API Gateway

Version 7.6.2

14 July 2020

Developer Guide

This documentation describes the following Axway software:

Axway API Gateway 7.6.2

Contents

Preface

This guide describes how to extend, leverage, and customize API Gateway to suit the needs of your environment. For example, this includes topics such as adding a custom filter to API Gateway, accessing configuration values dynamically at runtime, and creating custom scripts to run against API Gateway.

Who should read this guide

The intended audience for this guide is policy developers and system integrators.

Before creating your own custom filter you should understand exactly what message filters are, and how they are chained together to create a message policy. These concepts are documented in detail in the *API Gateway Policy Developer Guide*.

How to use this guide

This guide should be used in conjunction with the other guides in the API Gateway documentation set.

Before you begin customizing or extending API Gateway, review this guide thoroughly. The following is a brief description of the contents of each section:

Install the code samples on page 13 – Describes how to install the code samples used in the *API Gateway Developer Guide*.

Build the code samples on page 15 – Describes how to build the code samples.

Add a custom filter to API Gateway on page 17 – Describes several methods for adding custom filters to API Gateway.

Define user interfaces using declarative XML on page 37 – Describes how to use declarative XML to define Policy Studio user interface dialogs.

Unit test a filter using the Traffic Monitor API on page 42 – Describes how to unit test a custom filter using the Traffic Monitor API.

Java interfaces for extending API Gateway on page 44 – Describes several Java interfaces that you can use to extend API Gateway.

Access configuration values dynamically at runtime on page 55 – Describes how you can use selectors to access configuration values at runtime.

Key Property Store on page 59 – Introduces the API Gateway Key Property Store.

Entity Store on page 60 – Introduces the Entity Store and describes how to use the ES Explorer tool.

Debug custom Java code with a Java debugger on page 65 – Describes how to connect to API Gateway with a Java debugger.

Get diagnostics output from a custom filter on page 66 – Describes how to add diagnostics output from a custom filter.

Enable API Gateway with JMX on page 69 – Describes how to manage API Gateway using Java Management Extensions (JMX).

Automate tasks with Jython scripts on page 71 – Describes the Jython scripts that are provided with API Gateway.

API Gateway REST APIs on page 76 – Describes the REST APIs exposed by API Gateway.

Declarative UI reference on page 85 – Details the declarative XML UI elements that can be used to define the user interface of filters and dialogs in Policy Studio.

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 and code sample changes.

Note If you are upgrading from version 7.3.1 or earlier, and have developed custom filters, you must update your custom filter classes and recompile before upgrading. For more details, see the changes to classes described in *Changes in version 7.4.0* on page 11.

Changes in version 7.6.2

- Restructured the API Gateway Analytics information and added links to the new *API Gateway Analytics User Guide*.

Changes in version 7.6.1

No changes.

Changes in version 7.6.0

- Removed references to API Gateway Analytics.

Changes in version 7.5.3

- Added information on the CheckboxGroupAttribute element to the Declarative UI reference. For details, see *CheckboxGroupAttribute* on page 103.
- Updated or removed information on the `jabber` and `restJabber` code samples as these samples are no longer included in the code samples supplied with API Gateway.

Changes in version 7.5.2

- Replaced links to the API Gateway REST API documentation on Axway Support at https://support.axway.com with links to the the Axway Documentation portal at https://docs.axway.com where the API Gateway REST API documentation is now available in Swagger format.

Changes in version 7.5.1

- The following API documentation has been removed from the API Gateway installation.
  - `apidocs` – Documentation for the REST APIs provided by API Gateway.
  - `javadoc` – Javadoc for the API Gateway classes.

  This documentation is now available online:
  - API Gateway REST API available from Axway Support at https://support.axway.com
  - API Gateway Javadoc available from Axway Support at https://support.axway.com

Changes in version 7.4.2

- All sections in this guide were updated to reflect changes to the tree structure in Policy Studio.

Changes in version 7.4.1

- API Gateway is now built on JDK 1.8. As a result, the samples should also be built with JDK 1.8. For more details, see *Build the code samples* on page 15.
- Code samples in the guide were updated with the class changes that occurred in 7.4.0 (see *Changes in version 7.4.0* on page 11).
- Many classes changed to new packages and were moved to new JAR files and directories. Several of the JAR files were relocated to new directories in API Gateway and Policy Studio. To pick up the correct JARs and classes, ensure that you build against the latest CLASSPATH paths as defined in the `build.xml` files in the SDK samples. The JAR file naming convention is: `informative-name.version.(in some cases)version-qualifier`.
- The location of the Policy Studio help files for custom filters has changed. For more details, see *Create the Policy Studio classes* on page 28. Ensure that your local `build.xml` file for the code samples is updated with the latest SDK changes and that the CLASSPATH is correct.

Changes in version 7.4.0

- The developer samples in the `INSTALL_DIR/apigateway/samples/developer_guide` directory were updated with the following changes:
  - The `PropDef` class changed location from `com.vordel.circuit.PropDef` to `com.vordel.common.util.PropDef`.
  - The `Circuit` class changed location from `com.vordel.circuit.Circuit` to `com.vordel.config.Circuit`.

- The `SolutionPack` class is replaced with `ConfigContext` in method signatures. For example, any uses of `com.vordel.precipitate.SolutionPack` are changed to `com.vordel.config.ConfigContext`.
- The `LoadableModule` class changed location from `com.vordel.dwe.LoadableModule` to `com.vordel.config.LoadableModule`.
- The `test ClientResponse` class changed from `com.sun.jersey.api.client.ClientResponse` to `javax.ws.rs.core.Response`.
- The packages imported and used in the `RestJabberRequestClient Test` class have changed.

You must recompile the classes.

Install the code samples 1

Code samples demonstrating some of the tasks discussed in this guide are included in your installation of API Gateway in the `INSTALL_DIR/apigateway/samples/developer_guide` directory (for example, `C:\Axway-7.6.2\apigateway\samples\developer_guide`).

Alternatively, the associated code samples are available from Axway Support at https://support.axway.com as a zip file. This section describes how to install the samples.

Installation prerequisites

Before you install the code samples:

- Install API Gateway. You must install the API Gateway core server and Policy Studio, as the samples require certain classes that ship with these components to be on the CLASSPATH.
- To write custom message filters for API Gateway, you must install the samples on the same machine as API Gateway.

For more information on installing API Gateway, see the *API Gateway Installation Guide*.

Unzip the downloaded zip file

If you downloaded the samples from Axway Support at https://support.axway.com as a zip file, the zip file contains the following directory structure:

```
developer-guide-7.6.2\samples\developer_guide
```

Use your preferred zip utility to unzip the file to a suitable location (for example, `C:\samples`).

Location of code samples

The location `DEVELOPER_SAMPLES` is used throughout this guide to refer to the location of the code samples:

- If you have installed API Gateway, `DEVELOPER_SAMPLES` refers to the `INSTALL_DIR/apigateway/samples/developer_guide` directory.

- If you have installed the code samples from a zip file, as described in the preceding sections, `DEVELOPER_SAMPLES` refers to the location where you installed the samples (for example, the `C:\samples\developer-guide-7.6.2\samples\developer_guide` directory).

Build the code samples 2

API Gateway provides several code samples that demonstrate the tasks described in this guide, such as adding a custom filter or adding a message listener to API Gateway. This topic describes how to build the code samples.

Build prerequisites

API Gateway is built with JDK 1.8. To avoid `BadClassVersion` errors that might arise when deploying your sample classes with the API Gateway, you must also build the code samples with JDK 1.8.

Build the samples

Complete the following steps to build the samples:

1. Set the `VORDEL_HOME` and `POLICYSTUDIO_HOME` environment variables:
    - Set the `VORDEL_HOME` environment variable to point to the root of your Axway API Gateway installation. For example, if you installed API Gateway in `C:\Axway7.6.2\apigateway`, set `VORDEL_HOME` to this directory.
    - Set the `POLICYSTUDIO_HOME` environment variable to point to the root of your Policy Studio installation. For example, if you installed Policy Studio in `C:\Axway7.6.2\policystudio`, set `POLICYSTUDIO_HOME` to this directory.
2. Set the `JAVA_HOME` and `JUNIT_HOME` environment variables:
    - Set the `JAVA_HOME` environment variable to point to the root of a JDK 1.8 installation (for example, `C:\jdk1.8.0_07`).
    - Set the `JUNIT_HOME` environment variable to point to the directory containing your JUnit JAR file. The required version is 4.8.2 (for example, `junit_4.8.2.jar`).
3. Add Apache Ant to your `PATH` environment variable. For example, if Apache Ant is installed in `C:\ant`, add `C:\ant\bin` to your `PATH`. See the Apache Ant website for more information on Apache Ant.
4. To build and run each sample, follow these steps:
    a. Change to the directory where the sample is installed. Each sample is installed under `DEVELOPER_SAMPLES/SAMPLE_NAME` (for example, `DEVELOPER_SAMPLES/FilterInterceptorLoadableModule`).
    b. Open the `README.TXT` file and follow the instructions to build and run the sample.

Description of samples

The following code samples are included:

- `DEVELOPER_SAMPLES/FilterInterceptorLoadableModule` – Sample classes that implement Java interfaces. For more information, see *Java interfaces for extending API Gateway* on page 44.

Add a custom filter to API Gateway

3

You can extend the capability of API Gateway by adding a custom filter. There are several options for adding a custom filter:

- Write your custom requirement in Java and invoke it using the **Scripting Language** filter. You can use this approach to develop your business logic in a standard IDE and debug and test it in standalone mode before integrating with API Gateway. See *Use JavaScript to call existing Java code* on page 17.
- Write your custom requirement using the **Scripting Language** filter alone. See *Use JavaScript for custom requirements* on page 19.
- Write your custom filter using the API Gateway developer extension kit. Using this approach, a fully integrated filter is created that has the API Gateway runtime capability and appears in the filter palette in Policy Studio. See *Write a custom filter using the extension kit* on page 20.

The following examples all use different approaches to extend API Gateway by adding a custom filter.

The following table summarizes the different approaches for adding a custom filter:

Scripting	Writing a Java Filter
Quick way to reuse some functionality exposed in Java	Enterprise integration
No major development skills required	Development skills required
Does not appear in filter palette in Policy Studio	Filter appears in filter palette in Policy Studio
Possible approaches: • *Use JavaScript to call existing Java code* on page 17 • *Use JavaScript for custom requirements* on page 19	Possible approaches: • *Write a custom filter using the extension kit* on page 20

Use JavaScript to call existing Java code

In this approach, you write your custom requirement in Java and invoke it using JavaScript in a **Scripting Language** filter.

Follow these guidelines:

1. Create a Java class that meets your custom requirement.
2. Build a JAR file from the Java class and add it, and any third-party dependencies, to the API Gateway CLASSPATH and to the runtime dependencies in Policy Studio.
3. Create a policy (for example, called **InvokeJava**) in Policy Studio that contains only a **Scripting Language** filter. Configure the filter to invoke the Java code using JavaScript.

   Note We recommend that you select `JavaScript` in the **Language** field of the **Scripting Language** filter, and ensure that the JavaScript syntax in the script conforms with Nashorn engine syntax. For more information about migrating from Rhino to Nashorn, see the Rhino Migration Guide.

1. Configure API Gateway to invoke the policy. For more information, see *Invoke the policy* on page 18.
2. Test the policy using API Tester. For more information, see *Test the policy* on page 18.

Invoke the policy

To configure the API Gateway to invoke the new policy, follow these steps:

1. Under the **Environment Configuration > Listeners** node in Policy Studio, select the path (for example, **API Gateway > Default Services > Paths**).
2. On the resolvers window on the right, click **Add > Relative Path**.
3. Enter the following values on the dialog and click **OK**:
   - **When a request arrives that matches the path:** `/invokejava`
   - **Path Specific Policy:** Click the browse button and select the **InvokeJava** policy. This sends all requests received on the path configured above to your newly configured policy.
4. To deploy the new configuration to API Gateway, click the **Deploy** button on the toolbar or press **F6** and follow the instructions.

Test the policy

To test the configuration, follow these steps:

1. Start API Tester.
2. Click the arrow next to the Play icon and select **Request Settings**.
3. In the **Url** field, enter `http://localhost:8080/invokejava` to send the message to the relative path you configured above.
4. Click **Run** to send the message to API Gateway.

Tip Alternatively, you can test the policy by entering the URL `http://localhost:8080/invokejava` into any web browser.

Use JavaScript for custom requirements

In this approach, you write your custom requirement using the **Scripting Language** filter alone.

Follow these guidelines:

1. Create a policy (for example, called **InvokeScript**) in Policy Studio that contains only a **Scripting Language** filter. Configure the filter to invoke JavaScript that meets your custom requirement.

   Note We recommend that you select `JavaScript` in the **Language** field of the **Scripting Language** filter, and ensure that the JavaScript syntax in the script conforms with Nashorn engine syntax. For more information about migrating from Rhino to Nashorn, see the Rhino Migration Guide.

1. Configure API Gateway to invoke the policy. For more information, see *Use JavaScript for custom requirements* on page 19.
2. Test the policy using API Tester. For more information, see *Use JavaScript for custom requirements* on page 19.

Invoke the policy

To configure the API Gateway to invoke the new policy, follow these steps:

1. Under the **Environment Configuration > Listeners** node in Policy Studio, select the path (for example, **API Gateway > Default Services > Paths**).
2. On the resolvers window on the right, click **Add > Relative Path**.
3. Enter the following values on the dialog and click **OK**:
   - **When a request arrives that matches the path:** `/invokescript`
   - **Path Specific Policy:** Click the browse button and select the **InvokeScript** policy. This sends all requests received on the path configured above to your newly configured policy.
4. To deploy the new configuration to API Gateway, click the **Deploy** button on the toolbar or press **F6** and follow the instructions.

Test the policy

To test the configuration, follow these steps:

1. Start API Tester.
2. Click the arrow next to the Play icon and select **Request Settings**.
3. In the **Url** field, enter `http://localhost:8080/invokescript` to send the message to the relative path you configured above.
4. Click **Run** to send the message to API Gateway.

Tip Alternatively, you can test the policy by entering the URL `http://localhost:8080/invokescript` into any web browser.

Java and JavaScript translations

If you are using JavaScript in a **Scripting Language** filter to add a custom filter to API Gateway (see *Use JavaScript to call existing Java code* on page 17 or *Use JavaScript for custom requirements* on page 19), the following table provides some tips on translating from Java or starting with JavaScript.

Note We recommend that you select `JavaScript` in the **Language** field of the **Scripting Language** filter, and ensure that the JavaScript syntax in the script conforms with Nashorn engine syntax. For more information about migrating from Rhino to Nashorn, see the Rhino Migration Guide.

Java	Equivalent in JavaScript
`String x = new String("Hello World");`	`var x = new java.lang.String("Hello World");`

`import java.io.*;`	`var ioImport = new JavaImporter (Packages.java.io);`
	`with(ioImport) {`
	`…`
	`}`

`try {`
`}`
`catch (Exception exp) {`
`}`
	`try {`
	`}`
	`catch (exp) {`
	`}`

`Runnable x = new Runnable(){`
`public void run() {`
`// do something`
`}`
`});`
	`var v = new java.lang.Runnable() {`
	`run: function() {`
	`// do something`
	`}`
	`}`

`byte[] x = new byte[10];`	`var x = java.lang.reflect.Array.newInstance (java.lang.Byte.TYPE, 10);`
`for (FilterInvocation filterInvocation : invocation.path) {`	`for (i = 0; i < invocation.path.size(); i++) {`

Write a custom filter using the extension kit

Note The following sections refer to `jabber` sample code that is no longer included in the code samples supplied with API Gateway. We recommend that you use this section only as a general guide for writing a custom filter using the extension kit.

In this approach, you write your custom filter using the API Gateway developer extension kit. This section details how to write a custom message filter, called the **Jabber Filter** (API Gateway runtime component and Policy Studio configuration component). It also shows how to configure it as part of a policy in Policy Studio and then demonstrates how the filter sends an instant message to an account on Google Talk.

The steps required to build, integrate, configure, and test the supplied `JabberFilter` and `JabberProcessor` sample classes are as follows:

Step	Description
Create the TypeDoc on page 21	Every filter has an associated XML-based TypeDoc description file that contains the entity type definition. It defines the configuration field names for that filter and the corresponding data types for that filter.
Create the Filter class on page 23	Every message filter returns its corresponding Processor and Policy Studio classes.
Create the Processor class on page 24	The Processor class is the API Gateway runtime component that is responsible for processing the message. Every message filter has an associated Processor and Filter class.
Create the declarative UI XML file on page 26	The declarative XML file is used to define the user interface for filters and dialogs.
Create the Policy Studio classes on page 28	All filters are configured using Policy Studio. Every filter has a configuration wizard that enables you to set each of the fields defined in the entity that corresponds to that filter. You can then add the filter to a policy to process messages.
Build the classes on page 32	When the classes are written, you must build them and add them to the API Gateway and client CLASSPATH. Example classes are included in the `DEVELOPER_SAMPLES/jabber` directory.
Load the TypeDocs on page 34	You must register the TypeDoc created for the filter with the entity store.
Construct a policy on page 35	Construct a policy that sends an instant message to an account on Google Talk and echoes a message back to the client. Use the GUI component of the newly added filter to specify its configuration and test the functionality of the filter (and its configuration).

Create the TypeDoc

A *TypeDoc* is an XML file that contains entity type definitions. Entity type definitions describe the format of data associated with a configurable item. For more details on entity types, see *Entity types* on page 61.

All TypeDocs for custom filters must:

- Extend the `Filter` type
- Define a constant filter class (for example, JabberFilter)
- List the configuration fields for the entity

The following example shows how the TypeDoc lists the various fields that form the configuration data for the JabberFilter.

```
<entityStoreData>
  <entityType name="JabberFilter" extends="Filter">
      <constant name="class" type="string"
        value="com.vordel.jabber.filter.JabberFilter"/>
      <field name="fromEmailAddress" type="string" cardinality="1"/>
      <field name="password" type="string" cardinality="1"/>
      <field name="resourceName" type="string" cardinality="1"/>
      <field name="toEmailAddress" type="string" cardinality="1"/>
      <field name="messageStr" type="string" cardinality="1"/>
   </entityType>
</entityStoreData>
```

You can also provide internationalized log messages by specifying an `<entity>` block of type `InternationationalizationFilter` in the `<entityStoreData>` elements. For example:

```
<entityStoreData>
 <!-- Internationalization for logging / audit trail -->
 <entity xmlns="http://www.vordel.com/2005/06/24/entityStore"
   type="InternationalizationFilter">
    <key type="Internationalization">
        <id field="name" value="Internationalization Default"/>
    </key>
    <fval name="type">
        <value>JabberFilter</value>
    </fval>
    <fval name="logFatal">
        <value>Error in the Jabber Filter sending instant message.
          Error: ${circuit.exception}</value>
    </fval>
    <fval name="logFailure">
        <value>Failed in the Jabber Filter sending instant message</value>
    </fval>
    <fval name="logSuccess">
        <value>Success in the Jabber Filter sending instant message</value>
    </fval>
 </entity>
</entityStoreData>
```

Create the Filter class

A `Filter` class is responsible for returning the corresponding API Gateway runtime class and Policy Studio class.

The `Filter` class is responsible for the following tasks:

- Specifying the message attributes it requires, consumes, and generates.
- Returning the corresponding API Gateway runtime class (the Processor class).
- Returning the corresponding Policy Studio class.

The following code shows the members and methods of the `JabberFilter` class.

```
public class JabberFilter extends DefaultFilter {

    protected final void setDefaultPropertyDefs() {
        reqProps.add(new PropDef(MessageProperties.CONTENT_BODY,
          com.vordel.mime.Body.class));
    }

    @Override
    public void configure(ConfigContext ctx, com.vordel.es.Entity entity)
      throws EntityStoreException {
        super.configure(ctx, entity);
    }

    public Class getMessageProcessorClass() {
        return JabberProcessor.class;
    }

    public Class getConfigPanelClass() throws ClassNotFoundException {
        // Avoid any compile or runtime dependencies on SWT and other UI
        // libraries by lazily loading the class when required.
        return Class.forName("com.vordel.jabber.filter.JabberFilterUI");
    }
}
```

There are two important methods implemented in this class:

- `setDefaultPropertyDefs`
- `getMessageProcessorClass`

The `setDefaultPropertyDefs` method enables the filter to define the message attributes that it *requires*, *generates*, and *consumes* from the attributes message whiteboard.

The whiteboard contains all the available message attributes. When a filter generates message attributes, it puts them up on the whiteboard so that when another filter requires them, it can pull them off the whiteboard. If a filter consumes a message attribute, it is wiped from the whiteboard so that no other filter in the policy can use it.

The attributes are stored in sets of property definitions (`Set<PropDef>`). A property definition defines a property to type mapping. There are `reqProps`, `genProps`, and `consProps`, which are inherited from the Filter class.

In the case of the `JabberFilter` class, the `content.body` attribute, which is of type `com.vordel.mime.Body`, is required because the SOAP parameters must be extracted from the body of the HTTP request. The property definition is declared as follows:

```
protected final void setDefaultPropertyDefs() {
    reqProps.add(new PropDef(MessageProperties.CONTENT_BODY,
      com.vordel.mime.Body.class));
}
```

The next method is the `getMessageProcessorClass` method, which returns the API Gateway runtime component (the Processor class) that is associated with this Filter class. Each Filter class has a corresponding Processor class, which is responsible for processing the message.

Finally, the corresponding Policy Studio configuration class is returned by the `getConfigPanelClass` method, which in this case is the `com.vordel.jabber.filter.JabberFilterUI` class. This class is described in detail in *Create the Policy Studio classes* on page 28.

Create the Processor class

This is the API Gateway runtime component of the filter that is returned by the `getMessageProcessorClass` of the Filter class. The Processor class is responsible for performing the processing on messages. It uses the configuration data to process each message.

The following code shows how the Processor attaches to the Filter class and uses its data to process the message. It gets the configuration data using selectors to set up a connection to an XMPP server, creates a chat, and sends a message to a chat participant. The complete code for the class is available in the `DEVELOPER_SAMPLES/jabber` directory.

```
public class JabberProcessor extends MessageProcessor {

    …

    @Override
    public void filterAttached(ConfigContext ctx, com.vordel.es.Entity entity)
      throws EntityStoreException {
        super.filterAttached(ctx, entity);
        to = new Selector<String>(entity.getStringValue("toEmailAddress"),
          String.class);
        byte[] passwordBytes = entity.getEncryptedValue("password");
        if (passwordBytes != null) {
            try {
                passwordBytes = ctx.getCipher().decrypt(passwordBytes);
```

```
            } catch (GeneralSecurityException exp) {
                Trace.error(exp);
            }
        }
        String pass = new String(passwordBytes);
        password = new Selector<String>(pass, String.class);
        resourceName = new Selector<String>(entity.getStringValue("resourceName"),
          String.class);
        from = new Selector<String>(entity.getStringValue("fromEmailAddress"),
          String.class);
        messageStr = new Selector<String>(entity.getStringValue("messageStr"),
          String.class);
    }

    public boolean invoke(Circuit c, Message message)
      throws CircuitAbortException {
       XMPPConnection connection = null;
       try {
           ConnectionConfiguration config =
           new ConnectionConfiguration("talk.google.com", 5222, "gmail.com");
           connection = new XMPPConnection(config);
           SASLAuthentication.supportSASLMechanism("PLAIN", 0);
           connection.connect();
           connection.login(from.substitute(message), password.substitute(message),
             resourceName.substitute(message));
       } catch (org.jivesoftware.smack.XMPPException ex) {
           Trace.error("Error establishing connection to XMPP Server");
       }
       Chat chat = connection.getChatManager().createChat(to.substitute(message),
         new MessageListener(){
           @Override
           public void processMessage(Chat arg0,
             org.jivesoftware.smack.packet.Message arg1) {
               Trace.debug(arg1.getBody());
           }
       });
       try {
           chat.sendMessage(messageStr.substitute(message));
           connection.disconnect();
       } catch (org.jivesoftware.smack.XMPPException ex) {
           Trace.error("Error Delivering block");
       }
       return true;
    }
}
```

There are two important methods that must be implemented by every Processor class:

- `filterAttached`
- `invoke`

The `filterAttached` method should contain any API Gateway server-side initialization or configuration to be performed by the filter, such as connecting to third-party products or servers.

The `invoke` method is responsible for using the configuration data to perform the message processing. This method is called by API Gateway as it executes the series of filters in any given policy. In the case of the `JabberFilter`, the `invoke` method uses the configuration data to set up a connection to an XMPP server, creates a chat, sends a message to a chat participant, and disconnects from the XMPP server.

The `invoke` method can have the following possible results:

Result	Description
True	If the filter processed the message successfully (for example, successful authentication, schema validation passed, and so on), the `invoke` method should return a true result, meaning that the next filter on the success path for the filter is invoked.
False	If the filter processing fails (for example, the user was not authenticated, message failed integrity check, and so on), the `invoke` method should return false, meaning that the next filter on the failure path for the filter is invoked.
CircuitAbortException	If for some reason the filter cannot process the message at all (for example, if it cannot connect to an Identity Management server to authenticate a user), it should throw a `CircuitAbortException`. If a `CircuitAbortException` is thrown in a policy, the designated fault processor (if any) is invoked instead of any successive filters on either the success or failure paths.

Create the declarative UI XML file

The declarative UI XML file encapsulates the design of the user interface of filters and dialogs. It includes the markup UI elements and bindings to create the Jabber filter dialog within Policy Studio.

For more information on using declarative XML, see *Define user interfaces using declarative XML* on page 37. For a complete listing of the available elements and bindings, see *Declarative UI reference* on page 85.

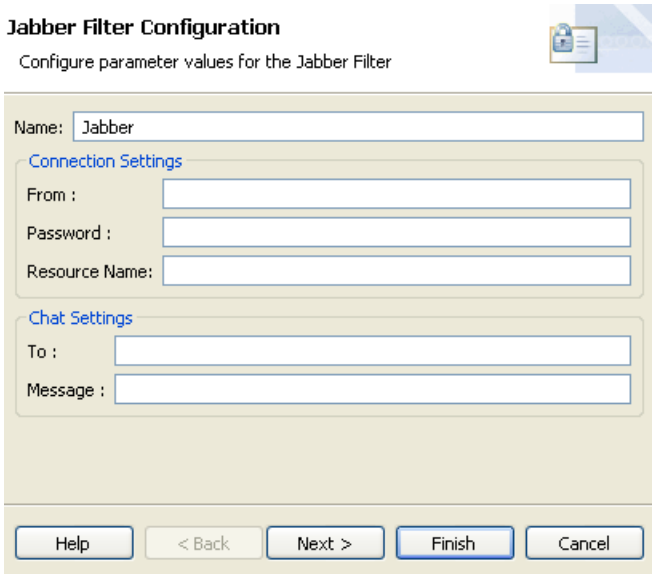
The following declarative XML shows the elements needed to create the Jabber filter dialog:

```
<ui>
 <panel columns="2">
  <NameAttribute />

  <!-- Connection settings -->
  <group label="CONNECTION_SETTINGS_LABEL"
```

```
    columns="2" span="2" fill="false">

    <TextAttribute field="fromEmailAddress"
      label="FROM_EMAIL_ADDRESS_LABEL"
      displayName="FROM_EMAIL_ADDRESS_DISP_NAME"/>
    <PasswordAttribute field="password"
      label="FROM_PASSWORD_LABEL"
      displayName="FROM_PASSWORD_DISP_NAME"/>
    <TextAttribute field="resourceName"
      label="RESOURCE_NAME_LABEL"
      displayName="RESOURCE_NAME_DISP_NAME"/>
  </group>

  <!-- Chat Settings -->
  <group label="CHAT_SETTINGS_LABEL"
    columns="2" span="2" fill="false">

    <TextAttribute field="toEmailAddress"
      label="TO_EMAIL_ADDRESS_LABEL"
      displayName="TO_EMAIL_ADDRESS_DISP_NAME"/>
    <TextAttribute field="messageStr"
      label="MESSAGE_LABEL"
      displayName="MESSAGE_DISP_NAME"/>
  </group>
 </panel>
</ui>
```

All declarative XML files start with `<ui>` elements. The preceding markup contains several `<TextAttribute>` elements and a `<PasswordAttribute>` element. Each element has a field attribute, which directly corresponds to the field definitions in the type definition, and a label attribute that correspond to localization keys in the `resources.properties` file.

The following figure shows the Jabber filter dialog that this XML creates.



Create the Policy Studio classes

The next step after defining the user interface is to write two GUI classes that enable the fields defined in the `JabberFilter` type definition to be configured. When the GUI classes and resources are built, the visual components can be used in Policy Studio to configure the filter and add it to a policy.

The following table describes the GUI classes and resources for the `JabberFilter`:

Class or Resource	Description
`JabberFilterUI.java`	This class lists the pages that are involved in a filter configuration window. Each filter has at least two pages: the main configuration page, and a page where log messages related to the filter can be customized. This class is returned by the `getConfigPanelClass` method of the `JabberFilter` class.
`JabberFilterPage.java`	This class loads the declarative XML file which defines the layout of the visual fields on the filter's main configuration window. For example, there are five fields on the configuration window for the **Jabber Filter** corresponding to the five fields defined in the entity type definition.
`resources.properties`	This file contains all text displayed in the GUI configuration window (for example, dialog titles, field names, and error messages). This means that the text can be customized or internationalized easily without needing to change the code.
`jabber.gif`	This image file is the icon that identifies the filter in Policy Studio, and is displayed in the filter palette.

The `JabberFilterUI` class, which is returned by the `getConfigPanelClass` method of the `JabberFilter` class, is responsible for the following:

- Listing the configuration pages that make up the user interface for the filter
- Naming the category of filters to which this filter belongs
- Specifying the name of the images to use as the icons and images for this filter

JabberFilterUI class

The code for the `JabberFilterUI` class is as follows:

```
public class JabberFilterUI extends DefaultGUIFilter
{
    public Vector<VordelPage> getPropertyPages() {
```

```
        Vector<VordelPage> pages = new Vector<VordelPage>();
        pages.add(new JabberFilterPage());
        pages.add(createLogPage());
        return pages;
    }

    public String[] getCategories() {
        return new String[]{_("FILTER_GROUP_JABBER")};
    }

    private static final String IMAGE_KEY = "jabberFilter";
    static {
        Images.getImageRegistry().put(IMAGE_KEY,
          Images.createDescriptor(JabberFilterUI.class, "jabber.gif"));
    }

    public String getSmallIconId() {
        return IMAGE_KEY;
    }

    public Image getSmallImage() {
        return Images.get(IMAGE_KEY);
    }

    public ImageDescriptor getSmallIcon() {
        return Images.getImageDescriptor(IMAGE_KEY);
    }
}
```

The following table describes the important methods:

Method	Description
`public Vector getPropertyPages()`	Initializes a Vector of the pages that make up the total configuration windows for this filter. Successive pages are accessible by clicking the **Next** button on the Policy Studio configuration window.
`public String[] getCategories()`	This method returns the names of the filter categories that this filter belongs to. The filter is displayed under these categories in the filter palette in Policy Studio. The **Jabber Filter** is added to the **XMPP Filters** category.
`public Image getSmallImage()`	The default image for the filter, which is registered in the static block in the preceding code, can be overridden by returning a different image here.
`public ImageDescriptor getSmallIcon()`	The default icon for the filter can be overridden by returning a different icon here.

A page only represents a single configuration window in Policy Studio. You can chain together several pages to form a series of configuration windows that together make up the overall configuration for a filter. By default, all filters consist of two pages: one for the filter configuration fields, and one for per-filter logging. However, more pages can be added if required. You can add additional pages to the configuration in the `getPropertyPages` method.

If you look at the `getPropertyPages` method of the `JabberFilterUI` class, you can see that the `JabberFilterPage` class forms one of the configuration windows (or pages) for the `JabberFilter`. The `JabberFilterPage` class is responsible for loading the declarative UI XML file that defines the layout of all the input fields that make up the configuration window for the `JabberFilter`.

JabberFilterPage class

The code for the `JabberFilterPage` class is as follows:

```
public class JabberFilterPage extends VordelPage
{
    public JabberFilterPage() {
        super("jabberPage");
        setTitle(_("JABBER_PAGE"));
        setDescription(_("JABBER_PAGE_DESCRIPTION"));
        setPageComplete(false);
    }

    public String getHelpID() {
        return "jabber.help";
    }

    public boolean performFinish() {
        return true;
    }

    public void createControl(Composite parent) {
        Composite panel =
          render(parent,
          getClass().getResourceAsStream("send_instant_message.xml"));
        setControl(panel);
        setPageComplete(true);
    }
}
```

There are four important interface methods that must be implemented in this class:

Method	Description
`public JabberFilterPage()`	The constructor performs some basic initialization, such as setting a unique ID for the page, and setting the title and description for the page. The text representing the page title and description are kept in the `resources.properties` file so that they can be localized or customized easily.

`public String getHelpID()`	This method is called by the Policy Studio help system. There is a **Help** button on every configuration page in Policy Studio. When you click this button, the help system is invoked. Every page has a help ID (for example, `jabber_help`) associated with it, which is mapped to an HTML help page. This mapping is defined in the following file under the directory where you have installed Policy Studio:
	`/plugins/com.vordel.rcp.policystudio.gateway.help_<version>/csh.xml`
	To define a mapping for the help page, follow these steps:
	1. Open the `csh.xml` file.
	2. Add the following XML to the file:
	`<context id="jabber_help"><description>Jabber Filter</description><topic label="Jabber Filter" href="Content/PolicyDevTopics/jabber.htm"/></context>`
	3. Create a help file called `jabber.htm` to contain the help for the filter in HTML format.
	All URLs specified in the `csh.xml` file are relative from the `/plugins/com.vordel.rcp.policystudio.gateway.help_<version>` directory of your Policy Studio installation.

`public boolean performFinish()`	This method gives you the chance to process the user-specified data before it is submitted to the entity store. For example, any validation on the data should be added to this method.
`public void createControl (Composite parent)`	This method is responsible for loading the declarative UI XML file that creates the configuration pages. Localization keys from the `resources.properties` file are used to give labels for the input fields in the XML file.

resources.properties file

Both the declarative UI XML file and the GUI classes use localized keys for all text that is displayed on the configuration window. This makes it easy to localize or customize all text displayed in Policy Studio. The localization keys and their corresponding strings are stored in the `resources.properties` file, which takes the following format:

```
#
# Palette category for Jabber filters
#
FILTER_GROUP_JABBER=XMPP Filters

#
# Properties for the JabberFilter Configuration Wizard
#
JABBER_PAGE=Jabber Filter Configuration
JABBER_PAGE_DESCRIPTION=Configure parameter values for the Jabber Filter

#
# Field labels and descriptions
#
CONNECTION_SETTINGS_LABEL=Connection Settings
FROM_EMAIL_ADDRESS_LABEL=From :
FROM_EMAIL_ADDRESS_DISP_NAME=Person sending the instant message
FROM_PASSWORD_LABEL=Password :
FROM_PASSWORD_DISP_NAME=Password of Person sending the message
RESOURCE_NAME_LABEL=Resource Name:
RESOURCE_NAME_DISP_NAME=Unique resource Name
CHAT_SETTINGS_LABEL=Chat Settings
TO_EMAIL_ADDRESS_LABEL=To :
TO_EMAIL_ADDRESS_DISP_NAME=Person receiving the instant message
MESSAGE_LABEL=Message :
MESSAGE_DISP_NAME=Message Content
```

The final resource is the `jabber.gif` file, which is displayed as the icon for the **Jabber Filter** in Policy Studio.

Build the classes

Perform the following steps to build the JAR file for the Jabber sample:

1. Change to the sample directory (`DEVELOPER_SAMPLES/jabber`).
2. Run the following command to compile the code and build the JAR:

   ```
   ant -f build.xml
   ```

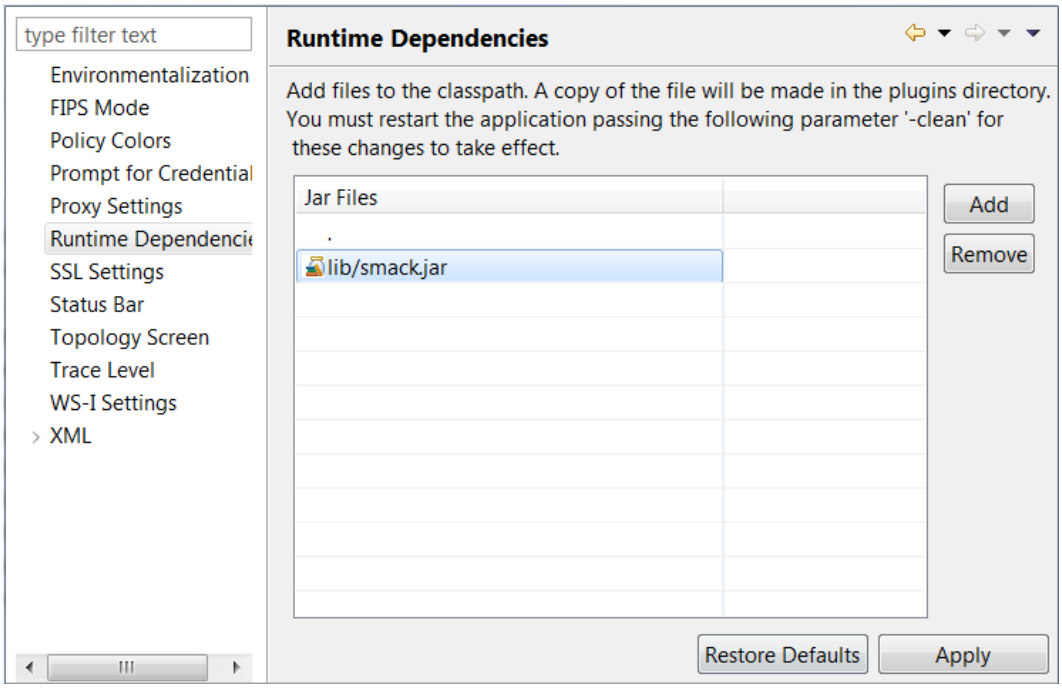
   Tip See the `README.TXT` file for additional instructions.

3. Add the new JAR and any third-party JAR files used by the Jabber classes (for example, the SMACK API JAR files) to the CLASSPATH for all API Gateways and Node Managers on a host by copying them to the `INSTALL_DIR/apigateway/ext/lib` directory.

   Alternatively, you can add the JARs to the CLASSPATH for a single API Gateway instance only, by copying them to the `INSTALL_DIR/apigateway/groups/GROUP_ID/INSTANCE_ID/ext/lib` directory.

4. Add the new JAR and any third-party JAR files used by the Jabber classes (for example, the SMACK API JAR files) to the runtime dependencies in Policy Studio. Select **Window > Preferences > Runtime Dependencies**, and click **Add** to browse to the new JAR and any third-party JARs, and add them to the list. Click **Apply** to save the changes.

   The following figure shows the runtime dependencies.



5. Restart the API Gateway instances and Node Managers.
6. Restart Policy Studio using the following command:

```
policystudio -clean
```

The extension kit includes all of the associated resources and classes to create the **Jabber Filter**.

Custom filter dependencies

If your custom filter introduces a dependency on a new third-party library, you must first check if the required library is already available under the following directory and sub-directories:

```
INSTALL_DIR/apigateway/system/lib
```

Note Any JAR file that you add under the following directories will be pushed ahead of `apigateway/system` JAR files on the CLASSPATH:

- `INSTALL_DIR/apigateway/ext/lib`
- `INSTALL_DIR/apigateway/groups/`*`GROUP_ID`*`/`*`INSTANCE_ID`*`/ext/lib`

For example, API Gateway ships with specific versions of several Apache Commons JARs. Introducing conflicting versions of these JARs could adversely affect the ability of the API Gateway and Node Manager to function correctly.

Load the TypeDocs

You can register the type definition for the **Jabber Filter** with the entity store using Policy Studio. When the entity type is registered, any time API Gateway needs to create an instance of the **Jabber Filter**, the instance contains the correct fields with the appropriate types.

Register using Policy Studio

To register the type definition using Policy Studio, perform the following steps:

1. Start Policy Studio, and connect to the API Gateway.
2. Select **File > Import > Import Custom Filters**.
3. Browse to the `Typeset.xml` file. A TypeSet file is used to group together one or more TypeDocs. This enables multiple TypeDocs to be added to the entity store in batch mode. The `JabberTypeSet.xml` file includes the following:

   ```
   <typeSet>
       <!-- JabberFilter Typedoc -->
       <typedoc file="JabberFilterDesc.xml" />
   </typeSet>
   ```

   When you import the TypeSet, the workspace refreshes. The new filter is available in the filter list.
4. To verify that the Jabber filter exists, select an existing policy in the Policy Studio tree, and you should see the **XMPP Filters** category in the palette, which contains the new custom **Jabber** filter.
5. Click the **Deploy** button in the toolbar to deploy the new custom filter.

   Tip You can also save the current configuration and deploy at a later point. For more information on managing deployments, see the *API Gateway DevOps Deployment Guide*.

Note Another way to verify that your new filter has been installed is to use the ES Explorer. You can use the ES Explorer tool for browsing the entity types and entity instances that have been registered with the Entity Store. For more information, see *Use the ES Explorer* on page 63.

Construct a policy

You can build policies using the policy editor in Policy Studio. To build a policy, you can drag message filters from the filters palette on the right on to the policy canvas. You can then link these filters using success paths or failure paths to create a network of filters.

Create the policy

To create a policy, perform the following steps:

1. In the Policy Studio tree, right-click the **Policies** node and select **Add Policy**.
2. Enter `Send Instant Message` as the name of the new policy in the dialog.
3. Drag a **Jabber** filter from the **XMPP** group onto the policy canvas.
4. Enter the following in the **Jabber** filter dialog:
    - **Name:** Name of the filter
    - **From:** User email address
    - **Password:** User password
    - **Resource Name:** Resource name (for example, `apigateway`)
    - **To:** Chat participant's email address
    - **Message:** Message to send
5. To check that the help is working correctly, click the **Help** button on the filter dialog.

    In *Create the Policy Studio classes* on page 28, as part of the `getHelp` method, you added a mapping to the `contexts.xml` file (in the `/plugins/com.vordel.rcp.policystudio.resources_<version>` directory of your Policy Studio installation). After restarting Policy Studio, the **Help** button should function correctly.
6. To add a **Reflect Message** filter, which echoes the client message back to the client, drag it from the **Utility** group onto the policy canvas.
7. Configure the **Reflect Message** filter as follows:
    - **Name:** Enter a name for the filter (or use the default)
    - **HTTP response code status:** Use the default value (`200`)
8. Connect the Jabber node to the Reflect Message node with a success path.
9. Right-click the **Jabber** filter, and select **Set as Start** to set it as the start filter for the policy.

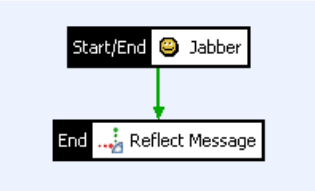
Invoke the policy

To configure the API Gateway to invoke the new policy, follow these steps:

1. Under the **Environment Configuration > Listeners** node in Policy Studio, select the path (for example, **API Gateway > Default Services > Paths**).

2. On the resolvers window on the right, click **Add > Relative Path**.
3. Enter the following values on the dialog and click **OK**:
    - **When a request arrives that matches the path:** `/send`
    - **Path Specific Policy:** Click the browse button and select the **Send Instant Message** policy. This sends all requests received on the path configured above to your newly configured policy.
4. To deploy the new configuration to API Gateway, click the **Deploy** button on the toolbar or press **F6** and follow the instructions.

The following diagram shows the complete policy:



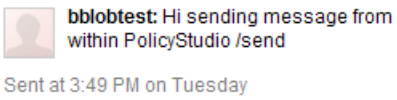
Test the policy

To test the configuration, follow these steps:

1. Start API Tester.
2. Click the arrow next to the Play icon and select **Request Settings**.
3. In the **Url** field, enter `http://localhost:8080/send` to send the message to the relative path you configured above.
4. Click **Run** to send the message to API Gateway.

Tip Alternatively, you can test the policy by entering the URL `http://localhost:8080/send` into any web browser.

API Gateway echoes the message back to the client using the **Reflect Message** filter after an instant message has been sent to an account on Google Talk. The following is an example of an instant message that appears on an account on Google Talk. This indicates that the newly added filter has worked successfully.



Define user interfaces using declarative XML

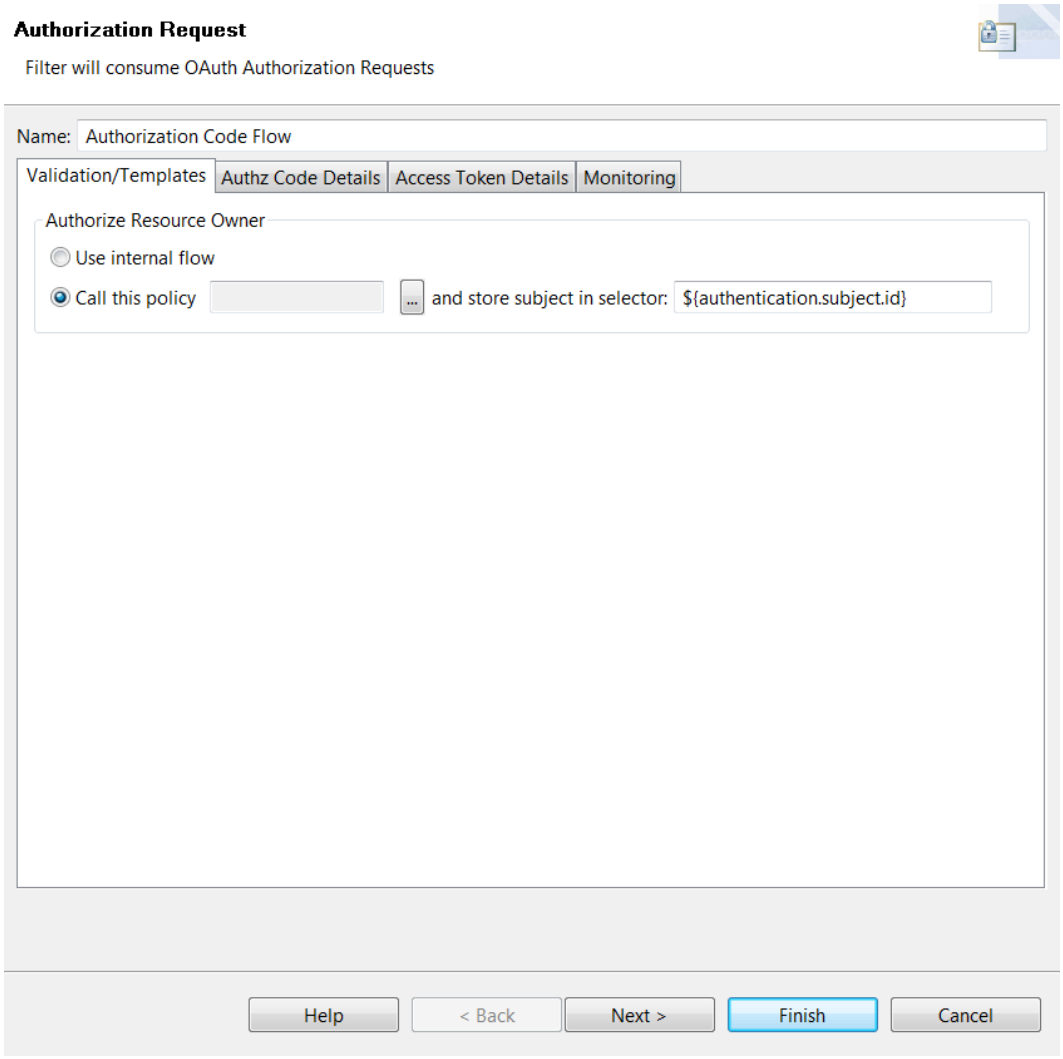
4

You can define the Policy Studio user interfaces for a custom filter using *declarative XML*. Declarative XML is a user interface markup language that is used to define UI elements and bindings, allowing you to quickly create dialogs within Policy Studio with minimal coding.

The defined UI elements map to Eclipse SWT (Standard Widget Toolkit) widgets and Axway ScreenAttributes (groups of SWT widgets backed by entity instances).

All declarative XML files start and end with a `<ui>` element. Each element has a field attribute that corresponds to a field definition in the type definition, and a label attribute that corresponds to a localization key in the `resources.properties` file.

The following is an example of a dialog in Policy Studio:



The following sections describe how the declarative UI XML file for this dialog is loaded within a Policy Studio class, and detail the declarative XML file that defines the dialog.

Note This dialog and the following code are for demonstration purposes only and might not reflect the current settings for this filter in Policy Studio.

Load the declarative XML file

The following code demonstrates how to load the declarative XML file within the Policy Studio class.

```
package com.vordel.client.manager.filter.oauth2.provider.authorize;

import org.eclipse.swt.widgets.Composite;
import com.vordel.client.manager.wizard.VordelPage;
```

```
public class OAuthAuthorizationRequestPage extends VordelPage {
    public OAuthAuthorizationRequestPage() {
        super ("ExampleFilterPage");
        setTitle("Put the title here");
        setDescription("Put the description here"));
        setPageComplete(true);
    }

    @Override
    public void createControl(Composite parent) {
        Composite container =
          render(parent, getClass().getResourceAsStream("declarative.xml"));
        setControl(container);
    }

    …

}
```

Declarative XML file

The following `declarative.xml` file defines the dialog in Policy Studio.

```
<ui>
  <panel columns="2" span="2">
    <NameAttribute />
    <tabFolder span="2">
      <tab label="OAUTH_APPLICATION_VALIDATION">
        <panel>
          <panel columns="2" fill="false">
            <TextAttribute field="kpsAlias"
              label="OAUTH_USE_KPS"
              required="true" />
          </panel>
          <group label="OAUTH_AUTHORIZATION_REQUEST_AUTHN" fill="false">
            <RadioGroupAttribute field="authNWith" columns="4">
              <choice value="INTERNAL_AUTHN_FLOW"
                label="USE_INTERNAL_FLOW" span="4" />
              <choice value="CIRCUIT_FLOW"
                label="OAUTH_AUTHORIZATION_REQUEST_AUTHN_USE_CIRCUIT" />
              <panel margin="1" columns="4" span="2" fill="false">
                <ReferenceSelector field="circuitPK"
                  selectableTypes="FilterCircuit"
                  searches="ROOT_CIRCUIT_CONTAINER,CircuitContainer"
```

```
                title="OAUTH_AUTHORIZATION_REQUEST_AUTHN_SELECT_CIRCUIT" />
              <TextAttribute field="subjectAttr"
                label="OAUTH_AUTHORIZATION_REQUEST_AUTHN_SELECTOR" />
            </panel>
          </RadioGroupAttribute>
        </group>
      </panel>
    </tab>
    <tab label="OAUTH_AUTHZ_CODE_GENERATION">
      <panel>
        <panel columns="3" fill="false">
          <ReferenceSelector field="authzCodeCache"
            selectableTypes="AuthzCodePersist"
            searches="OAuth2StoresGroup,AuthzCodeStoreGroup"
            label="OAUTH_AUTHORIZATION_REQUEST_STORE_CODE"
            title="OAUTH_AUTHORIZATION_REQUEST_CHOOSE_CACHE"
            required="true" />
        </panel>
        <panel columns="2" fill="false">
          <TextAttribute field="accessCodeTemplateLocation"
            label="ACCESS_CODE_TEMPLATE_LOCATION"
            required="true" />
        </panel>
        <group label="OAUTH_AUTHORIZATION_REQUEST_GENERATE_CODE"
          columns="4" fill="false">
          <NumberAttribute field="authzCodeLength"
            label="AUTHZ_CODE_LENGTH"
            required="true" min="10" />
          <NumberAttribute field="authzCodeExpiresInSecs"
            label="AUTHZ_CODE_EXPIRES_SECS"
            required="true" max="60O" />
        </group>
      </panel>
    </tab>
    <tab label="OAUTH_ACCESS_TOKEN_GENERATION">
      <scrollpanel>
        <panel>
          <include resource="accesstokengenerationtemplate.xml" />
          <include resource="refreshtokentemplate.xml" />
          <include resource="scopestemplate.xml" />
        </panel>
      </scrollpanel>
    </tab>

    …

  </tabFolder>
 </panel>
</ui>
```

This section describes some of the elements declared in the preceding XML file. For a complete listing of all the UI elements and bindings available, see *Declarative UI reference* on page 85.

This declarative XML file declares that the main panel of the dialog spans two columns. All the labels are obtained from the associated `resources.properties` file.

The `<panel>` tag renders an SWT Composite widget, which is usually employed to group other widgets.

The `<NameAttribute>` tag renders an SWT Label and accompanying Text widget. A `NAME` label must exist in the appropriate `resources.properties` file.

The `<tabFolder>` and `<tab>` attributes produce a tab folder with tabs using the labels supplied.

In the first tab it creates a text box and a group of two radio buttons. Beside the second radio button it creates a `<ReferenceSelector>`. The `<ReferenceSelector>` tag renders an SWT Label, Text, and Button control. When clicked, the button displays a reference browser to allow the user to easily select the required entity reference. The following table shows the main `<ReferenceSelector>` tag attributes.

Name	Description
`selectableTypes`	Specifies the entity types (as a comma separated list) that are selectable in the TreeViewer displayed in the Reference Selector dialog.
`searches`	Specifies the entity types (as a comma separated list) that are searchable for entities of those types specified by the `selectableTypes` attribute.

After the `<ReferenceSelector>`, it creates another text box to store the subject.

In the second tab it creates another `<ReferenceSelector>`, followed by a text box and a group of two `<NumberAttribute>` tags. The `<NumberAttribute>` tags render an SWT Label and accompanying Text widget. The Text widget only accepts numbers as input.

The third tab includes a `<scrollpanel>` tag. The `<scrollpanel>` tag renders an SWT ScrolledComposite widget. When rendered, the control automatically calculates the extent of its children so that the scroll bars are rendered correctly.

One of the following tags must be a direct child of `<scrollpanel>`:

- `<panel>`
- `<group>`
- `<tabFolder>`

For the **Access Token Details** tab it uses the `<panel>` tag. It also uses the `<include>` tag that allows another declarative XML file to be included inline in the parent including XML file.

For the remaining tabs it uses the `<include>` tag but introduces a new `<ButtonAttribute>` tag in the fourth tab. The `<ButtonAttribute>` tag renders an SWT Button widget with the `SWT.PUSH` style applied, backed by the specified entity field.

Unit test a filter using the Traffic Monitor API

5

The following example shows you how to create JUnit tests to test a custom filter using the Traffic Monitor API. Any JUnit test classes you write should extend and use the existing test classes that are shipped with API Gateway, as these test classes provide several assertions that are used to evaluate the responses returned.

The two main classes to use are:

- `TestClientResponse` – Uses Jersey client APIs to send GET and POST requests to API Gateway. The JAR files can be found in the `INSTALL_DIR/apigateway/system/lib/modules` directory.
- `TrafficMonitorClient` – Used to invoke the Traffic Monitor REST API, which monitors the traffic in and out of the API Gateway, and evaluate the responses returned. These classes are contained in `testClient.jar`, which can be found in the `INSTALL_DIR/apigateway/system/lib/` directory.

Note A Node Manager and an API Gateway instance must be running before the JUnit tests can be run.

Write a JUnit test for the Health Check policy filters

Perform the following steps to write a JUnit test for the Health Check policy filters:

1. Create a test class called `TestHealthCheck`. It should extend the `TestClientResponse` utility class, which contains several assertion methods that can be used to test the client responses returned from a web resource. For example:

```
import com.vordel.ops.TestClientResponse;

public class TestHealthCheck extends TestClientResponse {
…
}
```

2. Within the `setup` method, create a new instance of a `com.vordel.ops.TrafficMonitorClient`. This client contains several assertion methods that can be used to evaluate the response based on the traffic information in and out of the API Gateway, and the CorrelationId.

```
@BeforeClass
public static void setup() throws NodeManagerAPIException {
  client = new TrafficMonitorClient("https", "localhost", "8090",
    SERVER_ID, USERNAME, PASSWORD);
}
```

3. Create a test case that invokes a request and evaluates the response returned using the `TrafficMonitorClient`. Each filter of the policy can be evaluated to determine if it passed or failed.

```
import javax.ws.rs.core.Response;

@Test
public void testHealthCheck() {

    // Execute health check policy
    Response response = get("http://localhost:8080/healthcheck");
    assertStatusCode(response, 200);

    // Get its correlation id.
    String correlationId = getCorrelationId(response);

    // check HTTP header
    assertContainsHeader(response, "Host");

    // check HTTP header and value
    assertContainsHeaderWithValue(response, "Host", "localhost:8080");
    // Check that Set Message and Reflect Filters pass.
    client.assertFilterPassed(correlationId, "Set Message", "Reflect");

    // Ensure fault handlers did not fire.
    client.assertFilterOfTypeDidNotExecute(correlationId, "GenericError");
    client.assertFilterOfTypeDidNotExecute(correlationId, "JSONError");
    client.assertFilterOfTypeDidNotExecute(correlationId, "SOAPFault");

    client.assertNFiltersPassed(correlationId, 2);
    client.assertNFiltersFailed(correlationId, 0);
}
```

For more information on the client assertion methods, go to API Gateway Javadoc available from Axway Support at https://support.axway.com.

Java interfaces for extending API Gateway 6

This section describes the following Java interfaces that can be used to extend API Gateway:

- LoadableModule – Classes that implement this interface are used to instantiate long-lived objects in the API Gateway process. These objects can be loaded at startup or when a new configuration is deployed, and can be unloaded at shutdown.
- MessageCreationListener – Classes that implement this interface are used to track message creation.
- MessageListener – Classes that implement this interface are used to to track the changes in a message as it flows through API Gateway.

This section also provides examples of how to implement these interfaces.

Create a loadable module

This section describes the `LoadableModule` interface, and provides an example of a loadable module class that implements the interface. Another example of a class that implements `LoadableModule` can be found in the `DEVELOPER_SAMPLES/FilterInterceptorLoadableModule` directory.

LoadableModule interface

The `LoadableModule` interface provides methods that are invoked during startup or shutdown of the API Gateway, or when a new configuration is deployed.

A `LoadableModule` class can be loaded at startup, or when a new configuration is deployed, and can be unloaded at shutdown. Loadable modules are used to instantiate long-lived objects in the API Gateway server process (for example, a transport listener, a cache manager, an embedded broker, and so on).

The loadable module object itself is informed when it is loaded, reconfigured, and unloaded. The order in which all loadable modules are loaded and configured is specified explicitly with a `loadorder` field in the entity type description associated with that loadable module. If your loadable module depends on another loadable module then it must have a load order which is higher than the module on which it depends.

The base `LoadableModule` interface has three methods. These are used on startup of the API Gateway, on shutdown of the API Gateway, or when a new configuration is deployed to the API Gateway. The following example shows the methods.

```
public interface LoadableModule {


    …

    /**
     * Load a module into the process.
     * @param parent Loadable modules may nest - @parent provides access to
     * the containing LM, or is null for a top-level module.
     * @param entityType - The actual type of the entity that caused this
     * class to be constructed.
     */
    public void load(LoadableModule parent, String entityType)
            throws FatalException;
    /**
     * Unload the module from the process. This is called once
     * when the module is no longer required.
     */
    public void unload();

    /**
     * Configure the loadable module. Called if the entity for the
     * object changes.
     * Note that currently, modules are unloaded and reloaded for each
     * refresh - this behaviour should not be relied upon.
     */
    public void configure(ConfigContext pack, Entity object)
            throws EntityStoreException,FatalException;
}
```

A loadable module is normally designed as a singleton object so that only one instance exists. The same instance is returned to all filters accessing the loadable module. (For example, the `GlobalProperties` class is global and there is only one instance that is accessible to all parts of the application.)

Loadable module classes can be subclassed to provide extra information. For example, the `TransportModule` class provides settings to indicate what traffic information should be recorded for a specific protocol.

You can load a type definition for a `LoadableModule` class using the ES Explorer tool (see *Load a type definition* on page 63). You can also view the `LoadableModule` classes in the ES Explorer (see *Use the ES Explorer* on page 63).

LoadableModule example - TimerLoadableModule

This section describes how to create a loadable module using a simple example. The `TimerLoadableModule` class creates a timer and traces a message to the trace output at a set interval.

There are two parts to building this example:

1. *Create the TypeDoc definition for the loadable module* on page 46
2. *Create the loadable module implementation class* on page 47

Create the TypeDoc definition for the loadable module

A TypeDoc is an XML file that contains entity type definitions. Entity type definitions describe the format of data associated with a configurable item. For more details on entity types, see *Entity types* on page 61.

All TypeDocs for `LoadableModule` classes must:

- Extend the `LoadableModule` type or one of its subtypes (such as `NamedLoadableModule`)
- Define a constant `LoadableModule` class
- Define the `loadorder` that indicates in what order the loadable module is loaded and configured
- List the configuration fields for the entity

The following definition lists the various fields that form the configuration data for the `TimerLoadableModule` class.

```
<entityType name="TimerLoadableModule" extends="NamedLoadableModule">
    <constant name="_version" type="integer" value="0"/>
    <constant name="class" type="string"
      value="com.vordel.example.TimerLoadableModule "/>
    <constant name="loadorder" type="integer" value="20"/>
    <field name="delaySecs" type="integer" cardinality="1" default="30"/>
    <field name="periodSec" type="integer" cardinality="1" default="10"/>
    <field name="textMessage" type="string" cardinality="1" default="Hello world"/>
</entityType>
```

In this definition:

- `delaySecs` – Delay in milliseconds before task is to be executed
- `periodSec` – Time in milliseconds between successive task executions
- `textMessage` – Message to be output to the trace file

Create the loadable module implementation class

The API Gateway server-side implementation class is responsible for creating a timer and scheduling a task for repeated fixed-rate executions, beginning after a specified delay. Subsequent executions take place at regular intervals, separated by a specified period.

The following code shows the members and methods of the `TimerLoadableModule` class:

```
public class TimerLoadableModule implements LoadableModule {

  Timer timer = null;
  int initialDelay = 30 * 1000;
  int period = 10 * 1000;
  String message = "Hello world";

  @Override
  public void configure(ConfigContext solutionPack, Entity entity)
    throws EntityStoreException {

      if (timer != null)
        timer.cancel();

      // load the configuration settings
      initialDelay = entity.getIntegerValue("delaySecs") * 1000;
      period = entity.getIntegerValue("periodSec") * 1000;
      message = entity.getStringValue("textMessage");
      TimerTask task = new TimerTask() {
          public void run() {
              Trace.error(message);
          }
      };
      timer.scheduleAtFixedRate(task, initialDelay, period);
  }

  @Override
  public void load(LoadableModule loadableModule, String arg1) {
      timer = new Timer();
  }

  @Override
  public void unload() {
      // clean up
      if (timer != null)
          timer.cancel();
  }
}
```

The `load` method creates a `Timer` instance. The `unload` method terminates the timer, discarding any currently scheduled tasks. It does not interfere with a currently executing task (if it exists). When the timer is terminated, its execution thread terminates gracefully, and no more tasks can be scheduled on it.

The `configure` method loads the configuration data and creates a new `TimerTask` that traces a message to the trace output and schedules this task to be executed at a repeated fixed rate, beginning after a delay, with subsequent executions to take place at regular intervals, separated by a specified period.

See *Create a message listener* on page 49 for another example of a loadable module class that is used for monitoring messages passing through policies in an *interceptor*.

Note Currently, each loadable module is unloaded and recreated at reconfiguration time, so that the `configure` method is called only once for each loadable module. This behavior should not be relied upon.

Create a message creation listener

This section describes the `MessageCreationListener` interface, and provides an example of a message creation listener class that implements the interface. The sample code can be found in the `DEVELOPER_SAMPLES/FilterInterceptorLoadableModule` directory.

MessageCreationListener interface

The `MessageCreationListener` interface provides a method that is invoked when a message is created.

A `MessageCreationListener` class is used to track message creation. It is called when a message is created but before the originator of the message has populated any properties.

An example of its usage can be seen in the following `FilterInterceptor` class:

```
public class FilterInterceptor implements LoadableModule,
  MessageCreationListener, MessageListener, FilterInterceptorMBean
{
    …

    @Override
    public void load(LoadableModule parent, String typeName) {
        Message.addCreationListener(this);
        …
    }

    @Override
    public void unload() {
        Message.removeCreationListener(this);
        …
    }

    @Override
    public void messageCreated(Message msg, Object context) {
        msg.addMessageListener(this);
```

```
    }

    …
}
```

The message creation listener is added when the loadable module is loaded, and removed when it is unloaded. In this example, it adds a message listener when a message is created.

Create a message listener

This section describes the `MessageListener` interface, and provides an example of a message listener class that implements the interface. The sample code can be found in the `DEVELOPER_SAMPLES/FilterInterceptorLoadableModule` directory.

MessageListener interface

The `MessageListener` interface provides a set of callbacks that are invoked during the processing of a message as it passes through the processing engine of the API Gateway. The `MessageListener` interface provides callbacks which are invoked at certain points in the processing, for example just before a policy (circuit) is run, or before and after a message is processed by a filter. A message listener can be used to track the changes in a message as it flows through API Gateway, or to monitor the status of policies or filters as messages pass through them. Commonly it is used to gather statistics on message processing, which can then be used to give an indication of the status of API Gateway.

The `MessageListener` interface defines several methods that are invoked in conjunction with the methods or lifecycle events of the message. These include:

- Policy processing
    - `preCircuitProcessing` – Called when the message originator has completed initializing the message, and API Gateway is about to start processing in the policy-space.
    - `postCircuitProcessing` – Called when all processing in the policy-space is completed.
- Policy invocation
    - `preCircuitInvocation` – Called before the first filter in a given policy is invoked.
    - `postCircuitInvocation` – Called after a chain of filters in the policy has been invoked.

- Filter invocation
    - `preFilterInvocation` – This method is called immediately before a filter's `MessageProcessor` is invoked.
    - `postFilterInvocation` – This method is called when a filter's `MessageProcessor` has finished execution.
- `abortedCircuitInvocation` – Called if the policy exits because of a fault with one of the filters within it.
- `preFaultHandlerInvocation` – Called before attempting to handle a previous `CircuitAbortException` with specific fault-handling.
- `onMessageCompletion` – Called when a message has fully exited the system.

MessageListener example - FilterInterceptor

This section describes how to create a message listener using a simple example. The `FilterInterceptor` class counts the number of messages which pass, fail, or abort during processing of a request in the API Gateway.

There are two parts to building this example:

1. *Create the TypeDoc definition for the message listener* on page 50
2. *Create the message listener implementation class* on page 51

Create the TypeDoc definition for the message listener

A TypeDoc is an XML file that contains entity type definitions. Entity type definitions describe the format of data associated with a configurable item. For more details on entity types see *Entity types* on page 61.

In this example, the `FilterInterceptorLoadableModule` extends `NamedLoadableModule`. For more information on loadable module TypeDoc definitions, see *Create the TypeDoc definition for the loadable module* on page 46.

The following definition lists the various fields that form the configuration data for the `FilterInterceptorLoadableModule` class and declares an instance of the type.

```
<entityStoreData>
  <entityType name="FilterInterceptorLoadableModule" extends="NamedLoadableModule">
    <constant name="class" type="string"
      value="com.vordel.interceptor.FilterInterceptor"/>
    <constant name="loadorder" type="integer" value="1000000"/>
  </entityType>
</entityStoreData>
```

```
<entityStoreData>
```

```
  <entity type="FilterInterceptorLoadableModule">
    <fval name="name">
      <value>Filter Invocation Callback Listener</value>
    </fval>
  </entity>
</entityStoreData>
```

```
<typeSet>
  <typedoc file="FilterInterceptorLoadableModule.xml"/>
  <typedoc file="instance.xml"/>
</typeSet>
```

To add the `FilterInterceptorLoadableModule` type to the primary entity store, you can use the `publish.py` script. For example:

```
> cd INSTALL_DIR/apigateway/samples/scripts
> ./run.sh publish/publish.py
-i DEVELOPER_SAMPLES/FilterInterceptorLoadableModule/conf/typedoc/typeSet.xml
-t FilterInterceptorLoadableModule
-g "QuickStart Group" -n "QuickStart Server"
```

Alternatively, you can use the ES Explorer to add the type. For more information, see *Use the ES Explorer* on page 63.

Create the message listener implementation class

The API Gateway server-side implementation class is responsible for monitoring message creation and lifecycle events. On each lifecycle event it writes messages to the trace output.

The following is an extract of the `FilterInterceptor` class that can be found in the `DEVELOPER_SAMPLES/FilterInterceptorLoadableModule/src` directory.

```
public class FilterInterceptor implements LoadableModule,
  MessageCreationListener, MessageListener, FilterInterceptorMBean
{
    …
    @Override
    public void load(LoadableModule parent, String typeName) {
        Message.addCreationListener(this);
        …
    }

    @Override
    public void unload() {
        Message.removeCreationListener(this);
```

```
        …
    }

    @Override
    public void messageCreated(Message msg, Object context) {
        msg.addMessageListener(this);
    }
    …

    @Override
    public void preCircuitInvocation(Circuit circuit, Message message,
      Object context)
    {
        Trace.info("Circuit ["+circuit.getName()+"] about to invoke message: "
          +message.correlationId+", caller context is: "+context);
    }

    @Override
    public void postCircuitInvocation(Circuit circuit, Message message,
      boolean result, Object obj)
    {
        Trace.info("Circuit ["+circuit.getName()+"] has finished with message: "
          +message.correlationId+", result is :"+(result?"PASSED": "FAILED"));
    }
    …

    @Override
    public void preFilterInvocation(Circuit circuit, MessageProcessor processor,
      Message message, MessageProcessor caller, Object obj)
    {
        Filter f = processor.getFilter();
        String type = f.getEntity().getType().getName();
        Trace.info("["+f.getName()+"("+type+")] msg:"+message.correlationId);
    }

    @Override
    public void postFilterInvocation(Circuit circuit, MessageProcessor processor,
      Message message, int resultType, MessageProcessor caller, Object obj)
    {
        Trace.info("["+processor.getFilter().getName()+"] msg:"
          +message.correlationId+" Result: "+toString(resultType));
    }

    @Override
    public void onMessageCompletion(Message message) {
        Trace.info("Message ["+message.correlationId+"] completed.");
    }
    …
}
```

A `MessageListener` is registered with a message instance by first listening for a message creation event via the `addCreationListener(MessageCreationListener)` method, which is called during the loading of the `FilterInterceptor`, and then calling the `addMessageListener(MessageListener)` method on the message parameter. The message creation listener is unregistered during the unloading of the `FilterInterceptor`.

The `onMessageCompletion` method monitors the completion of a message in a policy so that resources can be cleaned up when the message is no longer useful. The preprocessing and postprocessing interceptor methods output trace information.

The following is an example of the trace output from the `FilterInterceptor` during the execution of a filter:

```
INFO 26/Feb/2013:11:26:12.064 [1698] Circuit [Send Instant Message] about to invoke
  message: 8a8ef28f512c9bce01980000, caller context
  is: com.vordel.dwe.http.HTTPPlugin@2b3fab
INFO 26/Feb/2013:11:26:14.017 [1698] [Jabber(JabberFilter)]
  msg:8a8ef28f512c9bce01980000
INFO 26/Feb/2013:11:26:16.658 [1698] [Jabber]
  msg:8a8ef28f512c9bce01980000 Result: SUCCESS
INFO 26/Feb/2013:11:26:20.799 [1698] Circuit [Send Instant Message] has finished
  with message: 8a8ef28f512c9bce01980000, result is :PASSED
```

Note To see the above output, you must build the FilterInterceptorLoadableModule sample and add the resulting `interceptor.jar` to the API Gateway CLASSPATH. See the `README.TXT` for more information on building the sample.

The following is a sample style sheet that can be used with the `removeType` script in the API Gateway to remove the `FilterInterceptorLoadableModule` and its instances from the primary entity store.

```
<?xml version="1.0" ?>
<stylesheet xmlns="http://www.w3.org/1999/XSL/Transform"
  version="1.0"
  xmlns:es="http://www.vordel.com/2005/06/24/entityStore">
    <template match="comment()|processing-instruction()">
      <copy />
    </template>
    <template match="@*|node()">
      <copy>
      <apply-templates select="@*|node()" />
      </copy>
    </template>
    <!-- Removing type and instances -->
    <template match=
      "/es:entityStoreData/es:entityType[@name='FilterInterceptorLoadableModule']"/>
    <template match=
      "/es:entityStoreData/es:entity[@type='FilterInterceptorLoadableModule']"/>
</stylesheet>
```

You can remove the type from the primary store by running the following command:

```
> cd INSTALL_DIR/apigateway/samples/scripts
> ./run.sh unpublish/unpublish.py
-i DEVELOPER_SAMPLES/FilterInterceptorLoadableModule/conf/remove.xslt
-t FilterInterceptorLoadableModule
-g "QuickStart Group" -n "QuickStart Server"
```

You can use the ES Explorer tool to view new types that were added, or to verify that types were removed. For more information, see *Use the ES Explorer* on page 63.

Access configuration values dynamically at runtime 7

You can access configuration values dynamically at runtime using *selectors*. A selector is a special syntax that enables API Gateway configuration settings to be evaluated and expanded at runtime, based on metadata values (for example, from message attributes, a Key Property Store (KPS), or environment variables).

For example, when a HTTP request is received, it is converted into a set of message attributes. Each message attribute represents a specific characteristic of the HTTP request, such as the HTTP headers, HTTP body, and so on.

For more information on using selectors, see the *API Gateway Policy Developer Guide*.

Example selector expressions

The *API Gateway Policy Developer Guide* includes some examples of selector expressions. The following table lists some more complex examples.

Selector expression	Result
`${kps.matrix.row.column}` `${kps.matrix["row"]["column"]}`	For a KPS with multiple read keys, the values for each key are provided in order. The result of the expression is also indexable: `set property test = ${kps.matrix.row}` `${test["column"]}` looks up the KPS for `[row/column]`.
`${content.body.getParameters().get("grant_type")}`	Gets the HTTP form post field called `grant_type`.

Selector expression	Result
`${content.body.getJSON().get('access_token').getTextValue()}`	If a body is of type `application/json` then it is automatically treated as a `com.vordel.mime.JSONBody`. A `JSONBody` object returns an `com.fasterxml.jackson.databind.JsonNode` object via a `getJSON()` call. For more information, see the Javadoc for JsonNode class. For example, if the body contains the following JSON content: `{ "access_token":"2YotnFZFEj", "token_type":"example", "expires_in":3600 }` this selector results in the value `2YotnFZFEj`.
`${content.body.getJSON().toString()}`	If a body is of type `com.vordel.mime.JSONBody`, this selector converts the JSON contained in the body to a string value.
`${environment.VINSTDIR}`	Accesses the environment variable `VINSTDIR`.

`${http.path[2]}`	If you have the filter **Extract REST Request Attributes** in your policy, this filter adds the incoming URI to the message whiteboard as a String array, so that you can index into the path. If the incoming path is `/thisisa/test`, using this type of selector results in the following attributes on the whiteboard:
	`${http.path[1]} = thisisa`
	`${http.path[2]} = test`

Database query results

You can use the **Retrieve from or write to database** filter to retrieve user attributes from a database or write user attributes to a database. You can select whether to place database query

results in message attributes on the **Advanced** tab of the filter. By default, the **Place query results into user attribute list** option is selected. For more information on the **Retrieve from or write to database** filter, see the *API Gateway Policy Developer Guide*.

The query results are represented as a list of properties. Each element in the list represents a query result row returned from the database. These properties represent pairs of attribute names and values for each column in the row. The **Prefix for message attribute** field in the filter is required to name the list of returned properties (for example, `user`).

Results in user attribute list

The following table shows some example selectors when the option **Place query results into user attribute list** is selected.

Selector expression	Result
`${user[0].NAME}`	`John`
`${user[0].LASTNAME}`	`Kennedy`
`${user[1].NAME}`	`Brian`
`${user[1].LASTNAME}`	`O'Connor`

You can also use standard Java function calls on the attributes. For example:

- `${user.size()}` – Number of properties (number of rows) retrieved from the database
- `${user[0].NAME.equals("John")}` – Returns true if the `NAME` attribute (value of column `NAME` in first row) is `"John"`

For more information, see the `java.util.ArrayList` and `java.lang.String` class interfaces.

Results not in user attribute list

The following table shows some example selectors when the option **Place query results into user attribute list** is not selected.

Selector Expression	Result
`${user.NAME[0]}`	`John`
`${user.LASTNAME[0]}`	`Kennedy`
`${user.NAME[1]}`	`Brian`
`${user.LASTNAME[1]}`	`O'Connor`

You can also use standard Java function calls on the attributes. For example:

- `${user.NAME.size()}` – Number of `NAME` attributes (number of rows with column `NAME`) retrieved from the database
- `${user.NAME[0].equals("John")}` – Returns true if the first `NAME` attribute (value of column `NAME` in first row) is `"John"`

For more information, see the `java.util.ArrayList` and `java.lang.String` class interfaces.

LDAP directory server search results

You can use the **Retrieve from directory server** filter to retrieve user profile data. For more information on the **Retrieve from directory server** filter, see the *API Gateway Policy Developer Guide*.

The 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. The **Prefix for message attribute field** in the filter is required to name the list of search results (for example, `user`).

The following table shows some example selectors:

Selector Expression	Result
`${user[0].memberOf[0]}`	`CN=Operator,OU=Sales`
`${user[0].memberOf[1]}`	`CN=Developer,OU=Dev`
`${user[0].memberOf[2]}`	`CN=Operator,OU=Support`
`${user[1].memberOf[0]}`	`CN=Operator,OU=Sales`

You can also use standard Java function calls on the attributes. For example:

- `${user.size()}` – Number of search results returned by the LDAP directory server
- `${user[0].memberOf.size()}` – Number of `memberOf` attribute values returned in the search result
- `${user[0].memberOf.contains("CN=Operator,OU=Sales")}` – Returns true if one of the returned `memberOf` attributes is `"CN=Operators,OU=Sales"`
- `${user[0].memberOf[0].equals("CN=Operator,OU=Sales")}` – Returns true if the first `memberOf` attribute is `"CN=Operators,OU=Sales"`

For more information, see `java.util.ArrayList` and `java.lang.String` class interfaces.

Key Property Store 8

A Key Property Store (KPS) is an external data store of API Gateway policy properties, which is typically read frequently, and seldom written to. Using a KPS enables metadata-driven policies, whereby policy configuration is stored in an external data store, and looked up dynamically when policies are executed.

For more information on KPS, see the *API Gateway Policy Developer Guide* and also the *API Gateway Key Property Store User Guide*.

Entity Store 9

Configuration data for API Gateway is stored in the Entity Store (ES). The Entity Store is an XML-based store that holds all configuration data required to run API Gateway. It contains policies, filters, certificates, resources, and so on. The Entity Store for a group of API Gateways can be found at the following location:

```
INSTALL_DIR/apigateway/groups/GROUP_ID/conf/CONFIG_UID
```

API Gateway runs with a number of separate stores that are combined as a federated entity store for the API Gateway's configuration.

The federated entity store is made up of component stores. Each component store is responsible for one or more *branch points* in the configuration tree. Each component store must be consistent in its own right:

- It must have all the entity types (see *Entity types* on page 61) required to describe its component entities
- It must also have valid hard references within. Hard references are fields that refer to other entities via their real primary keys (PKs). Soft references allow an entity in one store to reference an entity in another store. This is done via Portable ESPKs, which the federated entity store has the added ability, above other store flavors, to resolve to the correct entity when calling `getEntity(ESPK pk)`.

The following table lists the XML-based stores:

File name	Description
`CertStore.xml`	Contains certificates and private keys. Private keys are encrypted with the API Gateway passphrase.
`configs.xml`	The federated store which imports all other component stores.
`EnvSettingsStore.xml`	Contains environment settings for the configuration.
`ExtConnsStore.xml`	Contains the external connection information (for example, database, LDAP, and so on) that the runtime might connect to.
`ListenersStore.xml`	Contains the configuration for the HTTP ports and protocols that API Gateway listens on to receive messages to be processed.
`PrimaryStore.xml`	Contains the policies and filters to be applied to messages received by API Gateway.

File name	Description
`UserStore.xml`	User store containing user names and passwords and associated user roles.
`ResourcesRepository.xml`	Contains the resource information.

Entity types

An *entity type* is a description of an entity in the Entity Store. An entity type defines the following properties for a given entity:

- Constant fields, specifically name and value.
- Fields, specifically name, data type, and cardinality.

The entity type is known by its name, and is located in its inheritance tree by its parent type. All entity types are defined within the top-level `<entityStoreData>` element.

The following are additional properties of entity types:

- Each entity is an instance of an `EntityType`.
- Each entity is of exactly one entity type, and has a link to its defining type.
- Entity types can inherit properties from zero or one other entity types.
- The root `EntityType` is called `Entity`, so a basic entity is of type `Entity`.

A field in an entity can be of the following type:

- boolean
- string
- integer
- long
- utctime
- binary
- encrypted
- reference to another entity (either in the same component entity store or across a component store in the federated store)

A field must be assigned a cardinality. The possible cardinalities are:

- zero or one (?)
- zero or many (*)
- one or many (+)
- one (1)

The following shows an example.

```
<field name="name" type="string" cardinality="1" isKey="true" />
<field name="isEnabled" type="boolean" cardinality="1" default="true"/>
<field name="hosts" type="string" cardinality="*"/>
<field name="timeout" type="integer" cardinality="?" default="30000" />
<field name="certificates" type="^Certificate" cardinality="*" />
<field name="nextEntity" type="@Entity"/>
```

References to other entities

A field that refers to another entity within the same component entity store is called a *hard reference*. For example:

```
<field cardinality="1" name="category" type="@Category">
```

A field that refers to another entity in another component entity store is called a *soft reference*. For example:

```
<field cardinality="1" name="repository" type="^AuthnRepositoryBase"/>
```

Entity type definitions

Each configurable item has an entity type definition. The entity type definition is defined in an XML file known as the *TypeDoc*.

Entity types are analogous to class definitions in an object-oriented programming language. In the same way that instances of a class can be created in the form of objects, an instance of an entity type can also be created. Therefore it is useful to think of the entity type defined in a TypeDoc as a header file, and the entity itself as a class instance. All entities and their entity type definitions are stored in the Entity Store.

Every filter requires specific configuration data to perform its processing on the message. For example, a filter that extracts the values of two elements from a SOAP message, and adds them together, must be primed with the names and namespaces of those two elements.

Because a filter is a configurable item, it requires a new XML TypeDoc to be written containing an entity type definition for it. The entity type for a filter contains a set of configuration parameters and their associated data types and default values.

When an instance of the filter is added to a policy using Policy Studio, a corresponding entity instance is created and stored in the Entity Store. Whenever the filter instance is invoked, its configuration data is read from the entity instance in the Entity Store.

Use the ES Explorer

ES Explorer is a registry editor type tool which allows you to connect directly to an Entity Store (ES). Within the ES explorer you can perform various create, read, update, delete (CRUD) operations on the entities, and view the available types in the Entity Store. To open a federated entity store in ES Explorer, load the `configs.xml` file. Alternatively, to open a component entity store, load the XML file (for example, `PrimaryStore.xml`).

To start the ES Explorer, run this command:

```
INSTALL_DIR/apigateway/posix/bin/esexplorer
```

When ES Explorer is started you can load an entity store by selecting **Store > Connect** from the menu. In the **Connect to EntityStore** dialog browse to either the federated or component entity store to be opened.

Load a type definition

You can import a type definition or entity instances using ES Explorer by using a catalog called a `typeset`. The typeset is essentially a list of references to other files, each of which can contain type definitions or previously exported entity definitions. While you can import entity instances into the federated store, you must be careful when adding types to ensure that they get added to the appropriate component store, as each component store is responsible for only a subset of types visible from the federated view.

For example, if your TypeDoc is describing an entity type that defines a custom filter then you should add this to the Primary Store (`PrimaryStore.xml`), which is responsible for storing policies and filters. If your TypeDoc describes a listener for messages then you would add it to the Listener Store (`ListenersStore.xml`).

If you have followed the preceding steps to connect to a component store, you can add a typeset by right-clicking on the component store icon and selecting **Load a Type Set**. In the dialog you can specify the location of the `typeset.xml` file to add. This adds all types referenced by the typeset into the component store.

For an example of how to build up a typeset and have it indirect to a type definition for import, see the sample script `publish.py`, which is available in the `INSTALL_DIR/apigateway/samples/scripts/publish` directory. By default the `typeset.xml` file includes a `SimpleFilter.xml` file which contains a **SimpleFilter** entity type definition.

Locate entities using shorthand keys

Entities in the Entity Store can be located using shorthand keys. In the ES Explorer, you can right-click on a node and select **Print PortableESPK (shorthand)** to print out the shorthand key for it. For example, the following is the shorthand key for the **Set Message** filter in the Health Check policy:

```
/[CircuitContainer]name=Policy Library/[FilterCircuit]name=Health Check/
[ChangeMessageFilter]name=Set Message
```

You can then use the **Find an Entity** action to use the key to find that entity, or you can fashion a more general search string to find entities at a given level.

The following table shows some examples of shorthand keys:

Shorthand key	Description
`/[CircuitContainer]**/[FilterCircuit]`	Get all filter circuits at all levels.
`/[CircuitContainer]**/[FilterCircuit]/[Reflector]`	Get all filters of type Reflector.
`/[CircuitContainer]**/[FilterCircuit]/[Reflector]name=Reflect`	Get all filters of type Reflector, but restricted to Reflector filters with the name `Reflect`.
`/[CircuitContainer]**/[FilterCircuit]` `/[FilterCircuit]`	Get all policies in a configuration (you need two shorthand keys).

Note Forward slashes must be escaped, for example:

```
/[NetService]name=Service/[HTTP]name=Management Services/
[ServletApplication]uriprefix=\/manager\//[Servlet]uri=webManager
```

Debug custom Java code with a Java debugger

10

You can debug custom Java code running in API Gateway (for example, code for a custom filter), by attaching a remote debugger to API Gateway. To attach a remote debugger, add a JVM argument to API Gateway and restart it.

To change the JVM settings of an API Gateway instance, follow these steps:

1. Create a file called `jvm.xml` in the following location:

```
INSTALL_DIR/apigateway/groups/GROUP_ID/INSTANCE_ID/conf
```

2. Edit the `jvm.xml` file so that the contents are as follows:

```
<ConfigurationFragment>
  <VMArg name="-Xrunjdwp:transport=dt_socket,server=y,address=9999" />
</ConfigurationFragment>
```

3. Restart API Gateway.

When you restart the API Gateway instance with the above settings, it starts up and waits for a JVM debugger to connect to the process on port 9999. You can connect to port 9999 of the API Gateway instance using a Remote Java debug (for example, in the Eclipse IDE).

Get diagnostics output from a custom filter 11

You can configure API Gateway to output detailed diagnostic information for a specific custom filter by setting the trace level to DEBUG or DATA.

To change the trace level in Policy Studio, select the **Server Settings** node, and click **General**. Select DEBUG or DATA from the **Tracing level** field, and click **Save**.

For more information on tracing and logging see "Configure API Gateway diagnostic trace" in the *API Gateway Administrator Guide*

Add custom trace output to custom code

Note Some code extracts in this section are from the `jabber` sample that is no longer included in the code samples supplied with API Gateway.

To add custom trace information to custom code, you can add `Trace` statements within your code.

For example, the following code adds `Trace` statements to output the thread ID associated with the chat, which corresponds to the thread field of the SMACK XMPP message to a custom Jabber filter (see *Write a custom filter using the extension kit* on page 20).

```
…
import com.vordel.trace.Trace;
…

public class JabberProcessor extends MessageProcessor {
    …
    public boolean invoke(Circuit c, Message message)
      throws CircuitAbortException {
        …
        try {
            Trace.debug("Chat Thread ID is " + chat.getThreadID());
            chat.sendMessage(messageStr.substitute(message));
        } catch (org.jivesoftware.smack.XMPPException ex) {
            Trace.error("Error Delivering block");
        }
        …
    }
    …
}
```

The Chat Thread ID is output in the API Gateway trace file as follows:

```
DEBUG   10/Jun/2013:11:18:21.365 [01f4] run filter [Jabber] {
DEBUG   10/Jun/2013:11:18:22.880 [01f4] Chat Thread ID is VSx1B0
DEBUG   10/Jun/2013:11:18:23.037 [01f4] } = 1, filter [Jabber]
DEBUG   10/Jun/2013:11:18:23.037 [01f4] Filter [Jabber] completes in 672
milliseconds.
```

The trace level DATA can be used to provide more detailed information. To use the DATA level in the preceding example, change the `Trace.debug` statements to `Trace.data` statements.

Add custom log4j output to custom code

To output custom log4j information perform the following steps:

1. Update the `log4j2.xml` file, located in the `INSTALL_DIR/apigateway/system/conf` directory, to specify that the log4j appender sends output to the API Gateway trace file. For example:

   ```
   <Root level="debug">
   <AppenderRef ref="STDOUT" />
   <AppenderRef ref="VordelTrace" />
   </Root>
   ```

2. Add log4j statements to your code. Log4j is already on the API Gateway CLASSPATH. The following example shows the preceding code with log4j statements instead of Trace statements:

   ```
   …
   import org.apache.log4j.Logger;
   …
   public class JabberProcessor extends MessageProcessor {
       private static final Logger log = LogManager.getLogger
   (JabberProcessor.class.getName());
       …
       public boolean invoke(Circuit c, Message message)
         throws CircuitAbortException {
           …
           try {
               log.debug("Chat Thread ID is " + chat.getThreadID());
               chat.sendMessage(messageStr.substitute(message));
           } catch (org.jivesoftware.smack.XMPPException ex) {
               Trace.error("Error Delivering block");
           }
           …
       }
       …
   }
   ```

The following is output in the API Gateway trace file:

```
<Date> <Time> [Thread-xx] DEBUG com.vordel.jabber.filter.JabberProcessor - Chat
Thread ID is xxxxxx
```

Enable API Gateway with JMX

12

Java Management Extensions (JMX) allows remote clients to connect to a JVM and manage or monitor running applications in that JVM. MBeans are the controllable endpoints of your application where remote clients can observe application activity as well as control their inner workings.

For more information on implementing an MBean interface, see the `FilterInterceptor` example. The sample code can be found in the `DEVELOPER_SAMPLES/FilterInterceptorLoadableModule` directory. The `FilterInterceptor` class implements the `FilterInterceptorMBean` interface.

After you have set up your MBean, you need to tell the JMX infrastructure about the MBean so that it can be published to clients. This involves creating a unique name for the MBean and then registering it with the MBeanServer. For example:

```
…
try {
    MBeanServer mbs = ManagementFactory.getPlatformMBeanServer();
    // Construct the ObjectName for the MBean we will register
    ObjectName name = new ObjectName(
      "example:type=FilterInterceptorMBean");
    mbs.registerMBean(this, name);
} catch (Throwable t) {
    Trace.error(t);
}
…
```

After you have set up your MBeans and registered them with the MBeanServer you can view them in the management console that your JMX container supports (for example, JConsole). To use JConsole, add the following JVM argument to API Gateway and restart it.

Follow these steps:

1. Create a file called `jvm.xml` in the following location:

   ```
   INSTALL_DIR/apigateway/groups/GROUP_ID/INSTANCE_ID/conf
   ```

2. Edit the `jvm.xml` file so that the contents are as follows:

   ```
   <ConfigurationFragment>
     <VMArg name="-Dcom.sun.management.jmxremote"/>
   </ConfigurationFragment>
   ```

3. Restart API Gateway.

When you restart the API Gateway instance with the above settings you can connect using Java Monitoring and Management Console (`JConsole`) and view your MBeans. Launch `jconsole` (an executable in the `bin` directory of your Java JDK installation) and select the **MBeans** tab to see the **FilterInterceptorMBean** listed on the left. You can see the message total, success, failure, and abort counts.

Note In this case, the attributes are read-only, but in other cases they might be modifiable and you can change their settings.

Automate tasks with Jython scripts 13

API Gateway contains several sample scripts that let you automate various common administration tasks. The scripts are based on the Java scripting interpreter, Jython (http://www.jython.org). Scripts can be extended to suit your needs by following the Jython language syntax. All Jython scripts can be found in the following location:

```
INSTALL_DIR/apigateway/samples/scripts
```

To run a sample script, you can call the `run` shell in this directory and point it to the script to be run. For example, to run the `changeTrace.py` sample script:

```
> cd INSTALL_DIR/apigateway/samples/scripts
> ./run.sh config/changeTrace.py
```

The following table summarizes the Jython scripts that are available in the API Gateway installation.

Script category	Description
analyze	• Prints a list of all references in the API Gateway. For each reference it shows what store it originates from and to. • Checks if the API Gateway is locked down. • Gets a list of unresolved references between entities in the API Gateway.
apikeys	• Shows how to send an API key and associated secret as query parameters. • Shows how to send an API key only as query parameters. • Shows how to fetch a URL with HTTP basic authentication. • Shows how to send a signed query string. • Shows how to send an authenticated REST request.
cassandra	• Converts all Cassandra-based data stores to file-based data stores and redeploys.
certs	• Adds a new certificate to the certificate store from different sources.

Script category	Description
config	• Removes the sample service listeners and policies from the API Gateway. • Connects to a particular process and toggles tracing for the management port. • Connects to the API Gateway and sets an address to bind to on a given port of a given interface. • Updates the `maxInputLen`, `maxOuputLen`, and `maxRequestMemory` configuration in Node Managers and API Gateways.
environmentalize	• Marks fields for environmentalization and creates associated environment setting entries. • Gets the fields that have been marked for environmentalizing and outputs the associated value in environment settings. • Removes fields marked as environmentalized. • Creates a deployment archive from a policy package and an environment package.
io	• Imports and exports entities to and from an API Gateway configuration.
json	• Generates a JSON schema for a fully qualified class name.
migrate	• Downloads the current `.fed`, `.pol` and `.env` archives via Node Manager from an API Gateway. • Creates a deployment package from policy and environment packages. They can be obtained from a running API Gateway, from a source code repository (for example, Git or SVN), or via USB or FTP and so on. • Demonstrates a promotion of configuration from the development environment to the staging environment.
monitor	• Prints the filter details of each policy executed in a transaction. • Prints the success or failure status of each filter in a transaction.
oauth	• Provides OAuth 2.0 support.
passport	• Creates an Axway PassPort CSD based on the API service configuration. An Axway Component Security Descriptor (CSD) file is used when registering with Axway PassPort.

Script category	Description
publish	• Adds an entity type and any defined instances to an associated entity store.
unpublish	• Removes an entity type and any defined instances from an associated entity store.
topology	• Creates API Gateway instances in a group. • Gets the domain topology from the Admin Node Manager. • Gets the domain topology and outputs the IDs of the API Gateway instances.
securityconstraints	• Checks a configuration for FIPS, SuiteB, or SuiteBTS compliance.
users	• Connects to a particular process and adds a new user to the user store.
ws	• Registers a WSDL in an API Gateway. • Lists the web services in an API Gateway. • Removes a registered service from the API Gateway.

Java and Jython translations

The following table shows examples of translating Java to Jython.

Java	Jython equivalent
`import java.lang.Math;`	`from java.lang import Math`
Java does not support multiple inheritance.	Jython supports multiple inheritance.

Java	Jython equivalent

Class declaration:
`public class Hello {`
`  private String name = "John Doe";`
 	
`  Hello(String name) {`
`   this.name = name;`
`  }`
 	
`  public void greeting() {`
`    System.out.println("Hello" + name);`
`  }`
`}`
	Class declaration:
	`class Hello:`
	`  def __init__(self, name="John Doe"):`
	`    self.name = name`
	
	`  def greeting(self):`
	`    print "Hello, %s" % self.name`
	
	The `__init__` method is the constructor, and you can pass it default arguments for any arguments the user does not supply, as with the name argument. You can supply named parameters when you instantiate a class or call a method.
	The `self` term is a reference to the current instance of the class. It is the equivalent of the Java `this` reference. Calling it `self` is a Jython convention. You can use any name. Unlike Java, Jython requires you to prefix any instance variable with `self`. Failing to provide the `self` prefix usually results in an exception. When creating a bound method, the first parameter to the method is always `self`.

Variables can be public, protected, or private.	In Jython, variables and class methods can either be public or private. There is no protected level of visibility as there is in Java. They are public by default. To make them private, you can prefix them with a double underscore. For example, the following variable is private: `self.__name = name`
Variables have to be declared before use.	Variables do not have to be declared before use, nor do their types need to be declared. (They do, however, have to be assigned before use.) This is true both for local variables within a block, and for the data members of classes.

Java	Jython equivalent
Method signature: `public int getID(String name)`	Method signature: `def getID(name)` In Jython, types are not explicitly declared.

`for (Iterator i; i.hasNext();) {`
`  i.next();`
`}`
	`for i in list:`
	The `for` statement automatically iterates over Java Lists, Arrays, Iterators, and Enumerators.
	The `for` statement requires a sequence. It has the range() and xrange() functions to make sequences for the `for` statement:
	`for x in range(10, 0, -1):`

`if (x >= 0)`
`  System.out.println("x is valid for a factorial");`
`else`
`  System.out.println("x is invalid for a factorial");`
	The `if` statement also has the `if-elif-else` statement. The `else` is optional.
	`if x >= 0:`
	`  print x, "is valid for a factorial"`
	`else:`
	`  print x, "is invalid for a factorial"`
	or
	`if x == 0:`
	`  doThis()`
	`elif x == 1:`
	`  doThat()`
	`elif x == 2:`
	`  doTheOtherThing()`
	`else:`
	`  print x, "is invalid"`

`while( x < 20 ) {`
`  System.out.print("x : " + x );`
`  x++;`
`}`
	`while x<20:`
	`  print x`
	`  x = x+1`
	Jython has an additional `else` clause that is optional and is executed when the condition evaluates to `false`.

API Gateway REST APIs 14

API Gateway exposes some services as REST APIs. These APIs provide access and basic create, read, update, and delete (CRUD) operations for the service resources. API Gateway contains a Jersey Servlet (http://jersey.java.net/) that scans a predefined list of packages to identify RESTful resources to be exposed over HTTP or HTTPS.

Jersey REST services are exposed on the internal management HTTPS listener that is running on every API Gateway. This HTTPS listener is not accessible to the outside world and only accepts traffic over two-way SSL from the local Node Manager. Therefore, to call any REST service exposed on the management interface, you must call it via the Admin Node Manager using the Routing API.

The API Gateway REST APIs are available from the following locations:

- `INSTALL_DIR/apigateway/samples/swagger`
- Product APIs page on the Axway Documentation portal

  Note When viewing REST APIs on the Axway Documentation portal, the `consumes` field is not displayed if you are using `formData` type parameters in an API, due to limitations in the Swagger UI version.

API Gateway component REST APIs

The following table summarizes the API Gateway component REST APIs that are available:

API	Description
Router	The Router REST API is available in the Node Manager. It acts as a relay that forwards requests to the appropriate API Gateway registered with the Node Manager.
Management	The Management REST API is available in the Node Manager and all API Gateways. It provides the ability to retrieve the following API Gateway information: API Gateway name, group name, service type, product version, and the domain ID assigned to the Admin Node Manager on creation. This API can also be used to update the service name and group name.
Deployment	The Deployment REST API is available in the Node Manager. It provides the ability to manage deployment archives for API Gateways.
Configuration	The Configuration REST API is available in the API Gateway. It provides the ability to upload configurations to API Gateway Admin Node Manager instances. It is used in conjunction with the Deployment API.

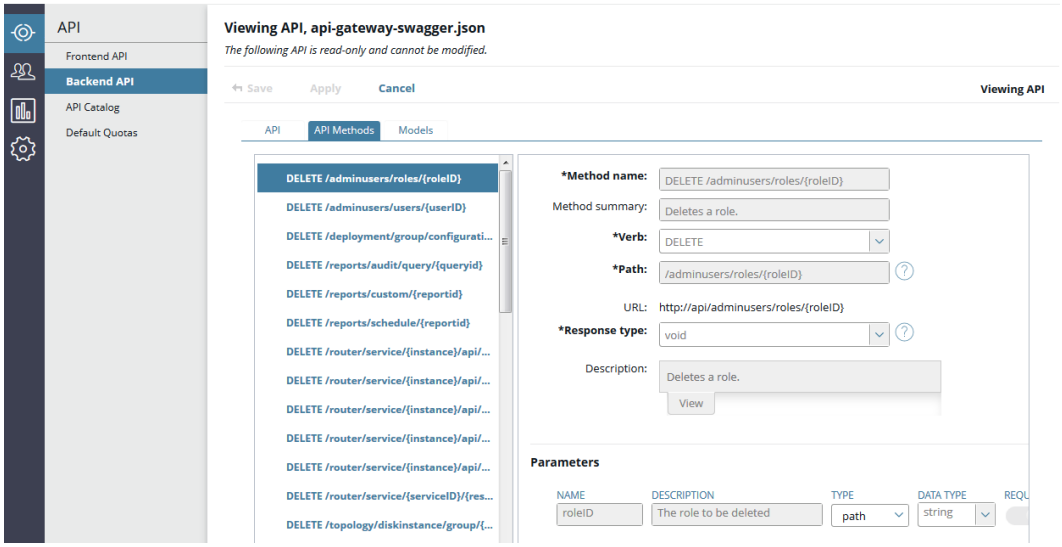
API	Description
API Manager	The API Manager REST API is available in the API Manager. It provides the ability to view and update the configured users, organizations, applications and events related to the API Manager.
Admin Users	The Admin Users REST API is available in the Node Manager. It provides the ability to manage administrator users and roles for the API Gateway installation.
Topology	The Topology REST API is available in the Node Manager. It provides the ability to manage hosts, groups, and services in the topology.
Traffic Monitor	The Traffic Monitor REST API is available in the API Gateway. It provides the ability to monitor traffic in and out of the API Gateway.
Service Manager	The Service Manager REST API is available in the Node Manager. It provides the ability to manage virtualized REST APIs configured in the topology.
Analytics	The Analytics REST API is available in API Gateway Analytics. It provides read-only access to the database audit log and audit message/payload details, metrics for charting, and CRUD for custom reporting.
RBAC	The RBAC (Role Based Access Control) REST API is available in the Node Manager. It ensures that only users with the assigned role can access parts of the management services exposed by the Admin Node Manager.
Monitoring	The Monitoring REST API is available in the API Gateway. It provides access to process summary details and listings of the real-time metrics for items that metrics are recorded for (for example, web services, external APIs, authenticated clients, external target servers, and so on).
KPS	The KPS REST API is exposed by the API Gateway and the Node Manager. The API Gateway interface provides a persistence mechanism. The Node Manager service provides administration functions.
Domain Audit	The Domain Audit REST API is available in the Node Manager and API Gateway. It provides the ability to read domain audit events recorded by the Node Manager and API Gateway.
Embedded Active MQ	The Embedded Active MQ REST API is exposed by the API Gateway.

Import the API Gateway REST API into API Manager

You can import the API Gateway REST API Swagger 2.0 definitions into API Manager in the same way that you import any other APIs. For example:

1. Click the **API Registration** > **Backend API** view in API Manager.
2. Click **New API** and select **Import Swagger API**.
3. In the **Import API** dialog, complete the following:
    - **Source**: Select Swagger definition file.
    - **File** or **URL**: Click the browse button to select the definition file. For example:
    - `INSTALL_DIR/apigateway/samples/swagger/api-gateway-swagger.json`
    - **API Name**: Enter a user-friendly name for the API. The default is **api-gateway-swagger.json**.
    - **Organization**: Select the organization from the list (for example, **API Development**).
4. Click **Import** to import the API Gateway API.

For more details, see the *API Manager User Guide*.



Add a Jersey-based REST API

Note The following sections refer to `restJabber` sample code that is no longer included in the code samples supplied with API Gateway. We recommend that you use this section only as a general guide for adding a Jersey-based REST API.

The following example shows how to add a Jersey REST service to your API Gateway and configure a corresponding servlet in Policy Studio. The REST service implements the SMACK API. The example also demonstrates how invoking a REST request sends an instant message to an account on Google Talk.

1. Annotate your Java class. The following example shows a code snippet of a Jersey-annotated Java class for the Smack API. The full class definition can be found in the `DEVELOPER_`

SAMPLES/restJabber directory. You must replace the username and password in the sample code with appropriate values.

```
@Path("/jabber")
public class RestJabberRequest {

    private static final String username = "yourEmailaddresst@here.com";
    private static final String password = "yourPassword";
    private static final String resource = "apiServer";
    XMPPConnection connection;

    // This method is called if TEXT_PLAIN is request
    @GET
    @Produces(MediaType.TEXT_PLAIN)
    @Path("{msg}/{to}")
    public String sendPlainMessage(@PathParam("msg") String msg,
            @PathParam("to") String to) {
        try {
            sendMessage(msg, to);
        } catch (XMPPException e) {
            System.err.println("Sending message failed:");
            e.printStackTrace();
        }
        return "Sent a message of : " + msg + " to " + to;
    }

    …

    private void sendMessage(String msg, String to) throws XMPPException {
        try {
            ConnectionConfiguration config =
              new ConnectionConfiguration("talk.google.com", 5222, "gmail.com");
            connection = new XMPPConnection(config);
            SASLAuthentication.supportSASLMechanism("PLAIN", 0);
            connection.connect();
            connection.login(username, password, resource);
            Chat chat =
              connection.getChatManager().createChat(to, new MessageListener(){
                @Override
                public void processMessage(Chat arg0, Message arg1) {
                    Trace.debug(arg1.getBody());
                }
            });
            chat.sendMessage(msg);
            connection.disconnect();
        } catch (org.jivesoftware.smack.XMPPException ex) {
            System.out.println("Exception throw");
        }
    }

}
```

2. Follow the instructions in the `README.TXT` in the sample directory to build the JAR file for the restJabber sample.
3. Add the new JAR and any third-party JAR files used by the restJabber classes (for example, the SMACK API JAR files) to the CLASSPATH for all API Gateways and Node Managers on a host by copying them to the `INSTALL_DIR/apigateway/ext/lib` directory.

   Alternatively, you can add the JARs to the CLASSPATH for a single API Gateway instance only, by copying them to the `INSTALL_DIR/apigateway/groups/GROUP_ID/INSTANCE_ID/ext/lib` directory.
4. Restart API Gateway. The REST Jabber service is now available.
5. Add your servlet application and servlet using Policy Studio or ES Explorer. See *Add a servlet using Policy Studio* on page 80.
6. Test the REST service. See *Test the REST Jabber service* on page 80.

Add a servlet using Policy Studio

To create a servlet using Policy Studio, perform the following steps:

1. Start Policy Studio, and connect to the API Gateway.
2. Select **Environment Configuration > Listeners > API Gateway > Default Services > Paths**.
3. Right-click **Paths** and select **Add Servlet Application**.
4. On the **General** tab, enter `/` in the **Relative Path** field.
5. Click **OK**.
6. Right-click the servlet application path you just created in the Paths window, and select **Add Servlet**.
7. Enter `smack` in the **Name** field.
8. Enter `smack` in the **URI** field.
9. Enter `org.glassfish.jersey.servlet.ServletContainer` in the **Class** field.
10. Click **Add** under the Servlet Properties table to add a new property with the following values:
    - Name: `jersey.config.server.provider.packages`
    - Value: `com.vordel.jabber.rest`
11. Click **OK**.
12. Click **Deploy** or press **F6** to deploy the new configuration on the API Gateway.

Test the REST Jabber service

To test the service, launch a web browser and enter the following URL:

```
http://localhost:8080/smack/jabber/{msg}/{to_email_address}
```

Replace `msg` and `to_email_address` in the URL with the message and the email address of the recipient.

Alternatively, you can execute the REST client that is included with the REST classes in the the `DEVELOPER_SAMPLES/restJabber` directory. Fill in the details of the message and the recipient's email address in the client class. You can then build and execute the client using the Ant targets supplied.

If the service is working correctly, an IM is sent and a string message is returned.

Get the ID of a group or API Gateway instance

Every group and API Gateway instance in a domain is assigned a unique ID, and this ID is required to route a REST request to an API Gateway instance. This section describes how to find the ID of a group and API Gateway instance in a number of ways:

- Use the **Print topology** option in the `managedomain` script. See *Print the topology using managedomain* on page 81.
- Call the Topology REST API using curl. See *Use curl to call the Topology REST API* on page 82.
- Use Jython code to query the Topology API. See *Use Jython to query the Topology API* on page 84.

Print the topology using managedomain

To run the `managedomain` script, enter the following commands:

```
> cd INSTALL_DIR/apigateway/posix/bin/
> managedomain --menu
```

Enter the domain user name and password when prompted. The topology management options are displayed as follows:

```
Topology Management:
  13) Print topology
  14) Check topologies are in synch
  15) Check the Admin Node Manager topology against another topology
  16) Synch all topologies
  17) Reset the local topology
```

Chose option 13, `Print topology`. This results in the following output:

```
Version: 2
Last updated: Wed Jan 30 10:17:20 GMT 2013

Hosts:
```

```
|
 ---127.0.0.1 [host-1]
Admin Node Manager:
|
 ---Node Manager on 127.0.0.1 [nodemanager-1] (https://127.0.0.1:8090)
Groups:
|
 ---QuickStart Group [group-2]
   |
    ---QuickStart Server [instance-1] (https://127.0.0.1:8085)
```

All IDs are shown in square brackets beside the node in the topology. In this example, the following are the names and associated IDs:

Type	Display name	ID
Group	QuickStart Group	group-2
API Gateway	QuickStart Server	instance-1

Use curl to call the Topology REST API

The Admin Node Manager is running the Topology API, and this can be called to return a list of groups and API Gateways running in the domain.

To call the API, execute the following curl command (replace `UNAME:PWD` with the domain user name and password):

```
curl --insecure --user UNAME:PWD https://127.0.0.1:8090/api/topology
```

The result is a JSON response with a format similar to the following.

```
{
    "result": {
        "id": "50fd7b96-6e8f-401e-b38c-eb77891e3aeb",
        "version": 2,
        "timestamp": 1428938393531,
        "productVersion": "7.4.1",
        "hosts": [
            {
                "id": "host-1",
                "name": "ITEM-A21575.wks.axway.int"
            }
        ],
        "groups": [
```

```
            {
                "id": "group-1",
                "name": "Node Manager Group",
                "tags": {},
                "services": [
                    {
                        "id": "nodemanager-1",
                        "name": "Node Manager on ITEM-A21575.wks.axway.int",
                        "type": "nodemanager",
                        "scheme": "https",
                        "hostID": "host-1",
                        "managementPort": 8090,
                        "tags": {
                            "internal_admin_nm": "true"
                        },
                        "enabled": true
                    }
                ]
            },
            {
                "id": "group-2",
                "name": "QuickStart Group",
                "tags": {},
                "services": [
                    {
                        "id": "instance-1",
                        "name": "QuickStart Server",
                        "type": "gateway",
                        "scheme": "https",
                        "hostID": "host-1",
                        "managementPort": 8085,
                        "tags": {},
                        "enabled": true
                    }
                ]
            }
        ],
        "uniqueIdCounters": {
            "NodeManager": 2,
            "Group": 3,
            "Host": 2,
            "Gateway": 2
        },
        "fips": false,
        "services": [
            {
                "id": "nodemanager-1",
                "name": "Node Manager on ITEM-A21575.wks.axway.int",
                "type": "nodemanager",
                "scheme": "https",
```

```
                "hostID": "host-1",
                "managementPort": 8090,
                "tags": {
                    "internal_admin_nm": "true"
                },
                "enabled": true
            },
            {
                "id": "instance-1",
                "name": "QuickStart Server",
                "type": "gateway",
                "scheme": "https",
                "hostID": "host-1",
                "managementPort": 8085,
                "tags": {},
                "enabled": true
            }
        ]
    }
}
```


All IDs are found in strings named `id` and are highlighted above.

Note The Admin Node Manager is itself within a group with the ID `group-1`.

In this example, the following are the names and associated IDs:

Type	Display name	ID
Group	QuickStart Group	group-2
API Gateway	QuickStart Server	instance-1

Use Jython to query the Topology API

You can call the Topology API from Jython scripts. The sample Jython script `INSTALL_DIR/apigateway/samples/scripts/topology/outputIDs.py` uses the Topology API to output the name and ID of the group and API Gateway instance.

```
> cd INSTALL_DIR/apigateway/samples/scripts
> ./run.sh topology/outputIDs.py
API Server 'QuickStart Server' has id 'instance-1' belongs to Group 'QuickStart
Group' with id 'group-2', it is running on …
```

Appendix A: Declarative UI reference

This appendix provides in-depth details about declarative XML, which is used in API Gateway to define the user interface of filters and dialogs within Policy Studio.

Declarative XML overview

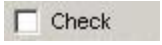
Declarative XML is a user interface markup language defining UI elements and bindings that allows you to quickly create dialogs within Policy Studio with minimal coding.

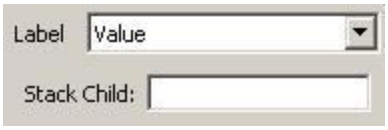
The defined elements map to Eclipse Standard Widget Toolkit (SWT) widgets and Axway ScreenAttributes (groups of SWT widgets backed by entity instances).

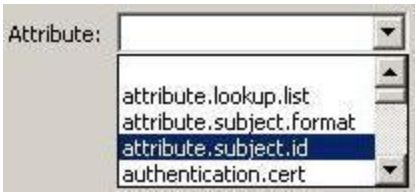
This topic describes in detail the UI elements and bindings.

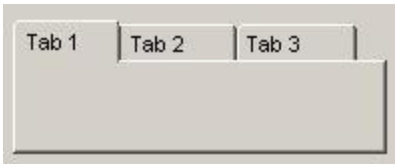
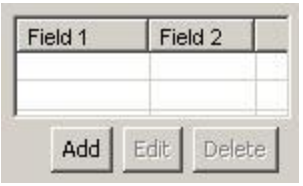
Element quick reference

The following table contains the available elements (in alphabetical order):

Element name	Example
ActorAttribute	
AgeAttribute	
AuthNRepositoryAttribute	
binding	
BitMaskAttribute	
Button	
ButtonAttribute	

Element name	Example
CategoryAttribute	
CertDNameAttribute	
certSelector	
CertTreeAttribute	
CheckboxGroupAttribute	
CircuitChainTable	
ComboAttribute	
ComboBinding	
ComboStackPanel	
condition	
ContentEncodingAttribute	
CronAttribute	
DirectoryChooser	
ESPKReferenceSummaryAttribute	
FieldTable	
FileChooserText	

Element name	Example
group	
HTTPStatusTableAttribute	
include	
label	
LifeTimeAttribute	
MsgAttrAttribute	
MultiValueTextAttribute	
NameAttribute	
NumberAttribute	
panel	
PasswordAttribute	
RadioGroupAttribute	
ReferenceSelector	
SamlAttribute	
SamlSubjectConfirmationAttribute	

Element name	Example
scrollpanel	
section	
SigningKeyAttribute	
SizeAttribute	
SoftRefListAttribute	
SoftRefTreeAttribute	
SpinAttribute	
tab	
tabFolder	
TablePage	
text	

Element name	Example
TextAttribute	
ui	
validator	
XPathAttribute	

The following sections detail the elements, including the available attributes.

Note In the listing of available attributes for each element, the attributes are identified as mandatory (M), optional (O), or conditional (C).

Elements A to C

ActorAttribute

Description

The <ActorAttribute> tag renders an SWT Combo widget with an optional Label. The combo box is populated with the following entry: "Current Actor/Role only". This is the default value for SOAP requests that contain a WS-Security block, and do not contain a SOAP Actor/Role attribute. An additional value can be entered if a WS_Security block with a specific Actor/Role is contained in the SOAP message.

Available attributes

Attribute	Description	M/O	Default
field	Specifies the name of the field of the entity backed by the rendered controls.	M	-
label	Specifies the ID of the resource containing the text to display on the Label.	O	-

Attribute	Description	M/O	Default

span	Value used in the creation of layout-data for the Button.
	Span represents the horizontal span of the following GridData:
	`GridData gridData = new GridData();`
	`gridData.horizontalAlignment = GridData.FILL;`
	`gridData.grabExcessHorizontalSpace = true;`
	`gridData.horizontalSpan = span;`
		O	1

required	Specifies whether or not the entity field is required	O	false

Sample XML

```
<ui>
   <panel indent="15" margin="0">
     <ActorAttribute label="ACTOR_LABEL" field="actor" required="true" />
   </panel>
</ui>
```

Rendered UI

The above XML renders the following UI:

AgeAttribute

Description

The <AgeAttribute> tag renders an SWT Label (optional), Text and Combo widgets, allowing you to specify a numeric age value, and select one of the following age types:

- Seconds
- Minutes
- Hours
- Days

Note The value that is persisted to the underlying entity is stored as milliseconds.

Available attributes

Attribute	Description	M/O	Default
field	Specifies the name of the field of the entity backed by the rendered controls.	M	-
label	Specifies the ID of the resource containing the text to display on the Label.	O	-

span	Value used in the creation of layout-data for the Button.
	Span represents the horizontal span of the following GridData:
	`GridData gridData = new GridData();`
	`gridData.horizontalAlignment = GridData.FILL;`
	`gridData.grabExcessHorizontalSpace = true;`
	`gridData.horizontalSpan = span;`
		O	1

required	Specifies whether or not the entity field is required	O	false

Sample XML

```
<ui>
  <panel columns="3" margin="0">
    <AgeAttribute field="maxAge" label="AGE_LABEL" required="true"/>
  </panel>
</ui>
```

Rendered UI

The above XML renders the following UI:

AuthNRepositoryAttribute

Description

The <AuthNRepositoryAttribute> tag renders an SWT Combo widget with an optional Label.

Authentication repositories are grouped into different types and each type of authentication repository has an associated group of filter types they are compatible with. For example, "Local Repositories" instances are compatible with the following filter types:

- HttpBasicFilter
- HttpDigestFilter
- WsBasicFilter
- WsDigestFilter
- WsUsernameFilter
- AttributeAuthnFilter
- FormAuthnFilter

The Combo widget is then populated with this list of instances that are compatible with this filter type.

Available attributes

Attribute	Description	M/O	Default
field	Specifies the name of the field of the entity backed by the rendered controls.	M	-
label	Specifies the ID of the resource containing the text to display on the Label.	O	-
filterType	When displaying the combo widget, only lists the authentication repositories which are compatible with this specific filter type.	M	-
refName	Specifies Field value of the referenced entity that will be displayed in the Combo box.	M	-
required	Specifies whether or not the entity field is required	M	-

Sample XML

```
<ui>
 <panel columns="2">
  <AuthNRepositoryAttribute label="REPOSITORY_LABEL" field="repository"
   filterType="FormAuthnFilter" refName="name" required="true"/>
 </panel>
</ui>
```

Rendered UI

The above XML renders the following UI:

binding

Description

The <binding> tag allows you to create a binding between various widgets.

Available attributes

Attribute	Description	M/O	Default
driver	Specifies the name of the attributes that designate as drivers separated by commas.	M	-
driven	Specifies a list of attributes separated by commas to be enabled/disabled.	M	-
class	Specifies the name of the class that performs and controls the binding between attributes.	M	-
uncheckOverride	This attribute only applies when using the Enabler class. It allows the Enabler to enable controls when a ButtonAttribute is not selected and disable them when a ButtonAttribute is selected. Specify 'enabled' to override the default Binding behavior.	O	-

Sample XML

The binding below specifies a binding between a <ButtonAttribute> and two <ComboAttribute> attributes. The binding is controlled in the Enabler class.

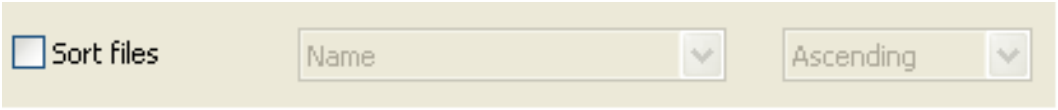
```
<ui>
 <ButtonAttribute field="sortFiles" label="SORT_FILES_LABEL"/>
 <panel columns="2" margin="0">
  <panel label="SORT_TYPE_LABEL" margin="0,0,0,7">
   <ComboAttribute field="sortType"
contentSource="com.vordel.client.manager.filter.dirscan.DirectoryScannerDialog.sortT
ype"
    includeBlank="false" readOnly="true" required="true" stretch="true" />
  </panel>
  <panel label="SORT_DIRECTION_LABEL" margin="0,0,0,7">
```

```
   <ComboAttribute field="sortDirection"
contentSource="com.vordel.client.manager.filter.dirscan.DirectoryScannerDialog.sortD
irection"
    includeBlank="false" readOnly="true" required="true" stretch="true" />
  </panel>
  <binding driver="SORT_FILES_LABEL" driven="SORT_TYPE_LABEL,SORT_DIRECTION_LABEL"
   class="com.vordel.client.ui.declarative.Enabler" />
 </panel>
</ui>
```

Rendered UI

The above XML renders the following UI:

When the Sort files button is disabled the "Sort type" and "Sort direction" Combo boxes are disabled.



When the Sort files button is enabled they are enabled also.

BitMaskAttribute

Description

The <BitMaskAttribute> tag renders a bank of SWT Button widgets, each with the `SWT.CHECK` style applied, and backed by the specified entity field.

Available attributes

Attribute	Description	M/O	Default
field	Specifies the name of the field of the entity backed by the rendered controls.	M	-

Attribute	Description	M/O	Default

columns	Value used in the creation of the layout data for the Composite. Columns represents the number of cell columns of the following GridLayout:
	`GridLayout gridLayout = new GridLayout();`
	`gridLayout.numColumns = columns;`
		O	1

required	Specifies whether or not the field is required	O	false

Each item in the bitmask is represented declaratively as a <choice> tag, which is a child of <BitMaskAttrbute>. The following table outlines the <choice> attributes:

Attribute	Description	M/O	Default
label	Label to be displayed for the check box	M	-
value	Integer value for this choice in the overall bitmask	M	-

Sample XML

```
<ui>
 <panel indent="15" margin="0">
  <BitMaskAttribute field="logMask" columns="3">
   <choice value="1" label="LOG_PAGE_LOG_LEVEL_FATAL"/>
   <choice value="2" label="LOG_PAGE_LOG_LEVEL_FAILURE"/>
   <choice value="4" label="LOG_PAGE_LOG_LEVEL_SUCCESS"/>
  </BitMaskAttribute>
 </panel>
</ui>
```

Rendered UI

The above XML renders the following UI:

button

Description

The <button> tag renders an SWT Button widget with the `SWT.PUSH` style applied.

Available attributes

Attribute	Description	M/O	Default
label	Used internally for callback purposes to allow the extents of the scroll panel to be set correctly. If an image is not specified, the label is also used as the text of the button.	M	-
image	Specifies the ID of the image to be used for the button. The ID must be specified in the `images.properties` file. An image takes precedence over a label, so if both are specified the image will be displayed, rather than the text.	O	-
tooltip	Specifies the tooltip text.	O	-
style	Specifies the style of the button. Possible values are: • check - renders a check box • radio - renders a radio button • push - renders a push button	O	push
selected	If the style attribute is specified and the value is set to "check" or "radio", it specifies whether or not the button is selected. Possible values are "true" and "false".	C	true

Sample XML

```
<ui>
 <panel columns="2">
  <button image="browse" label="BROWSE_TIP" tooltip="BROWSE_TIP" />
 </panel>
</ui>
```

Rendered UI

The above XML renders the following UI:

ButtonAttribute

Description

The <ButtonAttribute> tag renders an SWT Button widget with the `SWT.PUSH` style applied, and backed by the specified entity field.

Available attributes

Attribute	Description	M/O	Default
field	Specifies the name of the field of the entity backed by the rendered controls.	M	-
label	Specifies a textual label for the Button.	M	-
displayName	Specifies the name of the Button to be displayed in the event of an error.	O	-

span	Value used in the creation of layout data for the Button.
	Span represents the horizontal span of the following GridData:
	`GridData gridData = new GridData();`
	`gridData.horizontalAlignment = GridData.FILL;`
	`gridData.grabExcessHorizontalSpace = true;`
	`gridData.horizontalSpan = span;`
		O	1

on	Specifies the attribute value when the Button is checked.	O	-
off	Specifies the attribute value when the Button is unchecked.	O	-
required	Specifies whether or not the field is required.	O	false
trackChanges	Specifies whether or not changes will be tracked when the button state has changed. If set to "true" this calls the `trackChange()` method on the page on which the button is rendered.	O	false

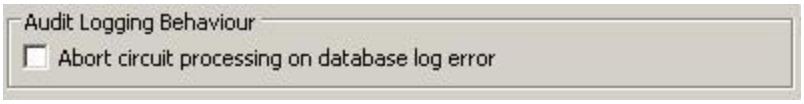
Sample XML

```
<ui>
 <group label="LOG_PAGE_ABORT_SECTION_LABEL">
  <ButtonAttribute field="abort" label="ABORT_PROCESSING_LABEL" span="2"/>
```

```
 </group>
</ui>
```

Rendered UI

The above XML renders the following UI:



CategoryAttribute

Description

The <CategoryAttribute> tag renders an SWT Combo widget with either the `SWT.BORER` or `READ_ONLY` style applied. It lists all the categories available.

Available attributes

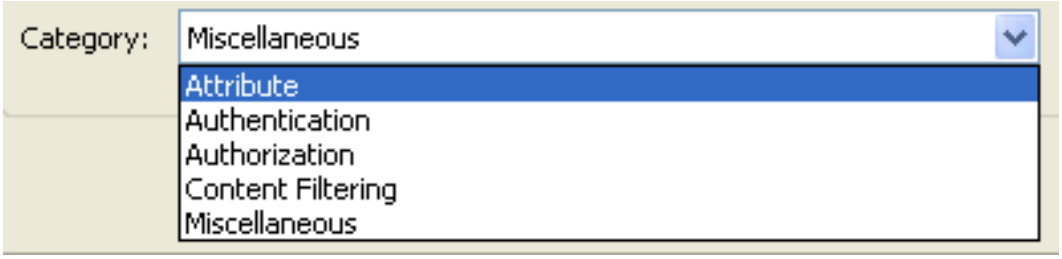
Attribute	Description	M/O	Default
label	Specifies a textual label for the Combo.	O	-
readonly	Specifies the style of the Combo. It can be either `SWT.READ_ONLY` or `SWT.BORDER`.	O	`SWT.READ_ONLY`

Sample XML

```
<ui>
 <CategoryAttribute label="LOG_CATEGORY_LABEL"/>
</ui>
```

Rendered UI

The above XML renders the following UI:



CertDNameAttribute

Description

The <CertDNameAttribute> tag renders a SWT Combo with a list of certificates.

Available attributes

Attribute	Description	M/O	Default
field	Specifies the name of the field of the entity backed by the rendered controls.	O	-
label	Specifies the ID of the resource containing the text to display on the Label (to the left of the combo box)	O	-
required	Specifies whether or not the entity field is required	O	false

Sample XML

```
<ui>
 <CertDNameAttribute label="ISSUER_DNAME_LABEL"
  field="issuerName" required="true" />
</ui>
```

Rendered UI

The above XML renders the following UI:

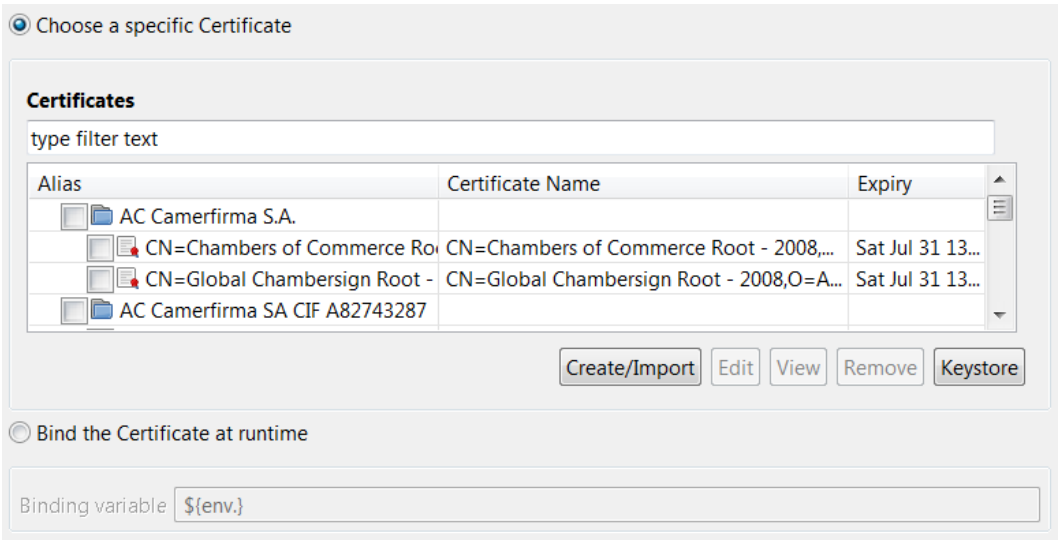
Note The combo box is longer in width but shortened here for clarity.

certSelector

Description

The <certSelector> tag renders a Label with an associated Button and read-only TextBox (which contains the alias of the selected certificate or (unset) if no certificate has been selected).

When the Button is clicked a certificate selection dialog similar to the following is displayed:



Available attributes

Attribute	Description	M/O	Default
field	Specifies the name of the field of the entity backed by the rendered controls.	M	-
label	Specifies a textual label to appear above the table.	O	-
buttonOnRight	Specifies whether or not the button is rendered to the right of the widgets.	O	false

Attribute	Description	M/O	Default
view	If "privateKey" is specified for this attribute the selection dialog that is displayed when the associated button is clicked will only display certificates that contain a private key.	O	-
required	Specifies whether or not the entity field is required.	O	false

Sample XML

```
<ui>
 <panel>
  <certSelector label="SECURITY_SERVER_CERTIFICATE" field="sslCertificate"
   required="false" view="privateKey" />
 </panel>
</ui>
```

Rendered UI

The above XML renders the following UI:

CertTreeAttribute

Description

The <CertTreeAttribute> tag renders a JFace TreeViewer, populated with certificate information read from the certificate store.

Available attributes

Attribute	Description	M/O	Default
field	Specifies the name of the field of the entity backed by the rendered controls.	M	-

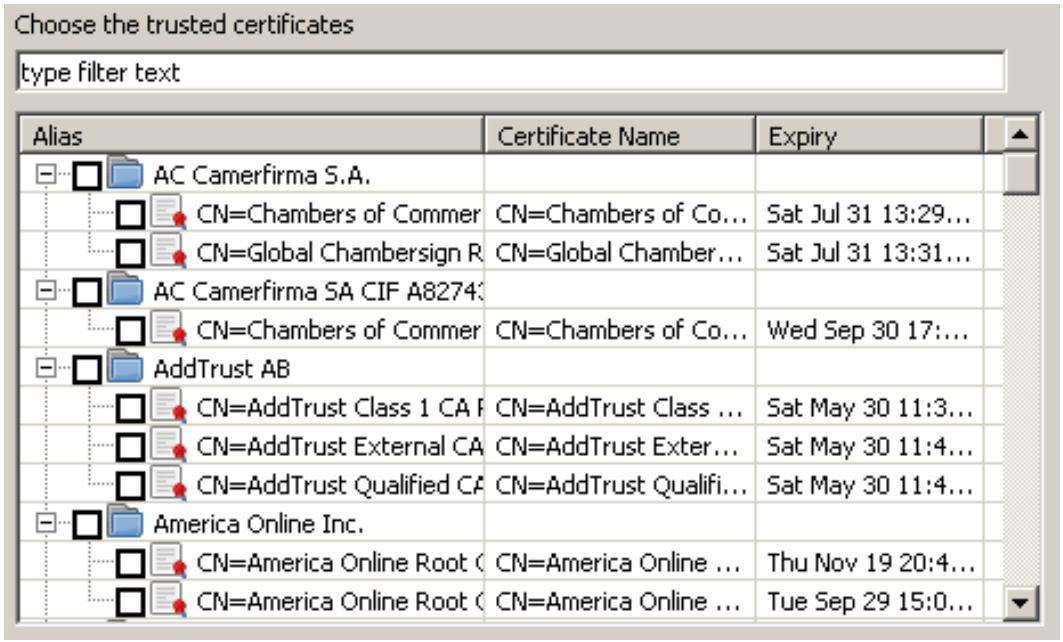
Attribute	Description	M/O	Default
label	Specifies a textual label to appear above the table.	O	-
multiSelect	Specifies whether or not multiple certificates can be selected in the tree.	O	true
keysOnly	Specifies whether or not to filter the tree and only show certificates that contain a private key. By default all certificates are displayed.	O	false
tableHeight	Specifies a height hint for the table.	O	-
required	Specifies whether or not the entity field is required.	O	false

Sample XML

```
<ui>
 <panel>
  <CertTreeAttribute label="SSL_CERTIFICATES_LABEL" field="sslUsers"
   required="false" multiSelect="false" keysOnly="true" tableHeight="300" />
 </panel>
</ui>
```

Rendered UI

The above XML renders the following UI:



CheckboxGroupAttribute

Description

The <CheckboxGroupAttribute> tag renders an SWT Composite with zero or more Buttons (style = `SWT.CHECK`) defined using <choice> tags as children.

CheckboxGroupAttribute attributes

Attribute	Description	M/O	Default
field	Specifies the name of the field of the entity backed by the rendered controls.	M	-

columns	Value used in the creation of layout data for the Composite. Columns represents the number of cell columns in the layout:
	`GridLayout layout = new GridLayout();`
	`layout.numColumns = columns;`
		O	1

required	Specifies whether or not the entity field is required.	O	false

choice attributes

Attribute	Description	M/O	Default
label	Specifies the ID of the resource containing the text to display on the Label.	M	-
value	Specifies one of the possible entity values for the 'field' defined in the CheckboxGroupAttribute tag. This value is tied to the button, and saved to the Entity if this button is selected.	M	-

span	Value used in the creation of layout data for the Button. Span represents the horizontal span of the following GridData:
	`GridData gridData = new GridData();`
	`gridData.horizontalAlignment = GridData.FILL;`
	`gridData.grabExcessHorizontalSpace = true;`
	`gridData.horizontalSpan = span;`
		O	1

Sample XML

The following example represents the multi-valued "options" entity field, of which there are three possible values: "value1", "value2", and "custom".

- If the first check box button is selected (represented by the first <choice> tag) the "options" acquire the value "value1".
- If the second check box button is selected the "options" acquire the two values: "value1" and "value2".
- If all three check boxes are selected the "options" acquire all three values: "value1", "value2" and "custom".

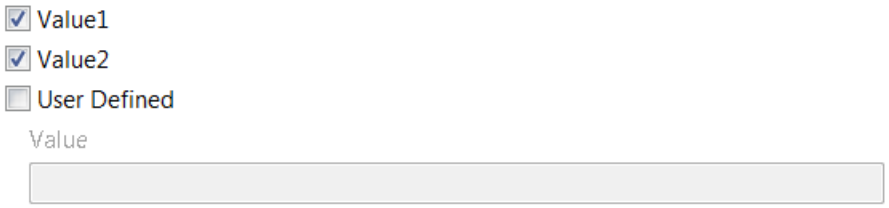
The <CheckboxGroupAttribute> tag is not restricted to having only <choice> tags as children. A good candidate is the <panel> container tag, as outlined in the example below. When the 'User Defined' choice is selected the children of the subsequent panel are enabled automatically. When the 'User Defined' choice is unselected, these children are disabled automatically.

```
<ui>
    <panel>
        <CheckboxGroupAttribute field="options" label="Select Values">
            <choice value="value1" label="Value1" />
            <choice value="value2" label="Value2" />
            <choice value="custom" label="User Defined" />
                <panel indent="15" margin="0">
                    <TextAttribute field="custom" label="Value"/>
                </panel>
        </CheckboxGroupAttribute>
    </panel>
</ui>
```

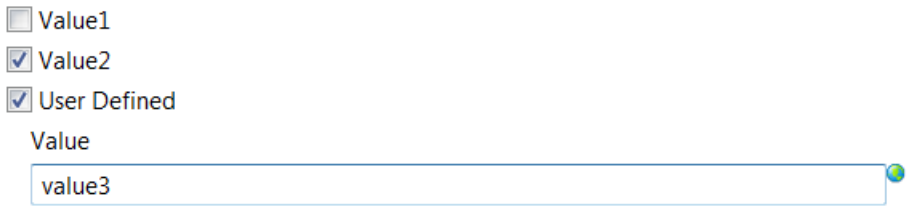
Rendered UI

The above XML renders the following UI.

The multi-valued "options" entity field has two values: "value1" and "value2":



The multi-valued "options" entity field has two values: "value2" and "custom":



CircuitChainTable

Description

The <CircuitChainTable> tag renders a Table widget. The table is populated with the field values of the backed entity.

Available attributes

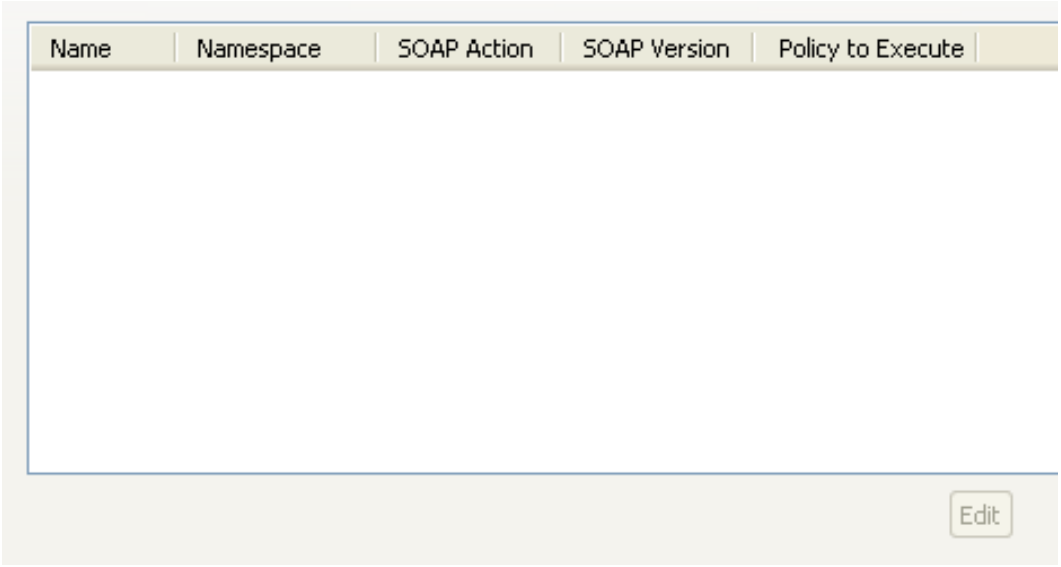
Attribute	Description	M/O	Default
flavor	Specifies the type of entity that is backed by each Table entry in the rendered controls.	M	-
setOrderable	Specifies whether the table has up and down buttons to traverse the entries.	O	true
setCapabilities	Specifies the CRUD capabilities that are allowed. They are separated by a comma [ADD, EDIT, DELETE]	O	-

Sample XML

```
<ui>
  <CircuitChainTable flavor="OperationCircuitReference"
   setOrderable="false" setCapabilities="EDIT"/>
</ui>
```

Rendered UI

The above XML renders the following UI:



ComboAttribute

Description

The <ComboAttribute> tag renders an SWT Combo widget, backed by the specified entity field.

Available attributes

Attribute	Description	M/O	Default
field	Specifies the name of the field of the entity backed by the rendered controls.	M	-
contentSource	Specifies a string array (Java String[]) or map (Java Map<>) with which to populate the combo box.	M	-
label	Specifies a textual label to appear to the left of the Combo	O	-
includeBlank	Specifies whether or not a blank item should be added as the first item to the combo box	O	false

Attribute	Description	M/O	Default
readOnly	Specifies whether or not the combo box is read-only; that is, whether or not the user can enter their own value as well as select from the drop-down list.	O	false
required	Specifies whether or not the entity field is required	O	false
stretch	Specifies whether or not the combo box stretches to fill the available horizontal space	O	false

Sample XML

```
<ui>
 <group label="TRACE_SETTINGS_LABEL" columns="2" span="2">
  <ComboAttribute field="traceLevel" label="TRACE_LEVEL_LABEL"
   required="true" readOnly="true"
   contentSource="com.vordel.client.manager.filter.util.TraceHelper.logLevels" />
 </group>
</ui>
```

Rendered UI

The above XML renders the following UI:

comboBinding

Description

The <comboBinding> tag allows you to create a binding between various widgets.

Available attributes

Attribute	Description	M/O	Default
driver	Specifies the name of the ComboAttribute that designates as the driver.	M	-

Attribute	Description	M/O	Default
driven	Specifies a list of Attributes separated by ',' to be enabled or disabled.	M	-
class	Specifies the name of the class that will perform and control the binding between attributes.	M	-
valueSelected	Designates the value to be initially selected.	M	null
enableDriven	Specifies by default where the attributes will be enabled or disabled on startup.	O	true

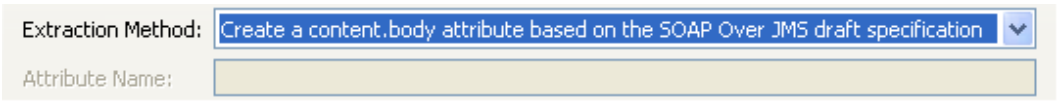
Sample XML

The binding below specifies a binding between a ComboAttribute and a TextAttribute.

```
<ui>
 <ComboAttribute field="extractMethod"
  label="JMS_CONSUMER_EXTRACTION_METHOD_LABEL"

contentSource="com.vordel.client.manager.filter.jms.JMSConsumerDialog.extractMethod
s"
  includeBlank="false" readOnly="true"
  required="true" stretch="true"/>
 <TextAttribute field="attributeName"
  label="JMS_MESSAGE_ATTRIBUTE_NAME"
  required="false"/>
 <comboBinding driver="JMS_CONSUMER_EXTRACTION_METHOD_LABEL"
  driven="JMS_MESSAGE_ATTRIBUTE_NAME"
  class="com.vordel.client.ui.declarative.ComboEnabler"
  valueSelected="1" enableDriven="false"/>
</ui>
```

Rendered UI

The above XML renders the following UI:



ComboStackPanel

Description

The <ComboStackPanel> tag primarily renders an SWT Combo widget, backed by the specified entity field. If the tag contains <panel> children it also renders those controls as part of an SWT StackLayout control, with each panel being 'flipped' when the selection in the combo box changes.

Available attributes

Attribute	Description	M/O	Default
field	Specifies the name of the field of the entity backed by the rendered combo box.	M	-
label	Specifies a textual label to appear to the left of the Combo	O	-
required	Specifies whether or not the entity field is required	O	false

span	Value used in the creation of layout-data for the Combo.
	Span represents the horizontal span of the following GridData:
	`GridData gridData = new GridData();`
	`gridData.horizontalAlignment = GridData.FILL;`
	`gridData.grabExcessHorizontalSpace = true;`
	`gridData.horizontalSpan = span;`
		O	1

Sample XML

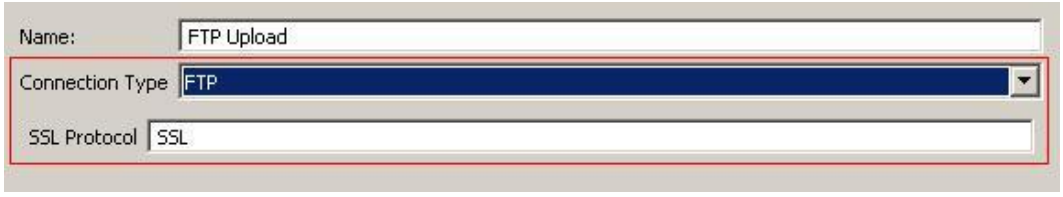
```
<ui>
 <panel columns="2">
  <NameAttribute />
  <ComboStackPanel field="connectionType" label="FTP_UPLOAD_CONNECTION_TYPE_LABEL">
   <panel label="FTP" columns="2">
    <TextAttribute label="FTP_UPLOAD_SSL_PROTOCOL_LABEL" field="sslProtocol"/>
   </panel>

   <panel label="FTPS" columns="2">
    <TextAttribute label="FTP_UPLOAD_SSL_PROTOCOL_LABEL" field="sslProtocol"/>
    <ButtonAttribute label="FTP_UPLOAD_SSL_IS_IMPLICIT" field="isImplicit" />
    <tabFolder span="2">
     <tab label="FTP_UPLOAD_SSL_TRUSTED_CERTS_TAB">
      <panel>
       <CertTreeAttribute label="FTP_UPLOAD_SSL_TRUSTED_CERTS_LABEL"
```

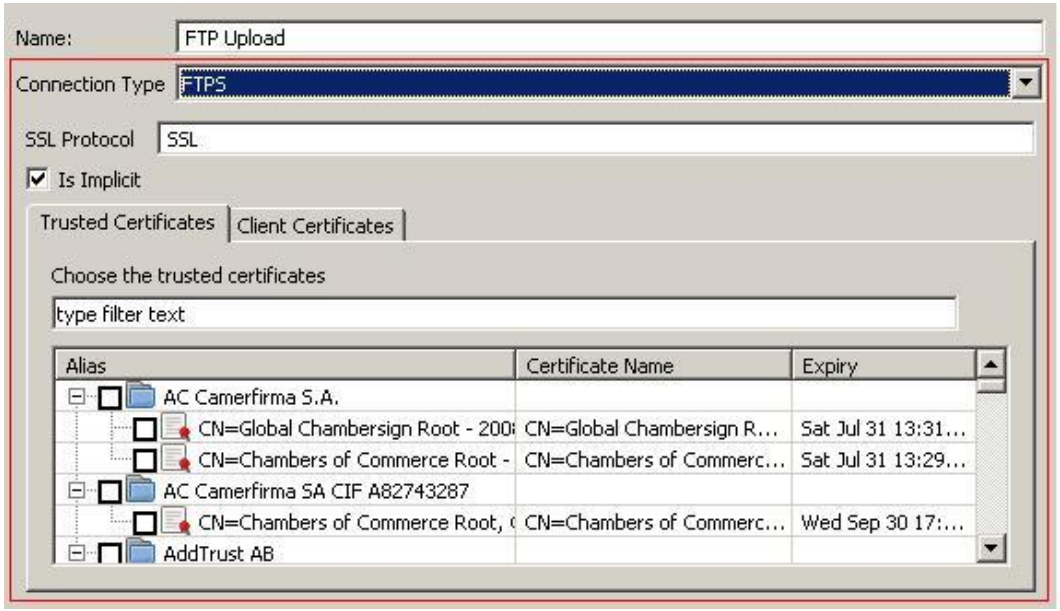
```
                field="trustedCerts" required="false"
                tableHeight="100" />
       </panel>
      </tab>
      <tab label="FTP_UPLOAD_SSL_CLIENT_CERTS_TAB">
       <panel>
        <CertTreeAttribute label="FTP_UPLOAD_SSL_CLIENT_CERTS_LABEL"
                field="clientCert" required="false"
                multiSelect="false" keysOnly="true"
                tableHeight="100" />
       </panel>
      </tab>
     </tabFolder>
    </panel>
   </ComboStackPanel>
  </panel>
</ui>
```

Rendered UI

The above XML renders the following UI:

Note The red rectangles are for illustrative purposes, and show the controls rendered by the ComboStackPanel and its children.





Condition

Description

The <Condition> tag introduces control statements.

Available attributes

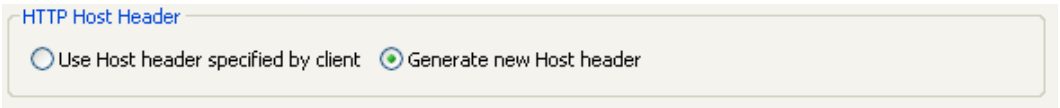
Attribute	Description	M/O	Default
criteria	Specifies the control statement. Currently "if" and "ifnot" are supported.	M	-
property	Specifies a property value to be evaluated.	M	-
type	Specifies the type of Condition. Currently only "JRE" is supported.	O	JRE
value	Specifies the value of the property. If not specified an attempt is made to get the value of the property.	O	-

Sample XML

```
<ui>
 <Condition criteria="ifnot" property="httphostheader.disabled">
   <group label="HOST_HEADER_GROUP_NAME" columns="2" fill="false">
     <RadioGroupAttribute field="forwardClientHostHeader" columns="2">
       <choice value="1" label="HOST_HEADER_FROM_CLIENT"/>
       <choice value="0" label="HOST_HEADER_FROM_VORDEL" />
     </RadioGroupAttribute>
   </group>
 </Condition> />
</ui>
```

Rendered UI

If the "httphostheader" property is not enabled nothing is displayed. If the "httphostheader" property is enabled it renders the following UI:



CronAttribute

Description

The <CronAttribute> tag renders an SWT Button widget and a SWT Text widget, backed by the specified entity field.

Available attributes

Attribute	Description	M/O	Default
field	Specifies the name of the field of the entity backed by the rendered controls.	M	-
label	Specifies a textual label to appear to the left of the Combo.	O	-
required	Specifies whether or not the entity field is required.	M	false

Sample XML

```
<ui>
   <CronAttribute field="expression"
    label="CRON_EXPRESSION_DIALOG_EXPRESSION_LABEL"
    required="true"/>
</ui>
```

Rendered UI

The above XML renders the following UI:

ContentEncodingAttribute

Description

The <ContentEncodingAttribute> tag renders an SWT Text widget and a SWT Button widget, backed by the specified entity field.

Available attributes

Attribute	Description	M/O	Default
field	Specifies the name of the field of the entity backed by the rendered controls.	M	-
label	Specifies the ID of the resource containing the text to display on the Label (to the left of the Text box)	O	-
trackChanges	Specifies whether or not changes will be tracked when the button state has changed. If set to "true" this will call the trackChange() method on the page on which the button is rendered	O	false

span	Value used in the creation of layout data for the Button..
	Span represents the horizontal span of the following GridData:
	`GridData gridData = new GridData();`
	`gridData.horizontalAlignment = GridData.FILL;`
	`gridData.grabExcessHorizontalSpace = true;`
	`gridData.horizontalSpan = span;`
		O	1

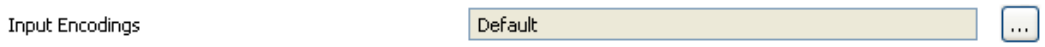
required	Specifies whether or not the entity field is required	O	false

Sample XML

```
<ui>
   <ContentEncodingAttribute field="inputEncodings"
    label="inputEncodings" trackChanges="true"/>
</ui>
```

Rendered UI

The above XML renders the following UI:



Elements D to M

DirectoryChooser

Description

The <DirectoryChooser> tag renders an SWT Label, Text and Button widget. When clicked, the button displays a directory browser to allow you to easily select a directory.

Available attributes

Attribute	Description	M/O	Default
label	Text for the label to be displayed	M	-
field	Specifies the name of the field of the entity backed by the rendered controls.	M	-

span	Value used in the creation of layout data for the Composite.
	Span represents the horizontal span of the following GridData:
	`GridData gridData = new GridData();`
	`gridData.horizontalAlignment = GridData.FILL;`
	`gridData.grabExcessHorizontalSpace = true;`
	`gridData.horizontalSpan = span;`
		O	1

required	Specifies whether or not the entity field is required	O	false

Sample XML

```
<ui>
 <group label="LOCATION_LABEL" span="2">
  <panel columns="2">
   <TextAttribute field="fileName" label="FILENAME_LABEL" required="true" />
   <DirectoryChooser field="directory" label="DIRECTORY_LABEL" required="true"
    span="2" />
  </panel>
 </group>
</ui>
```

Rendered UI

The above XML renders the following UI:

ESPKReferenceSummaryAttribute

Description

The <ESPKReferenceSummaryAttribute> tag renders an SWT Text and Button control. When clicked, the button displays a reference browser to allow you to easily select the required entity reference.

Available attributes

Attribute	Description	M/O	Default
label	Specifies the ID of the resource containing the text to display on the Label.	O	-
field	Specifies the name of the field of the entity backed by the rendered controls.	M	-
required	Specifies whether or not the entity field is required.	O	false
dialogTitle	Specifies the ID of the resource containing the text to display on the title bar of the reference browser dialog.	O	-
displayName	Specifies the ID of the resource containing the attribute/control name to be displayed in the event of an error.	O	-
selectableTypes	Specifies the entity types (as a comma separated list) that are selectable in the TreeViewer displayed in the Reference Selector dialog.	M	-
searches	Specifies the entity types (as a comma separated list) that are searchable for entities of those types specified by the "selectableTypes" attribute.	M	-

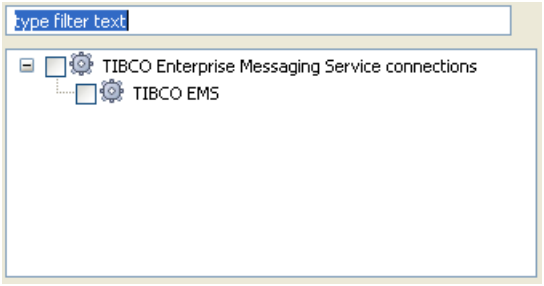
Sample XML

```
<ui>
 <ESPKReferenceSummaryAttribute
  displayName="EMS_CONSUMER_SELECT_CONNECTION_DISP_NAME"
  field="emsClient"
  searches="EMSClientGroup" selectableType="EMSClient"
  dialogTitle="EMS_CLIENT_DIALOG_TITLE" required="true" span="2" />
</ui>
```

Rendered UI

The above XML renders the following UI:

The following dialog is displayed when you click the browse button:



FieldTable

Description

The <FieldTable> tag renders an SWT TableViewer, along with a bank of buttons to allow you to enter a list of values, the type of which is based on the specified entity field.

Available attributes

Attribute	Description	M/O	Default
label	Text for the label to be displayed.	M	-

Attribute	Description	M/O	Default
field	Specifies the name of the field of the entity backed by the rendered controls.	M	-
helpID	Help identifier used for the Add/Edit dialogs associated with the table.	M	-

span	Value used in the creation of layout data for the controls.
	Span represents the horizontal span of the following GridData:
	`GridData gridData = new GridData();`
	`gridData.horizontalAlignment = GridData.FILL;`
	`gridData.grabExcessHorizontalSpace = true;`
	`gridData.horizontalSpan = span;`
		O	2

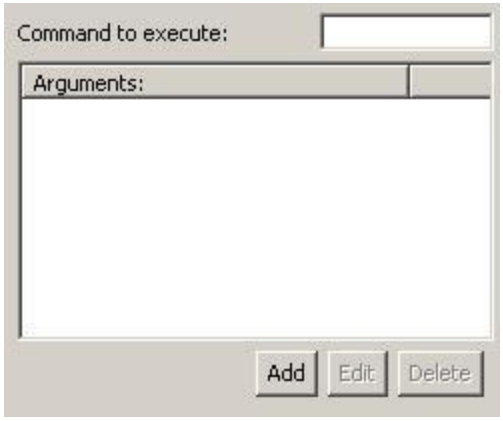
required	Specifies whether or not the entity field is required.	O	false
columnWidth	Specifies the width of the column in the table.	O	200
addDialogTitle	Specifies the title of the dialog that appears when the 'Add' button is clicked.	O	'Add'
editDialogTitle	Specifies the title of the dialog that appears when the 'Edit' button is clicked.	O	'Edit'
labelText	Specifies the text that appears on the label on the Add and Edit dialogs.	O	'Value'

Sample XML

```
<ui>
   <panel columns="2">
     <TextAttribute field="cmdLine" label="CMD_LINE_LABEL" required="true" />
     <FieldTable field="arguments" label="ARGUMENTS_LABEL" />
  </panel>
</ui>
```

Rendered UI

The above XML renders the following UI:



FileChooserText

Description

The <FileChooserText> tag renders an SWT Label, Text and Button widget. When clicked, the button displays a file browser to allow a user to easily select a file.

Available attributes

Attribute	Description	M/O	Default
label	Text for the label to be displayed	M	-
field	Specifies the name of the field of the entity backed by the rendered controls.	M	-

span	Value used in the creation of layout data for the Composite.
	Span represents the horizontal span of the following GridData:
	`GridData gridData = new GridData();`
	`gridData.horizontalAlignment = GridData.FILL;`
	`gridData.grabExcessHorizontalSpace = true;`
	`gridData.horizontalSpan = span;`
		O	1

required	Specifies whether or not the entity field is required	O	false

Sample XML

```
<ui>
  <panel columns="2">
    <FileChooserText field="fileIn" label="FILE_LABEL" required="true" span="2" />
  </panel>
</ui>
```

Rendered UI

The above XML renders the following UI:

group

Description

The <group> tag renders an SWT Group widget, which is usually used to group other widgets.

Available attributes

Attribute	Description	M/O	Default
label	Used to give the group a visual name and also employed internally for binding purposes to allow the control and its children to be enabled/disabled.	O	-

columns	Value used in the creation of the layout data for the Composite.
	Columns represent the number of cell columns of the following GridLayout:
	`GridLayout gridLayout = new GridLayout();`
	`gridLayout.numColumns = columns;`
		O	1

Attribute	Description	M/O	Default

span	Value used in the creation of layout data for the Composite.
	Span represents the horizontal span of the following GridData:
	`GridData gridData = new GridData();`
	`gridData.horizontalAlignment = GridData.FILL;`
	`gridData.grabExcessHorizontalSpace = true;`
	`gridData.horizontalSpan = span;`
		O	1

margin	Value used in the creation of the layout data for the Composite.
	Margin specifies the number of pixels to be used for the Composite. It can be specified as a single integer value whereby the following layout members will be set:
	`GridLayout gridLayout = new GridLayout();`
	`gridLayout.marginHeight = margin;`
	`gridLayout.marginWidth = margin;`
	`gridLayout.marginTop = margin;`
	`gridLayout.marginBottom = margin;`
	`gridLayout.marginLeft = margin;`
	`gridLayout.marginRight = margin;`
	Margin can also be specified as a list of 4 integer values, whereby the following layout members will be set:
	`GridLayout gridLayout = new GridLayout();`
	`StringTokenizer st =`
	`new StringTokenizer(margin, ",");`
	`gridLayout.marginTop = st.nextToken();`
	`gridLayout.marginBottom = st.nextToken();`
	`gridLayout.marginLeft = st.nextToken();`
	`gridLayout.marginRight = st.nextToken();`
		O	5

fill	Value used in the creation of the layout data for the Composite.
	Fill specifies that the layout should resize the Composite to fill both horizontally and vertically, and, depending on its parent, should grow horizontally and vertically if the space is available. Fill is usually used in conjunction with "span". Fill represents the following GridData:
	`GridData gridData = new GridData();`
	`gridData.horizontalAlignment = GridData.FILL;`
	`gridData.verticalAlignment = GridData.FILL;`
	`gridData.grabExcessHorizontalSpace = true;`
	`gridData.grabExcessVerticalSpace = true;`
	`gridData.horizontalSpan = span;`
		O	true

Attribute	Description	M/O	Default

verticalSpacing	Value used in the creation of the layout data for the Composite.
	Specifies the number of pixels from the bottom edge of one cell and the top edge of its neighboring cell underneath:
	`GridLayout gridLayout = new GridLayout();`
	`gridLayout.verticalSpacing = verticalSpacing;`
		O	5

indent	Value used in the creation of the layout data for the Composite.
	Specifies the number of pixels of horizontal margin that will be placed along the left and right edges of the layout. The following layout member will be set:
	`GridLayout gridLayout = new GridLayout();`
	`gridLayout.marginWidth = indent;`
		O	0

Sample XML

```
<ui>
 <group label="LOG_PAGE_CATEGORY_LABEL" columns="2">
  <CategoryAttribute label="LOG_CATEGORY_LABEL" required="true" />
 </group>
</ui>
```

Rendered UI

The above XML renders the following UI:

HTTPStatusTableAttribute

Description

The <HTTPStatusTableAttribute> tag renders a Table and a group SWT Buttons that appear as a Button bar. When clicked, the buttons display a dialog to add, edit, or delete a HTTP status code.

Available attributes

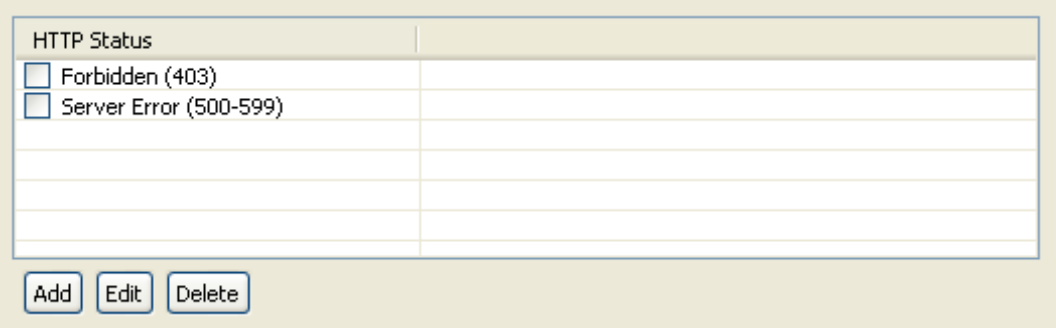
Attribute	Description	M/O	Default
field	Specifies the name of the field of the entity backed by the rendered controls.	M	-
tableHeight	Specifies the preferred height of the control	O	-
required	Specifies whether or not the entity field is required	O	false

Sample XML

```
<ui>
  <HTTPStatusTableAttribute field="retryHTTPRanges" tableHeight="100" />
</ui>
```

Rendered UI

The above XML renders the following UI:



include

Description

The <include> tag allows another declarative XML file to be included inline in the parent including XML file.

Available attributes

Attribute	Description	M/O	Default
resource	The name of the declarative XML file to be included inline in the declarative parent XML file.	M	
class	The class, including package, used to render the included resource. Any required string resources will need to be redefined in the local `resource.properties` file.	O	

Sample XML

```
<ui>
 <panel columns="2">
  <include resource="AuditSettings-page.xml"
   class="com.vordel.client.manager.AuditSettingsPage"/>
 </panel>
</ui>
```

label

Description

The <label> tag renders an SWT Label widget.

Available attributes

Attribute	Description	M/O	Default
label	Specifies the ID of the resource containing the text to display on the Label.	M	-

span	Value used in the creation of layout data for the Label..
	Span represents the horizontal span of the following GridData:
	`GridData gridData = new GridData();`
	`gridData.horizontalAlignment = GridData.FILL;`
	`gridData.grabExcessHorizontalSpace = true;`
	`gridData.horizontalSpan = span;`
		O	1

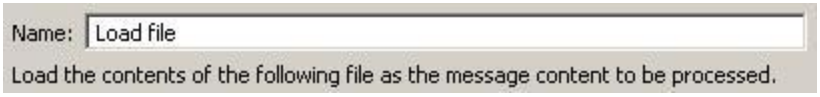
bold	Value used to specify whether or not the font is rendered bold.	O	false

Sample XML

```
<ui>
 <panel columns="2">
  <NameAttribute />
  <label label="FILE_TO_LOAD_SUMMARY" span="2" />
 </panel>
</ui>
```

Rendered UI

The above XML renders the following UI:



LifeTimeAttribute

Description

The <LifeTimeAttribute> tag renders a Label, Text, and a group of Spin controls, representing a time span. This allows you to enter values for Days, Hours, Minutes and Seconds.

Available attributes

Attribute	Description	M/O	Default
label	Specifies the ID of the resource containing the text to display on the Label to the left of all controls	O	-
field	Specifies the name of the field of the entity backed by the rendered controls.	M	-
required	Specifies whether or not the entity field is required	O	false

Sample XML

```
<ui>
 <panel columns="2">
  <panel span="2" columns="3" margin="0">
```

```
    <LifeTimeAttribute label="VALIDITY_LABEL" field="validity" required="true" />
  </panel>
 </panel>
</ui>
```

Rendered UI

The above XML renders the following UI:

NameAttribute

Description

The <NameAttribute> tag renders an SWT Label and accompanying Text widget. It is used to wrap the following TextAttribute:

```
<TextAttribute field="name" label="NAME_LABEL" required="true" />
```

The label `NAME_LABEL` must exist in the appropriate `resource.properties` file.

Sample XML

```
<ui>
  <panel columns="2">
    <NameAttribute />
  </panel>
</ui>
```

Rendered UI

The above XML renders the following UI:

MsgAttrAttribute

Description

The <MsgAttrAttribute> tag renders an SWT Label and accompanying Combo widget populated with a list of Axway message attributes.

Available attributes

Attribute	Description	M/O	Default
label	Specifies the ID of the resource containing the text to display on the Label.	M	-
field	Specifies the name of the field of the entity backed by the rendered controls.	M	-

span	Value used in the creation of layout data for the Composite.
	Span represents the horizontal span of the following GridData:
	`GridData gridData = new GridData();`
	`gridData.horizontalAlignment = GridData.FILL;`
	`gridData.grabExcessHorizontalSpace = true;`
	`gridData.horizontalSpan = span;`
		O	1

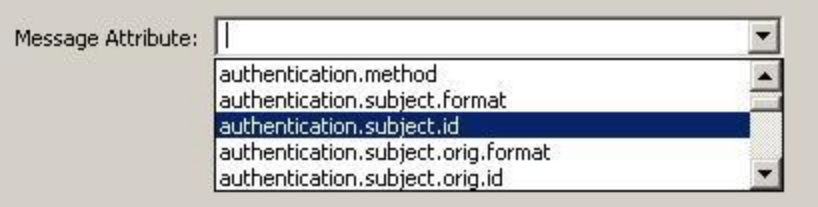
required	Specifies whether or not the entity field is required.	O	false

Sample XML

```
<ui>
 <panel columns="2">
  <MsgAttrAttribute field="sourceAttribute"
   label="STRING_REPLACE_SRC_ATTRIBUTE_LABEL" required="true"/>
 </panel>
</ui>
```

Rendered UI

The above XML renders the following UI:



MultiValueTextAttrAttribute

Description

The <MultiValueTextAttribute> tag is similar to the TextAttribute tag in that it renders an SWT Text widget (and optionally, a Label widget), backed by the specified field for the entity being configured. The difference is that it caters for multiple values interspersed with the specified separator.

Available attributes

Attribute	Description	M/O	Default
field	Specifies the name of the field of the entity backed by the rendered controls. The default value of the field will automatically appear in the Text widget.	M	-
label	Indicates that a Label should be rendered to the left of the Text widget. The value of this field is set to a resource identifier, specified in a `resource.properties` file.	O	-
required	Specifies whether or not the field is required. If required and the user does not enter a value, a warning dialog appears, prompting the user to enter a value for the field.	O	-

Attribute	Description	M/O	Default

span	Value used in the creation of layout data for the controls that are rendered.
	If a single-line control is being rendered, the span represents the horizontal span of the following GridData:
	`GridData gridData = new GridData();`
	`gridData.horizontalAlignment = GridData.FILL;`
	`gridData.grabExcessHorizontalSpace = true;`
	`gridData.horizontalSpan = colSpan;`
	If multiline control is being rendered, the span represents the horizontal span of the following GridData:
	`GridData gridData = new GridData();`
	`gridData.horizontalAlignment = GridData.FILL;`
	`gridData.verticalAlignment = GridData.FILL;`
	`gridData.grabExcessHorizontalSpace = true;`
	`gridData.grabExcessVerticalSpace = true;`
	`gridData.horizontalSpan = colSpan;`
		O	1

split	Specifies the string used as a separator to the list of multiple values.	O	,

Sample XML

```
<ui>
 <panel columns="2">
  <MultiValueTextAttribute field="extension" label="EXTENSION_LABEL"
   required="false" split=";"/>
 </panel>
</ui>
```

Rendered UI

The above XML renders the following UI:

In this case, `EXTENSION_LABEL` is resolved to the localized string "Extension:".

Elements N to S

NumberAttribute

Description

The <NumberAttribute> tag renders an SWT Label and accompanying Text widget. The Text widget only accepts numbers as input.

Available attributes

Attribute	Description	M/O	Default
label	Specifies the ID of the resource containing the text to display on the Label.	M	-
field	Specifies the name of the field of the entity backed by the rendered controls.	M	-

span	Value used in the creation of layout data for the Composite.
	Span represents the horizontal span of the following GridData:
	`GridData gridData = new GridData();`
	`gridData.horizontalAlignment = GridData.FILL;`
	`gridData.grabExcessHorizontalSpace = true;`
	`gridData.horizontalSpan = span;`
		O	1

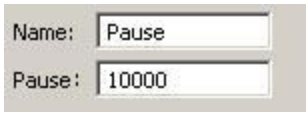
readOnly	Specifies whether or not the Text widget is editable.	O	false
min	Specifies the minimum permitted value for the number field.	O	0
max	Specifies the maximum permitted value for the number field.	O	100
required	Specifies whether or not the entity field is required.	O	false

Sample XML

```
<ui>
 <panel columns="2">
  <NameAttribute />
  <NumberAttribute field="pause" label="PAUSE_FOR_LABEL" required="true" />
 </panel>
</ui>
```

Rendered UI

The above XML renders the following UI:



panel

Description

The <panel> tag renders an SWT Composite widget, which is usually used to group other widgets.

Available attributes

Attribute	Description	M/O	Default
label	Used internally for binding purposes to allow the control and its children to be enabled/disabled.	O	-

columns	Value used in the creation of the layout data for the Composite.
	Columns represents the number of cell columns of the following GridLayout:
	`GridLayout gridLayout = new GridLayout();`
	`gridLayout.numColumns = columns;`
		O	1

span	Value used in the creation of layout data for the Composite.
	Span represents the horizontal span of the following GridData:
	`GridData gridData = new GridData();`
	`gridData.horizontalAlignment = GridData.FILL;`
	`gridData.grabExcessHorizontalSpace = true;`
	`gridData.horizontalSpan = span;`
		O	1

Attribute	Description	M/O	Default

margin	Value used in the creation of the layout data for the Composite.
	Margin specifies the number of pixels to be used for the Composite. It can be specified as a single integer value whereby the following layout members will be set:
	`GridLayout gridLayout = new GridLayout();`
	`gridLayout.marginHeight = margin;`
	`gridLayout.marginWidth = margin;`
	`gridLayout.marginTop = margin;`
	`gridLayout.marginBottom = margin;`
	`gridLayout.marginLeft = margin;`
	`gridLayout.marginRight = margin;`
	Margin can also be specified as a list of four integer values, whereby the following layout members will be set:
	`GridLayout gridLayout = new GridLayout();`
	`StringTokenizer st =`
	`new StringTokenizer(margin, ",");`
	`gridLayout.marginTop = st.nextToken();`
	`gridLayout.marginBottom = st.nextToken();`
	`gridLayout.marginLeft = st.nextToken();`
	`gridLayout.marginRight = st.nextToken();`
		O	5

fill	Value used in the creation of the layout data for the Composite.
	Fill specifies that the layout should resize the Composite to fill both horizontally and vertically, and, depending on its parent, should grow horizontally and vertically if the space is available. Fill is usually used in conjunction with "span". Fill represents the following GridData:
	`GridData gridData = new GridData();`
	`gridData.horizontalAlignment = GridData.FILL;`
	`gridData.verticalAlignment = GridData.FILL;`
	`gridData.grabExcessHorizontalSpace = true;`
	`gridData.grabExcessVerticalSpace = true;`
	`gridData.horizontalSpan = span;`
		O	true

Attribute	Description	M/O	Default

verticalSpacing	Value used in the creation of the layout data for the Composite.
	Specifies the number of pixels from the bottom edge of one cell and the top edge of its neighboring cell underneath.
	`GridLayout gridLayout = new GridLayout();`
	`gridLayout.verticalSpacing = verticalSpacing;`
		O	5

indent	Value used in the creation of the layout data for the Composite.
	Specifies the number of pixels of the horizontal margin that will be placed along the left and right edges of the layout. The following layout member will be set:
	`GridLayout gridLayout = new GridLayout();`
	`gridLayout.marginWidth = indent;`
		O	0

horizontalAlignment	Possible values are:
	• "center" - `SWT.CENTER`
	• "right" - `SWT.RIGHT`
	Value used in the creation of the layout data for the Composite.
	`GridData gridData = new GridData();`
	`gridData.horizontalAlignment = SWT.CENTER;`
		O	SWT.LEFT

verticalAlignment	Possible values are:
	• "center" - `SWT.CENTER`
	• "bottom" - `SWT.BOTTOM`
	Value used in the creation of the layout data for the Composite.
	`GridData gridData = new GridData();`
	`gridData.verticalAlignment = SWT.CENTER;`
		O	SWT.TOP

widthHint	Value used in the creation of the layout data for the Composite.
	`GridData gridData = new GridData();`
	`gridData.widthHint = 200;`
		O	

Attribute	Description	M/O	Default

heightHint	Value used in the creation of the layout data for the Composite.
	`GridData gridData = new GridData();`
	`gridData.heightHint = 300;`
		O	

minWidth	Value used in the creation of the layout data for the Composite.
	`GridData gridData = new GridData();`
	`gridData.minimumWidth = 300;`
		O	

minHeight	Value used in the creation of the layout data for the Composite.
	`GridData gridData = new GridData();`
	`gridData.minimumHeight = 300;`
		O	

Sample XML

```
<ui>
 <panel span=”2” columns="2" indent=”15” margin=”0” label=”SETTINGS_PANEL”>
  <ReferenceSelector field="connectionFailurePolicy"
   type="FilterCircuit"
   label="CONNECTION_FAILURE_POLICY_SELECTION_DIALOG_TITLE"
   title="CHOOSE_CONNECTION_FAILURE_POLICY" />
 </panel>
</ui>
```

Rendered UI

The above XML renders the following UI:

Note The red rectangle is for illustration purposes only.

PasswordAttribute

Description

The <PasswordAttribute> tag renders an SWT Label and accompanying Text widget. The Text widget has its user-entered text masked with the '*' character.

Available attributes

Attribute	Description	M/O	Default
label	Specifies the ID of the resource containing the text to display on the Label.	O	-
field	Specifies the name of the field of the entity backed by the rendered controls.	M	-

span	Value used in the creation of layout data for the Composite.
	Span represents the horizontal span of the following GridData:
	`GridData gridData = new GridData();`
	`gridData.horizontalAlignment = GridData.FILL;`
	`gridData.grabExcessHorizontalSpace = true;`
	`gridData.horizontalSpan = span;`
		O	1

required	Specifies whether or not the entity field is required.	O	false
widthHint	Specifies the preferred width of the control.	O	SWT.DEFAULT
heightHint	Specifies the preferred height of the control.	O	false
multiline	Specifies whether the control is a multiline Text widget. If this attribute is not present the control defaults to a single-line widget. No masking occurs if this attribute is set to "true". This restriction is coded in `Text.setEchoChar();`	O	false
wrap	Specifies whether the text should wrapped within the control. This attribute is conditional on the multiline attribute being present and set to true.	O	false

Attribute	Description	M/O	Default
vscroll	Specifies whether a vertical scrollbar should be rendered. This attribute is conditional on the multiline attribute being present and set to true.	C	false
hscroll	Specifies whether a horizontal scrollbar should be rendered. This attribute is conditional on the multiline attribute being present and set to true.	C	false

Sample XML

```
<ui>
 <group label="PROXY_SETTINGS_LABEL" columns="2" span="2">
  <TextAttribute field="username" label="PROXY_USERNAME_LABEL"/>
  <PasswordAttribute field="password" label="PROXY_PASSWORD_LABEL"/>
 </group>
</ui>
```

Rendered UI

The above XML renders the following UI:

RadioGroupAttribute

Description

The <RadioGroupAttribute> tag renders an SWT Composite with 0 or more Buttons (style = `SWT.RADIO`) defined using <choice> tags as children.

RadioGroupAttribute - Available attributes

Attribute	Description	M/O	Default
field	Specifies the name of the field of the entity backed by the rendered controls.	M	-

columns	Value used in the creation of layout data for the Composite. Columns represents the number of cell columns in the layout:
	`GridLayout layout = new GridLayout();`
	`layout.numColumns = columns;`
		O	1

required	Specifies whether or not the entity field is required.	O	false

choice - Available attributes

Attribute	Description	M/O	Default
label	Specifies the ID of the resource containing the text to display on the Label.	M	-
value	Specifies one of the possible entity values for the 'field' defined in the RadioGroupAttribute tag. This value will be tied to the button, and saved to the Entity if this button is selected.	M	-

span	Value used in the creation of layout data for the Button. Span represents the horizontal span of the following GridData:
	`GridData gridData = new GridData();`
	`gridData.horizontalAlignment = GridData.FILL;`
	`gridData.grabExcessHorizontalSpace = true;`
	`gridData.horizontalSpan = span;`
		O	1

Sample XML

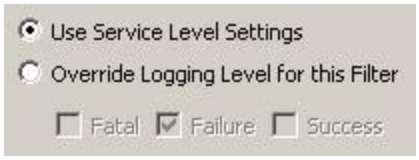
The following example represents the "logMaskType" entity field, of which there are two possible values: "SERVICE" and "FILTER". If the first radio button is selected (represented by the first <choice> tag) the logMaskField will acquire the value "SERVICE". If the second radio button is selected the logMaskField will acquire the value "FILTER".

The <RadioGroupAttribute> tag is not restricted to just having <choice> tags as children. A good candidate is the <panel> container tag, as outlined in the example below. When the 'USE_FILTER' choice is selected the children of the subsequent panel are enabled automatically. When the 'USE_ SERVICE' choice is selected, these children are disabled automatically.

```
<ui>
  <panel>
    <RadioGroupAttribute field="logMaskType" columns="1">
        <choice value="SERVICE" label="USE_SERVICE" />
        <choice value="FILTER" label="USE_FILTER" />
        <panel indent="15" margin="0">
            <BitMaskAttribute field="logMask" columns="3">
               <choice value="1" label="LEVEL_FATAL"/>
               <choice value="2" label="LEVEL_FAILURE"/>
               <choice value="4" label="LEVEL_SUCCESS"/>
            </BitMaskAttribute>
         </panel>
     </RadioGroupAttribute>
  </panel>
</ui>
```

Rendered UI

The above XML renders the following UI:



ReferenceSelector

Description

The <ReferenceSelector> tag renders an SWT Label, Text and Button control. When clicked, the button displays a reference browser to allow you to easily select the required entity reference.

Available attributes

Attribute	Description	M/O	Default
label	Specifies the ID of the resource containing the text to display on the Label.	O	-
field	Specifies the name of the field of the entity backed by the rendered controls.	M	-
required	Specifies whether or not the entity field is required.	O	false
title	Specifies the ID of the resource containing the text to display on the title bar of the reference browser dialog.	M	-
widthHint	Specifies a width hint of reference browser.	O	SWT.DEFAULT
displayName	Specifies the ID of the resource containing the attribute or control name to be displayed in the event of an error.	O	-
selectableTypes	Specifies the entity types (as a comma separated list) that are selectable in the TreeViewer displayed in the Reference Selector dialog.	M	-
searches	Specifies the entity types (as a comma separated list) that are searchable for entities of those types specified by the "selectableTypes" attribute.	M	-
trackChanges	Specifies whether or not to track selection changes.	O	false

Sample XML

```
<ui>
 <panel span="2" columns="2" indent="15" margin="0" label="SETTINGS_PANEL">
  <ReferenceSelector field="connectionFailurePolicy"
   selectableTypes="FilterCircuit"
   searches="ROOT_CIRCUIT_CONTAINER,CircuitContainer"
   label="POLICY_SELECTION_DIALOG_TITLE"
   title="CHOOSE_CONNECTION_FAILURE_POLICY" />
 </panel>
</ui>
```

Rendered UI

The above XML renders the following UI:

The following dialog is displayed when the browse button is clicked:

SamlAttribute

Description

The <SamlAttribute> tag renders a SWT Combo with a list of SAML versions.

Available attributes

Attribute	Description	M/O	Default
field	Specifies the name of the field of the entity backed by the rendered controls.	O	-
label	Specifies the ID of the resource containing the text to display on the Label (to the left of the combo box).	O	-
required	Specifies whether or not the entity field is required.	O	false

Sample XML

```
<ui>
 <SamlAttribute label="SAML_VERSION_REQUIRED_LABEL"
  field="samlVersion" required="true" />
</ui>
```

Rendered UI

The above XML renders the following UI:

SamlSubjectConfirmationAttribute

Description

The <SamlSubjectConfirmationAttribute> tag renders a SWT Combo with a list of available SAML subject confirmation methods. The Attribute is very specific to using the SAML filter. You can only have one <SamlSubjectConfirmationAttribute> per window.

When a SAML 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 table shows the available methods and their meanings :

Method	Meaning
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.
Bearer	The subject of the assertion is the bearer of the assertion.
SAML Artifact	The subject of the assertion is the user that presented a SAML Artifact to the API Gateway.
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 are signed if this option is selected.
	If the Method field is blank then no <ConfirmationMethod> block is inserted into the assertion.

Note The entity field defaults to "subjectConfirmationMethod".

Available attributes

Attribute	Description	M/O	Default
label	Specifies the ID of the resource containing the text to display on the Label	O	-
required	Specifies whether or not the entity field is required.	M	-

span	Value used in the creation of layout data for the Combo.
	Span represents the horizontal span of the following GridData:
	`GridData gridData = new GridData();`
	`gridData.horizontalAlignment = GridData.FILL;`
	`gridData.grabExcessHorizontalSpace = true;`
	`gridData.horizontalSpan = span;`
		O	1

columns	Value used in the creation of the layout data for the Composite.
	Columns represents the number of cell columns of the following GridLayout:
	`GridLayout gridLayout = new GridLayout();`
	`gridLayout.numColumns = columns;`
		O	2

Sample XML

```
<ui>
  <SamlSubjectConfirmationAttribute label="SAML_SUBJECT_CONF_METHOD_LABEL"
   required="true" />
</ui>
```

Rendered UI

The above XML renders the following UI:

scrollpanel

Description

The <scrollpanel> tag renders an SWT ScrolledComposite widget. When rendered the control automatically calculates the extent of its children so that the scroll bars are rendered correctly.

To facilitate ease-of-use, one of the following tags must be a direct child of scrollpanel:

- panel
- group
- tabFolder

Available attributes

Attribute	Description	M/O	Default
label	Used internally for callback purposes to allow the extents of the scrollpanel to be set correctly.	M	-

widthHint	Value used in the creation of the layout data for the Composite:
	`GridData gridData = new GridData();`
	`gridData.widthHint = 200;`
		O	

heightHint	Value used in the creation of the layout data for the Composite:
	`GridData gridData = new GridData();`
	`gridData.heightHint = 300;`
		O	

minWidth	Value used in the creation of the layout data for the Composite:
	`GridData gridData = new GridData();`
	`gridData.minimumWidth = 300;`
		O	

minHeight	Value used in the creation of the layout data for the Composite:
	`GridData gridData = new GridData();`
	`gridData.minimumHeight = 300;`
		O	

Sample XML

```
<ui>
 <scrollpanel>
  <panel columns="2">
    <TextAttribute field="name" label="EXCEPTION_NAME" required="true" />
```

```
    <TextAttribute field="name" label="EXCEPTION_NAME" required="true" />
    <TextAttribute field="name" label="EXCEPTION_NAME" required="true" />
    <TextAttribute field="name" label="EXCEPTION_NAME" required="true" />
    <TextAttribute field="name" label="EXCEPTION_NAME" required="true" />
    <TextAttribute field="name" label="EXCEPTION_NAME" required="true" />
  </panel>
 </scrollpanel>
</ui>
```

Rendered UI

The above XML renders the following UI:

section

Description

The <section> tag renders an SWT ExpandableComposite widget, which allows for groups of controls to be expanded or hidden from view.

To facilitate ease-of-use, one of the following tags must be a direct child of section:

- panel
- group
- tabFolder

Available attributes

Attribute	Description	M/O	Default
label	Specifies the text heading of the section.	M	-
expanded	Specifies whether or not the section is expanded.	O	false

Sample XML

```
<ui>
 <section label="RS_STATUS_LABEL" expanded="true">
  <panel columns="2">
   <TextAttribute field="name" label="RS_STATUS_LABEL" required="true" />
  </panel>
 </section>
</ui>
```

Rendered UI

The above XML renders the following UI:

The section can also be collapsed as follows:

SigningKeyAttribute

Description

The <SigningKeyAttribute> tag renders a SWT Radio Button and a Tab Folder which has three tabs whose content includes SWT Buttons, Radio Buttons, Combo boxes, and so on.

Available attributes

Attribute	Description	M/O	Default
subjectConfirmationNote	Specifies whether a generic signing panel is displayed or a specific SAML signing panel is displayed. By default the generic signing panel is displayed.	O	false
label	Specifies the ID of the resource containing the text to display on the Label.	O	-

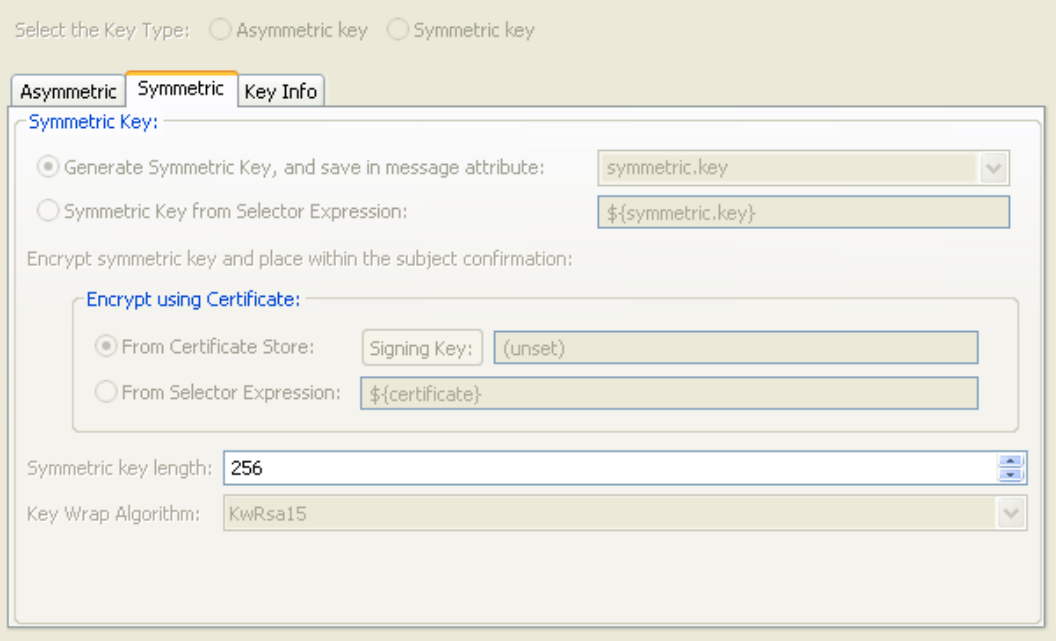
Attribute	Description	M/O	Default
required	Specifies whether or not the entity field is required.	O	false

Sample XML

```
<ui>
  <SigningKeyAttribute
   subjectConfirmationNote="SUBJECT_CONFIRMATION_ASYMMETRIC_NOTE_LABEL"
   required="false"/>
</ui>
```

Rendered UI

The above XML renders the following UI:



SizeAttribute

Description

The <SizeAttribute> tag renders an SWT Label (optional), Text, and Combo widgets, allowing you to specify a numeric size value, and select one of the following age types:

- Kb
- Mb
- Gb

Note The value that is persisted to the underlying entity is stored as bytes.

Available attributes

Attribute	Description	M/O	Default
field	Specifies the name of the field of the entity backed by the rendered controls.	M	-
label	Specifies the ID of the resource containing the text to display on the Label.	O	-

span	Value used in the creation of layout data for the Button.
	Span represents the horizontal span of the following GridData:
	`GridData gridData = new GridData();`
	`gridData.horizontalAlignment = GridData.FILL;`
	`gridData.grabExcessHorizontalSpace = true;`
	`gridData.horizontalSpan = span;`
		O	1

required	Specifies whether or not the entity field is required.	O	false

Sample XML

```
<ui>
 <panel columns="3" margin="0">
   <SizeAttribute field="maxDbSize" label="OPDB_MAX_DB_SIZE_LABEL" required="true"/>
 </panel>
</ui>
```

Rendered UI

The above XML renders the following UI:

SoftRefListAttribute

Description

The <SoftRefListAttribute> renders a SWT Combo that shows a list of entities of a certain type.

Available attributes

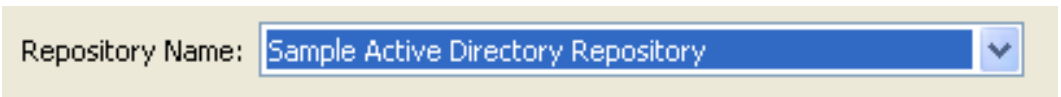
Attribute	Description	M/O	Default
field	Specifies the name of the field of the entity backed by the rendered controls.	M	-
label	Specifies the ID of the resource containing the text to display on the Label.	O	-
refName	Specifies Field value of the referenced entity that is displayed in the Combo box.	M	-
displayName	Specifies the name of the SoftRefListAttribute to be displayed in the event of an error.	O	-
src	Specifies the Shorthand Key to get the list of referenced entities of a particular type.	M	-
required	Specifies whether or not the entity field is required.	M	false

Sample XML

```
<ui>
 <SoftRefListAttribute label="WS_USERNAME_TOKEN_REPOSITORY_NAME_LABEL"
  field="repository" refName="name"
  src="/[AuthnRepositoryGroup]name=Authentication Repositories/
[AuthnRepositoryBaseGroup]allowedFilter=WsUsernameFilter/[AuthnRepositoryBase]" />
</ui>
```

Rendered UI

The above XML renders the following UI:



SoftRefTreeAttribute

Description

The <SoftRefTreeAttribute> renders a jFace TreeViewer that shows the policies of a certain type.

Available attributes

Attribute	Description	M/O	Default
field	Specifies the name of the field of the entity backed by the rendered controls.	M	-
searches	Comma separated list of Portable ESPKs. It will then display all the entities represented by each Portable ESPK.	M	-
selectableTypes	Specifies the type of entities that are selectable.	M	-
displayName	Specifies the name of the SoftRefTreeAttribute to be displayed in the event of an error.	O	-
span	Value used in the creation of layout data for the controls that are rendered. `GridData gridData = new GridData(GridData.FILL_BOTH); gridData.horizontalSpan = 1;`	O	1
width	Specifies the preferred width of the control.	O	300
height	Specifies the preferred height of the control.	O	300

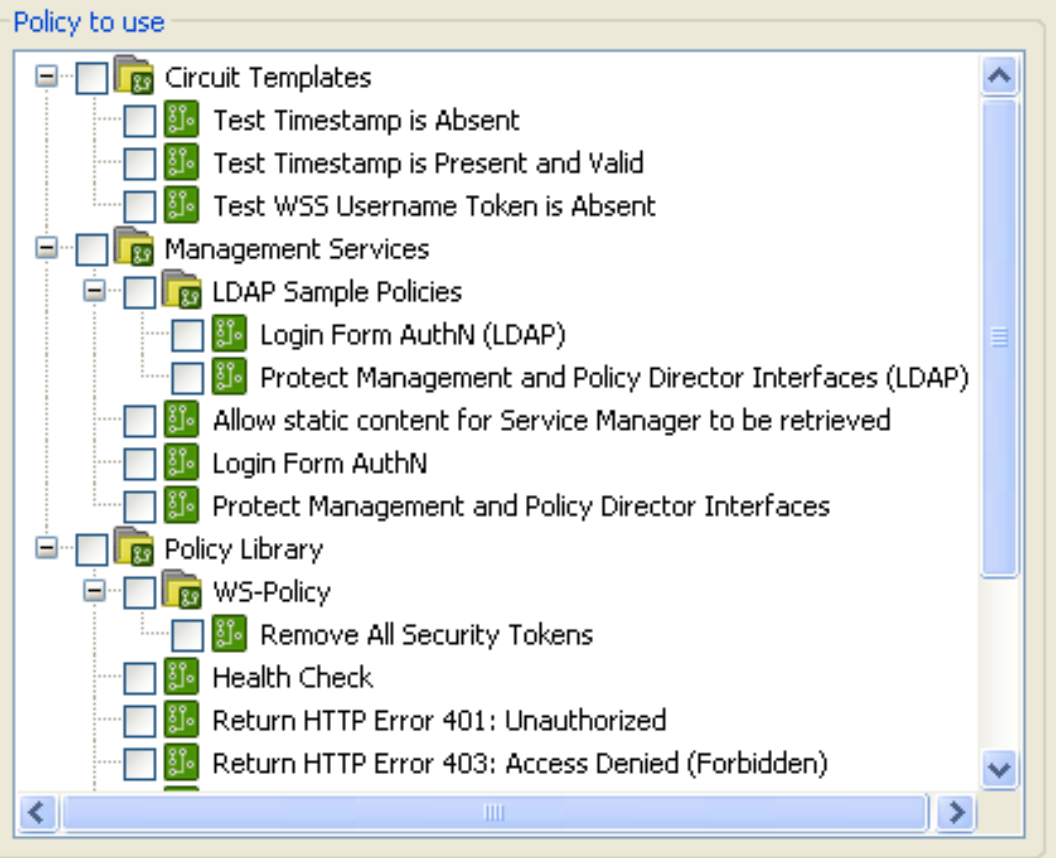
Sample XML

```
<ui>
 <panel columns="2">
  <group label="POP_POLICY_LABEL" span="1" columns="1">
   <SoftRefTreeAttribute field="filterCircuit"
    searches="/[CircuitContainer]**/[FilterCircuit],/[FilterCircuit]"
    selectableType="FilterCircuit"
    displayName="POP_CIRCUIT_DISP_NAME"
    height="80"
    width="100"
   />
  </group>
```

```
 </panel>
</ui>
```

Rendered UI

The above XML renders the following UI:



In this case, POP_POLICY_LABEL is resolved to the localized string "Policy to use".

SpinAttribute

Description

The <SpinAttribute> tag renders an SWT Text widget (and optionally, a Label widget), along with two buttons, one for incrementing the current entity value, and one for decrementing the value.

Available attributes

Attribute	Description	M/O	Default
field	Specifies the name of the field of the entity backed by the rendered controls. The default value of the field will automatically appear in the Text widget.	M	-
label	Specifies the ID of the resource containing the text to display on the Label (to the left of the spin control).	O	-
required	Specifies whether or not the field is required. If required and the user doesn't enter a value, a warning dialog will appear, prompting the user to enter a value for the field.	O	-
rangeMin	Specifies the minimum value for the permitted range for the entity value. rangeMax must also be present for the range to be set correctly.	C	-
rangeMax	Specifies the maximum value for the permitted range for the entity value. rangeMin must also be present for the range to be set correctly.	C	
unitLabel	Additional label to appear to the right of the spin control. Specifies the ID of the resource containing the text to display on the Label.	O	-

Sample XML

```
<ui>
 <panel columns="7">
  <SpinAttribute field="hrs" label="HOUR" required="true"
   rangeMin="0" rangeMax="23" />
  <SpinAttribute field="mins" label="MIN" required="true"
   rangeMin="0" rangeMax="59" />
  <SpinAttribute field="secs" label="SEC" required="true"
   rangeMin="0" rangeMax="59" />
 </panel>
</ui>
```

Rendered UI

The above XML renders the following UI:

Elements T to Z

tab

Description

The <tab> tag renders an SWT TabItem widget. They must be direct children of the <tabFolder>.

Available attributes

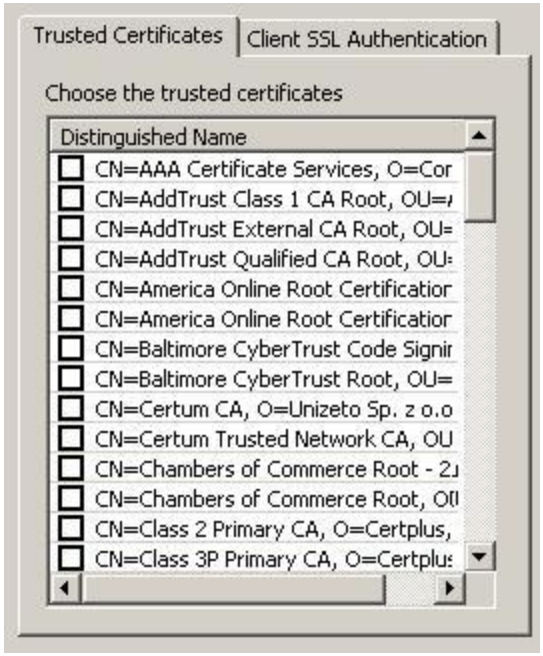
Attribute	Description	M/O	Default
label	Specifies the ID of the resource containing the text to display on the TabItem.	M	-

Sample XML

```
<ui>
 <panel columns="2" span="2">
  <tabFolder span="2">
   <tab label="CERTS">
   <tab label="SSL">
 </panel>
</ui>
```

Rendered UI

The above XML renders the following UI:



tabFolder

Description

The <tabFolder> tag renders an SWT TabFolder widget used to house TabItems (generated by using <tab> tags as children).

Available attributes

Attribute	Description	M/O	Default

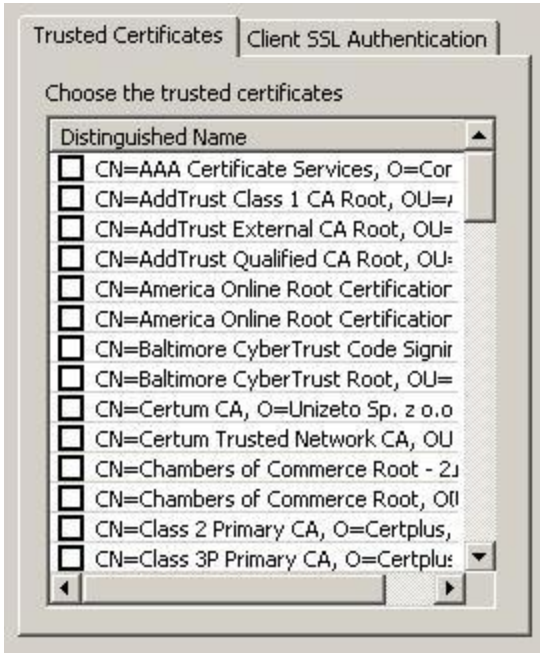
span	Value used in the creation of layout data for the controls that are rendered.
	`GridData gridData = new GridData();`
	`gridData.horizontalAlignment = GridData.FILL;`
	`gridData.verticalAlignment = GridData.FILL;`
	`gridData.grabExcessHorizontalSpace = true;`
	`gridData.grabExcessVerticalSpace = true;`
	`gridData.horizontalSpan = span;`
		O	1

Sample XML

```
<ui>
 <panel columns="2" span="2">
  <tabFolder span="2">
   <tab label="CERTS">
   <tab label="SSL">
 </panel>
</ui>
```

Rendered UI

The above XML renders the following UI:



TablePage

Description

The <TablePage> tag renders a jFace TableViewer that represents a list of entities of the specified type. These entities can be added, edited, and deleted using the available buttons.

Available attributes

Attribute	Description	M/O	Default
type	Specifies the type of the entity that is represented by the table.	M	-
dialogClass	Specifies the full name of the Java class used to implement the dialog employed to add/edit items in the table.	M	-
columnProperties	Specifies a comma-separated list of entity fields to be displayed in table columns.	M	-
sortColumns	Specifies a comma-separated list of columns to make sortable.	M	-
columnResources	Specifies a comma-separated list of resource IDs (from a `resource.properties` file) that resolve to the text to be displayed on the table columns.	M	-
columnWidths	Specifies a comma-separated list of column widths.	M	-
deleteText	Specifies the resource ID that resolves to the text to be displayed to the user when an item from the table is about to be deleted.	M	-

span	Value used in the creation of layout data for the controls that are rendered.
	`GridData gridData = new GridData();`
	`gridData.horizontalAlignment = GridData.FILL;`
	`gridData.verticalAlignment = GridData.FILL;`
	`gridData.grabExcessHorizontalSpace = true;`
	`gridData.grabExcessVerticalSpace = true;`
	`gridData.horizontalSpan = span;`
		O	1

dialog.setFlavor	Specifies a declarative XML file to use to construct the dialog used to add or edit the entities for this table.	O	-
tableHeight	Specifies the preferred height of the control.	O	SWT.DEFAULT

Sample XML

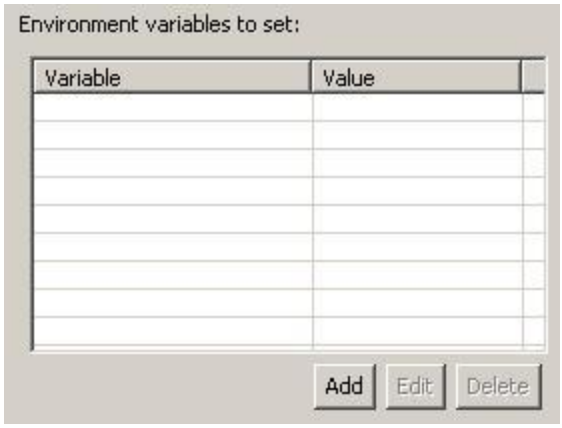
```
<ui>
 <panel columns="2">
  <TablePage type="Property"
   dialogClass="com.vordel.client.manager.filter.exec.EnvironmentVariableDialog"
   columnProperties="name,value" sortColumns="name,value"
   columnResources="COLUMN_VARIABLE,COLUMN_VALUE" columnWidths="300,200"
   deleteText="DELETE_VARIABLE_CONFIRMATION"
   dialog.setFlavor="environment_variable_dialog.xml" />
 </panel>
</ui>
```

Sample dialog flavor XML

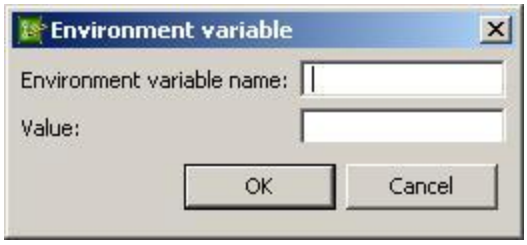
```
<panel columns="2">
 <TextAttribute field="name" label="NAME_LABEL" required="true"/>
 <TextAttribute field="value" label="VALUE_LABEL" required="false"/>
</panel>
```

Rendered UI

The above XML renders the following UI:



The dialog is rendered as follows:



text

Description

The <text> tag renders an SWT Text widget.

Available attributes

Attribute	Description	M/O	Default
label	Specifies the ID of the resource containing the text to display on the Label.	O	-
multiline	Specifies whether the control is a multiline Text widget. If this attribute is not present the control defaults to a single-line widget.	O	false
wrap	Specifies whether the text should be wrapped within the control. This attribute is conditional on the multiline attribute being present and set to true.	C	false
readOnly	Specifies whether or not the Text widget is read-only.	O	-

span	Value used in the creation of layout data for the controls that are rendered.
	If a single-line control is being rendered, span represents the horizontal span of the following GridData:
	`GridData gridData = new GridData();`
	`gridData.horizontalAlignment = GridData.FILL;`
	`gridData.grabExcessHorizontalSpace = true;`
	`gridData.horizontalSpan = colSpan;`
	If a multiline control is being rendered, span represents the horizontal span of the following GridData:
	`GridData gridData = new GridData();`
	`gridData.horizontalAlignment = GridData.FILL;`
	`gridData.verticalAlignment = GridData.FILL;`
	`gridData.grabExcessHorizontalSpace = true;`
	`gridData.grabExcessVerticalSpace = true;`
	`gridData.horizontalSpan = colSpan;`
		O	1

widthHint	Specifies the preferred width of the control.	O	SWT.DEFAULT
heightHint	Specifies the preferred height of the control.	O	SWT.DEFAULT

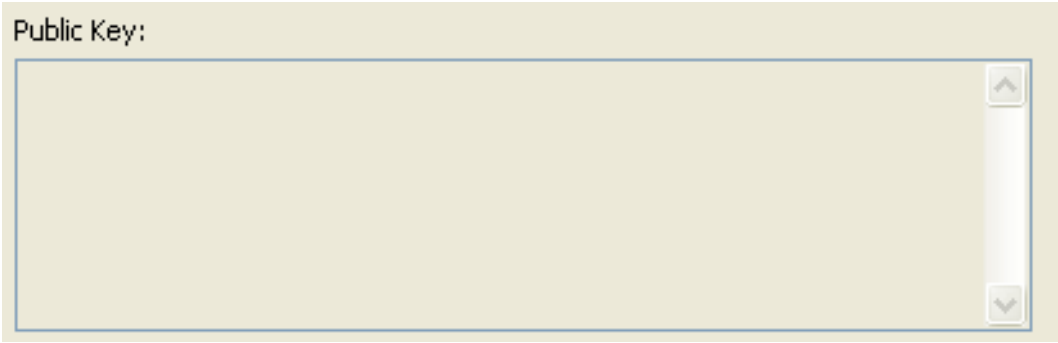
Attribute	Description	M/O	Default
vscroll	Specifies whether a vertical scroll bar should be rendered. This attribute is conditional on the multiline attribute being present and set to true.	C	-
hscroll	Specifies whether a horizontal scroll bar should be rendered. This attribute is conditional on the multiline attribute being present and set to true.	C	-

Sample XML

```
<ui>
 <text label="KEYPAIRS_PUBLICKEY_LABEL" multiline="true" vscroll="true"
  wrap="true" heightHint="100" widthHint="350" readOnly="true"/>
</ui>
```

Rendered UI

The above XML renders the following UI:



TextAttribute

Description

The <TextAttribute> tag renders an SWT Text widget (and optionally, a Label widget), backed by the specified field for the entity being configured.

Available attributes

Attribute	Description	M/O	Default
field	Specifies the name of the field of the entity backed by the rendered controls. The default value of the field will automatically appear in the Text widget.	M	-
label	Indicates that a Label should be rendered to the left of the Text widget. The value of this field is set to a resource identifier, specified in a `resources.properties` file.	O	-
readOnly	Specifies whether or not the Text widget is read-only.	O	-
required	Specifies whether or not the field is required. If required and the user does not enter a value, a warning dialog appears, prompting the user to enter a value for the field.	O	-

span	Value used in the creation of layout data for the controls that are rendered.
	If a single-line control is being rendered, span represents the horizontal span of the following GridData:
	`GridData gridData = new GridData();`
	`gridData.horizontalAlignment = GridData.FILL;`
	`gridData.grabExcessHorizontalSpace = true;`
	`gridData.horizontalSpan = colSpan;`
	If multiline control is being rendered, span represents the horizontal span of the following GridData:
	`GridData gridData = new GridData();`
	`gridData.horizontalAlignment = GridData.FILL;`
	`gridData.verticalAlignment = GridData.FILL;`
	`gridData.grabExcessHorizontalSpace = true;`
	`gridData.grabExcessVerticalSpace = true;`
	`gridData.horizontalSpan = colSpan;`
		O	1

widthHint	Specifies the preferred width of the control.	O	SWT.DEFAULT
heightHint	Specifies the preferred height of the control.	O	SWT.DEFAULT
multiline	Specifies whether the control is a multiline Text widget. If this attribute is not present the control defaults to a single-line widget.	O	false

Attribute	Description	M/O	Default
wrap	Specifies whether the text should be wrapped within the control. This attribute is conditional on the multiline attribute being present and set to true.	C	false
vscroll	Specifies whether a vertical scroll bar should be rendered. This attribute is conditional on the multiline attribute being present and set to true.	C	false
hscroll	Specifies whether the a horizontal scrollbar should be rendered. This attribute is conditional on the multiline attribute being present and set to true.	C	false

Sample XML

```
<ui>
 <panel columns="2">
  <TextAttribute field="name" label="EXCEPTION_NAME" required="true" />
 </panel>
</ui>
```

Rendered UI

The above XML renders the following UI:

In this case, EXCEPTION_NAME is resolved to the localized string "Name:".

ui

Description

The <ui> tag is the root of a declarative XML document.

Available attributes

This tag does not require any attributes.

Sample XML

```
<ui>
 <panel columns="2">
  <TextAttribute field="name" label="EXCEPTION_NAME" required="true" />
 </panel>
</ui>
```

validator

Description

The <validator> tag is used to include a validator class.

Available attributes

Attribute	Description	M/O	Default
class	Specifies the full name of the Java class used to validate input.	M	-

Sample XML

```
<ui>
 <validator
class="com.vordel.client.manager.filter.dirscan.DirectoryScannerDialogValidator" />
</ui>
```

XPathAttribute

Description

The <XPathAttribute> tag renders an SWT Combo widget and three SWT Button widgets within an SWT Composite displayed as a Button bar.

Available attributes

Attribute	Description	M/O	Default
field	Specifies the name of the field of the entity backed by the rendered controls.	M	-
xpathGroup	Specifies the entity type of all XPath expressions to be displayed.	M	-
label	Specifies the ID of the resource containing the text to display on the Label (to the left of the combo box).	O	-
required	Specifies whether or not the entity field is required.	O	false

span	Value used in the creation of layout data for the Button.
	Span represents the horizontal span of the following GridData:
	`GridData gridData = new GridData();`
	`gridData.horizontalAlignment = GridData.FILL;`
	`gridData.grabExcessHorizontalSpace = true;`
	`gridData.horizontalSpan = span;`
		O	1

Sample XML

```
<ui>
 <XPathAttribute field="insertTokenLocationXPath"
  xpathGroup="XPathTokenInsertionLocationGroup" />
</ui>
```

Rendered UI

The above XML renders the following UI: