <s:Body xmlns:axway/>="axway.com">
      <axway:getQuoteResponse>
         Every cloud has a silver lining
      <axway:getQuoteResponse>
   </s:Body>
</s:Envelope>
```

# Reflect message

## Overview

The **Reflect Message** filter echoes the HTTP request headers, body, and attachments back to the client.

See also *Reflect message and attributes* on page 449.

## Configuration

Configure the following settings:

**Name**:
Enter a name for the filter to display in a policy.

**HTTP response code status**:
Specify an HTTP status response code to return to the client.

# Reflect message and attributes

## Overview

The **Reflect Message and Attributes** filter echoes the HTTP request headers, body, and attachments back to the client. It also echoes back the message attributes that were stored in the message at the time when the message completed the policy.

See also *Reflect message* on page 448.

## Configuration

Enter a meaningful name for the filter to display in a policy.

# Remove attribute

## Overview

You can use the **Remove Attribute** filter to remove a specified message attribute from a request message or a response message, depending on where the filter is placed in the policy.

See also *Copy or modify attributes* on page 434.

## Configuration

**Name**:
Enter a suitable name for this filter to display in a policy.

**Attribute Name**:
Select or enter the message attribute name to be removed from the message (for example, `authentication.subject.password`).

# Set attribute

## Overview

The simple **Set Attribute** filter enables you to set the value of a specified message attribute.

See also *Copy or modify attributes* on page 434.

## Configuration

Complete the following fields:

- **Name**:
  Enter a meaningful name for the filter to display in a policy.
- **Attribute Name**:
  Enter the name of the message attribute in which to store a value.
- **Attribute Value**:
  Enter the value of the message attribute specified above.

# Set response status

## Overview

The **Set Response Status** filter is used to explicitly set the response status of a call. This status is then recorded as a message metric for use in reporting.

This filter is primarily used in cases where the fault handler for a policy is a **Policy Shortcut** filter. If the **Policy Shortcut** passes, the overall `fail` status still exists. You can use the **Set Response Status** filter to explicitly set the response status back to `pass`, if necessary.

See also *Policy shortcut* on page 445.

## Configuration

**Name**:
Enter a meaningful name for the filter to display in a policy.

**Response Status**:
Select **Pass** or **Fail** to set the response status.

# String replace

## Overview

The **String Replace** filter enables you to replace all or part of the value of a specified message attribute. You can use this filter to replace any specified string or substring in a message attribute. For example, changing the `from` attribute in an email, or changing all or part of a URL.

See also *Copy or modify attributes* on page 434.

## Configuration

To configure the **String Replace** filter, specify the following fields:

| | |
|---|---|
| **Name** | Enter the name of the filter to be displayed in a policy. |
| **Message Attribute** | Select the name of the message attribute to be replaced from the list. This is required. If this is not specified, a `MissingPropertyException` is thrown, which results in a `CircuitAbortException`. |
| **Specify Destination Attribute** | By default, the value of the specified **Message Attribute** is both the source and destination, and is therefore overwritten. To specify a different destination attribute, select this check box to enable the **Destination Attribute** field, and select a value from the list. |
| **Replacement String** | The string used to replace the value of the specified source attribute. You can specify this as a selector, which is expanded to the specified value at runtime (for example, `${http.request.uri}`). This is a required field if you specify the **Specify Destination Attribute**. |
| **Straight** | A match string used to search the value of the specified source attribute. You can specify this as a selector, which is expanded to the specified value at runtime. If a straight (exact) match is found, it is replaced with the specified **Replacement String** . |

| | |
|---|---|
| **Regexp** | A match string, specified as a regular expression, used to search the value of the specified source attribute. You can specify this as a selector, which is expanded to the specified attribute value at runtime. If a match is found, it is replaced with the specified **Replacement String** . For more details on selectors, see "Select configuration values at runtime" in the *API Gateway Policy Developer Guide*. |
| **First Match** | If a match is found, only replace the first occurrence. |
| **All Matches** | If a match is found, replace all occurrences. |

**Note** The possible paths available through this filter are `True` (even if no replacement takes place), and `CircuitAbort`. Under certain circumstances, if the **Replacement String** contains a selector, a `MissingPropertyException` can occur, which results in a `CircuitAbortException`.

# Switch on attribute value

## Overview

The **Switch on Attribute Value** filter enables you to switch to a specific policy based on the value of a configured message attribute. You can specify various switch cases (for example, contains, is, ends with, matches regular expression, and so on). Specified switch cases are evaluated in succession until a switch case is found, and the policy specified for that case is executed. You can also specify a default policy, which is executed when none of the switch cases specified in the filter is found.

See also *Policy shortcut* on page 445.

## Configuration

Complete the following configuration settings:

**Name**:
Enter a meaningful name for the filter to display in a policy. For example, the name might reflect the business logic of a specified switch case.

**Switch on selector expression**:
Enter or select the name of the message attribute selector to switch on (for example, `${http.request.path}`). This filter examines the specified message attribute value, and switches to the specified policy if this value meets a configured switch case.

**Case**:
You can add, edit, and delete switch cases by clicking the appropriate button on the right. All configured switch cases are displayed in the table. For more details, see *Add a switch case* on page 453.

**Default**:
This field specifies the default behavior of the filter when none of the specified switch cases are found in the configured message attribute value. Select one of the following options:

| | |
|---|---|
| **Return result of calling the following policy** | Click the browse button, and select a default policy to execute from the dialog (for example, **XML Threat Policy**). The filter returns the result of the specified policy. This option is selected by default. |
| **Return true** | The filter returns true. |
| **Return false** | The filter returns false. |

# Add a switch case

To add a switch case, click **Add**, and configure the following fields in the dialog:

**Comparison Type**:
Select the comparison type to perform with the configured message attribute. The available options include the following:

- `Contains`
- `Doesn't Contain`
- `Ends With`
- `Equals`
- `Does not Equal`
- `Matches Regular Expression`
- `Starts With`

All of these options are case insensitive, except for `Matches Regular Expression`.

**Compare with**:
Enter the value to compare the configured message attribute value with. For example, if you select a **Comparison Type** of `Matches Regular Expression`, enter the regular expression in this field.

**Policy**:
Click the browse button next to the **Policy** field, and select the policy to execute from the dialog (for example, **Remove All Security Tokens**). You can search for a specific policy by entering its name in the text box, and the policy tree is filtered automatically. The selected policy is executed when this switch case is found.

Click **OK** when finished. You can click **Add**, and repeat as necessary to add more switch cases to this filter. The switch cases are examined in the order in which they are listed in the table. You can alter the sequence in which the switch cases are evaluated by selecting a policy in the table and clicking the **Up** and **Down** buttons on the right.

# Time filter

## Overview

The **Time** filter enables you to block or allow messages on a specified time of day, or day of week, or both. You can input the time of day directly in the **Time** filter window, or configure message attributes to supply this information using the Java `SimpleDateFormat`, or specify a cron expression.

You can use the **Time** filter in any policy (for example, to block messages at specified times when a web service is not available, or has not been subscribed for by a consumer). In this way, this filter enables you to meter the availability of a web service and to enforce Service Level Agreements (SLAs).

## General settings

Configure the following general options:

**Name**:
Enter an appropriate name for this filter.

**Block Messages**:
Select this option to use this filter to block messages. This is the default option.

**Allow Messages**:
Select this option to use this filter to allow messages.

## Basic time settings

Select **Basic** to block or allow messages at specified times of the day. This is the default option. You can configure following settings:

**User defined time**:
Select this option to input the times to block or allow messages directly in this screen. This is the default option. Configure the following settings:

| | |
|---|---|
| **From** | The time to start blocking or allowing messages from in hours, minutes, and seconds. Defaults to `9:00:00`. |

| | |
|---|---|
| **To** | The time to end blocking or allowing messages in hours, minutes, and seconds. Defaults to `17:00:00`. |

**Time from attribute**:
Select this option to specify times to block or allow messages using configured message attributes. You can specify these attributes using selectors, which are replaced at runtime with the values of the specified message attributes set in previous filters or messages. For more details, see "Select configuration values at runtime" in the *API Gateway Policy Developer Guide*. You must configure the following settings:

| | |
|---|---|
| **From** | Message attribute that contains the time to start blocking or allowing messages from (for example, `$(message.starttime)`). Defaults to a time of `9:00:00`. |
| **To** | Message attribute that contains the time to end blocking or allowing messages (for example, `$(message.endtime)`). Defaults to a time of `17:00:00`. |
| **Pattern** | Message attribute that contains the time format based on the Java `SimpleDateFormat` class (for example, `$(message.pattern)`). This enables you to format and parse dates in a locale-sensitive manner. Day, month, years, and milliseconds are ignored. Defaults to a format of `HH:mm:ss`. |

**Days**:
To block or allow messages on specific days of the week, select the check boxes for those days. For example, to block messages on Saturday and Sunday only.

# Advanced time settings

Select **Advanced** to block or allow messages at specified times based on a cron expression. Configure the following setting:

**Cron Expression**:
Enter a cron expression or a message attribute that contains a cron expression in this field. Alternatively, click the browse button next to this field to select a preconfigured cron expression or to create and test a new cron expression. For more details, see "Configure cron expressions" in the *API Gateway Policy Developer Guide*.

For example, the following cron expression blocks all messages received on April 27 and 28 2012, at any time except those received between 10:00:01 and 10:59:59.

```
* * 0-9,11-23 27-28 APR ? 2012
```

The default value is `* * 9-17 * * ? *`, which specifies a time of 9:00:00 to 17:00:00 every day.

# Trace filter

The **Trace** filter outputs the current message attributes to the configured trace destinations. By default, output is traced to the system console.

For more details on API Gateway tracing, see the *API Gateway Administrator Guide*.

## Configuration

Configure the following settings:

**Name**:
Enter an appropriate name for the filter to display in a policy.

**Include the following text in trace**:
Enter an optional custom text message to include in the trace output.

**Trace Level**:
Select the trace level. `DATA` is the most verbose level, while `FATAL` is the least verbose.

**Include Attributes**:
Select this option to trace all current message attributes to the configured trace destination.

**Include Body**:
Select this option to trace the entire message body.

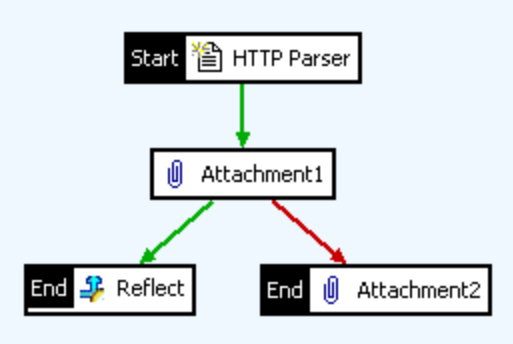
**Indent XML**:
If this option is selected, the XML message is pretty-printed (indented) before it is output to the trace destination.

# True filter

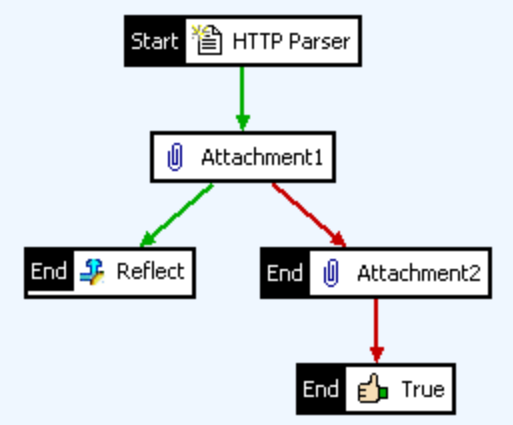
## Overview

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.




See also *False filter* on page 438.

# Configuration

Enter an appropriate name for this filter to display in a policy.

# Web service filters 22

The following filters are available in the Web Service category in Policy Studio:



# Return WSDL

## Overview

The **Return WSDL** filter returns a WSDL file from the web service repository.

**Note** This filter is configured automatically when auto-generating a policy from a WSDL file and is not normally manually configured. For a detailed example, see "Manage web services" in the *API Gateway Policy Developer Guide*.

This filter is used with the **Set Web Service Context** filter, which is used to identify web services. For more details, see *Set web service context* on page 458.

## Configuration

Enter a meaningful name for the filter.

# Set web service context

## Overview

The **Set Web Service Context** filter is used in a policy to determine the service to obtain resources from in the web service repository.

For example, by pointing this filter at a preconfigured `getQuote` service in the web service repository, the policy knows to return the WSDL for this particular service when a WSDL request is received. The **Return WSDL** filter is used in conjunction with this filter to achieve this.

**Note** The **Set Web Service Context** filter is configured automatically when auto-generating a policy from a WSDL file and is not normally manually configured. For a detailed example, see "Manage web services" in the *API Gateway Policy Developer Guide*.

## General settings

**Name**:
Enter a meaningful name for the filter to display in a policy.

## Service WSDL settings

The **Service WSDL** tab enables you to select the web service to obtain resources from in the web service repository.

Click the browse button to select a service definition (WSDL file) currently registered in the web service repository from the tree. To register a web service, right-click the default **APIs** > **Web Services** node, and select **Register Web Service**.

For more details on adding services to the web service repository, see "Manage web services" in the *API Gateway Policy Developer Guide*.

## Monitoring settings

The fields on this tab enable you to configure to configure whether API Gateway displays usage metrics data for this web service. For example, this information can be used by API Gateway Analytics to produce reports showing how and who is calling this web service. For details on the fields on this tab, see *Monitoring options* on page 466.

# Web service filter

## Overview

The **Web Service Filter** is used to control and validate requests to the web service and responses from the web service. Typically, this is automatically generated and populated as a **Service Handler** when a WSDL file is imported into the web service repository. For example, if you import the WSDL file for a web service named `ExampleService`, a **Service Handler for ExampleService** filter is automatically generated. However, you can also configure a **Web Service Filter** manually.

In cases where the imported WSDL file contains WS-Policy assertions, a number of policies are automatically created containing the filters required to generate and validate the relevant security

tokens (for example, SAML, WS-Security UsernameToken, and WS-Addressing headers). These policies perform the necessary cryptographic operations (for example, signing and encrypting) to meet the security constraints stipulated by the WS-Policy assertions.

# General settings

Configure the following general settings:

**Name**:
Enter an intuitive name for the filter to display in a policy.

**Web Service Context**:
Click the button on the right, and select a WSDL file currently registered in the web service repository from the tree to set the web service context. To register a web service, right-click the default **Web Services** node, and select **Register Web Service**. For more details, see "Manage web services" in the *API Gateway Policy Developer Guide*.

When you select a web service from the **Web Service Context** field, the **Message Interception Points** tab is automatically populated with resolvers for the operations exposed by that web service. Similarly, the **Routing** tab is automatically populated with the routing information for the selected web service.

# Routing settings

When routing to a service, you can specify a direct connection to the web service endpoint by using the URL in the WSDL or you can override this URL by entering a URL in the field provided.

Alternatively, in cases where the routing behavior is more complex, you can delegate to a custom routing policy, which takes care of the added complexity. The top-level radio buttons on the **Routing** tab allow for these alternative routing configurations.

**Direct Connection to Service Endpoint**:
Select this option to route to either the URL specified in the WSDL or a URL. The radio buttons in the **Routing Details** group enable you to choose between using the URL in the WSDL and providing an override. When providing an override, you can enter the new URL in the **URL** field. Alternatively, you can specify the URL as a selector so that the URL is built dynamically at runtime from the specified message attributes. For example:

```
${host}:${port}
${http.destination.protocol}://${http.destination.host}:${http.destination.port}
```

For more details on selectors, see "Select configuration values at runtime" in the *API Gateway Policy Developer Guide*.

In both cases, you can configure SSL settings, credential profiles for authentication, and other settings for the direct connection using the tabs in the **Connection Details** group. For more details, see *Connect to URL* on page 380.

**Delegate to Routing Policy**:
Select this setting to use a dedicated routing policy to send messages on to the web service. For example, you might have configured a dedicated routing policy that uses the JMS-based **Send to JMS** filter to route over JMS.

Click the browse button next to the **Routing Policy** field. Select the policy to use to route messages, and click **OK**. You can search for a specific policy by entering its name in the text box, and the policy tree is filtered automatically.

# Validation settings

The WSDL for a web service contains information about the SOAP Action, SOAP Operation, and the data types of the message parts used in a particular SOAP operation. API Gateway creates the following implicit validation incoming requests for the web service:

- **SOAPAction HTTP Header**:
  If a web service requires clients to send a certain SOAPAction HTTP header in all requests, API Gateway can check the value of this header in the incoming request against the value specified in the WSDL.
- **SOAP Operation and Namespace**:
  The WSDL defines the SOAP Operation and namespace to be used in the SOAP request. The SOAP Operation is defined as the first child element of the SOAP Body element. API Gateway can check the value of this element in an incoming SOAP request and its namespace against the values specified in the WSDL.
- **Relative Path**:
  The filter ensures that requests for this web service are received on the same URL as that specified in the `<service>` block of the WSDL.
- **SOAP Version**:
  API Gateway also validates requests by matching the SOAP protocol version in the message against the SOAP binding version (1.1 or 1.2) of the corresponding operation definition in the WSDL.

It is also common for a WSDL document to contain an XML schema that defines the format and types of the message parts in the request. This is usually the case for document/literal style SOAP requests, where a complete XML schema is embedded or imported into the `<wsdl:types>` block of the WSDL.

When using a WSDL to import a service into the web service repository, Policy Studio can extract the XML schema from the WSDL and configure API Gateway to validate incoming requests against it. Select **Use WSDL Schema** to validate incoming requests against the schema in the WSDL.

Alternatively, you can create a custom-built policy to validate the contents of incoming requests. To do this, select the **Delegate to Validation Policy** radio button, and the click the browse button next to the **Message Validation Policy** field. Select the policy to use to validate requests, and click **OK**.

# Configure message interception points

The configuration settings on the **Message Interception Points** tab determine how the request and response messages for the service are processed as they pass through API Gateway. Several message interception points are exposed to enable you to hook into different stages of the API Gateway's request processing cycle.

At each of these interception points, it is possible to run policies that are specific to that stage of the request processing cycle. For example, you can configure a logging policy to run just before the request has been sent to the web service and then again just after the response has been received.

Typically, the configuration settings on this window are automatically configured when importing a service into the web service repository based on information contained in the WSDL. In cases where the WSDL contains WS-Policy assertions, a number of policies are automatically generated and hooked up to perform the relevant security operations on the message.

For example, policies are created to insert SAML assertions, WS-Security `UsernameToken` elements, WS-Addressing headers, and WS-Security timestamps into the message. Similarly, filters are created to sign and encrypt the outbound message, if necessary, and to decrypt and validate the signature on the response from the web service.

## *Order of execution*

The order of execution of message interception points is as follows:

1. Request from client
2. User-defined request hooks
3. Request to service
4. Response from service
5. User-defined response hooks
6. Response to client

In steps 1, 3, 4, and 6, the execution order is as follows:

A. Before operation-specific policy

B. Operation-specific policy Shortcuts

C. After operation-specific policy

The overall order of all the message interception points is described in the sequence that follows.

## *1. Request from client*

This is the first message interception point, which enables you to run a policy against the request as it is received by API Gateway. Typically, this is where authentication and authorization events should occur.

- **1A) Before operation-specific policy**:
  This is usually where authentication policies should be configured because it is the earliest point in the request cycle that you can hook into. To select a policy to run at this point, click the browse button, and select the check box next to a previously configured policy.
- **1B) Operation-specific policy shortcuts**:
  To run policies that are specific to the different operations exposed by the web service, click **Edit** at the bottom of the table to set this up. For example, you can perform different validation on requests for the different operations.

  On the **Policy Shortcut Editor** dialog, enter the **Operation Namespace** and **Operation Name** in the fields provided. Enter a regular expression used to match the value of the SOAPAction HTTP header in the **SOAPAction Regular Expression** field. Finally, select the policy to run requests for this operation by clicking the browse button next to the **Policy Shortcut** field. Select the policy to run.
- **1C) After operation-specific policy**:
  This enables you to run a policy on the request *after* all the operation-level policies have been executed on the request. Select the appropriate policy as described earlier by clicking the browse button.

## 2. User-defined request hooks

You should primarily use this interception point to hook in your own custom-built *request* processing policies.

- **User-defined request policy**:
  Click to browse to your custom-built request processing policy.

## 3. Request to service

This enables you to alter the message before it is routed to the web service. For example, if the service requires the message to be signed and encrypted, you can configure the necessary policies here.

- **3A) Before operation-specific policy**:
  This enables you to run policies on the message *before* the operation-level policies are run. Select the policy to run as outlined in the previous sections.
- **3B) Operation-specific policy shortcuts**:
  Operation-level policies on the request to the web service can be run here. For example, if the input policy for a particular operation requires the body to be signed and encrypted, a **Locate XML Nodes** filter can be run here to mark the required nodes.
- **3C) After operation-specific policy**:
  This is the last interception point available before the message is routed on to the web service. For example, if certain operation-level policies have been run to mark parts of the message to be signed and encrypted, the signing and encrypting filters should be run here.

## 4. Response from service

This is executed on the response returned from the web service.

- **4A) Before operation-specific policy**:
  If the response from the web service is encrypted, this interception point enables you to decrypt the message *before* any of the operation-level policies are run on the decrypted message.
- **4B) Operation-specific policy shortcuts**:
  The policies configured at this point run on specific operation-level responses (for example, `getHelloResponse`) from the web service.
- **4C) After Operation-specific policy**:
  This should be used to run policies *after* the operation-level policies have been run. For example, this is the appropriate point to place an **XML Signature Verification** filter.

## 5. User-defined response hooks

You should primarily use this interception point to hook in custom-built *response* processing policies.

**User-defined response policy**:
Click to browse to your custom-built response processing policy.

## 6. Response to client

This enables you to process the response before it is returned to the client.

- **6A) Before operation-specific policy:**
  This enables you to process the message with a policy before the operation-level policies are run on the response.
- **6B) Operation-specific policy shortcuts:**
  The policies listed here are run on each operation response.
- **6C) After operation-specific policy:**
  This is the very last point at which you can run policies to process the response message before it is returned to the client. For example, if you are required to return a signed and encrypted response message to the client, the signing and encrypting should be done at this point.

# WSDL settings

You can expose an imported WSDL file to clients of API Gateway. A client can retrieve a WSDL for a service by appending `WSDL` to the query string of the relative path on which the service is accepting requests.

For example, if the service is accepting requests at the URL `http://server:8080/services/getHello`, the client can retrieve the WSDL on the following URL:

```
http://server:8080/services/getHello?WSDL
```

**Note** When API Gateway returns the WSDL to the client, it dynamically modifies the service URL of the original WSDL to point to the machine on which API Gateway is running. For example, the original WSDL contains the following service element, where `www.service.com` resolves to an internal IP address that is not accessible to the public Internet:

```
<wsdl:service name="GetHelloService">
   <wsdl:port name="GetHelloServiceSoap" binding="tns:ServiceSoap">
      <soap:address location="http://www.service.com/getHello"/>
   </wsdl:port>
</wsdl:service>
```

When API Gateway returns this WSDL to the client, it dynamically modifies the value of the `location` attribute to point to the name of the machine hosting API Gateway. In the following example, the `location` attribute has been modified to point to the API Gateway instance running on port 8080 on the `Axway_SERVER` host:

```
<wsdl:service name="GetHelloService">
   <wsdl:port name="GetHelloServiceSoap" binding="tns:ServiceSoap">
      <soap:address location="http://Axway_SERVER:8080/getHello"/>
   </wsdl:port>
</wsdl:service>
```

When the client receives the WSDL, it can automatically generate the SOAP request for the `getHello` service, which it then sends to the`Axway_SERVER` machine on port 8080.

Complete the following fields if you wish to expose the WSDL for this service to clients.

**Advertise WSDL to the Client**:
Select this option to publish the WSDL for the selected web service.

**Note** The exposed WSDL represents a *virtualized* view of the back-end web service. In this way, clients can retrieve the WSDL from API Gateway, generate SOAP requests, and send these requests to API Gateway. API Gateway then routes the requests on to the web service.

**WSDL Access Policy**:
To configure a policy to control or monitor access to the WSDL for this service, you can select the policy by clicking the browse button to the right of this field. Select the policy to run on requests to retrieve the WSDL.

# Monitoring options

The fields on this tab enable you to configure whether API Gateway displays usage metrics data for this web service. For example, this information can be used by API Gateway Analytics to produce reports showing who is calling this web service. You can configure the following fields:

- **Enable monitoring**:
  Select this option to enable monitoring for this web service in the **Monitoring** view in API Gateway Manager, and in API Gateway Analytics.
- **Which attribute is used to identify the client**:
  Enter the message attribute to use to identify authenticated clients. The default is `authentication.subject.id`, which stores the identifier of the authenticated user (for example, the user name or user's X.509 Distinguished Name).
- **Composite context**:
  This setting enables you to select a service context as a composite context in which multiple service contexts are monitored during the processing of a message. This setting is not selected by default.

  For example, API Gateway receives a message, and sends it to `serviceA` first, and then to `serviceB` . Monitoring is performed separately for each service by default. However, you can set a composite service context before `serviceA` and `serviceB` that includes both services. This composite service passes if both services complete successfully, and monitoring is also performed on the composite service context.

For more details on monitoring and reporting, see:

- *API Gateway Administrator Guide*
- *API Gateway Analytics User Guide*