Red Hat JBoss Fuse 6.3

SwitchYard Development Guide

Develop applications with SwitchYard

Last Updated: 2020-10-27

Red Hat JBoss Fuse 6.3 SwitchYard Development Guide

Develop applications with SwitchYard

JBoss A-MQ Docs Team
Content Services
fuse-docs-support@redhat.com

Legal Notice

Abstract

Use this guide to help you develop integrated applications with SwitchYard.

Table of Contents

CHAPTER 1. READ ME

1.1. BACK UP YOUR DATA

WARNING

Red Hat recommends that you back up your system settings and data before undertaking any of the configuration tasks mentioned in this book.

1.2. RED HAT DOCUMENTATION SITE

Red Hat's official documentation site is at https://access.redhat.com/site/documentation/. There you will find the latest version of every book, including this one.

1.3. EAP_HOME

EAP_HOME refers to the root directory of the Red Hat JBoss Enterprise Application Platform installation on which JBoss Fuse is deployed.

1.4. MODE

NOTE

Please note that Red Hat JBoss Fuse 6.0 does not support Red Hat JBoss Enterprise Application Platform run in Domain mode.

MODE is either **standalone** or **domain** depending on whether your instance of JBoss Enterprise Application Platform is running in standalone or domain mode. Substitute one of these whenever you see **MODE** in a file path in this documentation.

CHAPTER 2. JBOSS INTEGRATION AND SOA DEVELOPMENT

2.1. JBOSS INTEGRATION AND SOA DEVELOPMENT

The **JBoss Integration and SOA Development** plug-in is provided to support JBoss Fuse in JBoss Developer Studio. It provides the following features:

- Creation of SwitchYard projects
- Adding SwitchYard capabilities to existing Maven based JBoss Developer Studio projects
- Configuration of SwitchYard capabilities
- A graphical editor for editing SwitchYard application configuration
- Java2WSDL
- XML catalog entries for SwitchYard configuration schema
- Integration supporting the SwitchYard Maven plug-in (org.switchyard:switchyard-plugin)
- Support for workspace deployment of SwitchYard projects

The JBoss Integration and SOA Development plug-in is provided by the JBoss Development Studio Integration Stack.

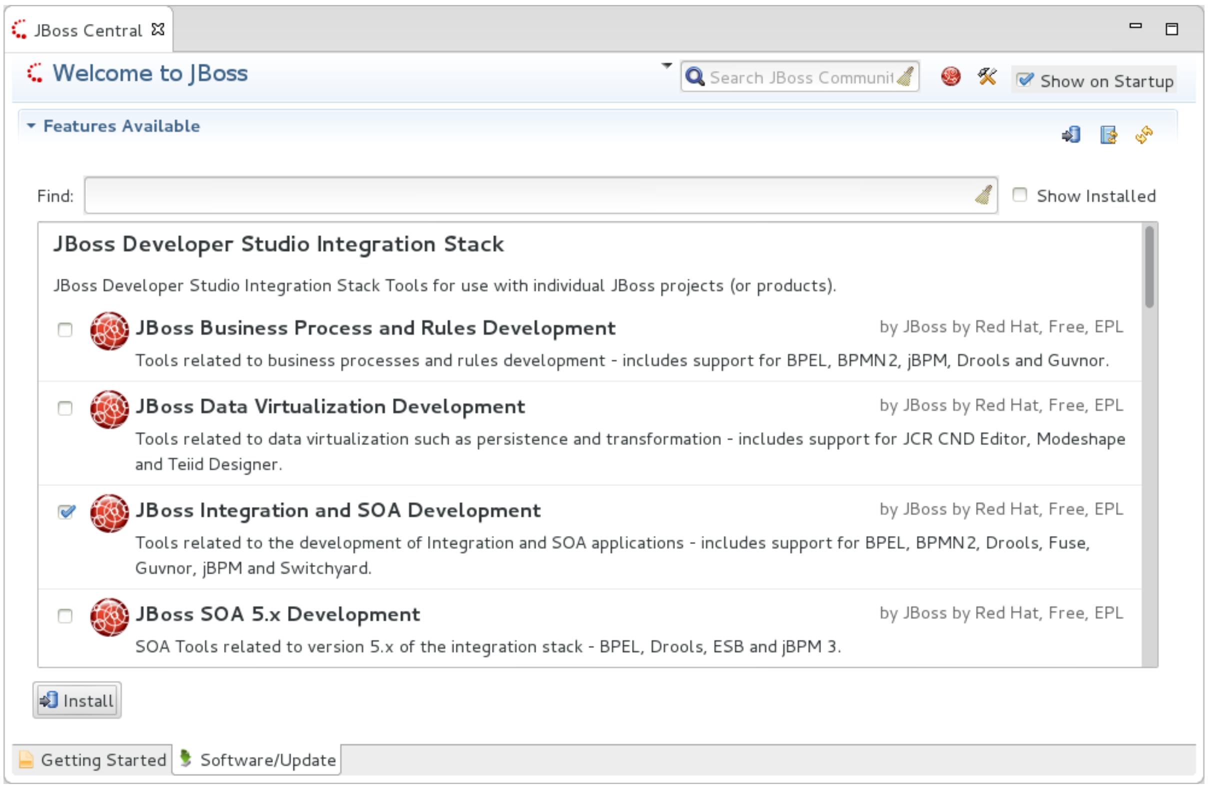
2.2. INSTALLING JBOSS DEVELOPER STUDIO INTEGRATION STACK

JBoss Developer Studio Integration Stack is not packaged as part of JBoss Developer Studio installations. These plug-ins must be installed independently through JBoss Central, as detailed in the procedure below.

Procedure 2.1. Install JBoss Developer Studio Integration Stack

1. Start JBoss Developer Studio.
2. In JBoss Central, select the **Software/Update** tab. Scroll through the list to locate **JBoss Developer Studio Integration Stack**. Select the check box next to **JBoss Integration and SOA Development** and click **Install**.

Figure 2.1. Find JBoss Developer Studio Integration Stack in JBoss Central Software/Update Tab



JBoss Developer Studio Integration Stack

JBoss Developer Studio Integration Stack Tools for use with individual JBoss projects (or products).

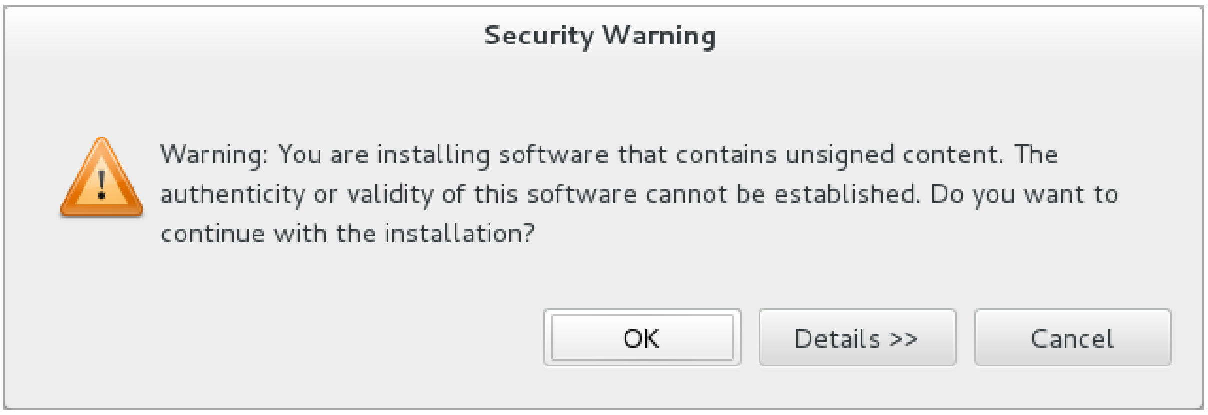
- [ ] JBoss Business Process and Rules Development — by JBoss by Red Hat, Free, EPL
  Tools related to business processes and rules development - includes support for BPEL, BPMN2, jBPM, Drools and Guvnor.
- [ ] JBoss Data Virtualization Development — by JBoss by Red Hat, Free, EPL
  Tools related to data virtualization such as persistence and transformation - includes support for JCR CND Editor, Modeshape and Teiid Designer.
- [✓] JBoss Integration and SOA Development — by JBoss by Red Hat, Free, EPL
  Tools related to the development of Integration and SOA applications - includes support for BPEL, BPMN2, Drools, Fuse, Guvnor, jBPM and Switchyard.
- [ ] JBoss SOA 5.x Development — by JBoss by Red Hat, Free, EPL
  SOA Tools related to version 5.x of the integration stack - BPEL, Drools, ESB and jBPM 3.

3. In the **Install** wizard, ensure the check boxes are selected for the software you want to install and click **Next**. It is recommended that you install all of the selected components.

4. Review the details of the items listed for install and click **Next**. After reading and agreeing to the license(s), click **I accept the terms of the license agreement(s)** and click **Finish**. The **Installing Software** window opens and reports the progress of the installation.

5. During the installation process you may receive warnings about installing unsigned content. If this is the case, check the details of the content and if satisfied click **OK** to continue with the installation.

Figure 2.2. Warning Prompt for Installing Unsigned Content



Security Warning

Warning: You are installing software that contains unsigned content. The authenticity or validity of this software cannot be established. Do you want to continue with the installation?

6. Once installing is complete, you are prompted to restart the IDE. Click **Yes** to restart now and **No** if you need to save any unsaved changes to open projects. Note that changes do not take effect until the IDE is restarted.

Once installed, you may need to complete additional configuration actions before you can use the individual JBoss Developer Studio Integration Stack components. For plug-in specific configuration information, see the appropriate Red Hat JBoss product documentation available from https://access.redhat.com/site/documentation on the Red Hat Customer Portal.

IMPORTANT

The installation method for early releases of JBoss Developer Studio Integration Stack may vary from that given here. For instructions, see the appropriate Red Hat JBoss product documentation available from https://access.redhat.com/site/documentation on the Red Hat Customer Portal.

2.3. HELPFUL TIPS

Honor all XML schema locations

After installation, go to **XML → XML Files → Validation → Preferences** and ensure **Honor all XML schema locations** check box is cleared. This prevents erroneous XML validation errors from appearing on **switchyard.xml** files.

DTD warning

Importing SwitchYard quickstarts into JBoss Developer Studio results in non-fatal warnings regarding **log4j.dtd**. These can be safely ignored. To stop receiving the warning, ensure the file **log4j.xml** is closed before starting a project.

JavaSE-1.6 error message

When commencing a project, a warning may be displayed saying "Build path specifies execution environment JavaSE-1.6". To disable this warning, go to your Java preferences and ensure that JavaSE-1.7 JDK is checked to support JavaSE-1.6 environments.

2.4. RUNNING QUICKSTARTS FROM JBOSS DEVELOPER STUDIO

Overview

This topic demonstrates how to import a quickstart application to JBoss Developer Studio and then deploy it to a running application server.

Prerequisites

The JBoss Integration and SOA Development tools must be installed from the JBoss Developer Studio Integration Stack.

1. Open JBoss Developer Studio.
2. Click **File → Import → Maven → Existing Maven Projects**
3. Select **Browse** and navigate to the quickstart directory, for example, ***EAP_HOME*/quickstarts/switchyard/bean-service** and then select **OK**.

   The import tool scans the directory to locate the associated **pom.xml** file.
4. Click **Finish**.
5. The quickstart is listed in the Project Explorer view. You can expand the project to explore its contents.

6. In the Project Explorer view, right-click on the project's name and click **Run as → Run on server → EAP**.

Result

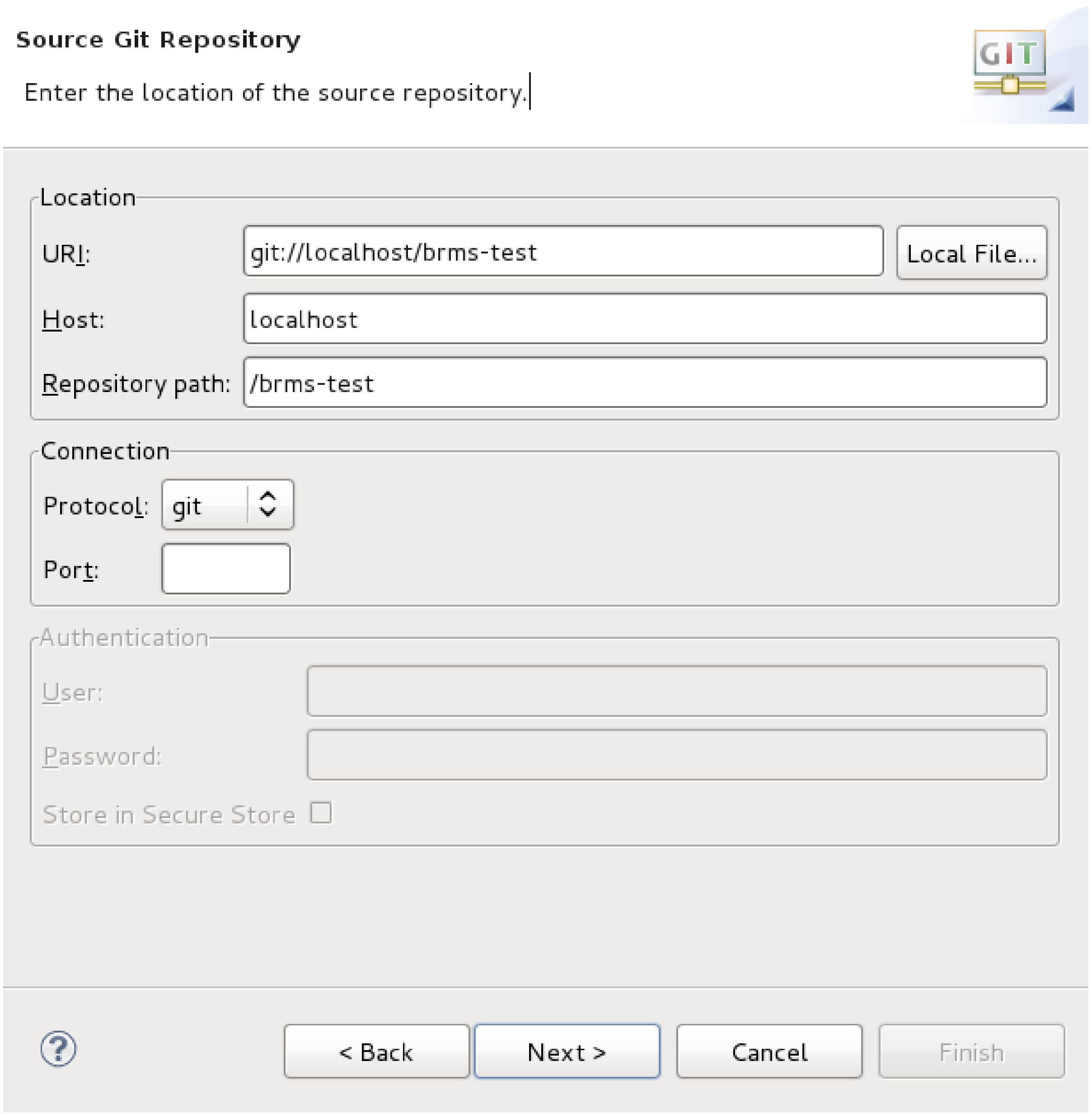
The quickstart application is deployed to the server and enabled by default.

2.5. IMPORT PROJECTS FROM A GIT REPOSITORY IN JBOSS DEVELOPER STUDIO

JBoss Developer Studio can be configured to connect to a central Git asset repository. The repository is where versions of rules, models, functions and processes are stored. This Git repository must already be defined by the KIE Workbench.

1. Start the Red Hat JBoss EAP server (if not already running) by selecting the server from the server tab and click the start icon.
2. Select **File → Import** and expand **Git**. Select **Projects from Git** and click Next.
3. Select the repository source as **URI** and click Next.
4. Enter the details of the Git repository in the next window and click Next.

Figure 2.3. Git Repository Details


Source Git Repository

Enter the location of the source repository.

Location

URI: git://localhost/brms-test

Host: localhost

Repository path: /brms-test

Connection

Protocol: git

Port:

Authentication

User:

Password:

Store in Secure Store ☐

5. Select which branch you want to import in the next window and click Next.

6. You are presented with the option to define the local storage for this project. Enter (or select) a non-empty directory, make any configuration changes and click Next.

7. Import the project as a general project in the next window and click Next. Name this project and click Finish.

2.6. SETTING A NEW RULES RUNTIME IN JBOSS DEVELOPER STUDIO

Setting this runtime provides an environment for new rules sets. It consists of a collection of jar files which are then utilized in rules creation. Once you have set up a runtime, you can go about adding and modifying rules in JBoss Developer Studio.

1. Download and unzip the runtime jars files located in the **jboss-brms-engine.zip** archive of the JBoss BRMS Deployable zip archive (available from Red Hat Customer Portal).

2. From the Red Hat JBoss Developer Studio menu, select **Window** and click **Preferences**.

3. Select **JBoss Rules → Installed JBoss Rules Runtimes → Runtime locations**.

4. Click **Add**; provide a name for the new runtime, and click **Browse** to navigate to the directory where the runtime is located.

5. Click **OK**, select the new runtime and click **OK** again. A dialog box indicates, if you have existing projects, that JBoss Developer Studio must be restarted to update the Runtime.

2.7. EDITING THE SWITCHYARD CONFIGURATION FILE

The JBoss Integration and SOA Development plug-in for JBoss Developer Studio provides a graphical editor for creating and maintaining your SwitchYard configuration file (**switchyard.xml**).

Assuming you have the plug-in installed, when you open a **switchyard.xml** file, by default it opens with the **SwitchYard Visual Multipage Editor**.

From here you can choose between three tabs (or views):

Design

This is the primary graphical interface for building your SwitchYard application. From here you can interact with and configure each of the application's entities, and add new entities from the **Palette**. The visual design is automatically converted into XML which you can view from the **Source** tab.

Domain

In this tab you can set additional configuration such as **Domain Properties** and **Security Configurations**. You can also enable message tracing from here.

Source

From the **Source** tab you can see the source XML which is generated automatically from the entities configured in the **Design** tab.

NOTE

Users cannot modify the **switchyard.xml** file directly from the **Source** tab.

Q: Why should I use the graphical editor?

A: The editor automatically manages dependencies for a project. For example, when you add a new binding or implementation to a composite, the editor adds the necessary info to **switchyard.xml** and the necessary Maven dependencies to the **pom.xml**. If you forget to update the **pom.xml**, the project fails to build (validate).

The editor automatically manages namespaces based on the features being used and the configuration level of the project.

The editor provides syntax and semantic validation, such as missing transformations and unused references.

Q: How can I modify the switchyard.xml file directly within JBoss Developer Studio?

A:

1. Navigate to the **src/main/resources/META-INF/switchyard.xml** file in the **Project Explorer** window.

2. Right-click on the file and select **Open With → XML Editor**.

IMPORTANT

Close the **switchyard.xml** file (as presented by the SwitchYard Visual Multipage Editor) before opening it with the XML Editor to avoid synchronization issues.

After completing your source edits, close the file and synchronize the model for the visual editor: right-click on the project in the **Project Explorer**, then select **Maven → Update Project**.

NOTE

The default editor for this file now is the XML Editor. To change it back to the graphical editor, right-click on the file and select **Open With → SwitchYard Visual Multipage Editor**.

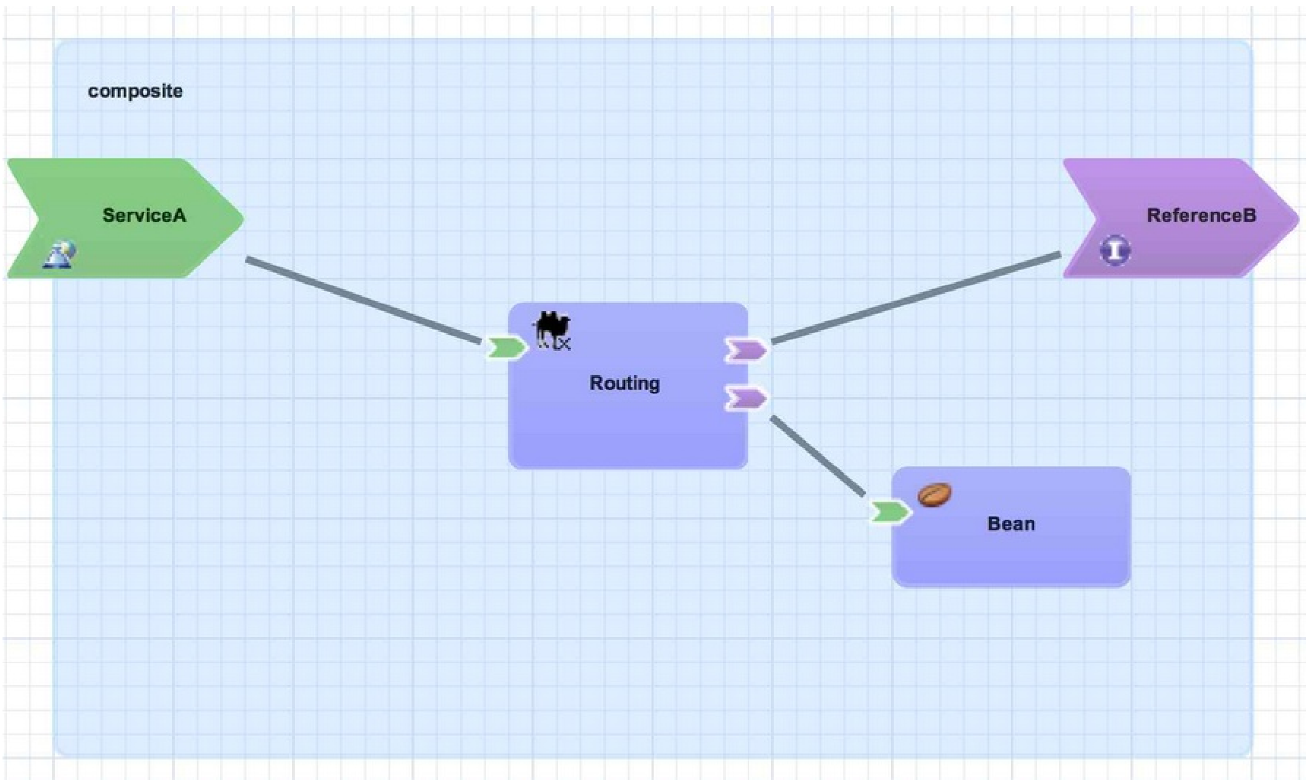
WARNING

Red Hat recommends using the graphical editor to prevent corruption of the **switchyard.xml** file. If the editor does not suit your needs, please consider submitting a request for enhancement.

CHAPTER 3. APPLICATION BASICS

This section introduces the basic building blocks of a SwitchYard application starting from an empty application and building up to the complete application as shown below:

Figure 3.1. Composite

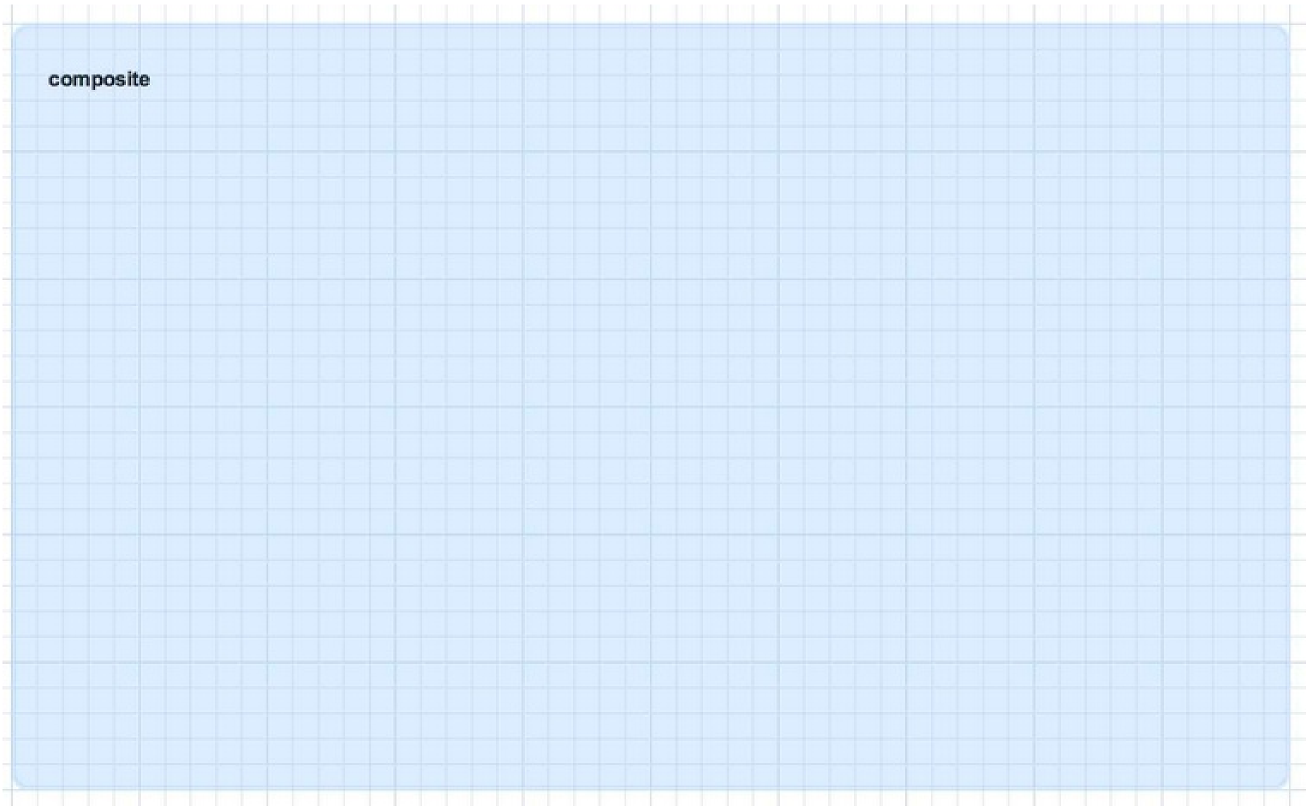


Each topic includes a visual representation of the **switchyard.xml** configuration file as designed in the SwitchYard graphical editor and the corresponding source XML which is automatically generated from the visual design.

3.1. COMPOSITE

A composite is displayed as a light blue rectangle and represents the boundary between what is inside your application and what is outside your application. A SwitchYard application consists of exactly one composite that has a **name** and a **targetNamespace**. The **targetNamespace** value is important as it allows names defined locally in the application (for example, service names) to be qualified and unique within a SwitchYard runtime.

Figure 3.2. Composite



Example 3.1. Sample Corresponding XML

```
<sca:composite name="example" targetNamespace="urn:example:switchyard:1.0">
</sca:composite>
```

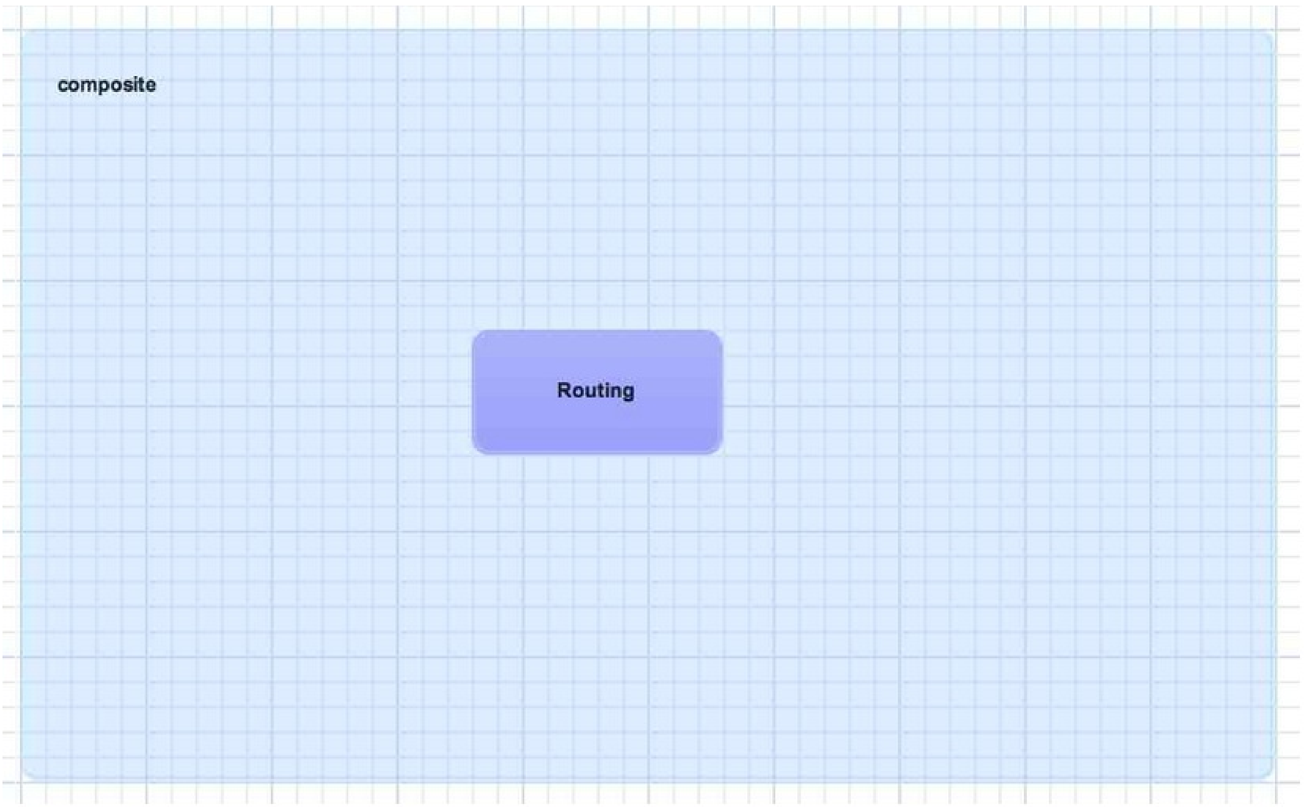
3.2. COMPONENT

A component is a modular container for application logic and consists of the following:

- 0 or 1 component service definitions
- 0 to many component reference definitions
- 1 implementation

Services and references allow a component to interact with other components, while the implementation provides the actual logic for providing or consuming services.

Figure 3.3. Component



Example 3.2. Sample Corresponding XML

```
<sca:component name="Routing">
</sca:component>
```

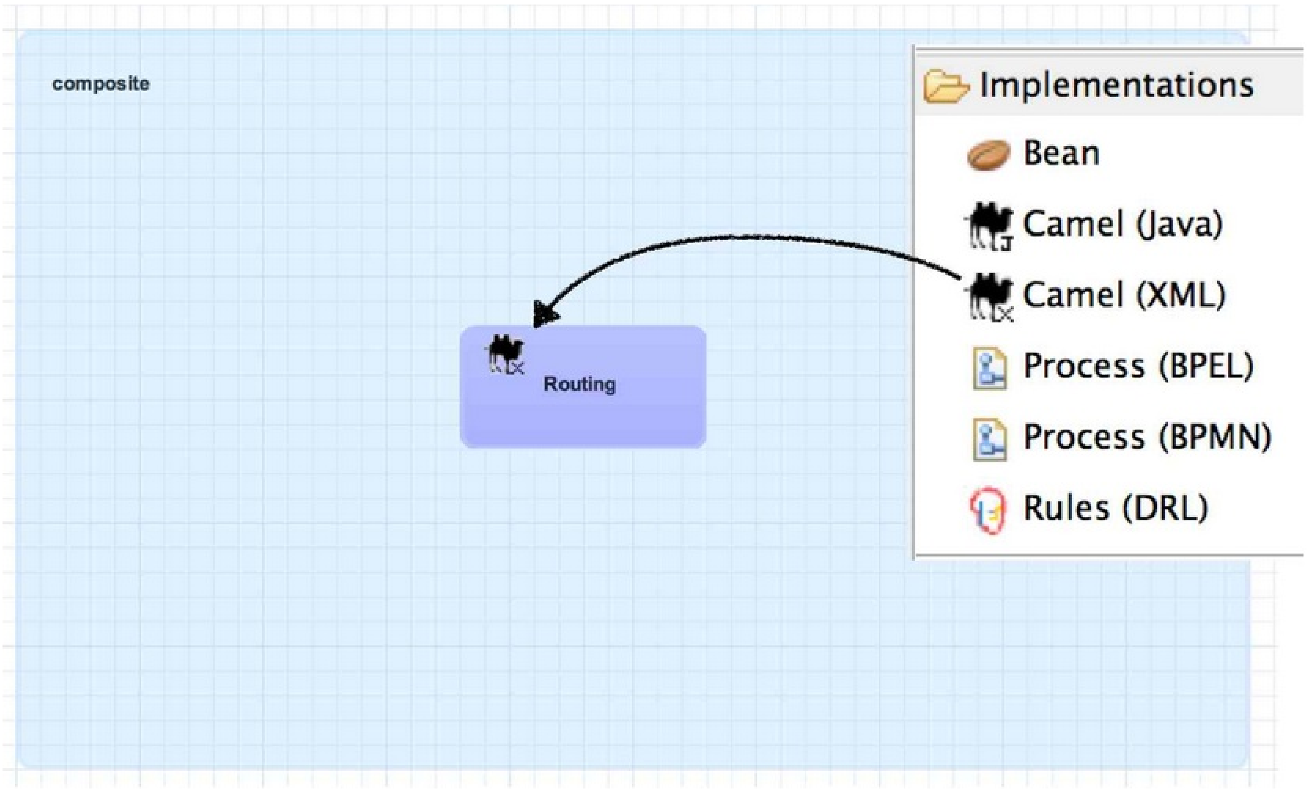
3.3. IMPLEMENTATION

An implementation acts as the brain of a service component and it is how implement your application logic. The following implementation options are available:

- Bean : allows a CDI Bean to consume or provide services using annotations
- Camel : EIP-style routing and service composition using the XML or Java DSL in Apache Camel
- BPMN 2 : service orchestration and human task integration expressed as BPMN 2 and executed using jBPM
- BPEL Process : web service orchestration using the OASIS Business Process Execution Language
- Rules : decision services based on Drools

Implementations are private to a component, which means external consumers and providers are not aware of the details of a component's implementation (implementation-hiding). All interactions with other components within an application and with external services are handled through component services and references.

Figure 3.4. Implementation



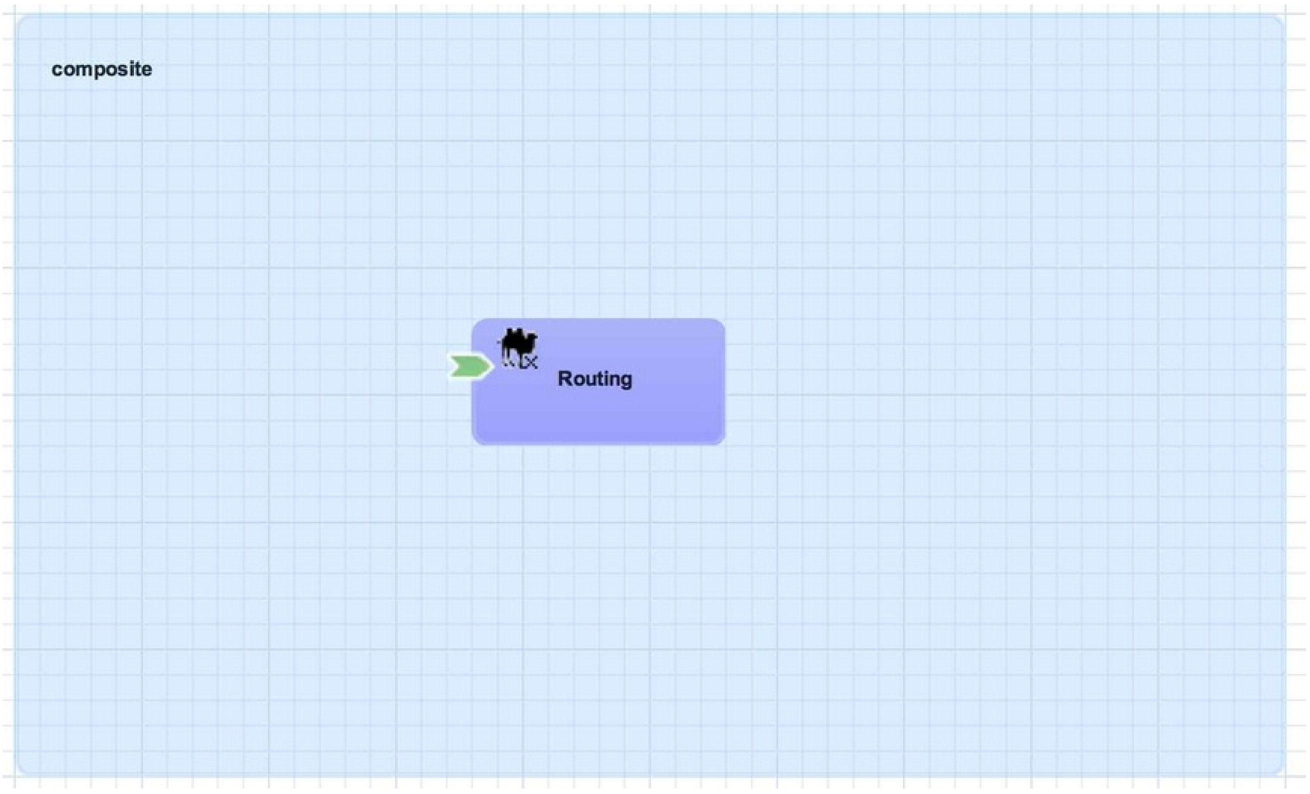
Example 3.3. Sample Corresponding XML

```
<sca:component name="Routing">
  <camel:implementation.camel>
    <camel:xml path="RoutingService.xml"/>
  </camel:implementation.camel>
</sca:component>
```

3.4. COMPONENT SERVICE

A component service is used to expose the functionality of an implementation as a service. All component services have a contract, which can be a Java interface, WSDL portType definition, or a set of named data types (interface.esb). Component services are private to an application, which means a component service can only be invoked by other components in the same application. In order to expose a component service to consumers external to the application, a component service can be promoted to a composite service. A component service can be promoted multiple times to create different composite services.

Figure 3.5. Component Service



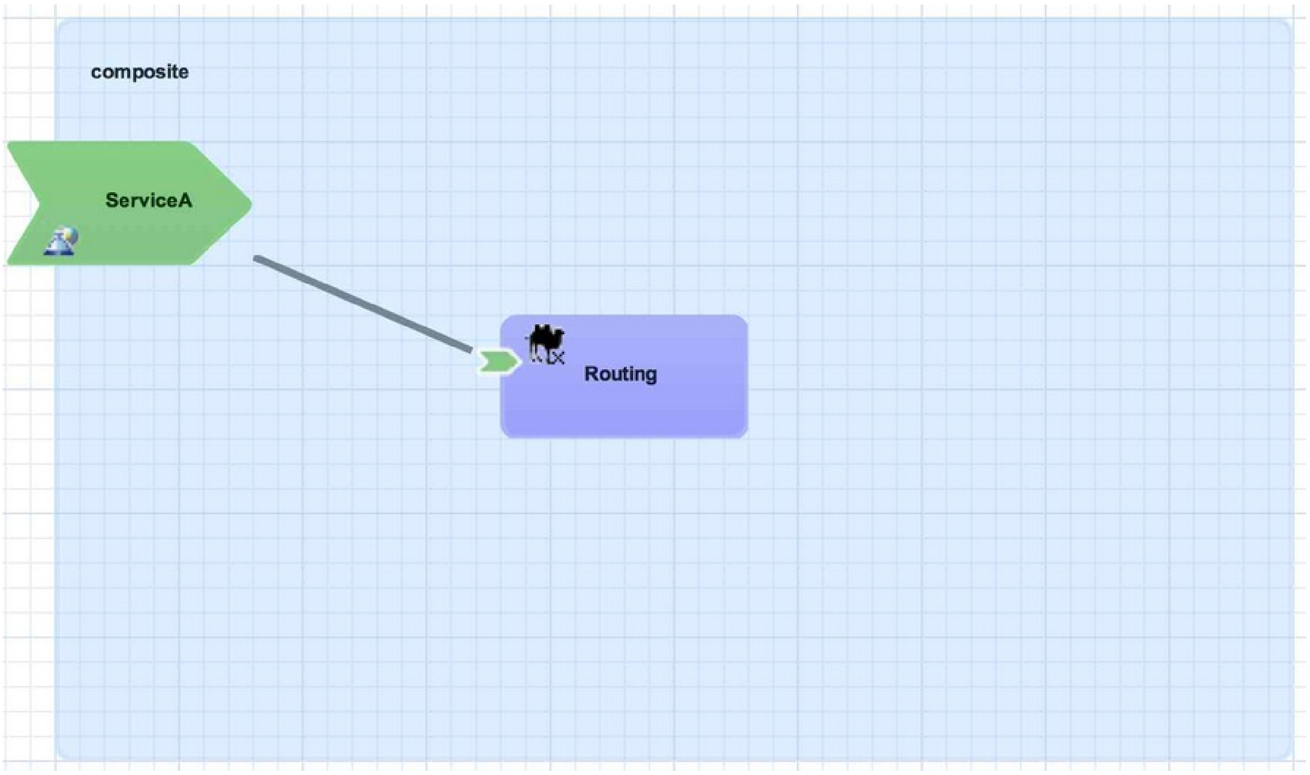
Example 3.4. Sample Corresponding XML

```
<sca:component name="Routing">
  <camel:implementation.camel>
    <camel:xml path="route.xml"/>
  </camel:implementation.camel>
  <sca:service name="ServiceA">
    <sca:interface.java interface="org.example.ServiceA"/>
  </sca:service>
</sca:component>
```

3.5. COMPOSITE SERVICE

A composite service represents an application service which is visible to other applications. A composite service can only be realized by promoting a component service within the application. The name and the interface of the composite service can be different from the component service. If the interface, or contract, of the composite service is different from the component service, be aware that a transformation may be required to map between the types defined in each interface. In our example application, the component service has a Java interface while the composite service has a WSDL interface. This means we must declare a transformer which maps between XML and Java to resolve the data type mismatch.

Figure 3.6. Composite Service



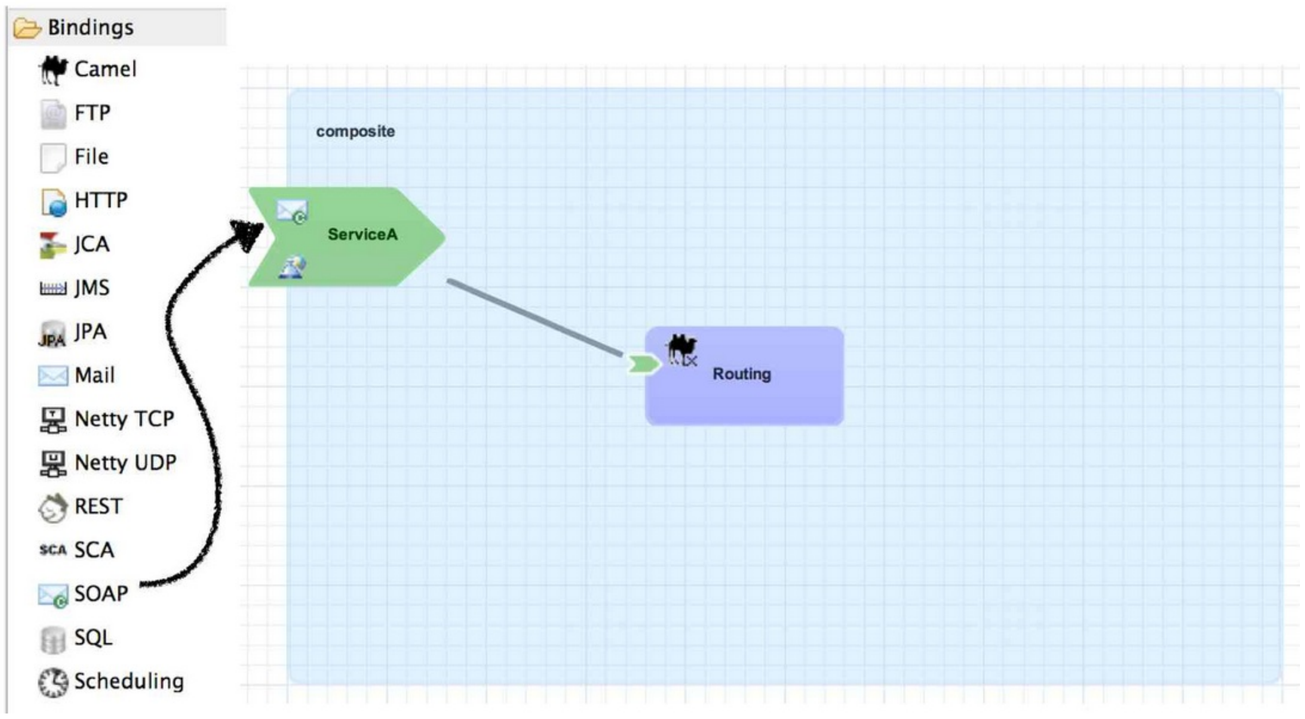
Example 3.5. Sample Corresponding XML

```
<sca:composite name="example" targetNamespace="urn:example:switchyard:1.0">
  <sca:service name="ServiceA" promote="Routing/ServiceA">
    <sca:interface.wsdl interface="ServiceA.wsdl#wsdl.porttype(ServiceAPortType)"/>
  </sca:service>
</sca:composite>
```

3.6. SERVICE BINDING

A service binding is used to define an access method for a composite service. Composite services can have multiple bindings, which allows a single service to be accessed in different ways. In most cases, a service binding represents a protocol or transport adapter (for example, SOAP, JMS, and REST). An important exception to this rule is the SCA binding, which allows services across applications in the same runtime to be wired together in memory. Regardless of the underlying binding details, a binding must always be used to facilitate inter-application communication in SwitchYard.

Figure 3.7. Service Binding



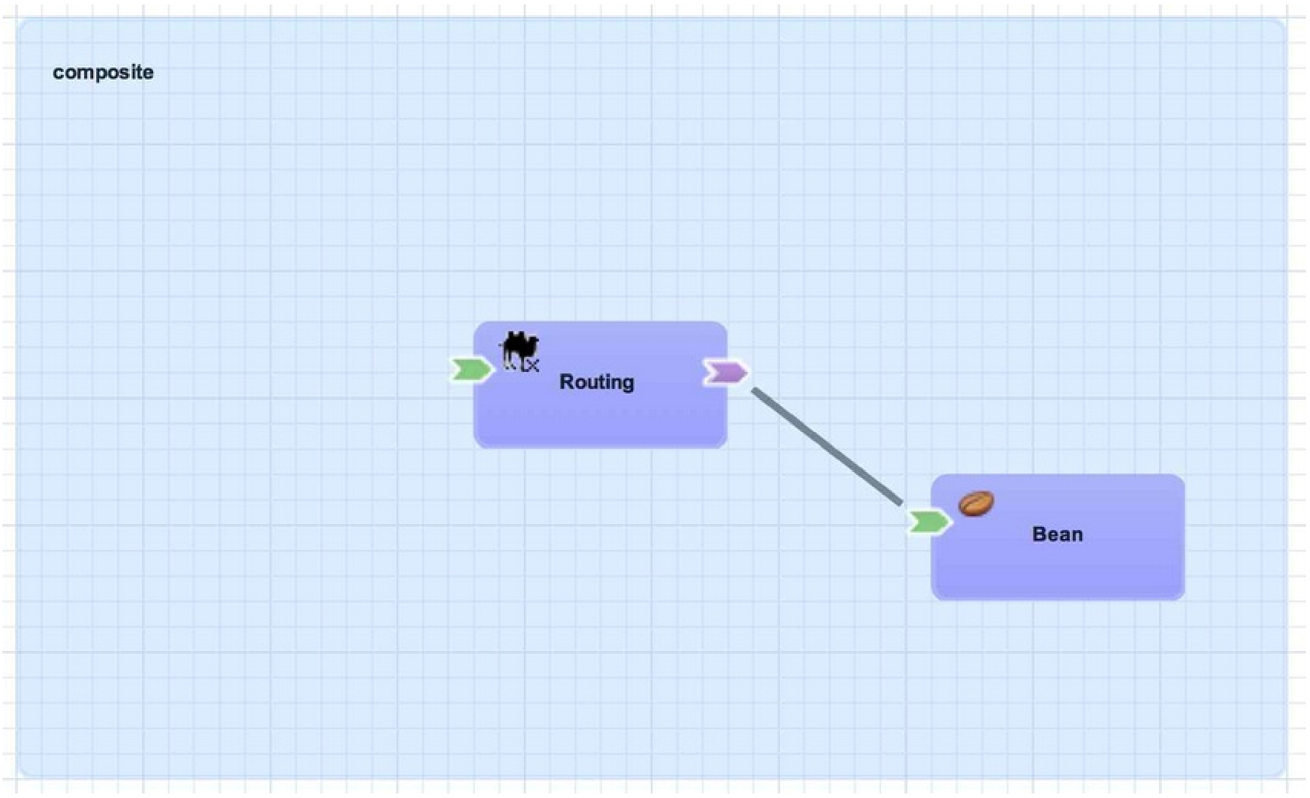
Example 3.6. Sample Corresponding XML

```
<sca:composite name="example" targetNamespace="urn:example:switchyard:1.0">
  <sca:service name="ServiceA" promote="Routing/ServiceA">
    <sca:interface.wsdl interface="ServiceA.wsdl#wsdl.porttype(ServiceAPortType)"/>
    <soap:binding.soap>
      <soap:wsdl>ServiceA.wsdl</soap:wsdl>
    </soap:binding.soap>
  </sca:service>
</sca:composite>
```

3.7. COMPONENT REFERENCE

A component reference allows a component to consume other services. A component reference can be wired to a service offered by another component in the same application or it can be wired to services outside the application with a composite reference. Similar to component services, all component references have a contract which allows a component to invoke services without knowing implementation or binding details. The picture below shows an example of wiring a reference on the Routing component to a service offered by the Bean component.

Figure 3.8. Component Reference



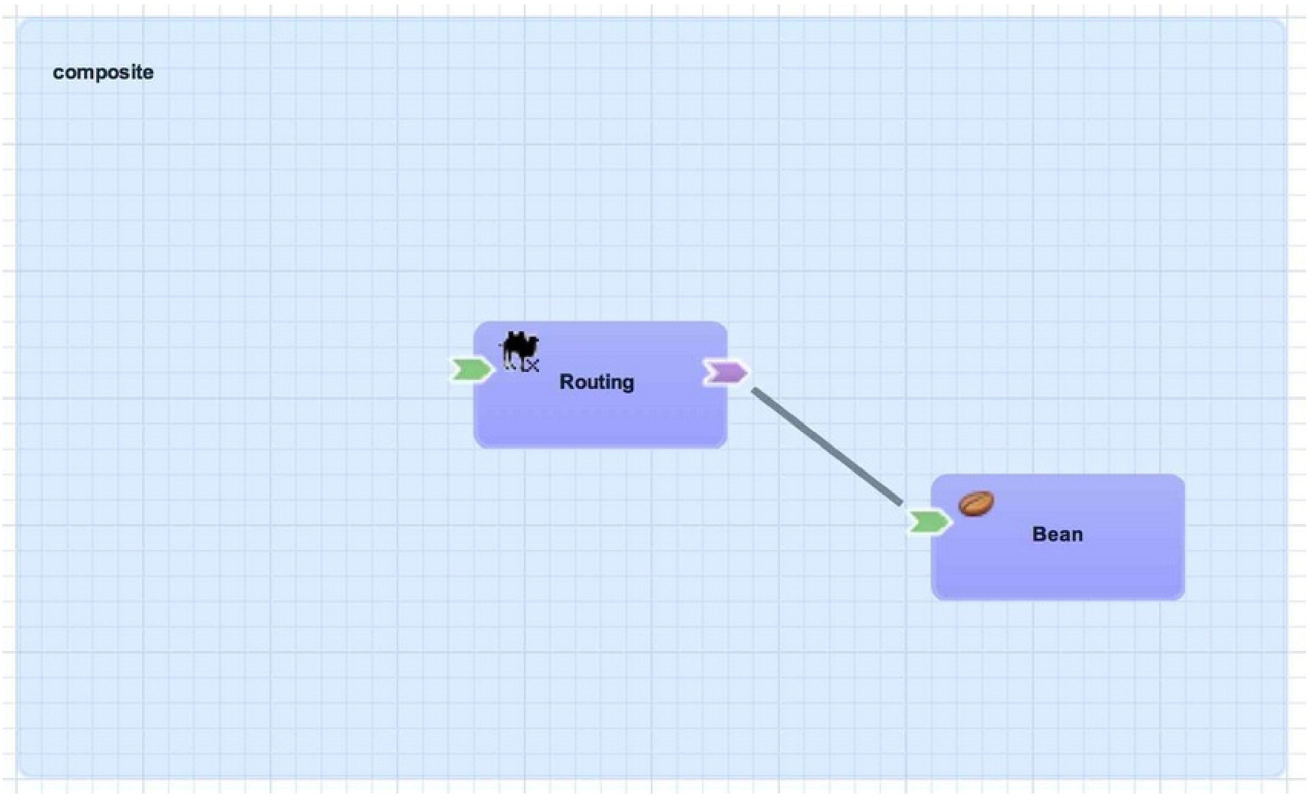
Example 3.7. Sample Corresponding XML

```
<sca:component name="Routing">
  <camel:implementation.camel>
    <camel:xml path="route.xml"/>
  </camel:implementation.camel>
  <sca:service name="ServiceA">
    <sca:interface.java interface="org.example.ServiceA"/>
  </sca:service>
  <sca:reference name="ServiceC">
    <sca:interface.java interface="org.example.ServiceC"/>
  </sca:reference>
</sca:component>
```

3.8. COMPOSITE REFERENCE

A composite reference allows a component reference to be wired to a service outside the application. Similar to composite services, bindings are used with composite references to specify the communication method for invoking the external service.

Figure 3.9. Composite Reference



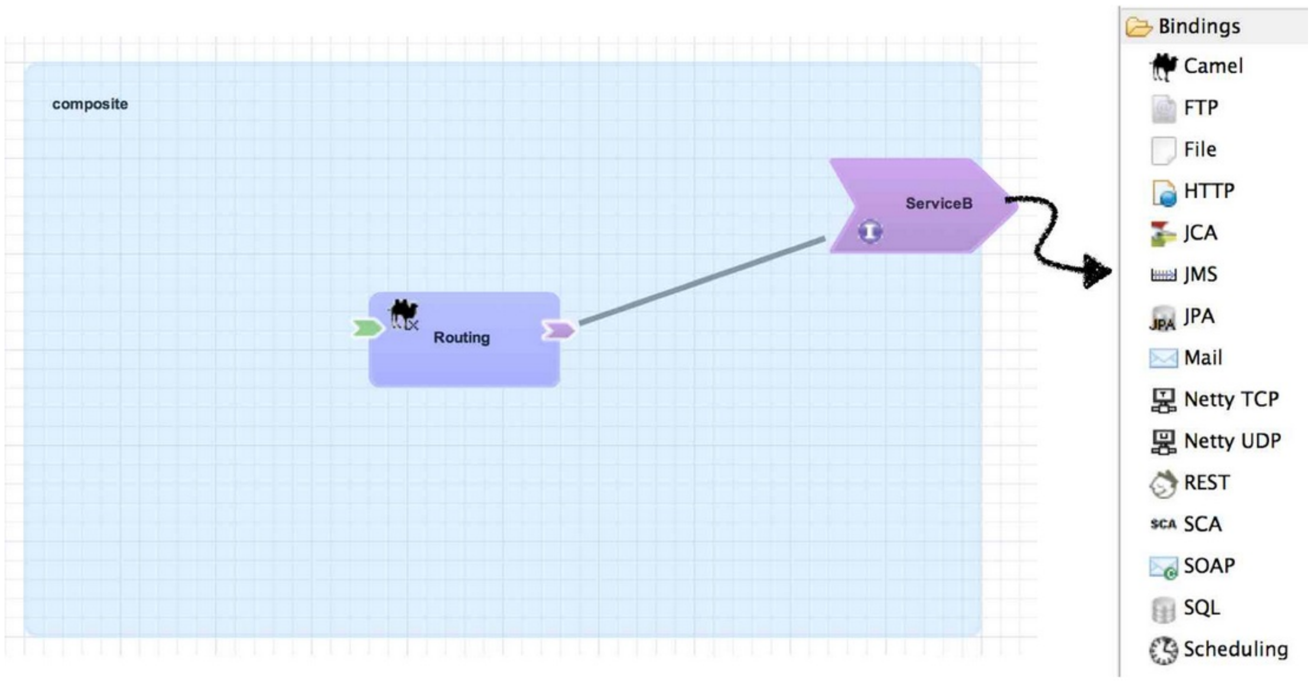
Example 3.8. Sample Corresponding XML

```
<sca:composite name="example" targetNamespace="urn:example:switchyard:1.0">
  <sca:reference name="ReferenceB" multiplicity="0..1" promote="Routing/ServiceB">
    <sca:interface.java interface="org.example.ServiceB"/>
  </sca:reference>
</sca:composite>
```

3.9. REFERENCE BINDINGS

A reference binding is used to define an access method for an external service with a composite reference. Unlike service bindings, there can only be one binding for each composite reference. The set of bindings available for references is identical to the set of bindings available for services, although the configuration values for a given binding may be different depending on whether it is used as a service binding or a reference binding.

Figure 3.10. Reference Bindings



Example 3.9. Sample Corresponding XML

```
<sca:composite name="example" targetNamespace="urn:example:switchyard:1.0">
  <sca:reference name="ReferenceB" multiplicity="0..1" promote="Routing/ServiceB">
    <sca:interface.java interface="org.example.ServiceB"/>
      <jms:binding.jms>
      <jms:queue>MyQueue</jms:queue>
      <jms:connectionFactory>#ConnectionFactory</jms:connectionFactory>
    </jms:binding.jms>
  </sca:reference>
</sca:composite>
```

NOTE

Although endpoint configuration can be static or dynamic (depending on the component), the endpoint reference itself is static. An example of a dynamic endpoint would be a file component configuration that maps properties (like file name) from the incoming binding and carries it to the outgoing reference. In this manner, the output file can be made to share the same name (in a different directory) as the incoming file that triggered the service invocation. The user did not have to specify a filename on the outgoing reference configuration - it was 'dynamic' and specified at runtime. Dynamic addressing for HTTP-based endpoints (such as SOAP, RESTEasy, and HTTP) is not currently available.

CHAPTER 4. SETTING UP THE SERVER

To make it possible for the tooling to manage a server, you need to add it to the Servers list. You can set up target runtime server using the **Servers** view. Once added, the server is displayed in the Servers view. Selecting a server in Servers view, enables you to start it, to stop it, or to delete its configuration. You can add multiple servers of the same type, as long as each uses a separate installation directory.

NOTE

If you have added a default runtime server when installing JBoss Developer Studio, you can see it in the **Servers** view.

4.1. ADD JBOSS EAP SERVER

1. In JBoss Developer Studio, click the **Servers** view. If the **Servers** view is not visible, click **Window → Show View → Servers**.

2. If no servers have been previously created then the Servers view displays a new server hyperlink. Click this link to create a new server.

   If there are one or more existing servers, right-click an existing server and click **New → Server**.

3. In the **Define a New Server** dialog, select a JBoss Enterprise Application Platform server from the **Select the server type** list.

4. The **Server's host name** and **Server name** fields are completed by default. In the **Server name** field, you can type a custom name by which to identify the server in the Servers view.

5. From the **Server runtime environment** list, select an existing server runtime environment for the application server type. Alternatively, to create a new runtime environment click **Add** and complete the fields and options as appropriate.

   NOTE

   If the Server runtime environment field is not shown, no server runtime environments exist for the selected application server type. To create a new server runtime environment without canceling the wizard, click **Next** and complete the fields and options as appropriate.

6. Click **Next**.

7. The server behavior options displayed vary depending on the selected application server type. To specify that the server life-cycle is to be managed from outside the IDE, select the **Server is externally managed** check box.

   To specify that the server should be launched to respond to requests on all hostnames, select the **Listen on all interfaces to allow remote web connections** check box.

   From the **location** list, select **Local**. Click **Next**.

NOTE

The Expose your management port as the server's hostname option, which enables management commands sent by the IDE to be successfully received by the server, is bypassed for local servers regardless of whether the check box is selected.

8. To select applications to deploy with this server, from the Available list select the applications and click **Add**. Applications to be deployed are detailed in the Configured list.

9. Click **Finish** to create the server.

Result

The new server appears in the list of servers in the **Servers** panel.

IMPORTANT

You can create multiple servers that use the same application server. But a warning is displayed if you try to simultaneously run more than one server on the same host. This is because multiple running servers on the same host can result in port conflicts.

CHAPTER 5. SWITCHYARD PROJECT

A SwitchYard project is a Maven based project with the following characteristics:

- a switchyard.xml file in the project's META-INF folder
- one or more SwitchYard runtime dependencies declared in the pom.xml file
- org.switchyard:switchyard-plugin Maven Old Java Object (MOJO) configured in the pom.xml file

A SwitchYard project may also contain a variety of other resources used to implement the application, for example: Java, BPMN2, DRL, BPEL, WSDL, XSD, and XML files.

The JBoss Developer Studio tooling supports the creation of new SwitchYard projects. The tooling also allows users to add SwitchYard capabilities to existing Maven projects.

5.1. CREATING A NEW SWITCHYARD PROJECT

Use the New SwitchYard Project wizard to create new SwitchYard project in your workspace.

The wizard creates a new project with the following structure:

- a **switchyard.xml** file in **src/main/resources/META-INF/** directory
- a **pom.xml** file declaring SwitchYard runtime dependencies and configuration for the switchyard-plugin MOJO
- a **beans.xml** file in **src/main/resources/META-INF/** directory
- a **beans.xml** file in **src/test/resources/META-INF/** directory
- a folder hierarchy for the specified Java package

Procedure 5.1. Create a New SwitchYard Project

1. Click **File → New → Project** to start the New Project wizard. Click **SwitchYard → SwitchYard Project** and then click **Next**.
2. Enter the name and location for the new project and then click **Next**.
3. Provide other details for the project.

   The library version can be specified directly using the Library Version field or indirectly by selecting a Target Runtime. Selecting a target runtime sets the library version to match the version provided by the target runtime.

   NOTE

   It is not necessary to select any components when creating the project. The tooling automatically configures components on the project as necessary.

4. Click **Finish** to create the new project.

The SwitchYard editor displays the **switchyard.xml** file for the newly created SwitchYard project after the wizard finishes.

5.2. IMPORTING EXISTING MAVEN PROJECT

You can import the existing Maven projects to your workspace.

Procedure 5.2. Import a Maven Project

1. Click **File → Import** to import the existing project. Click **Maven → Existing Maven Projects** and then click **Next**.
2. Click **Browse** and navigate to the directory where the project's **pom.xml** is located. Click **OK**.
3. The **pom.xml** file for the existing project is displayed in the Projects section. Click the check box to select the desired POM file.
4. Click **Finish** to start the importing process.

The SwitchYard editor displays the **switchyard.xml** file for newly created SwitchYard project after the wizard finishes.

5.3. ADDING SWITCHYARD CAPABILITIES TO EXISTING PROJECTS

SwitchYard capabilities may be added to existing projects in the workspace.

NOTE

SwitchYard capabilities can only be added to Maven projects.

Procedure 5.3. Add SwitchYard Capabilities to Existing Project

1. Right-click the project and click **SwitchYard → Configure Capabilities** to add (or modify) SwitchYard capabilities.
2. The **SwitchYard Settings** dialog is displayed. Modify the settings as required. Click **OK** to update the configuration.

   Changes are reflected in the **pom.xml** file and **switchyard.xml** file.

5.4. EDITING SWITCHYARD PROJECTS

5.4.1. Editing the SwitchYard Configuration File

The JBoss Integration and SOA Development plug-in for JBoss Developer Studio provides a graphical editor for creating and maintaining your SwitchYard configuration file (**switchyard.xml**).

Assuming you have the plug-in installed, when you open a **switchyard.xml** file, by default it opens with the **SwitchYard Visual Multipage Editor**.

From here you can choose between three tabs (or views):

Design

This is the primary graphical interface for building your SwitchYard application. From here you can interact with and configure each of the application's entities, and add new entities from the **Palette**. The visual design is automatically converted into XML which you can view from the **Source** tab.

Domain

In this tab you can set additional configuration such as **Domain Properties** and **Security Configurations**. You can also enable message tracing from here.

Source

From the **Source** tab you can see the source XML which is generated automatically from the entities configured in the **Design** tab.

NOTE

Users cannot modify the **switchyard.xml** file directly from the **Source** tab.

Q: **Why should I use the graphical editor?**

A: The editor automatically manages dependencies for a project. For example, when you add a new binding or implementation to a composite, the editor adds the necessary info to **switchyard.xml** and the necessary Maven dependencies to the **pom.xml**. If you forget to update the **pom.xml**, the project fails to build (validate).

The editor automatically manages namespaces based on the features being used and the configuration level of the project.

The editor provides syntax and semantic validation, such as missing transformations and unused references.

Q: **How can I modify the switchyard.xml file directly within JBoss Developer Studio?**

A:

1. Navigate to the **src/main/resources/META-INF/switchyard.xml** file in the **Project Explorer** window.

2. Right-click on the file and select **Open With → XML Editor**.

IMPORTANT

Close the **switchyard.xml** file (as presented by the SwitchYard Visual Multipage Editor) before opening it with the XML Editor to avoid synchronization issues.

After completing your source edits, close the file and synchronize the model for the visual editor: right-click on the project in the **Project Explorer**, then select **Maven → Update Project**.

NOTE

The default editor for this file now is the XML Editor. To change it back to the graphical editor, right-click on the file and select **Open With → SwitchYard Visual Multipage Editor**.

WARNING

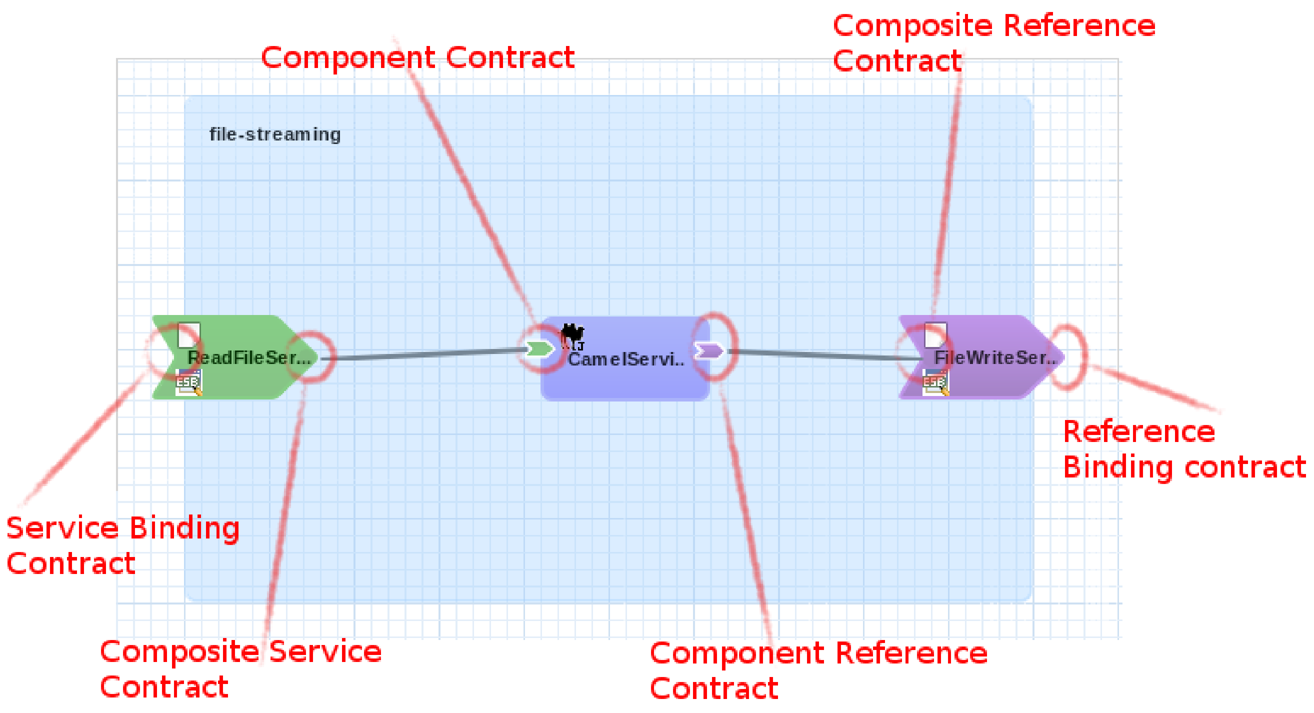
Red Hat recommends using the graphical editor to prevent corruption of the **switchyard.xml** file. If the editor does not suit your needs, please consider submitting a request for enhancement.

CHAPTER 6. SWITCHYARD CONTRACTS

6.1. SWITCHYARD CONTRACTS

SwitchYard uses a component service to expose the functionality of an implementation. All component services and implementations have service contracts. You can define a contract depending on Component Implementation and Service Binding.

A simple service has following contracts:


Component Contract
Composite Reference Contract
file-streaming
ReadFileSer...
CamelServi..
FileWriteSer..
Reference Binding contract
Service Binding Contract
Composite Service Contract
Component Reference Contract

Component Contracts

- Component Service
- Component Reference

Composite Contracts

- Composite Service
- Composite Reference

Binding Contracts

- Service Binding
- Reference Binding

6.2. COMPONENT CONTRACTS

Component Contract can be either a Component Service or a Component Reference. A Component Service is used to expose the functionality of an implementation as a service. A Component Reference allows a component to consume other services. A component contract can be defined in three different ways in SwitchYard:

- Java: Using a Java interface.
- WSDL: Using a port type in a WSDL file.
- ESB: Using a virtual interface definition. (No real file is used).

A component contract in SwitchYard has the following characteristics. All are optional:

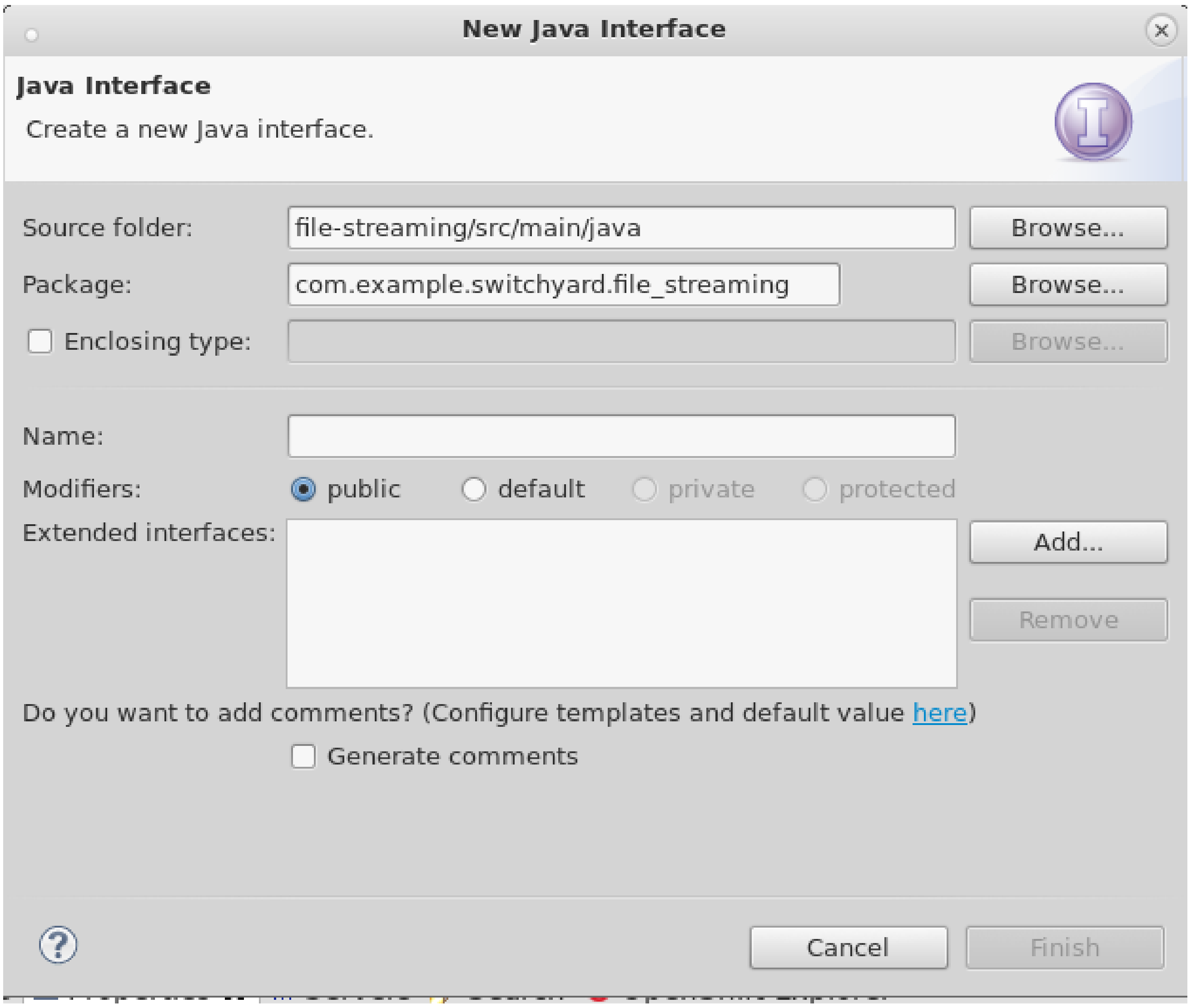
- argument: If used, this is the message content. It is optional as there can be operations that don’t expect a message (for example, REST GET, Scheduled operations). Used in Exchanges of type IN_ONLY and IN_OUT.
- return type: If used, this is the message content for the response. Used only in Exchanges of type IN_OUT.
- exceptions: If used, this is the message content for the response in case of an Exception. Used in Exchanges of type IN_ONLY and IN_OUT.

NOTE

Contracts in Camel can be defined with empty parameters, and users can still work with the message content.

Java contract

A Java contract is defined by a Java Interface.



Source folder: file-streaming/src/main/java

Package: com.example.switchyard.file_streaming

Enclosing type:

Name:

Modifiers: [✓ public] default private protected

Extended interfaces:

Do you want to add comments? (Configure templates and default value here)

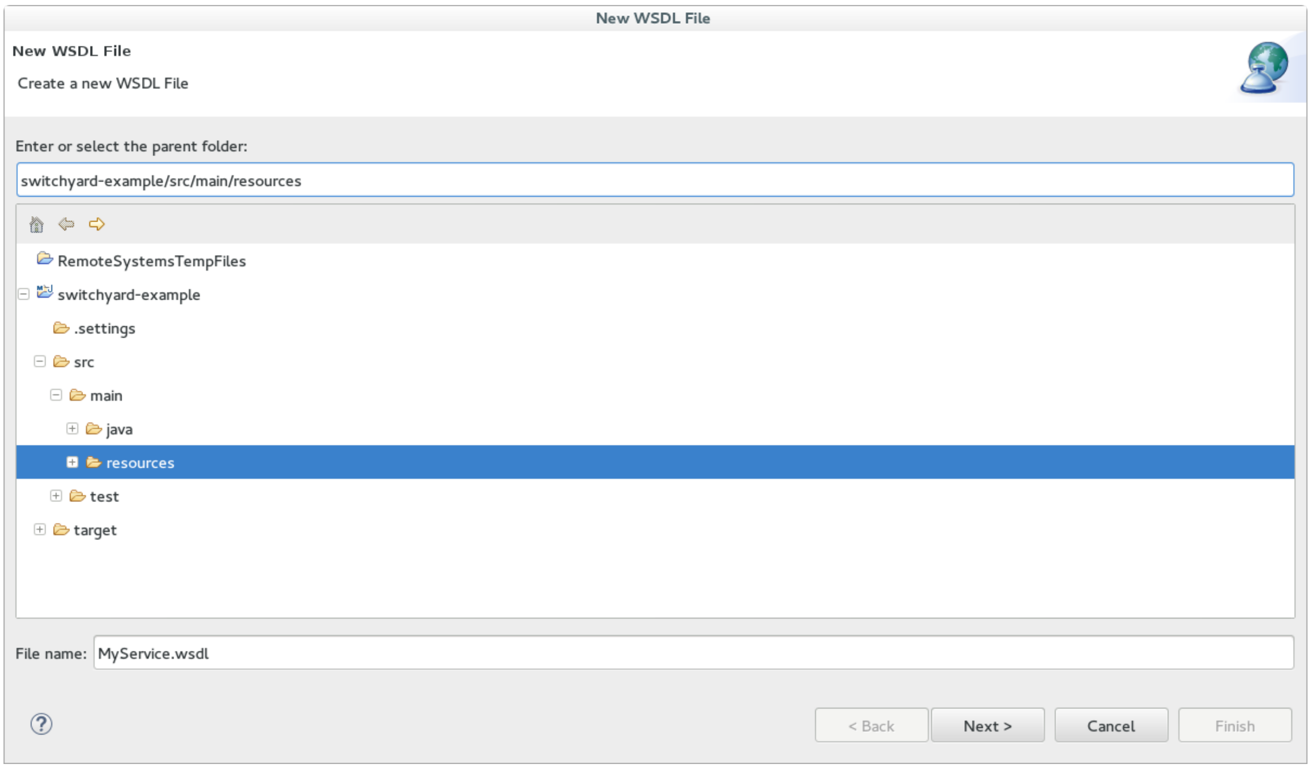
Generate comments

NOTE

Java components require Java contracts.

WSDL contract

A WSDL contract is defined by a port type in a WSDL file.



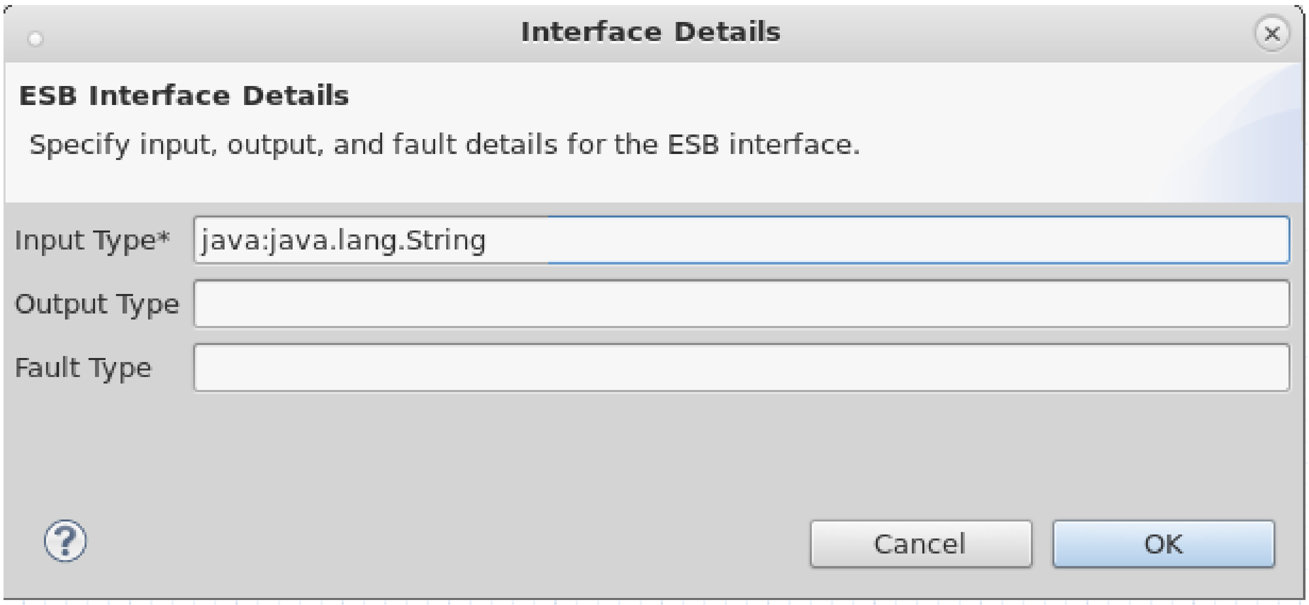
Enter or select the parent folder:

switchyard-example/src/main/resources

File name: MyService.wsdl

ESB contract

An ESB contract is a virtual contract (no file required) that declares the types of the input, output and exception types.



Specify input, output, and fault details for the ESB interface.

Input Type* java:java.lang.String

Output Type

Fault Type

NOTE

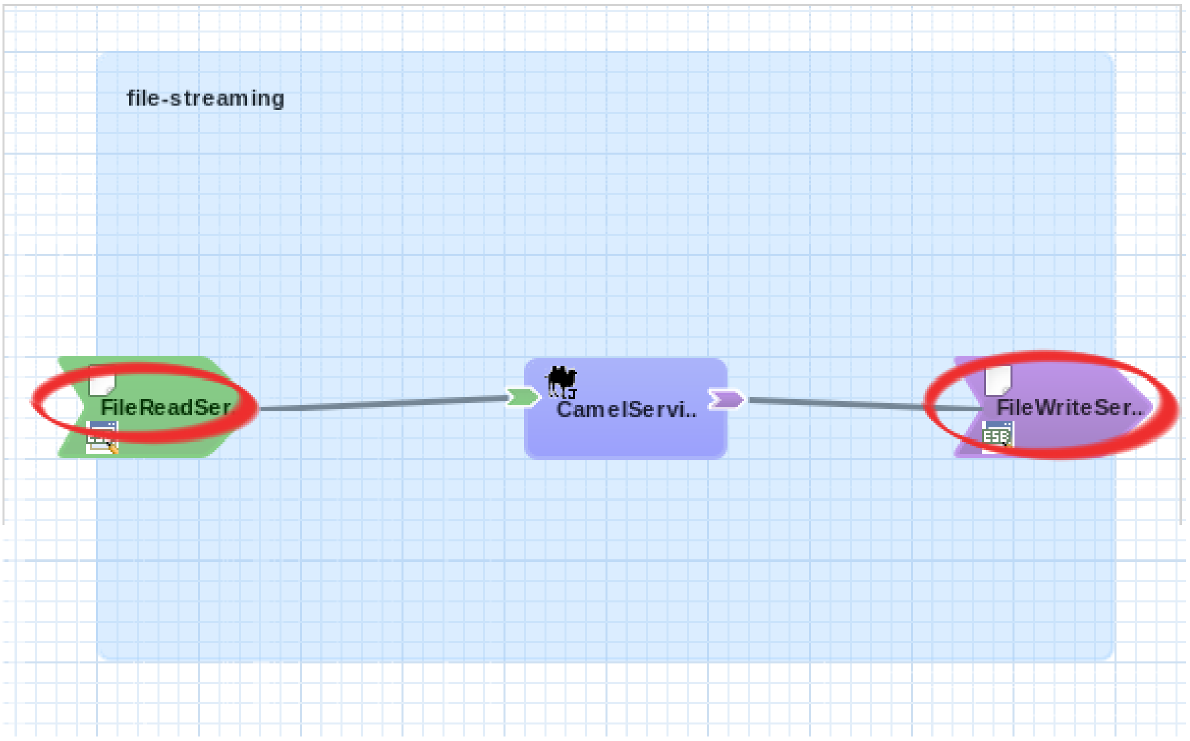
ESB contract may be used in components with one single operation.

6.3. TRANSFORMATIONS BETWEEN CONTRACTS

Transformation between contracts is necessary when a message flows in SwitchYard, and there are different types in the contracts on both ends of the channel. Implicit or explicit transformations happen wherever necessary. The extension points define how the transformation between types in contracts must happen.

Composite Service Binding and Composite Service / Composite Reference and Composite Reference Binding

Contract differences are handled in the ServiceHandlers when composing and decomposing the SwitchYard message. Any difference in contracts is handled in the Message composer.



NOTE

Camel and HttpMessageComposers carry out Camel Implicit Transformations if required. If target type is JAXB, the class must have @XMLRootElement annotation.

Composite Service and Component Service / Component Reference and Composite Reference

Contract differences are handled by transformers defined in the composite application, which are applied by the ExchangeHandler chain during the execution of the Exchange. The transformers usually map from an origin type to a destination type.

When the Exchange is between a composite service and a component service:

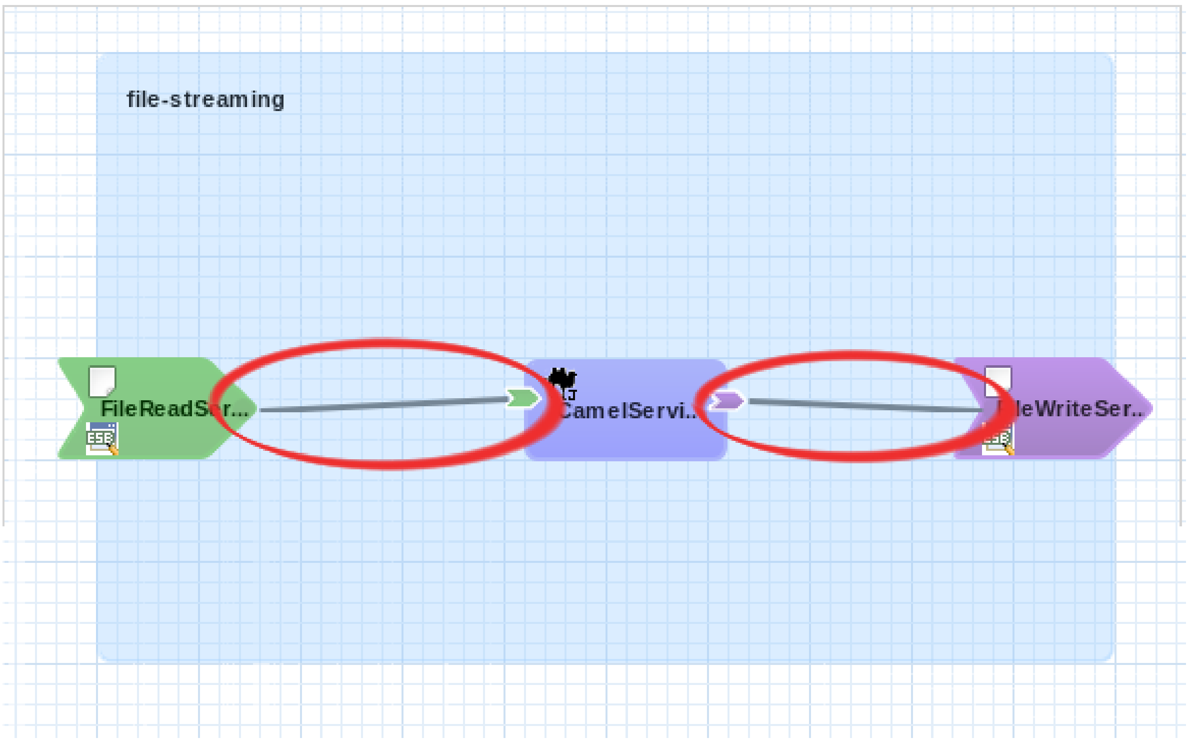
- In the IN phase, from is the argument's type of the composite service and the to is the type in the component service.
- In the OUT phase, from is the return/exception type of the component service and the to is the return/exception type of the composite service.

When the Exchange is between a component reference and a composite reference:

- In the IN phase, from is the argument's type of the component reference and the to is the type in the composite reference.
- In the OUT phase, from is the return/exception type of the composite reference and the to is the return/exception type of the component reference.

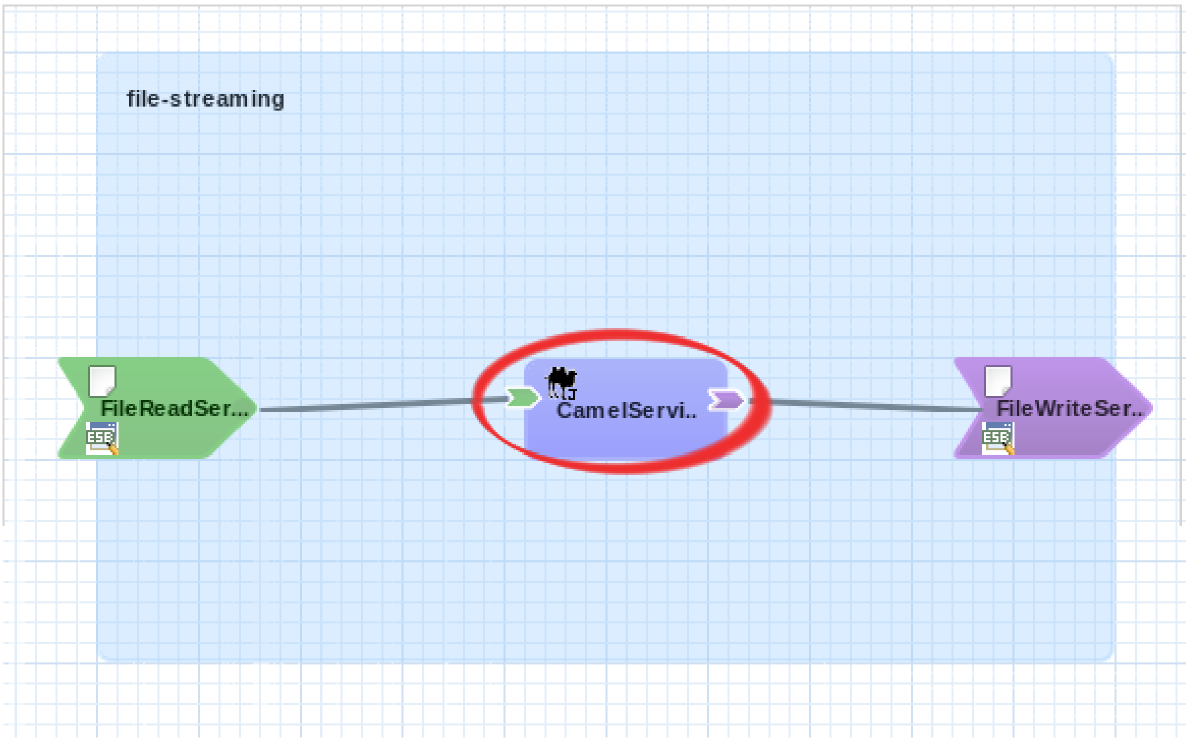
NOTE

Some implicit transformations happen (without declaring a transformer) if SwitchYard supports automatic transformation between both types, as declared by Camel Transformer.



Component Service and Component Reference

Contract differences are handled in the component implementation, and has to be explicitly transformed.



IMPORTANT

- If a Composite Service does not declare a contract, it uses the contract defined by the promoted Component Service.
- Every Component can have one Service.
- Binding name can be null. In this case, a binding name is automatically generated with "ServiceName+BindingType+i".
- When the input parameter of a service contract is empty, the message does not change, it is in its original form (e.g. java.io.Reader for streaming bindings like http, File,...)

CHAPTER 7. PACKAGING AND DEPLOYMENT FOR SWITCHYARD

SwitchYard supports the following packaging types for deployment:

JAR

JAR is the default packaging type for SwitchYard apps.

WAR

WAR files are useful when you need to include additional libraries with your application or you have web application resources (e.g. JSPs, JSF, etc.).

For an example of packaging as a WAR, see the quickstart at ***EAP_HOME*/quickstarts/switchyard/demos/orders**.

EAR

EAR files support multiple application modules in a single deployment along with additional libraries.

EAR deployments allow multiple SwitchYard applications to be included in a single deployable archive. Keep in mind that the SwitchYard applications included in an EAR are still considered separate applications from a lifecycle and visibility (both class loading and service) standpoint. For an example of deploying a SwitchYard application as an EAR, see the quickstart at ***EAP_HOME*/quickstarts/switchyard/ear-deployment**.

7.1. DEPLOYMENT FOR SWITCHYARD

You can deploy SwitchYard applications to the server using file-based deployment, CLI, maven plugin, and the management console. See the JBoss EAP deployment documentation for more information.

7.2. DEPLOY A WAR FILE FOR SWITCHYARD

To prepare a WAR file for deployment to SwitchYard, the SwitchYard component dependencies (switchyard-*.jar) must be excluded as they are provided by the Fuse Service Works container. There are two methods of excluding the SwitchYard dependencies:

1. **Mark the SwitchYard dependencies as provided.**

   To do this, edit the pom.xml file for the WAR file. Mark the **switchyard-*.jar** dependiencies as **provided.**

```
<dependency>
  <groupId>org.switchyard</groupId>
  <artifactId>switchyard-api</artifactId>
  <scope>provided</scope>
</dependency>
<dependency>
  <groupId>org.switchyard.components</groupId>
  <artifactId>switchyard-component-bean</artifactId>
  <scope>provided</scope>
</dependency>
...
```

Next, configure **switchyard-plugin** in the pom.xml file:

```
<plugin>
  <groupId>org.switchyard</groupId>
  <artifactId>switchyard-plugin</artifactId>
  <configuration>
    ...
    <!-- Output to war directory -->
    <outputFile>${project.build.directory}/${project.build.finalName}/WEB-
INF/switchyard.xml</outputFile>
  </configuration>
  <executions>
    <execution>
      <goals>
        <goal>configure</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

2. **Configure the Maven WAR plugin to exclude SwitchYard dependencies**

   An alternative to marking the SwitchYard dependencies as provided, is to use the Maven WAR plugin to exclude SwitchYard dependencies. Edit the pom.xml file to include the following entry:

```
<plugin>
  <artifactId>maven-war-plugin</artifactId>
  <configuration>
    <failOnMissingWebXml>false</failOnMissingWebXml>
    <packagingExcludes>
      WEB-INF/lib/*.jar,
      WEB-INF/classes/META-INF/switchyard.xml
    </packagingExcludes>
    <webResources>
      <resource>
        <directory>target/classes/META-INF</directory>
        <targetPath>WEB-INF</targetPath>
        <includes>
          <include>switchyard.xml</include>
        </includes>
      </resource>
    </webResources>
  </configuration>
</plugin>
```

NOTE

The preferred method is option 1, Mark the SwitchYard dependencies as provided.

CHAPTER 8. SERVICE IMPLEMENTATIONS

8.1. BEAN

8.1.1. Bean Service Component

The Bean Component is a pluggable container in SwitchYard which allows Java classes (or beans) to provide and consume services. This means that you can implement a service by simply annotating a Java class. It also means you can consume a service by injecting a reference to that service directly into your Java class.

NOTE

In a SwitchYard Beans service implementation **java:comp/BeanManager** lookup via JNDI is not supported. You can use **@InjectBeanManager**, which is supported.

8.1.2. Bean Services

Bean Services are standard CDI beans with a few extra annotations. This also opens up the possibilities of how SwitchYard is used; you can now expose existing CDI-based beans in your application as services to the outside world or consume services within your bean.

8.1.3. Create a Bean Service

Prerequisites

- Name: the name of the Java class for your bean service.
- Service Name: the name of the service your bean provides.
- Interface: the contract for the service being provided. Java is the only valid interface type for bean services.

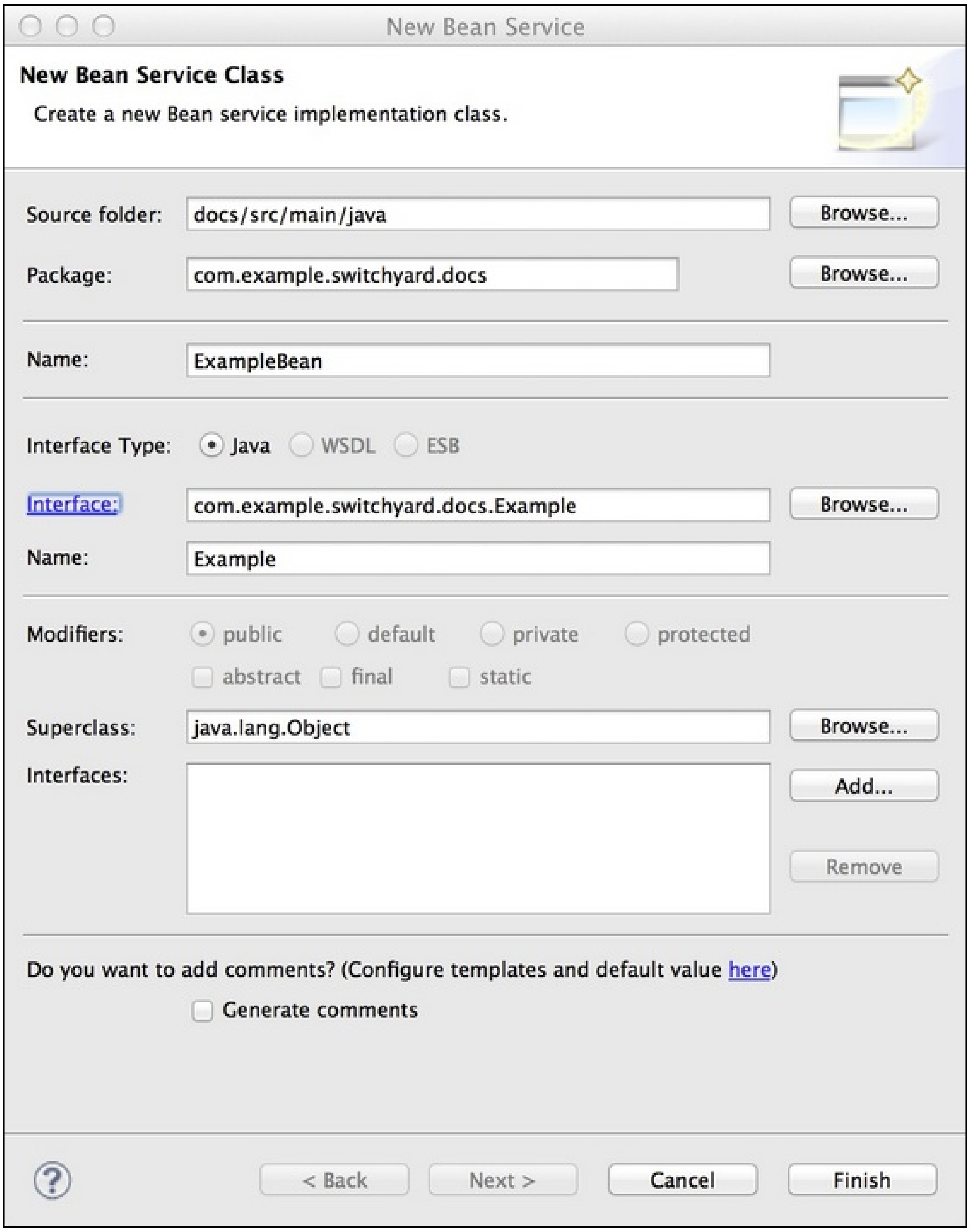
Procedure 8.1. Task

1. Create a new Bean Service Class in the SwitchYard Editor JBoss Developer Studio plug-in.

   NOTE

   If you provide the interface value first, it automatically generates default names for Name and Service Name.

Figure 8.1. Creating a New Bean Service


New Bean Service

New Bean Service Class

Create a new Bean service implementation class.

Source folder: docs/src/main/java

Package: com.example.switchyard.docs

Name: ExampleBean

Interface Type: (•) Java ( ) WSDL ( ) ESB

Interface: com.example.switchyard.docs.Example

Name: Example

Modifiers: (•) public ( ) default ( ) private ( ) protected

[ ] abstract [ ] final [ ] static

Superclass: java.lang.Object

Interfaces:

Do you want to add comments? (Configure templates and default value here)

[ ] Generate comments

The example above shows **ExampleBean** as the name of the class and **com.example.switchyard.docs.Example** as the interface.

2. Click **Finish**.

Your new class looks like this:

```
package com.example.switchyard.docs;

import org.switchyard.component.bean.Service;
```

```
@Service(Example.class)
public class ExampleBean implements Example {
}
```

The @Service annotation allows the SwitchYard CDI Extension to discover your bean at runtime and register it as a service. The value of the annotation (**Example.class** in the above example) represents the service contract for the service. Every bean service must have an @Service annotation with a value identifying the service interface for the service.

3. After creating a bean service, complete the service definition by adding one or more operations to the service interface and a corresponding implementation in your bean:

```
package com.example.switchyard.docs;

import org.switchyard.component.bean.Service;

@Service(Example.class)
public class ExampleBean implements Example {

   public void greet(Person person) {
      // implement service logic here for greet operation
   }
}
```

8.1.4. Providing a Service

Procedure 8.2. Task

- In order to provide a service with the Bean component, add an @Service annotation to your bean:

```
 @Service(SimpleService.class)
public class SimpleServiceBean implements SimpleService {
   public String sayHello(String message) {
      System.out.println("*** Hello message received: " + message);
      return "Hi there!!";
   }
}

public interface SimpleService {
   String sayHello(String message);
}
```

The ***SimpleService*** interface represents the Service Interface that defines the service operations exposed by SwitchYard.

8.1.5. @Service

The @Service annotation allows the SwitchYard CDI Extension to discover your bean at runtime and register it as a service. The value of the annotation (SimpleService.class in the above example) represents the service contract for the service. Every bean service must have an @Service annotation with a value identifying the service interface for the service.

The **META-INF/beans.xml** file in your deployed application tells the JBoss application server to look for beans in this application and to activate the CDI. The ***SwitchYardCDIServiceDiscovery*** CDI extension picks up the @Service beans and makes them available to the application deployer. The service can now be invoked from other services within SwitchYard or bound to a wire protocol through SwitchYard gateways.

8.1.6. Consuming a Service

Procedure 8.3. Task

- In order to consume a SwitchYard service from within a CDI bean, add a @Reference annotation to your bean:

```
@Service(ConsumerService.class)
public class ConsumerServiceBean implements ConsumerService {

  @Inject
  @Reference
   private SimpleService service;

   public void consumeSomeService() {
      service.sayHello("Hello");
   }
}

public interface ConsumerService {

   void consumeSomeService();
}
```

  By default, SwitchYard expects a service reference to be declared with a name which matches the Java type used for the reference. In the above example, the SimpleService type expects a service reference called "SimpleService" in your SwitchYard configuration. However, the @Reference annotation also accepts a service name if the service reference name does not match the Java type name of the contract. For example:

```
@Reference("urn:myservices:purchasing:OrderService")
private OrderService orders;
```

8.1.7. @Reference

The @Reference annotation enables CDI beans to consume other services. The reference can point to services provided in the same application by other implementations, or to a service that is hosted outside of SwitchYard and exposed over JMS, SOAP, or FTP. The JBoss application server routes the invocations made through this reference through the SwitchYard exchange mechanism.

8.1.8. ReferenceInvoker

Although the @Reference annotation injects a reference using the Java interface of the reference contract, it does not allow you to use SwitchYard API constructs like the **Message** and **Context** interfaces.

When invoking a reference from a Bean service, the ReferenceInvoker enables you to access an attachment or a context property.

To use a ReferenceInvoker, replace the service contract interface type with a ReferenceInvoker type. It allows SwitchYard to inject the correct instance automatically.

NOTE

Red Hat recommends you create a new instance of ReferenceInvocation each time you want to invoke a service using ReferenceInvoker.

For example, here is an instance which uses a ReferenceInvoker to invoke SimpleService.

```
@Inject
@Reference("SimpleService")
private ReferenceInvoker service;

public void consumeSomeService(String consumerName) {
  service.newInvocation("sayHello")
    .setProperty("myHeader", "myValue")
    .invoke(consumerName);
}
```

8.1.9. Invocation Properties

While it is a best practice to write your service logic to the data that is defined in the contract (the input and output message types), there can be situations where you need to access contextual information like message headers such as received file name in your implementation. To facilitate this, the Bean component allows you to access the SwitchYard Exchange Context instance associated with a given Bean Service Operation invocation.

Invocation properties represent the contextual information (like message headers) in your bean implementation.

8.1.10. Accessing Invocation Properties

Procedure 8.4. Task

- To enable access to the invocation properties, add a ***Context*** property to your bean and annotate it with the CDI @Inject annotation:

```
package com.example.switchyard.docs;

import javax.inject.Inject;

import org.switchyard.Context;
import org.switchyard.component.bean.Service;
@Service(SimpleService.class)
public class SimpleServiceBean implements SimpleService {

@Inject
private Context context;

public String sayHello(String message) {
        System.out.println("*** Funky Context Property Value: " +
context.getPropertyValue("funkyContextProperty"));
        return "Hi there!!";
    }
}
```

Here, the Context interface allows your bean logic to get and set properties in the context.

NOTE

You can invoke the Context instance only within the scope of one of the Service Operation methods. If you invoke it outside this scope, it results in an UnsupportedOperationException error.

8.1.11. @Inject

The @Inject annotation lets you define an injection point that is injected during bean instantiation. Once your beans are registered as Services, they can be injected as CDI beans using @Inject annotation.

8.1.12. Implementation Properties

Implementation properties represent environmental properties that are defined in the SwitchYard application descriptor (**switchyard.xml**) for your bean implementation.

8.1.13. Accessing Implementation Properties

Implementation properties represent environmental properties that you have defined in the SwitchYard application descriptor (**switchyard.xml**) for your bean implementation. Implementation properties in SwitchYard are the properties that you can configure on a specific service implementation. That is, you can make the property value available to service logic executing inside an implementation container. Here is an example:

```
<sca:component name="SimpleServiceBean">
      <bean:implementation.bean
class="com.example.switchyard.switchyard_example.SimpleServiceBean"/>
      <sca:service name="SimpleService">
        <sca:interface.java interface="com.example.switchyard.switchyard_example.SimpleService">
            <properties>
               <property name="userName" value="${user.name}"/>
            </properties>
        </sca:interface.java>
      </sca:service>
    </sca:component>
```

Procedure 8.5. Task

- To access the Implementation Properties, add an @Property annotation to your bean class identifying the property you want to inject:

```
package com.example.switchyard.docs;

import org.switchyard.component.bean.Property;
import org.switchyard.component.bean.Service;
```

```
@Service(SimpleService.class)
public class SimpleServiceBean implements SimpleService {

    @Property(name="userName")
    private String name;

    @Override
    public String sayHello(String message) {
        return "Hello " + name + ", I got a message: " + message;
    }

}
```

Here, the @Property annotation is used for injecting the ***user.name*** property.

8.1.14. @Property

The @Property annotation enables you to identify the implementation property that you want to inject in a bean.

8.2. BPM

8.2.1. BPM Component

SwitchYard implements the Business Process Management (BPM) functionality through the BPM Component. The BPM Component is a pluggable container in SwitchYard that allows you to expose a business process as a service. Using the BPM component, you can start a process, signal a process event, or abort a process.

IMPORTANT

A JBoss Fuse subscription includes an entitlement to use embedded BPM as a SwitchYard component only. All other uses (for example, with Apache Camel) require a separate BPM subscription.

8.2.2. Create a BPM Service

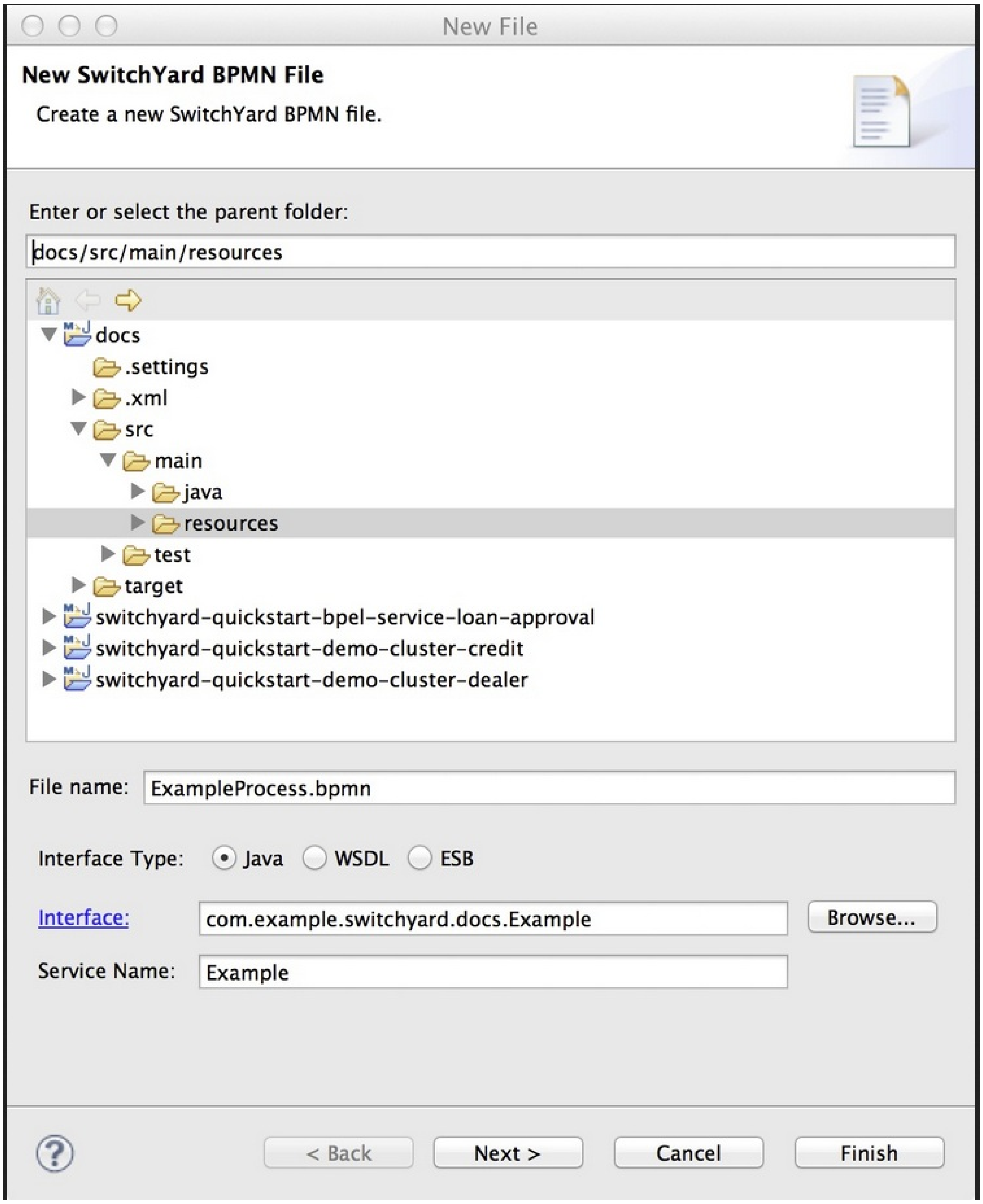
Prerequisites

- File Name: The file name of the new BPMN 2 Process definition.
- Interface Type: The contract for the service provided by your bean. BPM supports Java and WSDL contract types.
- Service Name: The name of the service provided by your bean.

Procedure 8.6. Task

1. Create a new BPMN file in the SwitchYard Editor JBoss Developer Studio plug-in.
2. Input the values into the SwitchYard Editor's New SwitchYard BPMN File Screen.

Figure 8.2. New SwitchYard BPMN File Screen


New File

New SwitchYard BPMN File

Create a new SwitchYard BPMN file.

Enter or select the parent folder:

docs/src/main/resources

```
▼ docs
    .settings
  ▶ .xml
  ▼ src
    ▼ main
      ▶ java
      ▶ resources
    ▶ test
  ▶ target
▶ switchyard-quickstart-bpel-service-loan-approval
▶ switchyard-quickstart-demo-cluster-credit
▶ switchyard-quickstart-demo-cluster-dealer
```

File name: ExampleProcess.bpmn

Interface Type: [✓ Java] WSDL ESB

Interface: com.example.switchyard.docs.Example

Service Name: Example

3. Click Finish.

   This creates a new service component definition for your process service and an empty BPMN process definition.

4. After creating a new BPM Service, create the BPMN 2 process definition and configure the BPM service component to interact with that process definition.

8.2.3. Process Interaction

You define interaction with a process by the actions you add to a BPM service component. Study this sample code which is for a service contract:

```
package org.switchyard.userguide;
public interface MyService {
    public void start(String data);
    public void signal(String data);
    public void stop(String data);
}
```

By using actions, you can map an operation in the service contract to one of the following interactions with a business process:

START_PROCESS

Operations configured with the START_PROCESS action type start new process instances.

When you start your process (actually, any interaction with a service whose implementation is bpm), the processInstanceId is put into the SwitchYard context at Scope.EXCHANGE and is fed back to your client in a binding-specific way. For SOAP, it is in the form of a SOAP header in the SOAP response envelope:

```
<soap:Header>
    <bpm:processInstanceId xmlns:bpm="urn:switchyard-component-
bpm:bpm:1.0">1</bpm:processInstanceId>
</soap:Header>
```

In future process interactions, you need to send back that same processInstanceId, so that the correlation is done properly. For SOAP, that means including the same SOAP header that was returned in the response to be sent back with subsequent requests.

IMPORTANT

If you are using persistence, the sessionId is also available in the Context, and must be fed back as well. It looks the same as the processInstanceId in the SOAP header.

SIGNAL_EVENT

Operations configured with the SIGNAL_EVENT action type have to signal the associated process instance. The processInstanceId must be available in the Context so the correct process instance is correlated.

There are two other pieces of information that are needed when signaling an event:

- The "id". In BPMN2 lexicon, this is known as the "signal id", but in jBPM can also be known as the "event type". This is set as the id of the annotation.

  NOTE

  In BPMN2, a signal looks like this:

  ```
  <signal id="foo" value="bar"/>
  ```

  In jBPM, it is the signal id that is respected, not the name. This might require you to tweak a tooling-generated id if you want to customize its name.

- The "event object". This is the data representing the event itself. There are two ways to pass in the event object:

    1. The first way is through a Context object. The way in which this is passed is similar to the processInstanceId, and it is known as "signalEvent". If you use this, you are limited to a String type.

    2. The second way is through the Message content object itself (that is, your payload). If the signalEvent Context property is absent, the content of the Message is used as the event object.

ABORT_PROCESS_INSTANCE

If an operation is configured with the ABORT_PROCESS_INSTANCE action type, associated process instances are aborted. Note that the processInstanceId must be available in the Context so the correct process instance is correlated.

8.2.4. Use Process Variables

Prerequisites

- JBoss Developer Studio jBPM Plug-In

Procedure 8.7. Use Process Variables

1. Click on the white space around a process or on any of your process nodes.

2. Access the Properties view.

3. Declare the variablenames at the process level and in the Parameter Mapping (and possibly Result Mapping).

8.2.5. Mappings

Mappings are the way to move data in or out of the action for that operation. You can specify as many mappings as you like for an action, and they get grouped as globals, inputs or outputs:

Mapping variables from your SwitchYard Exchange, Context or Message into jBPM process variables can be done with expressions. Each action for a process service has a distinct set of variable mappings.

- Global mappings are used to provide data that is applicable to the entire action, and is often used in classic in/out param (or data-holder/provider) fashion. An example of a global mapping is a global variable specified within a Drools Rule Language (DRL) file.

- Input mappings are used to provide data that represents parameters being fed into an action. An example of an input mapping for BPM is a process variable used while starting a business process. For Rules, it could be a fact to insert into a rules engine session.

- Output mappings are used to return data from an action. An example of an output mapping is a BPM process variable that you want to set as the outgoing (response) message’s content.

NOTE

The onlyexpressionType supported currently is MVEL, so you do not have to specify it. The expression itself can be any MVEL expression

8.2.6. expressionType Properties

These variables are available by default:

exchange

The current org.switchyard.Exchange.

context

The current org.switchyard.Context.

message

The current org.switchyard.Message.

Whatever the resultant value of the expression is constitutes the data that is made available to the action.

```
expression="message.content"
```

This is the same as **message.getContent()**.

```
expression="context[‘foo’]" scope="IN"
```

This is the same as **context.getProperty("foo", Scope.IN).getValue()** in a null-safe manner.

NOTE

Specifying the scope attribute only matters if you use the context variable inside your expression. If you don’t specify a scope, the default Context access (which is done like a Map, if you picked up on that), is done with Scope.EXCHANGE for global mappings, Scope.IN for input mappings, and Scope.OUT for output mappings.

Specifying the variable attribute is often optional, but this depends on the usage. For example, if you are specifying a global variable for a rule, or a process variable to put into (or get out of) a BPM process, then it is required. However, if the result of the expression is to be used as facts for rule session insertion, then specifying a variable name is not applicable.

Here is some XML sample code:

```
    <mapping expression="theExpression" expressionType="MVEL" scope="IN"
variable="theVariable"/>
```

8.2.7. Consuming a Service

There are two ways of consuming Services with the SwitchYard BPM component:

1. By invoking the BPM implementation through a gateway binding. Since the BPM component exposes a Java interface fronting the business process, you can use any of the bindings provided by SwitchYard. (You could, for example, use either a SOAP Binding or a Camel Binding.)

2. By invoking other SwitchYard Services from inside a BPM process itself. To do this, you can use the SwitchYardServiceWorkItemHandler, which is provided out-of-the-box. (To make authoring BPMN2 processes easier, SwitchYard provides a widget for the Eclipse BPMN2 Modeler visual editor palette.)

8.2.8. SwitchYard Service Task Properties

You can use the following properties to configure the SwitchYard Service task.

Service Naming Properties:

ServiceName

This is the name of the SwitchYard service to invoke. It is a mandatory property

ServiceOperationName

This is the name of the operation within the SwitchYard service to invoke. It is an optional property. (The default behavior is to use the single method name in the service interface, if there is just one.)

Content I/O Properties:

ContentInputName

This is the process variable into which the message content is placed. It is an optional property. The default value is contentInput.

ContentOutputName

The process variable from which the message content is obtained. It is an optional property. The default value is contentOutput.

Fault-Handling Properties:

FaultResultName

This is the name of the output parameter (in other words, the result variable) under which the exception is stored. It is optional.

FaultSignalId

This is the bpmn signal id (or event type) that is used to signal an event in the same process instance. The event object is the exception. This is an optional property.

FaultWorkItemAction

This property determines what happens after a fault occurs. If it is set to null, nothing is done. If set to complete, the current work item (that is, the SwitchYard service task) is completed. If it is set to abort, the current work item is aborted. (The default setting is null.) This property is optional.

8.2.9. SwitchYard Service Fault Handling

The **SwitchYardServiceWorkItemHandler** class is used for fault handling in the Switchyard Service tasks during process execution. It executes service references according to the SwitchYard's fault-handling properties that you have configured. The SwitchYard's fault-handling properties define the behavior of the **SwitchYardServiceWorkItemHandler** class when a fault is encountered during the execution of the service reference.

You can use the **SwitchYardServiceWorkItemHandler** class for fault handling in the following scenarios:

- If you want to have a split gateway in your process flow, to inspect a process variable for any occurrence of a fault. To achieve this, you need to set the following fault-handling properties in your SwitchYard Service task:
    - FaultResultName: Specifying the FaultResultName property enables the **SwitchYardServiceWorkItemHandler** class to make the fault available as an output parameter of the task. You can then associate it with a process variable, and inspect for existence in your split gateway.
    - FaultWorkItemAction: Specifying the FaultWorkItemAction property to complete enables the process to continue on to your split gateway.
- If you want to have a single shared path of fault-handling in your process flow. To achieve this, you need to set the following fault-handling properties in your SwitchYard Service task:
    - FaultSignalId: Specifying the FaultSignalId property same as the Signal ID you specified in your bpmn2 process definition, enables you to add an event node in your process that is triggered with this signal id. The flow starting from this event node is your fault handling path. The **SwitchYardServiceWorkItemHandler** class then signals the proper event with the configured ID.

8.2.10. Using The Standard BPMN2 Service Task

You can invoke SwitchYard Services using the standard BPMN2 Service Task. You can use the Service Task icon from the BPMN2 Editor palette and configure its properties. To configure the Service Task, keep in mind the following points:

- The ***<serviceTask>*** attribute invokes SwitchYard when it has an ***implementation="##SwitchYard"*** attribute.
- The ServiceName is derived from the BPMN2 interfaceImplementationRef.
- The ServiceOperationName is derived from the BPMN2 operationImplementationRef.
- The ContentInputName is always called Parameter.
- The ContentOutputName is always called Result.

8.2.11. Resources

A resource represents an artifact that your BPM process uses at runtime. A resource can be a properties file or a Drools Rule Language file. You can configure the list of resources available to your process in the BPM service component.

8.2.12. WorkItemHandler Interface

A work item handler is responsible for executing work items of a specific type. They represent the glue code between an abstract, high-level work item that is used inside the process and the implementation of this work item. You can add your own code into the business process using the WorkItemHandler. To do so, implement the **org.kie.runtime.process.WorkItemHandler** interface and add a handler definition to your BPM service component.

8.3. BPEL

8.3.1. BPEL Component

The BPEL Component is a pluggable container in SwitchYard that allows you to expose a WS-BPEL business process as a service through an interface defined using WSDL.

8.3.2. Providing a Service with the BPEL Component

Procedure 8.8.

1. Define your process using WS-BPEL within JBoss Developer Studio (with JBoss Integration and SOA Development tooling installed).

2. Define a WSDL interface for the BPEL service.

3. Define a Deployment Descriptor using the ODE Deployment Descriptor editor bundled with JBoss Tools.

4. Add the component containing the implementation and service interface to the SwitchYard configuration.

8.3.3. Example of BPEL Component Configuration

Here is an example of the component section of the SwitchYard configuration:

```
<sca:component name="SayHelloService">
      <bpel:implementation.bpel process="sh:SayHello"/>
      <sca:service name="SayHelloService">
        <sca:interface.wsdl interface="SayHelloArtifacts.wsdl#wsdl.porttype(SayHello)"/>
      </sca:service>
 </sca:component>
```

The BPEL component contains a single ***implementation.bpel*** element that identifies the fully qualified name of the BPEL process. This component may also contain one or more service elements defining the WSDL port types through which the BPEL process can be accessed.

In the packaged Switchyard application, ensure that the BPEL process associated with this fully qualified name must be present within the root folder of the distribution, along with the deployment descriptor (**deploy.xml**). Here is an example of the deployment descriptor for the BPEL process referenced above:

```
<deploy xmlns="http://www.apache.org/ode/schemas/dd/2007/03"
 xmlns:examples="http://www.jboss.org/bpel/examples">

<process name="examples:SayHello">
 <active>true</active>
  <retired>false</retired>
  <process-events generate="all"/>
    <provide partnerLink="client">
```

```
      <service name="examples:SayHelloService" port="SayHelloPort"/>
    </provide>
  </process>
  </deploy>
```

8.3.4. Consuming a Service from a BPEL Process

To enable a BPEL process to invoke other services, you need to define the WSDL interface representing the service to be consumed, using an invoke element within the deployment descriptor. For example, in the **deploy.xml** file:

```
<process name="ls:loanApprovalProcess">
    <active>true</active>
    <process-events generate="all"/>
    <provide partnerLink="customer">
      <service name="ls:loanService" port="loanService_Port"/>
    </provide>
    <invoke partnerLink="assessor" usePeer2Peer="false">
      <service name="ra:riskAssessor" port="riskAssessor_Port"/>
    </invoke>
</process>
```

Here, the ***usePeer2Peer*** property informs the BPEL engine not to use internal communications for sending messages between BPEL processes that may be executing within the same engine, and instead pass messages through the SwitchYard infrastructure.

For each consumed service, you can then create a reference element within the SwitchYard configuration to locate the WSDL file and identify the port type associated with the required WSDL service or port, as shown in the **switchyard.xml** file below:

```
<sca:component name="loanService">
    <bpel:implementation.bpel process="ls:loanApprovalProcess" />
    <sca:service name="loanService">
      <sca:interface.wsdl interface="loanServicePT.wsdl#wsdl.porttype(loanServicePT)"/>
    </sca:service>
    <sca:reference name="riskAssessor">
      <sca:interface.wsdl interface="riskAssessmentPT.wsdl#wsdl.porttype(riskAssessmentPT)"/>
    </sca:reference>
</sca:component>
```

8.3.5. Property Injection into a BPEL Process

You can inject properties into your BPEL process definition by using the **SwitchYardPropertyFunction.resolveProperty()** XPath custom function. The ***bpel:copy*** section copies Greeting property value into the **ReplySayHelloVar** variable in example shown below:

```
  <bpel:copy>
    <bpel:from
xmlns:property="java:org.switchyard.component.bpel.riftsaw.SwitchYardPropertyFunction"
      expressionLanguage="urn:oasis:names:tc:wsbpel:2.0:sublang:xpath2.0">
       <![CDATA[concat(property:resolveProperty('Greeting'),
$ReceiveSayHelloVar.parameters/tns:input)]]>
    </bpel:from>
    <bpel:to part="parameters" variable="ReplySayHelloVar">
```

```
      <bpel:query queryLanguage="urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0"><!
[CDATA[tns:result]]></bpel:query>
    </bpel:to>
  </bpel:copy>
```

8.3.6. Maintaining Multiple Versions of a BPEL Process

You can use the BPEL processes to implement long lasting stateful business processes. However the BPEL process may need to change, over the course of its lifetime, to accommodate new requirements. This introduces the problem of how to deal with the existing active instances of the BPEL process that may not complete for weeks, months or even years. To deal with multiple version of a BPEL process and to enable new requirements to be introduced (while still preserving the original process definitions associated with existing active process instances), you can associate a version number with the BPEL process by adding it as a suffix to the BPEL file name.

For example, if your BPEL process is located in the **HelloWorld.bpel** file, then you can simply add a hyphen followed by the version number, such as **HelloWorld-32.bpel**. This indicates that this is the thirty second version of this BPEL process. Whenever you define a new version of the BPEL process, package it in the SwitchYard application along side the previous versions of the BPEL process. It is important that the older version of the BPEL process remain in the SwitchYard application until there are no longer any active process instances associated with that version. You need to then re-deploy the SwitchYard application, without undeploying the previous version. If you undeploy the previous version of the SwitchYard application, the BPEL engine deletes all outstanding active instances associated with the deleted process definitions.

You can version the BPEL process but not the WSDL interfaces. So you must ensure that any changes made to the WSDL interfaces are backward compatible, so that both the new and older versions of the BPEL (that still have active process instances) are not affected by the changes.

8.3.7. Structure of a SwitchYard BPEL Application

For SwitchYard BPEL applications, the artifacts within the **src/main/resources** folder are structured differently. The **switchyard.xml** configuration file is located in the META-INF folder. However, the BPEL deployment descriptor (**deploy.xml**), and the BPEL process definition are located in the root folder. You can locate the WSDL interface definitions, and any accompanying XSD schemas in the sub-folders. You must ensure that the BPEL process and SwitchYard BPEL component configuration define the correct relative path for the artifacts.

Here is an example that shows the structure of the say_hello SwitchYard BPEL quickstart:

```
say_hello
    src/main/java
    src/main/resources
         META-INF
          switchyard.xml
       deploy.xml
       SayHello.bpel
       SayHelloArtifacts.wsdl
    JRE System Library [JavaSE-1.6]
    src
    pom.xml
```

8.4. CAMEL

8.4.1. Camel Services

Camel services allow you to leverage the core routing engine of Apache Camel to route between services in SwitchYard. Camel endpoints function as protocol adapters, exposing services hosted in SwitchYard to the outside world and allowing external services to be invoked from within SwitchYard. You can define the Camel routes using Java DSL or XML and deploy them within SwitchYard to handle pipeline orchestration between SwitchYard services.

8.4.2. Create a Camel Service

Prerequisites

- Name: the name of the Java class or XML file for your bean service.
- Service Name: the name of the service your bean provides.
- Interface: the contract for the service being provided. Camel supports Java and WSDL contract types.

Procedure 8.9. Create a Camel Service

1. Create a new Camel Route file resource in the SwitchYard Editor JBoss Developer Studio plug-in.
2. Decide whether to use DSL or XML dialect for the route. (Functionally, they are more or less equivalent, the choice is determined by how you want to express your routing logic.)
3. If you want create a Java DSL route, select the "Camel (Java)" implementation type. For XML, use the "Camel (XML)" type.
4. Input the values into the SwitchYard Editor's New Route File Screen.
5. Click Finish.

8.4.3. Guidelines of Camel Route

These are some general guidelines to keep in mind when creating either type of route:

- There is only one route per service.
- The consumer or "from" endpoint in a route is always a "switchyard" endpoint and the endpoint name must equal the service name. This is default behavior in the tooling.
- To consume other services from within your route, only use "switchyard" consumer (in other words "to") endpoints. This keeps your routing logic independent of the binding details for consumed services.

8.4.4. Java DSL Route

You can define Camel routes using the Java Domain Specific Language (DSL). To implement the Java DSL, extend the **RouteBuilder** class. As there can be only one route per service, you can have only one **RouteBuilder** class for each Camel routing service in your application. You can then add logic to your routing service in the **configure()** method as shown below:

```
package com.example.switchyard.docs;
```

```
import org.apache.camel.builder.RouteBuilder;

public class CamelServiceRoute extends RouteBuilder {
    /**
     * The Camel route is configured through this method.  The from:
     * endpoint is required to be a SwitchYard service.
     */
    public void configure() {
        // TODO Auto-generated method stub
        from("switchyard://Example").log(
                "Received message for 'Example' : ${body}");
    }
}
```

8.4.5. XML Route

You can define Camel routes using XML. To define Camel routes in XML files, use the ***<route>*** tag with the namespace "http://camel.apache.org/schema/spring" as shown below:

```
<?xml version="1.0" encoding="ASCII"?>
<route xmlns="http://camel.apache.org/schema/spring">
 <from uri="switchyard://Example"/>
 <log message="Example - message received: ${body}"/>
</route>
```

You can have only one file containing a route definition for each XML routing service in your application.

8.4.6. Consuming Services From Camel Routes

You can invoke another service from your Camel route by using the SwitchYard producer endpoint (***switchyard://***) within your route as shown below:

```
switchyard://[service-name]?operationName=[operation-name]
```

Here,

- service-name: Name of the SwitchYard service. This value should match the name of a service reference defined on the service component for the route.
- operation-name: Name of the service operation you want to invoke. This is only used on references and is optional if the target service only has a single operation.

The example below illustrates the default XML route modified to invoke a SwitchYard service:

```
<?xml version="1.0" encoding="ASCII"?>
  <route xmlns="http://camel.apache.org/schema/spring">
     <from uri="switchyard://Example"/>
     <log message="Example - message received: ${body}"/>
          <!-- Invoke hasItem operation on WarehouseService -->
     <to uri="switchyard://WarehouseService?operationName=hasItem"/>
  </route>
```

8.4.7. Message Exchange Pattern

Exchanges are of central importance in Apache Camel, because the exchange is the standard form in which messages are propagated through routing rules. An exchange object consists of a message, augmented by metadata. Using an Exchange object makes it easy to generalize message processing to different message exchange patterns (MEP). See Apache Camel Development Guide for more information about exchange objects and message exchange patterns in Camel.

It is possible to temporarily implement an InOut (Request/Response) MEP when the default has been set as non Request/Response. For example, in the **quickstarts/switchyard/camel-jpa-binding** example from the quickstarts on EAP, you can change the **storeGreeting()** method to be InOut by manipulating the call.

This is the original call. No return value is specified.

```
void storeGreeting(Greet event);
```

Specify a return value and the established MEP will be overriden and will become an InOut MEP.

```
int storeGreeting(Greet event);
```

8.4.8. Using Scripting Languages

Camel supports dynamic scripting languages inside the XML and Java DSL route logic. You can use the scripting languages in the following ways:

1. You can use them to create a predicate in a message filter as shown below:

```
public class ScriptingBuilder extends RouteBuilder
{
public void configure()
{
from("switchyard://Inbound").filter().javaScript("request.getHeader('myHeader') !=
null").to("switchyard://Outbound");
}
}
```

2. You can use them to implement your service using ***transform*** element as shown below:

```
public class ScriptingImplementationBuilder extends RouteBuilder {
public void configure()
{
from("switchyard://Inbound").transform().groovy("classpath:script.groovy");
// classpath resource
from("switchyard://InboundBsh").transform().language("beanshell", "file:script.bsh");
// file system resource
}
}
```

Additionally, you can generate responses by using predefined variables like request, response, or exchange inside your script.

8.4.9. Supported Scripting Languages

SwitchYard supports the following scripting languages inside the XML and Java DSL route logic:

- BeanShell
- JavaScript
- Groovy
- Ruby
- Python

8.4.10. Using CDI Beans in Camel Routes

The Camel component provides a convenient and powerful mechanism for routing between SwitchYard services using Java DSL or XML routing definitions. When creating these routes, you may need to add logic to the service pipeline that is specific to the Camel route. Instead of implementing your logic as a service, you can add a CDI bean to your application and call that as a bean from within your Camel route. SwitchYard integrates the CDI Bean Manager with the Camel Bean Registry to allow you to reference CDI Beans in your Camel routes. For example, you can use the following CDI bean inside your SwitchYard Camel Routes:

```
@Named("StringSupport")
@ApplicationScoped
public class StringUtil
 {
    public String trim(String string)
    {
     return string.trim();
    }
 }
```

The SwitchYard Camel Route logic implemented with this CDI bean looks like this:

```
public class ExampleBuilder extends RouteBuilder
{
    public void configure()
    {
       from("switchyard://ExampleBuilder")
           .split(body(String.class).tokenize("\n"))
           .filter(body(String.class).startsWith("sally:"))
           .to("bean:StringSupport");
    }
}
```

Any Java class annotated with ***@Named*** annotation in your application is available through Camel's Bean registry.

8.4.11. Injecting Implementation Properties in Camel Routes

SwitchYard integrates with the Properties Component in Camel to make system and application properties available inside your route definitions. You can inject properties into your camel route using ***{{propertyName}}*** expression, where ***propertyName*** is the name of the property.

For example, the following camel route expects the ***user.name*** property to be injected in the last ***<Log>*** statement:

```
<route xmlns="http://camel.apache.org/schema/spring" id="CamelTestRoute">
  <log message="ItemId [${body}]"/>
  <to uri="switchyard://WarehouseService?operationName=hasItem"/>
  <log message="Title Name [${body}]"/>
  <log message="Properties [{{user.name}}]"/>
</route>
```

8.5. RULES

8.5.1. Rules Component

The Rules component is a pluggable container in SwitchYard which allows you to expose business rules as a service. You can add a custom interface to your rules and annotate its methods to define which methods should execute the rules. The Rules component supports Drools as the rule engine.

IMPORTANT

A JBoss Fuse subscription includes an entitlement to use embedded BRMS (Drools) as a SwitchYard component only. All other uses (for example, with Apache Camel) require a separate BRMS subscription.

NOTE

In a SwitchYard Rules service implementation **java:comp/BeanManager** lookup via JNDI is not supported. There is no supported alternative.

8.5.2. Create a Rules Service

Prerequisites

- File Name: the name of the file that is used to create a new template rules definition.
- Service Name: the name of the service that your rules provide.
- Interface Type: the contract for the service being provided. Rules services support Java and WSDL contract types.
- Package Name: package name used for the new Rules file.

Procedure 8.10. Create a Rules Service

1. Create a new SwitchYard Rules file in the SwitchYard Editor JBoss Developer Studio plug-in.
2. The MyService interface can be as simple as this, with no SwitchYard-specific imports:

```
package com.example.switchyard.docs;
public interface Example {
    public void process(MyData data);
  }
```

3. The generated rule template looks like this:

```
package com.example.switchyard.docs
import org.switchyard.Message
global Message message

rule "RulesExample"
    when
        // insert conditional here
    then
        // insert consequence here
        System.out.println("service: ExampleService, payload: " + message.getContent());
end
```

4. Input the values into the SwitchYard Editor's SwitchYard Rules File screen.

5. Click Finish.

8.5.3. Stateless and Stateful Rules Executions

Introduction

By default, service method invocation creates a new Drools knowledge session, execute it given the passed-in domain data and then be disposed cleanly.

However, it is possible to configure SwitchYard so that a stateful knowledge session is used. To do this, you use the **FIRE_ALL_RULES** action type instead of **EXECUTE**.

There is also a capability which allows you to insert facts into a stateful knowledge session without firing the rules. In this case, use the **INSERT** action type.

8.5.4. Stateless Knowledge Session

Stateless session that does not utilize inference is the simplest use case for JBoss Rules. A stateless session can be called like a function. It can be passed some data and then receive some results back.

Here are some common use cases for stateless sessions:

Validation

Is this person eligible for a mortgage?

Calculation

Compute a mortgage premium.

Routing and Filtering

Filter incoming messages, such as emails, into folders.

Send incoming messages to a destination.

8.5.5. Stateful Knowledge Session

A stateful knowledge session is one which persists in memory, allowing it to span multiple invocations. They can change iteratively over time. In contrast to a Stateless Session, **the dispose()** method must be called afterwards to ensure there are no memory leaks, as the Knowledge Base contains references to Stateful Knowledge Sessions when they are created. StatefulKnowledgeSession also supports the **BatchExecutor** interface, like StatelessKnowledgeSession, the only difference being that the **FireAllRules** command is not automatically called at the end for a Stateful Session.

Here are some common use cases for Stateful Sessions:

Monitoring
: Stock market monitoring and analysis for semi-automatic buying.

Diagnostics
: Fault finding and medical diagnostics

Logistics
: Parcel tracking and delivery provisioning

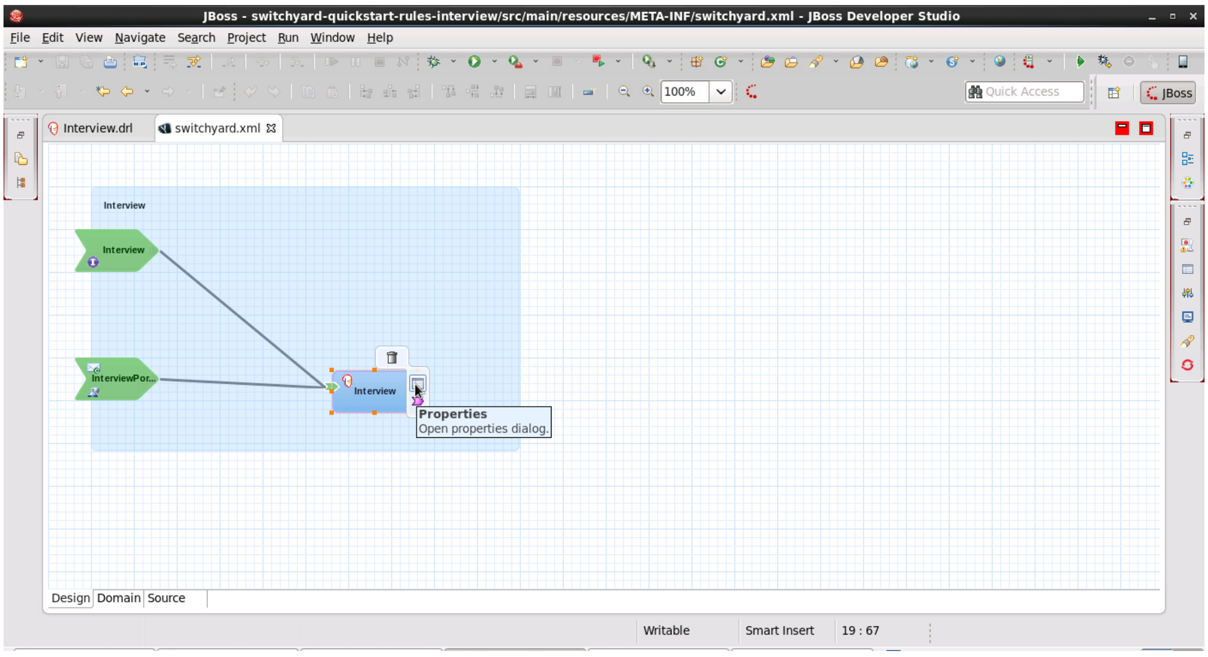
Compliance
: Validation of legality for market trades.

8.5.6. Mapping Global Variables

You can map variables from your SwitchYard Exchange, Context or Message into JBoss Rules globals with MVEL expressions.

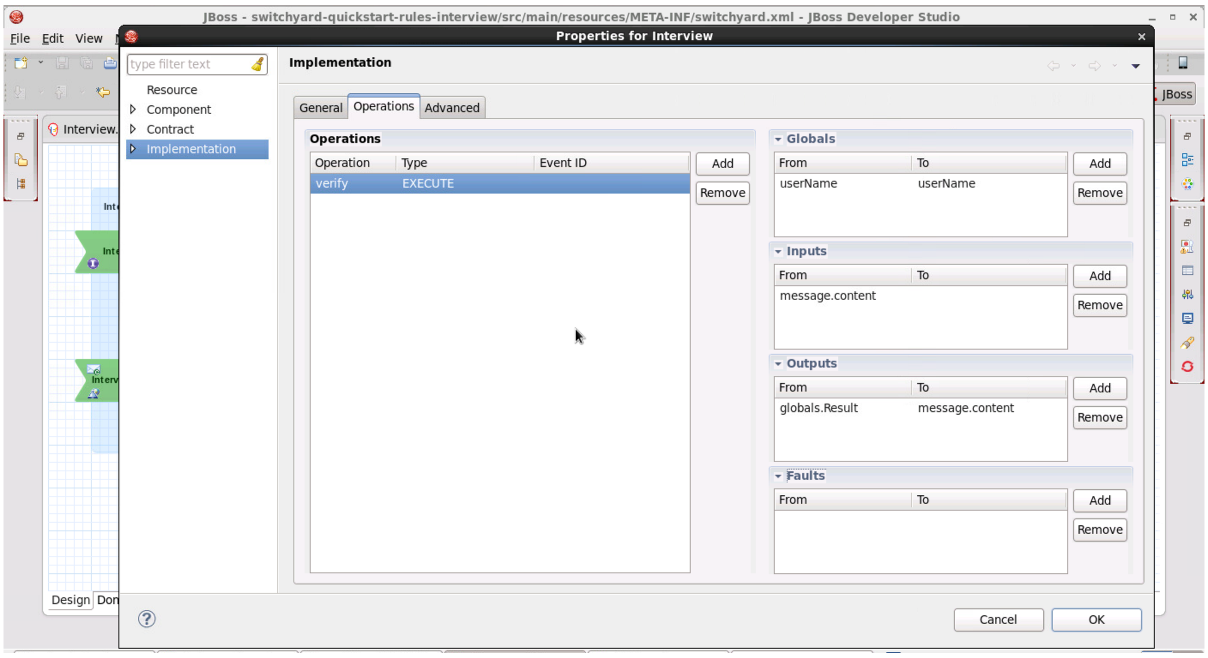
1. To map the variables, hover the mouse over the Rules component in **switchyard.xml** and click Properties icon.

   Figure 8.3. Properties dialog for Rules Component



2. In the Properties dialog, click **Implementation**. From the right hand side panel, click **Operations** tab to view Operation Mapping tooling window.

Figure 8.4. Operations Mapping


JBoss - switchyard-quickstart-rules-interview/src/main/resources/META-INF/switchyard.xml - JBoss Developer Studio

Properties for Interview

Implementation

General | Operations | Advanced

Operations

Operation	Type	Event ID
verify	EXECUTE	

Globals

From	To
userName	userName

Inputs

From	To
message.content	

Outputs

From	To
globals.Result	message.content

Faults

From	To

8.5.7. Map Global Variables

1. Configure the rules implementation.

NOTE

Your expression can use the variables exchange (**org.switchyard.Exchange**), context (**org.switchyard.Context**) or message (**org.switchyard.Message**).

Context can be accessed in the expression as a java.util.Map. When accessing it as such, the default properties Scope is IN. This can be overridden using the contextScope attribute by changing it to OUT or EXCHANGE

2. Use these global variables in your JBoss Rules Drools Rule Language file:

```
package example

global java.lang.String service
global java.lang.String messageId
global com.example.Payload payload

rule "Example"
    when
        ...
    then
        ...
        System.out.println("service: " + service + ", messageId: " + messageId + ", payload: " +
payload);
end
```

NOTE

In a more realistic scenario, the payload would be accessed in rule firing because it was inserted into the session directly by the RulesExchangeHandler, and thus evaluated (rather than being accessed as a global).

8.5.8. Mapping Facts

The object which is inserted into the rules engine as a fact by default is the SwitchYard message's content. However, you can override this by specifying your own fact mappings.

8.5.9. Notes About Mapping Facts

- If you specify your own fact mappings, the SwitchYard message's content is not inserted as a fact. You can add a fact mapping with an expression of "message.content" if you want to still include it.
- There is no point in specifying the "variable" attribute of the mappings (as done for global mappings), as the result of each expression is inserted as a nameless fact.

  For stateless execution, the full list of facts is passed to the StatelessKnowledgeSessions' **execute(Iterable)** method.

  For stateful execution, each fact is individually inserted into the StatefulKnowledgeSession.
- If the result of the mapping expression implements Iterable (for example, a Collection), then the result is iterated over, and each iteration is inserted as a separate fact, rather than the parent Iterable itself being inserted. This is not recursive behavior (because it is only done once).

8.5.10. Auditing a Service

SwitchYard supports basic audit mechanism to audit the Drools rules execution. For auditing a service, SwitchYard requires a CDI environment to run. You can write custom auditors for your service and use listeners to monitor them. For example, you can use listeners to audit a BPM process and save the audit details into a database while the process progresses. The listener then reports the audit details at a later time.

8.5.11. Consuming a Service from the Rules Component

Consuming a SwitchYard Service from within the Rules Component leverages the Channels capability. You can configure channels within your process that executes business rules.

8.6. KNOWLEDGE SERVICES

8.6.1. Knowledge Services

Knowledge Services are SwitchYard services that leverage Knowledge, Innovation and Enterprise (KIE) and provide knowledge based content as outputs. The Knowledge Services leverage Drools and jBPM. Drools and jBPM are tightly integrated under KIE, and hence both SwitchYard's BPM component and Rules component share most of their runtime configuration.

8.6.2. Actions

When you invoke a SwitchYard operation, it results in the execution of the corresponding Action. Actions are how Knowledge Services know how to map service operation invocations to their appropriate runtime counterparts. For example, when you invoke a method **myOperation**, it may result in execution of actions like execute some business rules or start a business process. Here is an example of Actions attribute illustrating **myOperation** method and an action of type ***ACTION_TYPE***:

```
<actions>
 <action id="myId" operation="myOperation" type="ACTION_TYPE">
  <globals>
   <mapping/>
  </globals>
  <inputs>
   <mapping/>
  </inputs>
  <outputs>
   <mapping/>
  </outputs>
 </action>
</actions>
```

Here, the placeholder ***ACTION_TYPE*** can hold values for the actual ***ActionTypes*** specific to the BPM or Rules components.

8.6.3. Mappings

You can use Mappings to move data in or out of the action for an operation. You can specify as many mappings as you like for an action. The mappings are categorized as global, input, and output mappings:

- Global Mappings: Use the global mappings to provide data that is applicable to the entire action. You can use them in an in/out parameter or a data-holder/provider structure. An example of a global mapping is a global variable specified within a Drools Rule Language (DRL) file.
- Input Mappings: Use the input mappings to provide data that represents parameters provided to an action. An example of an input mapping for BPM is a process variable used while starting a business process. An example of an input mapping for Rules is a fact to insert into a rules engine session.
- Output Mappings: Use the output mappings to return data out of an action. An example of an output mapping is a BPM process variable that you want to set as the outgoing (response) message's content.

8.6.4. MVEL expressionType

The mappings support a default ***expressionType*** called MVEL. You can use following variables with MVEL:

- exchange: The current **org.switchyard.Exchange**.
- context: The current **org.switchyard.Context**.
- message: The current **org.switchyard.Message**.

Here are some examples of the expressions using default variables:

```
expression="message.content" - This is the same as message.getContent().
expression="context['foo']" scope="IN" - This is the same as context.getProperty("foo",
Scope.IN).getValue(), in a null-safe manner.
```

NOTE

Specify the scope attribute only if you use the context variable inside your expression. If you do not specify a scope, the default Context access is done with ***Scope.EXCHANGE*** for global mappings, ***Scope.IN*** for input mappings, and ***Scope.OUT*** for output mappings.

It is important to specify a global variable for a rule, or a process variable to put into (or get out of) a BPM process. However, if you need to use the result of an expression as facts for rule session insertion, then you need not specify a variable name. Here is an XML example of how you can specify variable name:

```
<mapping expression="theExpression" expressionType="MVEL" scope="IN"
variable="theVariable"/>
```

8.6.5. Channels

Drools support the use of Channels that are the exit points in your Drools Rule Language (DRL) file. Channels must implement **org.kie.runtime.Channel**. Here is an example illustrating the use of channels in a method:

```
package com.example
rule "example rule"
   when
      $f : Foo ( bar > 10 )
   then
      channels["Bar"].send( $f.getBar() );
end
```

The following example illustrates use of channels in XML:

```
<channels>
   <channel class="com.example.BarChannel" name="Bar"/>
</channels>
```

8.6.6. SwitchYard Service Channel

SwitchYard provides an out-of-the-box channel called SwitchYard Service channel, which allows you to invoke (one-way) other SwitchYard services directly and easily from your DRL. Here is an example:

```
<channel name="HelloWorld" reference="HelloWorld" operation="greet"/>
```

Here,

- class: The channel implementation class. Default value is SwitchYardServiceChannel.
- name: The channel name.

- reference: The service reference qualified name.
- operation: The service reference operation name.
- input: The service reference operation input name.

8.6.7. Listeners

You can use Listeners to monitor specific types of events that occur during Knowledge execution. For example, you can use listeners to audit a BPM process and save the audit details into a database while the process progresses. The listener then reports the audit details at a later time. Drools and jBPM provide many out-of-the-box listeners. If you write your own listener, ensure that it implements **java.util.EventListener**.

To register your listener, you must implement one of the **KIE/Drools/jBPM Listener** interfaces. For example:

- **org.drools.event.WorkingMemoryEventListener**
- **org.drools.event.AgendaEventListener**
- **org.kie.event.process.ProcessEventListener**

Here is an example of listener implementation:

```
<listeners><listener class="org.drools.event.DebugProcessEventListener"/>
<listener class="org.kie.event.rule.DebugWorkingMemoryEventListener"/>
<listener class="com.example.MyListener"/>
</listeners>
```

8.6.8. Loggers

Loggers are special types of listeners, which you can use to output the events that occur during Knowledge execution. Support for loggers use a dedicated configuration element. You can log events to the console or a file. If events log are directed to a file, you can open those logs with the JBoss Developer Studio or JBoss Developer Studio Integration Stack. Here is an example of a logger implementation:

```
<loggers>
   <logger interval="2000" log="myLog" type="THREADED_FILE"/>
   <logger type="CONSOLE/>
</loggers>
```

8.7. MANIFEST

8.7.1. Manifest

The Manifest is where you specify from where the "intelligence" of the component is to come. For the BPM component, you need to specify, at the minimum, the location of the BPMN 2 process definition file. For the Rules component, you can specify the location of DRL, DSL, DSLR or XLS files.

8.7.2. Ways of Configuring the Manifest

NOTE

The following code examples assume there is a DRL file located on the classpath at **com/example/MyRules.drl**.

There are two ways to to configure the Manifest:

- with a KIE Container. (This relies upon the existence of a **META-INF/kmodule.xml** configuration file.)

  Here is the sample **META-INF/kmodule.xml** file:

  ```
  <kmodule xmlns="http://jboss.org/kie/6.0.0/kmodule">
      <kbase name="com.example">
          <ksession name="my-session"/>
      </kbase>
  </kmodule>
  ```

  Here is the sample XML file:

  ```
  <manifest>
      <container sessionName="my-session"/>
  </manifest>
  ```

  In addition to the sessionName attribute, you can also specify baseName and releaseId, if you desire.

  To enable it to scan for updates, simply set **scan="true"** and, optionally,

  ```
  scanInterval=<# of milliseconds>
  ```

- with a manually defined list of resources.

  Here is the sample XML file:

  ```
  <manifest>
      <resources>
          <resource location="com/example/MyProcess.bpmn" type="BPMN2"/>
          <resource location="com/example/MyRules.drl" type="DRL"/>
      </resources>
  </manifest>
  ```

IMPORTANT

These two options are mutually exclusive: You have to choose one or the other.

8.8. PROPERTIES

8.8.1. Properties

Properties allow you to provide "hints" to the underlying KIE/Drools/jBPM runtime on how certain options are configured. Use them to avoid having to expose every single KIE/Drools/jBPM option as a configurable element or attribute within SwitchYard.

8.8.2. Add a Property

Procedure 8.11. Add a Property

1. Launch the **JBoss Developer Studio**'s SwitchYard Editor.
2. Open a Type Hierarchy that has a root of **org.kie.conf.Option**.
3. Here you can see the full list. Here is one example:

```
<properties>
    <property name="drools.clockType" value="pseudo"/>
    <property name="drools.eventProcessingMode" value="stream"/>
</properties>
```

CHAPTER 9. GATEWAYS

9.1. WHAT IS A GATEWAY

Gateway is a network point that enables the SwitchYard applications to interact with services outside the application. They provide connectivity to/from external systems

9.2. BINDINGS

The Service Component Architecture (SCA) binding provides a means by which SwitchYard services and SwitchYard-aware clients communicate with one another. It facilitates inter-application communication within a SwitchYard runtime and provides clustering of SwitchYard services in two or more SwitchYard instances.

This section provides details on the out of the box gateway bindings provided with the SwitchYard application.

9.2.1. SOAP

The SOAP component in SwitchYard provides SOAP-based web service binding support for services and references.

NOTE

More information about incorporating WS-Security is available in this guide.

See Also:

- Section 9.2.1.4.1, "Enable WS-Security"

9.2.1.1. Binding Services with SOAP

You can expose composite-level services as a SOAP-based web service using the

```
<binding.soap>
```

binding definition. The following configuration options are available:

wsdl

This is the location of the WSDL used to describe the web service endpoint. A relative path can be used if the WSDL is included in the deployed application. If the WSDL is located outside the application, then you can use a **file:** or **http:** URL.

socketAddr

This is the IP Socket Address to be used. The value can be in the form of **hostName/ipAddress:portNumber**, **hostName/ipAddress** or **:portNumber**.

wsdlPort

This is the port name in the WSDL to use. If you leave it unspecified, the first port definition in the WSDL is used for the service endpoint.

contextPath

This is an additional context path for the SOAP endpoint. (The default is none.)

NOTE

By default, the JBossWS-CXF stack is enabled on JBoss EAP, so the **socketAddr** parameter is ignored. However, this parameter can be used for standalone usage.

Example 9.1. Sample SOAP Service Binding

```
<sca:composite name="orders" targetNamespace="urn:switchyard-quickstart-demo:orders:0.1.0">
    <sca:service name="OrderService" promote="OrderService">
        <soap:binding.soap>
            <soap:wsdl>wsdl/OrderService.wsdl</soap:wsdl>
            <soap:socketAddr>:9000</soap:socketAddr>
        </soap:binding.soap>
    </sca:service>
</sca:composite>
```

9.2.1.2. Binding References with SOAP

You can bind references with SOAP to make SOAP-based web services available to SwitchYard services. The following configuration options are available:

wsdl

This is the location of the WSDL used to describe the web service endpoint. A relative path can be used if the WSDL is included in the deployed application. (If the WSDL is located outside the application, then you can use a file: or http: URL.)

wsdlPort

This is the port name in the WSDL to use. If you leave it unspecified, the first port definition in the WSDL is used for the service endpoint.

endpointAddress

This SOAP endpoint address overrides the address specified in the WSDL. This is an optional property. If you do not specify it, the endpoint address specified in the WSDL is used instead.

timeout

This is the requests timeout value in milliseconds.

proxy

This is the HTTP Proxy settings for the endpoint.

basic/ntlm

This is the authentication configuration for the endpoint.

Example 9.2. Sample SOAP Reference Binding

```
<sca:composite name="orders" targetNamespace="urn:switchyard-quickstart-demo:orders:0.1.0">
    <sca:reference name="WarehouseService" promote="OrderComponent/WarehouseService"
multiplicity="1..1">
            <soap:binding.soap>
               <soap:wsdl>wsdl/OrderService.wsdl</soap:wsdl>
               <soap:endpointAddress></soap:endpointAddress>
            </soap:binding.soap>
    </sca:reference>
</sca:composite>
```

9.2.1.2.1. Proxy Configuration

If you need the SOAP reference to pass through a proxy server, then provide the proxy server configuration using the proxy element. The following configuration options are available:

- type : The proxy type. This can be HTTP or SOCKS. The default is HTTP.
- host : The proxy host.
- port : The proxy port (optional).
- user : The proxy user (optional).
- password : The proxy password (optional).

Example 9.3. Sample SOAP Proxy Configuration

```
<sca:composite name="orders" targetNamespace="urn:switchyard-quickstart-demo:orders:0.1.0">
    <sca:reference name="WarehouseService" promote="OrderComponent/WarehouseService"
multiplicity="1..1">
        <soap:binding.soap>
            <soap:wsdl>wsdl/OrderService.wsdl</soap:wsdl>
            <soap:endpointAddress>[</soap:endpointAddress>
            <soap:proxy>
                <soap:type>HTTP</soap:type>
                <soap:host>192.168.1.2</soap:host>
                <soap:port>9090</soap:port>
                <soap:user>user</soap:user>
                <soap:password>password</soap:password>
            </soap:proxy>
        </soap:binding.soap>
    </sca:reference>
</sca:composite>
```

9.2.1.2.2. Authentication Configuration

If the SOAP reference endpoint is secured using BASIC or NTLM, then you can provide the authentication configuration using the BASIC or NTLM elements. The following configuration options are available:

- user : The user name.

- password : The password.
- host : The authentication host (optional).
- port : The authentication port (optional).
- realm : The authentication realm (optional, applicable only for BASIC).
- domain: The Windows domain for authentication (optional, applicable only for NTLM).

Example 9.4. Sample NTLM Authentication Configuration

```
<sca:composite name="orders" targetNamespace="urn:switchyard-quickstart-demo:orders:0.1.0">
  <sca:reference name="WarehouseService" promote="OrderComponent/WarehouseService"
multiplicity="1..1">
      <soap:binding.soap>
         <soap:wsdl>wsdl/OrderService.wsdl</soap:wsdl>
         <soap:endpointAddress>[</soap:endpointAddress>
         <soap:ntlm>
            <soap:user>user</soap:user>
            <soap:password>password</soap:password>
            <soap:domain>domain</soap:domain>
         </soap:ntlm>
      </soap:binding.soap>
  </sca:reference>
</sca:composite>
```

9.2.1.3. Enabling SOAP Message Logging for SOAP Binding

To log inbound and outbound SOAP messages on SOAP binding, turn on **DEBUG** level logging for **switchyard.component.soap.InboundHandler** and **switchyard.component.soap.OutboundHandler**.

- Add the following to **EAP_HOME/standalone/configuration/standalone.xml**:

```
<logger category="org.switchyard.component.soap.InboundHandler">
   <level name="DEBUG" />
</logger>
<logger category="org.switchyard.component.soap.OutboundHandler">
   <level name="DEBUG" />
</logger>
```

9.2.1.4. WS-Security

9.2.1.4.1. Enable WS-Security

Procedure 9.1. Enable WS-Security

1. Define a Policy within your WSDL and reference it with a PolicyReference inside your binding.
2. Configure your <soap.binding> with an <endpointConfig> to ensure that JBossWS-CXF is configured appropriately.

3. Configure your <soap.binding> with an <inInterceptors> section, including the appropriate JBossWS-CXF <interceptor> to handle incoming SOAP requests.

4. Include a **WEB-INF/jboss-web.xml** file in your application with a <security-domain> specified, so that JBossWS-CXF knows which modules to use for authentication and role mapping.

9.2.1.4.2. Sample WS-Security Configurations

JBoss Fuse provides the **policy-security-wss-username** quickstart application as an example. The following are the pertinent sections:

- **META-INF/WorkService.wsdl**:

```
<binding name="WorkServiceBinding" type="tns:WorkService">
  <wsp:PolicyReference URI="#WorkServicePolicy"/>
  ...
</binding>
<wsp:Policy wsu:Id="WorkServicePolicy">
  <wsp:ExactlyOne>
    <wsp:All>
      <sp:SupportingTokens xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-
securitypolicy/200702">
        <wsp:Policy>
          <sp:UsernameToken sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-
securitypolicy/200702/IncludeToken/AlwaysToRecipient">
            <wsp:Policy>
              <sp:WssUsernameToken10/>
            </wsp:Policy>
          </sp:UsernameToken>
        </wsp:Policy>
      </sp:SupportingTokens>
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
```

- **META-INF/switchyard.xml**:

```
<binding.soap xmlns="urn:switchyard-component-soap:config:1.0">
  <wsdl>META-INF/WorkService.wsdl</wsdl>
  <contextPath>policy-security-wss-username</contextPath>
  <endpointConfig configFile="META-INF/jaxws-endpoint-config.xml"
configName="SwitchYard-Endpoint-Config"/>
  <inInterceptors>
    <interceptor
class="org.jboss.wsf.stack.cxf.security.authentication.SubjectCreatingPolicyInterceptor"/>
  </inInterceptors>
</binding.soap>
```

- **META-INF/jaxws-endpoint-config.xml**:

```
<jaxws-config xmlns="urn:jboss:jbossws-jaxws-config:4.0"
xmlns:javaee="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:jboss:jbossws-jaxws-config:4.0 schema/jbossws-jaxws-
config_4_0.xsd">
```

```
    <endpoint-config>
      <config-name>SwitchYard-Endpoint-Config</config-name>
      <property>
          <property-name>ws-security.validate.token</property-name>
          <property-value>false</property-value>
      </property>
    </endpoint-config>
  </jaxws-config>
```

- **WEB-INF/jboss-web.xml**:

```
<jboss-web>
   <security-domain>java:/jaas/other</security-domain>
</jboss-web>
```

With these in place, JBossWS-CXF intercepts incoming SOAP requests, extract the **UsernameToken**, attempt to authenticate it against the LoginModule(s) configured in the application server's "other" security domain, and provide any authorized roles. If successful, the request is handed over to SwitchYard, which processes it further, including enforcing your own policies. In the case of WS-Security, SwitchYard does not attempt a second clientAuthentication, but instead respects the outcome from JBossWS-CXF.

NOTE

If the original clientAuthentication fails, this is a "fail-fast" scenario, and the request is not channeled into SwitchYard.

9.2.1.4.3. Signature and Encryption Support

To setup WS-Security encryption, ensure the following:

- The Policy in your WSDL must reflect the added requirements. See Section 9.2.1.4.3.1, "Sample Endpoint Configurations", Section 9.2.1.4.3.2, "Sample Client Configurations", and Section 9.2.1.4.3.3, "Endpoint Serving Multiple Clients".
- Configure your <soap.binding> with an <endpointConfig> to ensure JBossWS-CXF is configured appropriately.

For example:

META-INF/switchyard.xml

```
<binding.soap xmlns="urn:switchyard-component-soap:config:1.0">
  <wsdl>META-INF/WorkService.wsdl</wsdl>
  <contextPath>policy-security-wss-username</contextPath>
  <endpointConfig configFile="META-INF/jaxws-endpoint-config.xml" configName="SwitchYard-
Endpoint-Config">
</binding.soap>
```

META-INF/jaxws-endpoint-config.xml

```
<jaxws-config xmlns="urn:jboss:jbossws-jaxws-config:4.0"
xmlns:javaee="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
xsi:schemaLocation="urn:jboss:jbossws-jaxws-config:4.0 schema/jbossws-jaxws-
config_4_0.xsd">
  <endpoint-config>
    <config-name>SwitchYard-Endpoint-Config</config-name>
    <property>
      <property-name>ws-security.callback-handler</property-name>
      <property-
value>org.switchyard.quickstarts.demo.policy.security.wss.signencrypt.WorkServiceCallbackHandler
/property-value>
    </property>
    <property>
      <property-name>ws-security.encryption.properties</property-name>
      <property-value>META-INF/bob.properties</property-value>
    </property>
    <property>
      <property-name>ws-security.encryption.username</property-name>
      <property-value>alice</property-value>
    </property>
    <property>
      <property-name>ws-security.signature.properties</property-name>
      <property-value>META-INF/bob.properties</property-value>
    </property>
    <property>
      <property-name>ws-security.signature.username</property-name>
      <property-value>bob</property-value>
    </property>
  </endpoint-config>
</jaxws-config>
```

META-INF/bob.properties

```
org.apache.ws.security.crypto.provider=org.apache.ws.security.components.crypto.Merlin
org.apache.ws.security.crypto.merlin.keystore.type=jks
org.apache.ws.security.crypto.merlin.keystore.password=password
org.apache.ws.security.crypto.merlin.keystore.alias=bob
org.apache.ws.security.crypto.merlin.file=META-INF/bob.jks
```

See Also:

- Section 9.2.1.4.4, “Sample CXF Interceptor Configurations”

9.2.1.4.3.1. Sample Endpoint Configurations

An endpoint declares all the abstract methods that are exposed to the client. You can use endpoint configurations to include property declarations. The endpoint implementations can be associated with a given endpoint configuration using the @EndpointConfig annotation. The following steps describe a sample endpoint configuration:

1. Create the web service endpoint using JAX-WS. Use a contract-first approach when using WS-Security as the policies declared in the WSDL are parsed by the Apache CXF engine on both server and client sides. Here is an example of WSDL contract enforcing signature and encryption using X 509 certificates:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<definitions targetNamespace="http://www.jboss.org/jbossws/ws-
extensions/wssecuritypolicy" name="SecurityService"
        xmlns:tns="http://www.jboss.org/jbossws/ws-extensions/wssecuritypolicy"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
        xmlns="http://schemas.xmlsoap.org/wsdl/"
        xmlns:wsp="http://www.w3.org/ns/ws-policy"
        xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-
utility-1.0.xsd"
        xmlns:wsaws="http://www.w3.org/2005/08/addressing"
        xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
  <types>
    <xsd:schema>
      <xsd:import namespace="http://www.jboss.org/jbossws/ws-extensions/wssecuritypolicy"
schemaLocation="SecurityService_schema1.xsd"/>
    </xsd:schema>
  </types>
  <message name="sayHello">
    <part name="parameters" element="tns:sayHello"/>
  </message>
  <message name="sayHelloResponse">
    <part name="parameters" element="tns:sayHelloResponse"/>
  </message>
  <portType name="ServiceIface">
    <operation name="sayHello">
      <input message="tns:sayHello"/>
      <output message="tns:sayHelloResponse"/>
    </operation>
  </portType>
  <binding name="SecurityServicePortBinding" type="tns:ServiceIface">
    <wsp:PolicyReference URI="#SecurityServiceSignThenEncryptPolicy"/>
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
    <operation name="sayHello">
      <soap:operation soapAction=""/>
      <input>
        <soap:body use="literal"/>
      </input>
      <output>
        <soap:body use="literal"/>
      </output>
    </operation>
  </binding>
  <service name="SecurityService">
    <port name="SecurityServicePort" binding="tns:SecurityServicePortBinding">
      <soap:address location="http://localhost:8080/jaxws-samples-wssePolicy-sign-encrypt"/>
    </port>
  </service>

  <wsp:Policy wsu:Id="SecurityServiceSignThenEncryptPolicy"
xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
    <wsp:ExactlyOne>
      <wsp:All>
        <sp:AsymmetricBinding
xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
          <wsp:Policy>
```

```
            <sp:InitiatorToken>
              <wsp:Policy>
                <sp:X509Token
sp:IncludeToken="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy/IncludeToken/Always
ToRecipient">
                  <wsp:Policy>
                    <sp:WssX509V1Token11/>
                  </wsp:Policy>
                  </sp:X509Token>
              </wsp:Policy>
            </sp:InitiatorToken>
            <sp:RecipientToken>
              <wsp:Policy>
                <sp:X509Token
sp:IncludeToken="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy/IncludeToken/Never"
>
                  <wsp:Policy>
                    <sp:WssX509V1Token11/>
                  </wsp:Policy>
                </sp:X509Token>
              </wsp:Policy>
            </sp:RecipientToken>
            <sp:AlgorithmSuite>
              <wsp:Policy>
                <sp-cxf:Basic128GCM xmlns:sp-cxf="http://cxf.apache.org/custom/security-
policy"/>
              </wsp:Policy>
            </sp:AlgorithmSuite>
            <sp:Layout>
              <wsp:Policy>
                <sp:Lax/>
              </wsp:Policy>
            </sp:Layout>
            <sp:IncludeTimestamp/>
            <sp:EncryptSignature/>
            <sp:OnlySignEntireHeadersAndBody/>
            <sp:SignBeforeEncrypting/>
          </wsp:Policy>
        </sp:AsymmetricBinding>
        <sp:SignedParts xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
          <sp:Body/>
        </sp:SignedParts>
        <sp:EncryptedParts xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
          <sp:Body/>
        </sp:EncryptedParts>
        <sp:Wss10 xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy">
          <wsp:Policy>
            <sp:MustSupportRefIssuerSerial/>
          </wsp:Policy>
        </sp:Wss10>
      </wsp:All>
    </wsp:ExactlyOne>
  </wsp:Policy>
</definitions>
```

You can generate the service endpoint using the **wsconsume** tool and then use a @EndpointConfig annotation as shown below:

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.basic;

import javax.jws.WebService;
import org.jboss.ws.api.annotation.EndpointConfig;

@WebService
(
  portName = "SecurityServicePort",
  serviceName = "SecurityService",
  wsdlLocation = "WEB-INF/wsdl/SecurityService.wsdl",
  targetNamespace = "http://www.jboss.org/jbossws/ws-extensions/wssecuritypolicy",
  endpointInterface = "org.jboss.test.ws.jaxws.samples.wsse.policy.basic.ServiceIface"
)
@EndpointConfig(configFile = "WEB-INF/jaxws-endpoint-config.xml", configName = "Custom
WS-Security Endpoint")
public class ServiceImpl implements ServiceIface
{
  public String sayHello()
  {
    return "Secure Hello World!";
  }
}
```

2. Use the referenced **jaxws-endpoint-config.xml** descriptor to provide a custom endpoint configuration with the required server side configuration properties as shown below. This tells the engine which certificate or key to use for signature, signature verification, encryption, and decryption.

```
<?xml version="1.0" encoding="UTF-8"?>
<jaxws-config xmlns="urn:jboss:jbossws-jaxws-config:4.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:javaee="http://java.sun.com/xml/ns/javaee" xsi:schemaLocation="urn:jboss:jbossws-
jaxws-config:4.0 schema/jbossws-jaxws-config_4_0.xsd">
 <endpoint-config>
  <config-name>Custom WS-Security Endpoint</config-name>
  <property>
   <property-name>ws-security.signature.properties</property-name>
   <property-value>bob.properties</property-value>
  </property>
  <property>
   <property-name>ws-security.encryption.properties</property-name>
   <property-value>bob.properties</property-value>
  </property>
  <property>
   <property-name>ws-security.signature.username</property-name>
   <property-value>bob</property-value>
  </property>
  <property>
   <property-name>ws-security.encryption.username</property-name>
   <property-value>alice</property-value>
```

```
    </property>
    <property>
      <property-name>ws-security.callback-handler</property-name>
      <property-
value>org.jboss.test.ws.jaxws.samples.wsse.policy.basic.KeystorePasswordCallback</prope
rty-value>
    </property>
  </endpoint-config>
</jaxws-config>
```

Here,

- The **bob.properties** configuration file includes the WSS4J Crypto properties which in turn links to the keystore file, type, alias, and password for accessing it. For example:

```
org.apache.ws.security.crypto.provider=org.apache.ws.security.components.crypto.Merlin

org.apache.ws.security.crypto.merlin.keystore.type=jks
org.apache.ws.security.crypto.merlin.keystore.password=password
org.apache.ws.security.crypto.merlin.keystore.alias=bob
org.apache.ws.security.crypto.merlin.keystore.file=bob.jks
```

- The callback handler enables Apache CXF to access the keystore. For example:

```
package org.jboss.test.ws.jaxws.samples.wsse.policy.basic;

import java.io.IOException;
import java.util.HashMap;
import java.util.Map;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.UnsupportedCallbackException;
import org.apache.ws.security.WSPasswordCallback;

public class KeystorePasswordCallback implements CallbackHandler {
  private Map<String, String> passwords = new HashMap<String, String>();

  public KeystorePasswordCallback() {
    passwords.put("alice", "password");
    passwords.put("bob", "password");
  }

  /**
   * It attempts to get the password from the private
   * alias/passwords map.
   */
  public void handle(Callback[] callbacks) throws IOException,
UnsupportedCallbackException {
    for (int i = 0; i < callbacks.length; i++) {
      WSPasswordCallback pc = (WSPasswordCallback)callbacks[i];

      String pass = passwords.get(pc.getIdentifier());
```

```
            if (pass != null) {
               pc.setPassword(pass);
               return;
            }
         }
      }

      /**
       * Add an alias/password pair to the callback mechanism.
       */
      public void setAliasPassword(String alias, String password) {
         passwords.put(alias, password);
      }
   }
```

3. Assuming the **bob.jks** keystore is properly generated and contains the server Bob's full key as well as the client Alice's public key, you can proceed to packaging the endpoint. Here is the expected content:

```
 /dati/jbossws/stack/cxf/trunk $ jar -tvf ./modules/testsuite/cxf-tests/target/test-libs/jaxws-
samples-wsse-policy-sign-encrypt.war
     0 Thu Jun 16 18:50:48 CEST 2011 META-INF/
   140 Thu Jun 16 18:50:46 CEST 2011 META-INF/MANIFEST.MF
     0 Thu Jun 16 18:50:48 CEST 2011 WEB-INF/
   586 Thu Jun 16 18:50:44 CEST 2011 WEB-INF/web.xml
     0 Thu Jun 16 18:50:48 CEST 2011 WEB-INF/classes/
     0 Thu Jun 16 18:50:48 CEST 2011 WEB-INF/classes/org/
     0 Thu Jun 16 18:50:48 CEST 2011 WEB-INF/classes/org/jboss/
     0 Thu Jun 16 18:50:48 CEST 2011 WEB-INF/classes/org/jboss/test/
     0 Thu Jun 16 18:50:48 CEST 2011 WEB-INF/classes/org/jboss/test/ws/
     0 Thu Jun 16 18:50:48 CEST 2011 WEB-INF/classes/org/jboss/test/ws/jaxws/
     0 Thu Jun 16 18:50:48 CEST 2011 WEB-INF/classes/org/jboss/test/ws/jaxws/samples/
     0 Thu Jun 16 18:50:48 CEST 2011 WEB-
INF/classes/org/jboss/test/ws/jaxws/samples/wsse/
     0 Thu Jun 16 18:50:48 CEST 2011 WEB-
INF/classes/org/jboss/test/ws/jaxws/samples/wsse/policy/
     0 Thu Jun 16 18:50:48 CEST 2011 WEB-
INF/classes/org/jboss/test/ws/jaxws/samples/wsse/policy/basic/
  1687 Thu Jun 16 18:50:48 CEST 2011 WEB-
INF/classes/org/jboss/test/ws/jaxws/samples/wsse/policy/basic/KeystorePasswordCallback.clas

   383 Thu Jun 16 18:50:48 CEST 2011 WEB-
INF/classes/org/jboss/test/ws/jaxws/samples/wsse/policy/basic/ServiceIface.class
  1070 Thu Jun 16 18:50:48 CEST 2011 WEB-
INF/classes/org/jboss/test/ws/jaxws/samples/wsse/policy/basic/ServiceImpl.class
     0 Thu Jun 16 18:50:48 CEST 2011 WEB-
INF/classes/org/jboss/test/ws/jaxws/samples/wsse/policy/jaxws/
   705 Thu Jun 16 18:50:48 CEST 2011 WEB-
INF/classes/org/jboss/test/ws/jaxws/samples/wsse/policy/jaxws/SayHello.class
  1069 Thu Jun 16 18:50:48 CEST 2011 WEB-
INF/classes/org/jboss/test/ws/jaxws/samples/wsse/policy/jaxws/SayHelloResponse.class
  1225 Thu Jun 16 18:50:44 CEST 2011 WEB-INF/jaxws-endpoint-config.xml
     0 Thu Jun 16 18:50:44 CEST 2011 WEB-INF/wsdl/
  4086 Thu Jun 16 18:50:44 CEST 2011 WEB-INF/wsdl/SecurityService.wsdl
  653 Thu Jun 16 18:50:44 CEST 2011 WEB-INF/wsdl/SecurityService_schema1.xsd
 1820 Thu Jun 16 18:50:44 CEST 2011 WEB-INF/classes/bob.jks
  311 Thu Jun 16 18:50:44 CEST 2011 WEB-INF/classes/bob.properties
```

Here, the jaxws classes generated by the tools and a basic web.xml referencing the endpoint bean are also included:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app
  version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
  <servlet>
    <servlet-name>TestService</servlet-name>
    <servlet-class>org.jboss.test.ws.jaxws.samples.wsse.policy.basic.ServiceImpl</servlet-
class>
  </servlet>
  <servlet-mapping>
    <servlet-name>TestService</servlet-name>
    <url-pattern>/*</url-pattern>
  </servlet-mapping>
</web-app>
```

NOTE

If you are deploying the endpoint archive to JBoss Application Server 7, add a dependency to **org.apache.ws.security** module in the **MANIFEST.MF** file:

```
Manifest-Version: 1.0
Ant-Version: Apache Ant 1.7.1
Created-By: 17.0-b16 (Sun Microsystems Inc.)
Dependencies: org.apache.ws.security
```

9.2.1.4.3.2. Sample Client Configurations

On the client side, use the **wsconsume** tool to consume the published WSDL and then invoke the endpoint as a standard JAX-WS one as shown below:

```
QName serviceName = new QName("http://www.jboss.org/jbossws/ws-extensions/wssecuritypolicy",
"SecurityService");
URL wsdlURL = new URL(serviceURL + "?wsdl");
Service service = Service.create(wsdlURL, serviceName);
ServiceIface proxy = (ServiceIface)service.getPort(ServiceIface.class);

((BindingProvider)proxy).getRequestContext().put(SecurityConstants.CALLBACK_HANDLER, new
KeystorePasswordCallback());
((BindingProvider)proxy).getRequestContext().put(SecurityConstants.SIGNATURE_PROPERTIES,
   Thread.currentThread().getContextClassLoader().getResource("META-INF/alice.properties"));
((BindingProvider)proxy).getRequestContext().put(SecurityConstants.ENCRYPT_PROPERTIES,
   Thread.currentThread().getContextClassLoader().getResource("META-INF/alice.properties"));
```

```
((BindingProvider)proxy).getRequestContext().put(SecurityConstants.SIGNATURE_USERNAME,
"alice");
((BindingProvider)proxy).getRequestContext().put(SecurityConstants.ENCRYPT_USERNAME,
"bob");

proxy.sayHello();
```

The WS-Security properties are set in the request context. Here, the ***KeystorePasswordCallback*** is same as that on the server side. The **alice.properties** file is the client side equivalent of the server side **bob.properties** file and references the **alice.jks** keystore file, which has been populated with client Alice's full key as well as server Bob's public key:

```
org.apache.ws.security.crypto.provider=org.apache.ws.security.components.crypto.Merlin
org.apache.ws.security.crypto.merlin.keystore.type=jks
org.apache.ws.security.crypto.merlin.keystore.password=password
org.apache.ws.security.crypto.merlin.keystore.alias=alice
org.apache.ws.security.crypto.merlin.keystore.file=META-INF/alice.jks
```

The Apache CXF WS-Policy engine consumes the security requirements in the contract and ensures that a valid secure communication is in place for interacting with the server endpoint.

9.2.1.4.3.3. Endpoint Serving Multiple Clients

n the endpoint and client configuration examples, the server side configuration implies that the endpoint is configured for serving a given client which a service agreement is established for. In real world scenarios, a server should be able to decrypt and encrypt messages coming from and being sent to multiple clients. Apache CXF supports that through the ***useReqSigCert*** value for the ***ws-security.encryption.username*** configuration parameter. The referenced server side keystore then needs to contain the public key of all the clients that are expected to be served.

9.2.1.4.4. Sample CXF Interceptor Configurations

For adding a CXF Interceptor, perform the following configuration settings:

- **META-INF/switchyard.xml**

```
<binding.soap xmlns="urn:switchyard-component-soap:config:1.0">
  <wsdl>META-INF/WorkService.wsdl</wsdl>
  <contextPath>policy-security-wss-username</contextPath>
  <inInterceptors>
    <interceptor class="com.example.MyInterceptor"/>
  </inInterceptors>
</binding.soap>
```

- **com/example/MyInterceptor.java**

```
public class MyInterceptor extends WSS4JInterceptor {
  private static final PROPS;
```

```
    static {
      Map<String,String> props = new HashMap<String,String>();
      props.put("action", "Signature Encryption");
      props.put("signaturePropFile", "META-INF/bob.properties");
      props.put("decryptionPropFile", "META-INF/bob.properties");
      props.put("passwordCallbackClass", "com.example.MyCallbackHandler");
      PROPS = props;
    }
    public MyInterceptor() {
      super(PROPS);
    }
}
```

- **META-INF/bob.properties**

```
org.apache.ws.security.crypto.provider=org.apache.ws.security.components.crypto.Merlin
org.apache.ws.security.crypto.merlin.keystore.type=jks
org.apache.ws.security.crypto.merlin.keystore.password=password
org.apache.ws.security.crypto.merlin.keystore.alias=bob
org.apache.ws.security.crypto.merlin.file=META-INF/bob.jks
```

9.2.1.5. Attachments

9.2.1.5.1. SOAP with Attachments

By default, any attachment sent with a SOAP Envelope is passed around in a SwitchYard Message as an attachment. The default **SOAPMessageComposer** handles this.

9.2.1.5.2. SOAP with MTOM/XOP

To support Message Transmission Optimization Mechanism (MTOM), the underlying stack sends and receives attachments as MIME multipart messages. One additional configuration in SwitchYard that allows you to expand an xop:include's SOAP Message is **mtom**. When the corresponding **xopExpand** attribute is set to true, the xop:include element is replaced with the contents from the MIME attachment.

Example 9.5.

```
<soap:binding.soap xmlns:soap="urn:switchyard-component-soap:config:1.0">
    <soap:wsdl>Foo.wsdl</soap:wsdl>
    <soap:endpointAddress></soap:endpointAddress>
    <soap:mtom enabled="true" xopExpand="true"/>
</soap:binding.soap>
```

You can enable MTOM by overriding as shown above, or by using WSDL policy as shown below:

```
<definitions targetNamespace="urn:switchyard-component-soap:test-ws:1.0" name="ImageService"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:tns="urn:switchyard-component-soap:test-ws:1.0"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap12/"
    xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"
    xmlns:wsp="http://www.w3.org/ns/ws-policy"
    xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdl"
    xmlns:xmime="http://www.w3.org/2005/05/xmlmime"
    xmlns:wsoma="http://www.w3.org/2007/08/soap12-mtom-policy"
    xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd">
...
    <wsp:Policy wsu:Id="ImageServicePortBinding_policy">
        <wsp:ExactlyOne>
            <wsp:All>
                <wsoma:MTOM/>
            </wsp:All>
        </wsp:ExactlyOne>
    </wsp:Policy>
...
  <binding name="ImageServicePortBinding" type="tns:ImageService">
    <wsp:PolicyReference URI="#ImageServicePortBinding_policy"/>
    <soap:binding transport="http://www.w3.org/2003/05/soap/bindings/HTTP/" style="document"/>
    <operation name="resize">
      <soap:operation soapAction=""/>
      <input>
        <soap:body use="literal"/>
      </input>
      <output>
        <soap:body use="literal"/>
      </output>
    </operation>
  </binding>
```

9.2.1.6. WS-Addressing

SwitchYard runtime provides support for WS-A (WS-Addressing) through the underlying SOAP stack. To enable WS-A, you can either set a policy or use the ***UseAdrressing*** element in the WSDL as shown below:

```
<definitions targetNamespace="urn:switchyard-component-soap:test-ws:1.0"
name="HelloAddressingService"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:tns="urn:switchyard-component-soap:test-ws:1.0"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap12/"
    xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"
    xmlns:wsp="http://www.w3.org/ns/ws-policy"
    xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdl"
    xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd">
...
    <wsp:Policy wsu:Id="HelloSOAPAddressingServicePortBinding_policy">
        <wsp:ExactlyOne>
            <wsp:All>
                <wsam:Addressing wsdl:required="false">
                    <wsp:Policy/>
                </wsam:Addressing>
             </wsp:All>
          </wsp:ExactlyOne>
       </wsp:Policy>
...
    <binding name="HelloSOAPAddressingServicePortBinding" type="tns:HelloAddressingService">
      <wsp:PolicyReference URI="#HelloSOAPAddressingServicePortBinding_policy"/>
      <soap:binding transport="http://www.w3.org/2003/05/soap/bindings/HTTP/" style="document"/>
      <operation name="sayHello">
        <soap:operation soapAction=""/>
        <input>
          <soap:body use="literal"/>
        </input>
        <output>
          <soap:body use="literal"/>
        </output>
      </operation>
    </binding>
...
    <binding name="HelloSOAPAddressingServicePortBinding2" type="tns:HelloAddressingService2">
      <wsaw:UsingAddressing required="true" />
      <soap:binding transport="http://www.w3.org/2003/05/soap/bindings/HTTP/" style="document"/>
      <operation name="sayHello">
        <soap:operation soapAction=""/>
        <input>
          <soap:body use="literal"/>
        </input>
        <output>
          <soap:body use="literal"/>
        </output>
      </operation>
    </binding>
```

9.2.2. HTTP

9.2.2.1. HTTP Component

The HTTP component in SwitchYard provides HTTP-based binding support for services and references in SwitchYard.

9.2.2.2. Binding Services with HTTP

To expose composite-level services as an HTTP-based service, use the

```
<binding.http>
```

binding definition. You can use these configuration options:

operationSelector

This is the specification of the operation to use for the message exchange.

contextPath

This is the context path for the HTTP endpoint.

Here is an example HTTP service binding:

```
<sca:service name="QuoteService" promote="StockService/QuoteService">
   <http:binding.http>
      <selector:operationSelector operationName="getPrice"/>
      <http:contextPath>http-binding/quote</http:contextPath>
   </http:binding.http>
</sca:service>
```

NOTE

Do not edit the **SwitchYard.xml** file directly. Red Hat recommends using the JBoss Developer Studio SwitchYard Editor to edit the **SwitchYard.xml**file.

9.2.2.3. Binding References with HTTP

When you bind a reference with HTTP, it makes HTTP-based services available to SwitchYard services.

The following configuration options are available to you for binding.http when binding references:

address

This is a URL that points to an HTTP endpoint. It is optional and if you do not specify it, it defaults to http://127.0.0.1:8080/.

method

This is the HTTP method used for invoking the endpoint. (The default is GET.)

contentType

This is the HTTP content type header that must be set on the request.

Here is an example HTTP reference binding:

```
<sca:reference name="Symbol" promote="StockService/SymbolService" multiplicity="1..1">
   <http:binding.http>
      <http:address>http://localhost:8080/http-binding/symbol</http:address>
      <http:method>POST</http:method>
      <http:contentType>text/plain</http:contentType>
   </http:binding.http>
</sca:reference>
```

9.2.3. RESTEasy

9.2.3.1. About RESTEasy

RESTEasy is a portable implementation of the JAX-RS Java API. It also provides additional features, including a client side framework (the RESTEasy JAX-RS Client Framework) for mapping outgoing requests to remote servers, allowing JAX-RS to operate as a client or server-side specification.

9.2.3.2. RESTEasy Component

The RESTEasy component provides REST-based binding support for services and references in SwitchYard.

9.2.3.3. Binding Services with RESTEasy

To expose composite-level services as a REST-based service, use the

```
<binding.rest>
```

binding definition. The following configuration options are available for binding.rest when you are binding services:

interfaces

This is a comma separated list of interfaces or empty classes with JAX-RS annotations.

contextPath

This is an additional context path for the REST endpoint. (The default is setting is none.)

Here is an example REST service binding:

```
<sca:service name="OrderService" promote="OrderService/OrderService">
    <rest:binding.rest>

<rest:interfaces>org.switchyard.quickstarts.rest.binding.OrderResource,org.switchyard.quickstarts.rest.b
inding.TestResource</rest:interfaces>
        <rest:contextPath>rest-binding</rest:contextPath>
    </rest:binding.rest>
</sca:service>
```

For more information on RESTEasy Component's REST binding features, refer to the **rest-binding** quickstart.

9.2.3.4. Binding References with RESTEasy

When you bind a reference with RESTEasy, it makes REST-based services available to SwitchYard services.

The following configuration options are available to you for binding.rest when binding references:

address

This is a URL that points to the root path of resources. This is only applicable for Reference bindings. It is optional and if you do not specify it, it defaults to http://127.0.0.1:8080/.

interfaces

This is a comma-separated list of interfaces or abstract or empty classes with JAX-RS annotations.

contextPath

This is an additional context path for the REST endpoint. (The default is none.)

Here is an example HTTP reference binding. Note the resource URLs start from http://localhost:8080/rest-binding:

```
<sca:reference name="Warehouse" promote="OrderService/Warehouse" multiplicity="1..1">
    <rest:binding.rest>
        <rest:interfaces>org.switchyard.quickstarts.rest.binding.WarehouseResource</rest:interfaces>
        <rest:address>http://localhost:8080</rest:address>
        <rest:contextPath>rest-binding</rest:contextPath>
    </rest:binding.rest>
</sca:reference>
```

9.2.3.5. Proxy Configuration

If the REST reference needs to pass through a proxy server then the proxy server configuration can be provided using the proxy element. The following configuration options are available:

- **host** : The proxy host.
- **port** : The proxy port (optional).
- **user** : The proxy user (optional).
- **password** : The proxy password (optional).

Example 9.6.

```
<sca:reference name="Warehouse" promote="OrderService/Warehouse" multiplicity="1..1">
    <rest:binding.rest>

<rest:interfaces>org.switchyard.quickstarts.rest.binding.WarehouseResource</rest:interfaces>
        <rest:address></rest:address>
        <rest:proxy>
            <rest:host>host</rest:host>
            <rest:port>8090</rest:port>
            <rest:user>Beal</rest:user>
            <rest:password>conjecture</rest:password>
        </rest:proxy>
    </rest:binding.rest>
</sca:reference>
```

9.2.3.6. Authentication Configuration

If the REST reference endpoint is secured using BASIC/NTLM, then the authentication configuration can be provided using the **basic** or **ntlm** elements. The following configuration options are available:

- **user** : The authentication user.

  password : The authentication password.

  realm/**domain** : The authentication realm or the Windows domain.

Example 9.7. Sample NTLM Authentication Configuration

```
<sca:reference name="Warehouse" promote="OrderService/Warehouse" multiplicity="1..1">
    <rest:binding.rest>
```

```
<rest:interfaces>org.switchyard.quickstarts.rest.binding.WarehouseResource</rest:interfaces>
        <rest:address></rest:address>
        <rest:ntlm>
            <rest:user>user</rest:user>
            <rest:password>password</rest:password>
            <rest:domain>domain</rest:domain>
        </rest:ntlm>
    </rest:binding.rest>
</sca:reference>
```

9.2.4. JCA

9.2.4.1. Java Connector Architecture (JCA) Transport

The Java Connector Architecture (JCA) Transport is a Java-based piece of architecture that works as a service integrator. It is a connector that links application servers and enterprise information systems.

9.2.4.2. JCA Adapter

A JCA Adapter provides outbound and inbound connectivity between Enterprise Information Systems (for example, mainframe transaction processing and database systems), application servers, and enterprise applications. It controls the inflow of messages to Message-Driven Beans (MDBs) and the outflow of messages sent from other Java EE components. It also provides a variety of options to fine tune your messaging applications.

9.2.4.3. JCA Gateway

The JCA gateway allows you to send and receive messages to and from EIS by means of the JCA ResourceAdapter.

9.2.4.4. Binding Services with JCA Message Inflow

You can bind composite-level services to an EIS with JCA message inflow using the **<binding.jca>** binding definition. You require the following configuration options for **binding.jca**

- ***operationSelector***: Specification of the service operation used for invoking the message exchange. For more details, see Section 9.4.2, “Types of Operation Selectors” .
- ***inboundConnection***
  - ***resourceAdapter***
    - ***@name***: Name of the ResourceAdapter archive. Ensure that the resource adapter is deployed on the JBoss application server before you deploy the SwitchYard application which has JCA binding.
  - ***activationSpec***
    - ***property***: Properties for injecting into the ActivationSpec instance. Provide properties that are specific to the ResourceAdapter implementation.
- ***inboundInteraction***
  - ***listener***: A fully qualified name (FQN) of the listener interface. When you use JMSEndpoint,

specify the **javax.jms.MessageListener** and when you use CCIEndpoint, specify the **javax.resource.cci.MessageListener**. Otherwise, you may need to specify EIS specific listener interface according to its ResourceAdapter. Also ensure that the endpoint class implements this listener interface.

- ***endpoint***
  - ***@type***: An FQN of the endpoint implementation class. There are two built-in endpoints namely **org.switchyard.component.jca.endpoint.JMSEndpoint** and **org.switchyard.component.jca.endpoint.CCIEndpoint**. These two endpoints have corresponding listeners. If neither JMSEndpoint nor CCIEndpoint is applicable for the EIS you are binding to, then you need to implement its own Endpoint class according to the ResourceAdapter implementation. The endpoint class should be a subclass of **org.switchyard.component.jca.endpoint.AbstractInflowEndpoint**.
  - ***property***: Properties for injecting into the endpoint class. The JMSEndpoint does not require a property. The CCIEndpoint requires connectionFactoryJNDIName property.
- ***transacted***: The boolean value to indicate whether the endpoint needs a transaction or not. Its value is true by default.
- ***batchCommit***: If you define this element, multiple incoming messages are processed in one transaction. The transaction is committed when the number of processed messages reach to batchSize, or batchTimeout milliseconds pass since the start of the transaction. Transaction reaper thread watches the inflight transaction, and once batch timeout occurs the transaction reaper thread commits it.
  - ***@batchSize***: The number of messages to be processed in one transaction.
  - ***@batchTimeout***:The batch timeout in milliseconds.

Here is an example that binds a service to HornetQ:

```
<sca:composite name="JCAInflowExample" targetNamespace="urn:userguide:jca-example-
service:0.1.0">
   <sca:service name="JCAService" promote="SomeService">
      <jca:binding.jca>
         <selector:operationSelector operationName="onMessage"/>
         <jca:inboundConnection>
            <jca:resourceAdapter name="hornetq-ra.rar"/>
            <jca:activationSpec>
               <jca:property name="destinationType" value="javax.jms.Queue"/>
               <jca:property name="destination" value="ServiceQueue"/>
            </jca:activationSpec>
         </jca:inboundConnection>
         <jca:inboundInteraction>
            <jca:listener>javax.jms.MessageListener</jca:listener>
            <jca:endpoint type="org.switchyard.component.jca.endpoint.JMSEndpoint"/>
            <jca:transacted>true</jca:transacted>
            <jca:batchCommit batchSize="10" batchTimeout="5000"/>
         </jca:inboundInteraction>
      </jca:binding.jca>
   </service>
   <!-- sca:component definition omitted -->
</sca:composite>
```

9.2.4.5. Binding References with JCA Outbound

Composite-level references can be bound to a EIS with JCA outbound using the binding.jca binding definition. The following configuration options are required for **binding.jca**:

- ***outboundConnection***
- - ***resourceAdapter***
    - ***@name***: Name of the ResourceAdapter archive. Ensure that the resource adapter is deployed on the JBoss application server before you deploy the SwitchYard application which has the JCA binding.
  - ***connection***
    - ***@jndiName***: JNDI name to which the ConnectionFactory is bound.
- ***outboundInteraction***
  - ***connectionSpec***: This is the configuration for **javax.resource.cci.ConnectionSpec**. Note that the JMSProcessor does not use this option.
  - - ***@type***:This is the FQN of the ConnectionSpec implementation class.
    - ***property***: These are the properties to be injected into ConnectionSpec instance.
  - ***interactionSpec***: This is the configuration for the **_javax.resource.cci.InteractionSpec**. Note that the JMSProcessor does not use this option.
    - ***@type***:This is the FQN of the InteractionSpec implementation class.
    - ***property***: These are the properties to be injected into InteractionSpec instance.
  - ***processor***
    - ***@type***: This is the FQN of the class which processes outbound delivery. There are two built-in processors, **org.switchyard.component.jca.processor.JMSProcessor** and **org.switchyard.component.jca.processor.CCIProcessor**. If neither JMSProcessor nor CCIProcessor is applicable for the EIS to which you have to bind, then you need to implement the EIS' own processor class according to the ResourceAdapter implementation. Note that this class should be a subclass of **org.switchyard.component.jca.processor.AbstractOutboundProcessor**.
    - ***property***: These are the properties to be injected into processor instance. JMSProcessor needs destination property to specify target destination. CCIProcessor needs recordClassName property to specify record type to be used to interact with EIS. If you use CCIProcessor with the record type other than MappedRecord and IndexedRecord, you need to implement the corresponding RecordHandler.

  Here is an example of a JCA reference binding to HornetQ:

```
<sca:composite name="JCAReferenceExample" targetNamespace="urn:userguide:jca-example-
reference:0.1.0">
   <sca:reference name="JCAReference" promote="SomeComponent/SomeReference"
multiplicity="1..1">
      <jca:binding.jca>
         <jca:outboundConnection>
            <jca:resourceAdapter name="hornetq-ra.rar"/>
```

```
                <jca:connection jndiName="java:/JmsXA"/>
            </jca:outboundConnection>
            <jca:outboundInteraction>
                <jca:processor type="org.switchyard.component.jca.processor.JMSProcessor">
                    <jca:property name="destination" value="ReferenceQueue"/>
                </jca:processor>
            </jca:outboundInteraction>
        </jca:binding.jca>
    </sca:reference>
</sca:composite>
```

9.2.5. JMS

9.2.5.1. SwitchYard JMS Binding

The Java Message Service binding in SwitchYard provides support for asynchronous communication with messaging providers for services and references. The JMS binding is built on top of camel-jms and supports most of options for this endpoint.

9.2.5.2. Generic JMS Options

You can apply these options to <binding.jms>

- queue or topic: this is the destination name from which resources are consumed or to which they shall be sent.
- connectionFactory: this is the name of the connection factory instance to use.
- username
- password
- clientId
- durableSubscriptionName
- concurrentConsumers
- maxConcurrentConsumers
- disableReplyTo
- preserveMessageQos
- deliveryPersistent
- priority
- explicitQosEnabled
- replyTo
- replyToType
- requestTimeout

- selector
- timeToLive
- transacted
- transactionManager

Here is what a JMS service binding looks like:

```
<sca:composite name="camel-binding" targetNamespace="urn:switchyard-quickstart:camel-
binding:0.1.0">
    <sca:service name="GreetingService" promote="GreetingService">
        <camel:binding.jms>
            <camel:queue>INCOMING_GREETS</camel:queue>
            <camel:connectionFactory>#connectionFactory</camel:connectionFactory>
            <camel:username>esb</camel:username>
            <camel:password>rox</camel:password>
            <camel:selector>RECEIVER='ESB' AND SENDER='ERP'<camel:selector>
        </camel:binding.jms>
    </sca:service>
</sca:composite>
```

Here is what a JMS reference binding looks like:

```
<sca:composite name="camel-binding" targetNamespace="urn:switchyard-quickstart:camel-
binding:0.1.0">
    <sca:reference name="GreetingService" promote="camel-binding/GreetingService"
multiplicity="1..1">
        <camel:binding.jms>
            <camel:topic>GREETINGS_NOTIFICATION</camel:topic>
            <camel:connectionFactory>#connectionFactory</camel:connectionFactory>
            <camel:priority>8</camel:priority>
        <camel:binding.jms>
    </sca:reference>
</sca:composite>
```

9.2.6. File

9.2.6.1. File Binding

The file binding in SwitchYard provides filesystem level support for services and references. The file binding is built on top of camel-file and supports the endpoint options listed in the following sections.

9.2.6.2. Generic File Options

You can apply these options to <binding.file>:

- directory: directory name for consuming and producing files
- autoCreate: automatically creates directory if a directory does not exist
- bufferSize: write buffer size

- fileName: file name filter for consumer or file name pattern for producer
- flatten: skip path and just use file name
- charset: charset used for reading or writing file

Here is what a File service binding looks like:

```
<sca:composite name="camel-binding" targetNamespace="urn:switchyard-quickstart:camel-
binding:0.1.0">
    <sca:service name="GreetingService" promote="GreetingService">
        <camel:binding.file>
            <camel:directory>target/input</camel:directory>
            <camel:fileName>test.txt</camel:fileName>
            <camel:consume>
               <camel:initialDelay>50</camel:initialDelay>
               <camel:delete>true</camel:delete>
            </camel:consume>
        </camel:binding.file>
    </sca:service>
</sca:composite>
```

Supported options for binding services with files are:

- delete
- recursive
- noop
- preMove
- move
- moveFailed
- include
- exclude
- idempotent
- idempotentRepository
- inProgressRepository
- filter
- inProgressRepository
- sorter
- sortBy
- readLock
- readLockTimeout

- readLockCheckInterval
- readLockTimeout
- exclusiveReadLockStrategy
- processStrategy
- startingDirectoryMustExist
- directoryMustExist
- doneFileName

Here is what a File reference binding looks like:

```
<sca:composite name="camel-binding" targetNamespace="urn:switchyard-quickstart:camel-
binding:0.1.0">
    <sca:reference name="GreetingService" promote="camel-binding/GreetingService"
multiplicity="1..1">
        <camel:binding.file>
            <camel:directory>target/output</camel:directory>
            <camel:autoCreate>false</camel:autoCreate>
            <camel:produce>
                <camel:fileExist>Override</camel:fileExist>
            </camel:produce>
        <camel:binding.file>
    </sca:reference>
</sca:composite>
```

Supported options for binding references with files are:

- fileExist
- tempPrefix
- tempFileName
- keepLastModified
- eagerDeleteTargetFile
- doneFileName

9.2.7. FTP FTPS SFTP

9.2.7.1. FTP Binding

SwitchYard provides support for remote file systems on both service and reference. The ftp binding is built on top of camel-ftp and supports the endpoint options listed in the following sections.

9.2.7.2. Generic FTP FTPS SFTP Options

You can apply these options to <binding.ftp>,<binding.ftps>, and <binding.sftp>:

- host

- port
- username
- password
- binary
- connectTimeout
- disconnect
- maximumReconnectAttempts
- reconnectDelay
- separator
- stepwise
- throwExceptionOnConnectFailed

Here is what a FTP service binding and FTP reference binding looks like.

```
cat switchyard.xml
<xml version="1.0" encoding="UTF-8"><switchyard xmlns="urn:switchyard-config:switchyard:1.0">
<composite xmlns="http://docs.oasis-open.org/ns/opencsa/sca/200912" name="camel-ftp-binding"
targetNamespace="urn:switchyard-quickstart:camel-ftp-binding:0.1.0">
<service name="GreetingService" promote="GreetingService/GreetingService">
<ftp:binding.ftp xmlns:ftp="urn:switchyard-component-camel-ftp:config:1.0">
<ftp:directory>/</ftp:directory>
<ftp:host>localhost</ftp:host>
<ftp:port>2222</ftp:port>
<ftp:username>camel</ftp:username>
<ftp:password>isMyFriend</ftp:password>
<ftp:consume>
<ftp:initialDelay>50</ftp:initialDelay>
<ftp:delay>50</ftp:delay>
</ftp:consume>
</ftp:binding.ftp>
</service>
<component name="GreetingService">
<implementation.bean xmlns="urn:switchyard-component-bean:config:1.0"
class="org.switchyard.quickstarts.camel.ftp.binding.GreetingServiceBean"/>
<service name="GreetingService">
<interface.java interface="org.switchyard.quickstarts.camel.ftp.binding.GreetingService"/>
</service>
</component>
</composite>
</switchyard>
```

The supported options for a FTP service are same as that for file. However, binding a reference with file can be used to store outcome of service on remote server. All File reference properties are supported in FTP reference binding.

9.2.7.3. Specific FTP FTPS SFTP Options

You can apply these options to <binding.ftp>:

- passiveMode
- timeout
- soTimeout
- siteCommand

You can apply these options to <binding.ftps>:

- securityProtocol
- isImplicit
- execPbsz
- execProt
- disableSecureDataChannelDefaults

You can apply these options to <binding.sftp>:

- knownHostsFile
- privateKeyFile
- privateKeyFilePassphrase

9.2.8. TCP UDP

9.2.8.1. TCP UDP Binding

SwitchYard provides support for network level integration with TCP and UPD protocols. The TCP and UDP binding is built on top of camel-netty and supports most of options for this endpoint.

IMPORTANT

The **camel-netty** component is deprecated since JBoss Fuse 6.3 and will be replaced by the **camel-netty4** component in a future release of JBoss Fuse.

9.2.8.2. Generic TCP UDP Options

You can apply these options to <binding.tcp> and <binding.udp>:

- host
- port
- receiveBufferSize
- sendBufferSize
- reuseAddress

- encoders
- decoders
- allowDefaultCodec
- workerCount
- sync
- disconnect

Here is what a TCP/UDP service binding looks like:

```
<sca:composite name="camel-binding" targetNamespace="urn:switchyard-quickstart:camel-binding:0.1.0">
    <sca:service name="GreetingService" promote="GreetingService">
        <camel:binding.tcp>
            <camel:host>localhost</camel:host>
            <camel:port>3939</camel:port>
            <camel:allowDefaultCodec>false</camel:allowDefaultCodec>
            <camel:sync>false</camel:sync>
        </camel:binding.tcp>
        <camel:binding.udp>
            <camel:host>localhost</camel:host>
            <camel:port>3940</camel:port>
            <camel:allowDefaultCodec>false</camel:allowDefaultCodec>
            <camel:sync>false</camel:sync>
        </camel:binding.udp>
    </sca:service>
</sca:composite>
```

9.2.8.3. Specific TCP UDP Options

You can apply these options to <binding.tcp>:

- textline
- tcpNoDelay
- keepAlive

You can apply these options to <binding.udp>:

- broadcast

9.2.8.4. SSL Configuration Options

This endpoint supports SSL. You can use the following parameters to configure it:

- ssl: turn on SSL
- sslHandler: custom SSL Handler to use
- passphrase: bean reference to String instance used to open KeyStore

- securityProvider: name of Java security provider
- keyStoreFormat
- keyStoreFile: reference to File instance which is loaded into java KeyStore
- trustStoreFile: reference to File instance
- sslContextParametersRef: if this parameter is specified, it must be an bean reference to an instance of **org.apache.camel.util.jsse.SSLContextParameters** where you may specify all necessary parameters at once

9.2.9. JPA

9.2.9.1. JPA Binding

The JPA binding in SwitchYard provides support for consuming and storing JPA entities. It supports both service binding for entity consumption and reference for entity storing. The JPA binding is built on top of camel-jpa and supports most of options for this endpoint.

9.2.9.2. Generic JPA Options

You can apply these options to <binding.jpa>

- entityClassName
- persistenceUnit
- transactionManager

Here is what a JPA service binding looks like:

```
<sca:composite name="camel-binding" targetNamespace="urn:switchyard-quickstart:camel-
binding:0.1.0">
    <sca:service name="GreetingService" promote="GreetingService">
        <camel:binding.jpa>

<camel:entityClassName>org.switchyard.quickstarts.camel.jpa.binding.domain.Greet</camel:entityClass
Name>
            <camel:persistenceUnit>JpaEvents</camel:persistenceUnit>
            <camel:transactionManager>#jtaTransactionManager</camel:transactionManager>
            <camel:consume>
                <camel:consumeLockEntity>false</camel:consumeLockEntity>
                <camel:consumer.transacted>true</camel:consumer.transacted>
            </camel:consume>
        </camel:binding.jpa>
    </sca:service>
</sca:composite>
```

Supported options for binding services with JPA are:

- consumeDelete
- consumeLockEntity

- maximumResults
- consumer.query
- consumer.namedQuery
- consumer.nativeQuery
- consumer.resultClass
- consumer.transacted

Here is what a JPA reference binding looks like:

```
<sca:composite name="camel-binding" targetNamespace="urn:switchyard-quickstart:camel-
binding:0.1.0">
    <sca:reference name="GreetingService" promote="camel-binding/GreetingService"
multiplicity="1..1">
        <camel:binding.jpa>

<camel:entityClassName>org.switchyard.quickstarts.camel.jpa.binding.domain.Greet</camel:entityClass
Name>
            <camel:persistenceUnit>JpaEvents</camel:persistenceUnit>
            <camel:produce>
                <camel:flushOnSend>false</camel:flushOnSend>
            </camel:produce>
        </camel:binding.jpa>
    </sca:reference>
</sca:composite>
```

Supported options for binding references with JPA are:

- flushOnSend
- usePersist

9.2.10. SQL

9.2.10.1. SQL Binding

The SQL binding in SwitchYard provides database read/write support for references in SwitchYard. This binding is built on top of camel-sql.

9.2.10.2. Generic SQL Options

You can apply these options to <binding.sql>:

query

SQL query to execute

dataSourceRef

Data Source name

batch

Turn on JDBC batching

placeholder
A placeholder sign used to replace parameters in query

You may use the following two additional attributes for binding.sql element when used with service reference:

- period
- initialDelay

Here is what a SQL service binding looks like:

```
<sca:composite name="camel-binding" targetNamespace="urn:switchyard-quickstart:camel-
binding:0.1.0">
    <sca:service name="GreetingService" promote="GreetingService">
        <camel:binding.sql period="10s">
            <camel:query>SELECT * FROM greetings</camel:query>
            <camel:dataSourceRef>java:jboss/datasources/GreetDS</camel:dataSourceRef>
        </camel:binding.sql>
    </sca:service>
</sca:composite>
```

Here is what a SQL reference binding looks like:

```
<sca:composite name="camel-binding" targetNamespace="urn:switchyard-quickstart:camel-
binding:0.1.0">
    <sca:reference name="GreetingDatabaseStore" promote="camel-binding/GreetingDatabaseStore"
multiplicity="1..1">
        <camel:binding.sql>
            <camel:query>INSERT INTO greetings (name) VALUES (#)</camel:query>
            <camel:dataSourceRef>java:jboss/datasources/GreetDS</camel:dataSourceRef>
        <camel:binding.sql>
    </sca:reference>
</sca:composite>
```

9.2.11. Mail

9.2.11.1. Mail Binding

The Mail binding in SwitchYard provides support for consuming and sending mail messages. It supports both service binding for mail consumption and reference for message sending. The Mail binding is built on top of camel-mail and supports most of options for this endpoint.

9.2.11.2. Generic Mail Options

You can apply these options to <binding.mail>:

- host
- port

- username
- password
- connectionTimeout

  You may also use the secure attribute to identify usage of secured connection (pop3s/imaps/smtps).

Here is what a Mail service binding looks like:

```
<sca:composite name="camel-binding" targetNamespace="urn:switchyard-quickstart:camel-
binding:0.1.0">
    <sca:service name="GreetingService" promote="GreetingService">
        <camel:binding.mail>
            <camel:host>localhost</camel:host>
            <camel:username>camel</camel:username>
            <camel:consume accountType="pop3">
                <camel:copyTo>after-processing</camel:copyTo>
            </camel:consume>
        </camel:binding.mail>
    </sca:service>
</sca:composite>
```

Supported options for binding services with Mail are:

- folderName
- fetchSize
- unseen
- delete
- copyTo
- disconnect

  You may specify additional attribute accountType to choose mail protocol. The possible values for this attribute are pop3 or imap. Default is imap.

Here is what a Mail reference binding looks like:

```
<sca:composite name="camel-binding" targetNamespace="urn:switchyard-quickstart:camel-
binding:0.1.0">
    <sca:reference name="GreetingService" promote="camel-binding/GreetingService"
multiplicity="1..1">
        <camel:binding.mail>
            <camel:host>localhost</camel:host>
            <camel:username>camel</camel:username>
            <camel:produce>
                <camel:subject>Forwarded message</camel:subject>
                <camel:from>camel@localhost</camel:from>
                <camel:to>rider@camel</camel:to>
            </camel:produce>
        </camel:binding.mail>
    </sca:reference>
</sca:composite>
```

Supported options for binding references with Mail are:

- subject
- from
- to
- CC
- BCC
- replyTo

9.2.12. Quartz

9.2.12.1. Quartz Binding

The Quartz binding in SwitchYard provides support for triggering services with a given cron expression. The Quartz binding is built on top of camel-quartz.

9.2.12.2. Generic Quartz Options

You can apply these options to <binding.quartz>:

- name: name of the job
- cron: execution expression

Here is what a Quartz service binding looks like:

```
<sca:composite name="camel-binding" targetNamespace="urn:switchyard-quickstart:camel-binding:0.1.0">
    <sca:service name="GreetingService" promote="GreetingService">
        <camel:binding.quartz>
            <camel:name>GreetingJob</camel:name>
            <camel:cron>0 0/5 * * * ?</camel:cron>
        </camel:binding.quartz>
    </sca:service>
</sca:composite>
```

9.2.13. Timer

9.2.13.1. Timer Binding

The Timer binding in SwitchYard provides support for triggering services with fixed timer. It is a lightweight alternative for Quartz. The file binding is built on top of camel-timer.

9.2.13.2. Generic Timer Options

You can apply these options to <binding.timer>:

- name: name of the timer
- time
- pattern
- period
- delay
- fixedRate
- daemon

Here is what a Timer service binding looks like:

```
<sca:composite name="camel-binding" targetNamespace="urn:switchyard-quickstart:camel-
binding:0.1.0">
    <sca:service name="GreetingService" promote="GreetingService">
        <camel:binding.timer>
            <camel:name>GreetingTimer</camel:name>
            <camel:time>2012-01-01T12:00:00</camel:time>
            <camel:pattern>yyyy-MM-dd'T'HH:mm:ss</camel:pattern>
            <camel:delay>1000</camel:delay>
            <camel:fixedRate>true</camel:fixedRate>
        </camel:binding.timer>
    </sca:service>
</sca:composite>
```

9.2.14. SEDA

9.2.14.1. SEDA Binding

The SEDA binding in SwitchYard provides asynchronous service binding between camel route and SwitchYard service. This binding is built on top of camel-seda.

NOTE

The SEDA queue is not persistent.

9.2.14.2. Generic SEDA Options

You can apply these options to <binding.seda>:

- name: name of the queue
- size: the maximum capacity of the SEDA queue (the number of messages it can hold)
- concurrentConsumers
- waitForTaskToComplete

- timeout
- multipleConsumers
- limitConcurrentConsumers

Here is what a SEDA service binding looks like:

```
<sca:composite name="camel-binding" targetNamespace="urn:switchyard-quickstart:camel-
binding:0.1.0">
    <sca:service name="GreetingService" promote="GreetingService">
        <camel:binding.seda>
            <camel:name>GreetingQueue</camel:name>
            <camel:size>6</camel:size>
            <camel:concurrentConsumers>2</camel:concurrentConsumers>
        </camel:binding.seda>
    </sca:service>
</sca:composite>
```

9.2.15. Camel URI

9.2.15.1. Camel Binding

Camel binding support in SwitchYard allows Camel components to be used as gateway bindings for services and references within an application.

9.2.15.2. Generic Camel Options

Every Camel component binding supported by SwitchYard has its own configuration name-space with one exception. Bindings for Direct, SEDA, Timer and Mock share the same name-space: urn:switchyard-component-camel-core:config:1.0.

Composite-level services can be bound to a Camel component using the <binding.uri> binding definition. The following configuration options are available for binding.uri:

configURI

This contains the Camel endpoint URI used to configure a Camel component instance.

operationSelector

This is the specification of the operation to use for the message exchange. (This setting is not used for CXFRSconfigurations.)

NOTE

binding.uri is not linked with any specific component. It allows usage of 3rd party camel components which are not part of distribution.

Here is a sample service binding that uses a Camel component:

```
<sca:composite name="SimpleCamelService" targetNamespace="urn:userguide:simple-camel-
service:0.1.0">
```

```
    <sca:service name="SimpleCamelService" promote="SimpleComponent/SimpleCamelService">
        <camel:binding.uri configURI="file://target/input?
fileName=test.txt&amp;initialDelay=50&amp;delete=true">
            <selector:operationSelector operationName="print"/>
        </camel:binding.uri>
    </sca:service>
    <!-- sca:component definition omitted -->
</sca:composite>
```

Binding a reference with Camel is very similar to binding a service. The only significant difference is that specification of the operationSelector is not required on reference bindings. Logically reference elements points to outgoing communication (which may, for example, be a service called by SwitchYard):

```

<sca:composite name="orders" targetNamespace="urn:switchyard-quickstart-demo:orders:0.1.0">
    <sca:reference name="WarehouseService" promote="OrderComponent/WarehouseService"
multiplicity="1..1">
        <camel:binding.uri configURI="file://target/output"/>
    </sca:reference>
</sca:composite>
</sca:composite>
```

9.2.16. SCA

The SCA binding provides a means by which SwitchYard services and SwitchYard-aware clients can communicate with one another. There are three basic use cases for the SCA binding:

- Facilitate inter-application communication within a SwitchYard runtime. The SCA binding can be used to link a composite reference in one application to a composite service in another application.
- Provide a remote invocation endpoint for external clients using RemoteInvoker. This allows a stand-alone client to communicate with a SY application.
- Allow clustering of SwitchYard services in two or more SwitchYard instances.

9.2.16.1. SCA Service Bindings

An SCA binding can be added to composite-level services to make that service available to other applications and remote clients through a SwitchYard internal communication protocol.

There is only one configuration option available for SCA bindings:

- clustered : when enabled, the service is published in the distributed SY runtime registry so that other cluster instances can discover and consume the service.

Regardless of the clustering setting, all services with an SCA binding are invokable through the SwitchYard remote invoker endpoint. The default URL for this endpoint is http://localhost:8080/switchyard-remote. The hostname and port for this endpoint are based on the default HTTP listener defined in AS 7.

9.2.16.2. SCA Reference Bindings

An SCA binding can be added to a composite-level reference to invoke services provided in other SwitchYard applications deployed locally or in a cluster.

The following configuration parameters can be used with an SCA reference binding:

- clustered : if enabled, the reference binding discovers remote SY service endpoints in a cluster.
- load balancing : the name of a load balancing strategy to be used with clustering. Two out of the box options available are "RoundRobinStrategy" and "RandomStrategy". You can also specify a custom load balance strategy by implementing LoadBalanceStrategy.
- target service : allows you to override the name of the service being invoked in the case where the target application uses a service name different from the reference name (default is that reference and service name match).
- target namespace : allows you to override the namespace of the service being invoked. By default, all applications in SwitchYard use a different namespace, so keep this setting in mind when invocations occur across application boundaries.

9.2.16.3. Remote Transaction Propagation

If you invoke remote SwitchYard service through SCA binding under the active JTA transaction, SwitchYard runtime propagates its transaction context. Let's say we have ServiceA and ServiceB, where ServiceA is deployed on NodeA and ServiceB is on NodeB. ServiceA invokes ServiceB through SCA reference binding besides. If ServiceA is processing under the active JTA transaction, SwitchYard runtime embeds its transaction context into the remote invocation message when ServiceA invokes ServiceB. Then ServiceB extracts that transaction context and create subordinate JTA transaction, which means ServiceB is processed under the subordinate transaction of ServiceA's transaction, so those transactions could synchronize. Please note that transaction policy must allow this behavior on both sides of ServiceA and ServiceB.

In order to achieve this remote transaction propagation, XTS must be enabled in AS 7 configuration. Following changes must be applied:

```
--- standalone-ha.xml    2013-10-09 22:09:32.085300978 +0900
+++ standalone-ha-xts.xml    2013-10-16 11:40:57.198147545 +0900
@@ -25,6 +25,7 @@
         <extension module="org.jboss.as.webservices"/>
         <extension module="org.jboss.as.weld"/>
         <extension module="org.switchyard"/>
+        <extension module="org.jboss.as.xts"/>
 </extensions>
 <management>
    <security-realms>
@@ -405,6 +406,9 @@
    <extension identifier="org.apache.camel.soap"/>
    </extensions>
 </subsystem>
+<subsystem xmlns="urn:jboss:domain:xts:1.0">
+   <xts-environment url="http://${jboss.bind.address:127.0.0.1}:8080/ws-c11/ActivationService"/>
+</subsystem>
 </profile>
     <interfaces>
         <interface name="management">
```

9.2.17. MQTT

The MQTT binding in SwitchYard provides support for asynchronous communication with MQTT messaging providers. It supports both sides - service and reference. The MQTT binding is built on top of camel-mqtt (http://camel.apache.org/mqtt.html) and supports most options for this endpoint.

9.2.17.1. Generic options

You can apply the following options to the **<binding.mqtt>** definition:

- **host** : The host that you want to connect to.
- **localAddress** : The local address.
- **connectAttemptsMax** : The maximum number of connect attempts.
- **reconnectAttemptsMax** : The maximum number of reconnect attempts.
- **reconnectDelay** : The time in milliseconds between reconnect attempts.
- **reconnectBackOffMultiplier** : The multiplier to use to the delay between connection attempts.
- **reconnectDelayMax** : The maximum time in milliseconds between reconnect attempts.
- **userName** : user name
- **password** : password
- **qualityOfService** : The MQTT Quality of Service. Possible values are: **AtMostOnce**, **AtLeastOnce**, or **ExactlyOnce**.
- **byDefaultRetain** : The default retain policy.
- **mqttTopicPropertyName** : The property name for the MQTT topic.
- **mqttRetainPropertyName** : The property name for the MQTT Retain policy.
- **mqttQosPropertyName** : The property name for the MQTT Quality of Service.
- **connectWaitInSeconds** : Delay in seconds to wait before establishing the connection.
- **disconnectWaitInSeconds** : Delay in seconds to wait before disconnecting the connection.
- **sendWaitInSeconds** : Delay in seconds to wait before sending the message.

9.2.17.2. Binding Services with MQTT

You can specify the MQTT Topic name with the **subscribeTopicName** option.

Example 9.8. Example MQTT service binding

```
<sca:composite name="camel-mqtt-binding" targetNamespace="urn:switchyard-quickstart:camel-
mqtt-binding:0.1.0">
    <sca:service name="GreetingService" promote="GreetingService/GreetingService">
        <mqtt:binding.mqtt name="Greet">
            <mqtt:userName>karaf</mqtt:userName>
            <mqtt:password>karaf</mqtt:password>
            <mqtt:subscribeTopicName>camel/mqtt/test/input</mqtt:subscribeTopicName>
        </mqtt:binding.mqtt>
    </sca:service>
</sca:composite>
```

9.2.17.3. Binding References with MQTT

You can specify the MQTT Topic name to publish with the **publishTopicName** option.

Example 9.9. Example MQTT reference binding

```
<sca:composite name="camel-mqtt-binding" targetNamespace="urn:switchyard-quickstart:camel-
mqtt-binding:0.1.0">
    <sca:reference name="StoreReference" multiplicity="0..1"
promote="GreetingService/StoreReference">
        <mqtt:binding.mqtt name="Store">
            <mqtt:userName>karaf</mqtt:userName>
            <mqtt:password>karaf</mqtt:password>
            <mqtt:publishTopicName>camel/mqtt/test/output</mqtt:publishTopicName>
        </mqtt:binding.mqtt>
    </sca:reference>
</sca:composite>
```

9.3. MESSAGE COMPOSITION

9.3.1. Message Composers

A MessageComposer composes or decomposes a native binding message to or from SwitchYard's canonical message. It does so in three steps:

1. Construct a new target message instance.
2. Copy the content of the message.
3. Delegate the header/property mapping to a ContextMapper.

A SOAPMessageComposer and CamelMessageComposer are included with SwitchYard. These implementations are used by their associated bindings but you can override these with your own implementations.

9.3.2. Create a Custom Message Composer

Procedure 9.2. Create a Custom Message Composer

1. Implement the **org.switchyard.component.common.composer.MessageComposer** interface:

```
public interface MessageComposer<T> {
    ContextMapper<T> getContextMapper();
    MessageComposer<T> setContextMapper(ContextMapper<T> contextMapper);
    Message compose(T source, Exchange exchange, boolean create) throws Exception;
    public T decompose(Exchange exchange, T target) throws Exception;
}
```

2. Specify your implementation in your **switchyard.xml** file:

```
<binding.xyz ...>
    <messageComposer class="com.example.MyMessageComposer"/>
</binding.xyz>
```

9.3.3. Custom Message Composer Properties

- The **getContextMapper()** and **setContextMapper()** methods are bean properties. If you extend BaseMessageComposer, both properties are implemented for you.
- Your **compose()** method needs to take the data from the passed-in source native message and compose a SwitchYard Message based on the specified Exchange. The create parameter determines whether or not we ask the Exchange to create a new Message, or use the existing one.
- Your **decompose()** method needs to take the data from the SwitchYard Message in the specified Exchange and "decompose it" into the target native message.

9.4. OPERATION SELECTORS

9.4.1. Operation Selector

Use the **OperationSelector** to determine which service operation should be invoked for the message exchange.

9.4.2. Types of Operation Selectors

SwitchYard provides the following options for Operation Selectors.

IMPORTANT

Operation selectors are used in combination with a service binding to help SwitchYard determine the target operation for a service invocation. When a service only has a single operation, an operation selector should not be used. If an operation selector is used for a service with a single operation, failure to assign an operation in the operation selector will not result in an error. If an operation selector fails to assign an operation for a service with multiple operations an error is reported.

Static Operation Selector

You can specify an operation name in the configuration. Here is what a Static Operation Selector configuration looks like:

```
<hornetq:binding.hornetq>
    <swyd:operationSelector operationName="greet" xmlns:swyd="urn:switchyard-
config:switchyard:1.0"/>
```

```
(... snip ...)
</hornetq:binding.hornetq>
```

XPath Operation Selector

You can specify an XPath location which contains an operation name to be invoked in the message contents. Here is what an XPath Operation Selector configuration looks like:

```
<jca:binding.jca>
    <swyd:operationSelector.xpath expression="//person/language" xmlns:swyd="urn:switchyard-config:switchyard:1.0"/>
(... snip ...)
</jca:binding.jca>
```

For this configuration, if you specify the following message content, then the operation **spanish()** is invoked:

```
<person>
    <name>Fernando</name>
    <language>spanish</language>
</person>
```

Regex Operation Selector

You can specify a regular expression to find an operation name to be invoked in the message contents. Here is what a Regex Operation Selector configuration looks like:

```
<http:binding.http>
    <swyd:operationSelector.regex expression="[a-zA-Z]*Operation" xmlns:swyd="urn:switchyard-config:switchyard:1.0"/>
(... snip ...)
</http:binding.http>
```

For this configuration, if you specify the following message content, then the operation **regexOperation()** is invoked:

```
xxx yyy zzz regexOperation aaa bbb ccc
```

Java Operation Selector

You can specify a Java class which is able to determine the operation to be invoked. Here is what a Java Operation Selector configuration looks like:

```
<jca:binding.jca>
    <swyd:operationSelector.java class="org.switchyard.example.MyOperationSelectorImpl"
xmlns:swyd="urn:switchyard-config:switchyard:1.0"/>
(... snip ...)
</jca:binding.jca>
```

Here, the **org.switchyard.example.MyOperationSelectorImpl** has to implement **org.switchyard.selector.OperationSelector** or be a subclass of concrete OperationSelector classes for each service bindings. You can override the **selectOperation()** method as you like.

Following are the default OperationSelector implementation for each service bindings:

- Camel : **org.switchyard.component.camel.selector.CamelOperationSelector**
- JCA/JMS : **org.switchyard.component.jca.selector.JMSOperationSelector**
- JCA/CCI : **org.switchyard.component.jca.selector.CCIOperationSelector**
- HTTP : **org.switchyard.component.http.selector.HttpOperationSelector**

9.5. THROTTLING

SwitchYard provides support for message throttling. Throttling is configured on a composite service and applies to all requests received through all gateways configured on the service. If you have multiple bindings on a composite service, they share the same throttling configuration. If you want a given service to have two different throttling settings (one per binding), then promote the service twice and provide a unique setting for each composite service.

SwitchYard provides the following throttling options:

- timePeriod : (optional; defaults to 1000) The time period, in milliseconds, over which requests are counted. The message count is reset at the beginning of each period.

  The setting for the time period is static - it can not change at runtime.

- maxRequests : The maximum number of requests that can be processed within a specified timePeriod. Processing is delayed until the start of the next period for any messages received after the limit is reached.

  The setting for the number of messages is dynamic - it can change at runtime.

NOTE

Throttling of individual gateways is not supported.

9.5.1. Example

Here's an example configuration, restricting the OrderService to handle at most one request every ten seconds. Note that the **sy:throttling** element is located within an **sca:extensions** element.

```
<sca:composite name="orders" targetNamespace="urn:switchyard-quickstart-demo:orders:0.1.0">
  <sca:service name="OrderService" promote="OrderService">
    <soap:binding.soap>
      <soap:wsdl>wsdl/OrderService.wsdl</soap:wsdl>
      <soap:socketAddr>:9000</soap:socketAddr>
    </soap:binding.soap>
    <sca:extensions>
      <sy:throttling maxRequests="1" timePeriod="10000"/>
    </sca:extensions>
  </sca:service>
</sca:composite>
```

You can specify the same configuration using the SwitchYard editor as well.

CHAPTER 10. TRANSFORMER

10.1. WHAT IS A TRANSFORMER

It is not possible for the message providers to know what format of message is expected by every consumer. Therefore Transformers are used to enable the loose-coupling of message providers and message consumers. Transformers convert message content to various formats, thereby playing an important role in connecting service consumers and providers.

10.2. TRANSFORMATION IN SWITCHYARD

Transformation represents a change to the format or representation of a message's content. The representation of a message is the Java contract used to access the underlying content. For example, **java.lang.String** and **org.example.MyFancyObject**. The format of a message refers to the actual structure of the data itself. Examples of data formats include XML, JSON, CSV, and EDI. Here is an example of message content in XML format:

```
<MyBook>
  <Chapter1>
  <Chapter2>
</MyBook>
```

You can also represent XML in Java as a String. For example:

```
String content = "<MyBook>...";
```

Transformation plays an important role in connecting service consumers and providers, as the format and representation of message content can be quite different between the two. For example, a SOAP gateway binding is likely use a different representation and format for messages than a service offered by a Java Bean. In order to route services from the SOAP gateway to the Bean providing the service, the format and representation of the SOAP message needs to change. Implementing the transformation logic directly in the consumer or provider pollutes the service logic and can lead to tight coupling. SwitchYard allows you to declare the transformation logic outside the service logic and inject into the mediation layer at runtime.

10.3. ADDING TRANSFORMATION TO A SWITCHYARD APPLICATION

You can specify transformation of message content in the descriptor of your SwitchYard application (**switchyard.xml**). The qualified name of the type being transformed from as well as the type being transformed to are defined along with the transformer implementation. This allows transformation to be a declarative aspect of a SwitchYard application, as the runtime automatically registers and executes transformers in the course of a message exchange. Here is an example of message content transformation:

```
<transforms>
   <transform.java bean="MyTransformerBean"
                  from="{urn:switchyard-quickstart-demo:orders:1.0}submitOrder"
                  to="java:org.switchyard.quickstarts.demos.orders.Order"/>
</transforms>
```

10.4. CHAINING TRANSFORMERS

JBoss Fuse Service Works 6.0 allows you to chain transformers. Say you have a transformer that transforms from A to B and another that transforms from B to C. If you then have an invocation where the consumer is using type A and the provider is using type C, then the runtime will invoke the A->B and the B->C transformers to achieve A->C.

For example, for a SOAP message, you could define two transformers:

```
<transform:transform.java from="{http://web.service.com/}GetStatisticsResponse" to="
{urn:web.service.com}StatisticsCData" bean="CDataReturn"/>
<transform:transform.jaxb from="{urn:web.service.com}StatisticsCData"
to="java:com.sample.model.Statistics" contextPath="com.sample.model"/>
```

The first transformer transforms the SOAP reply which contains the **GetStatisticsResponse** element into an intermediate object which is just the content of the return element (CDATA containing the element), implemented in Java as follows:

```
@Named("CDataReturn")
public class CDataExtractor {

    @Transformer(from = "{http://web.service.com/}GetStatisticsResponse",
                 to = "{urn:web.service.com}StatisticsCData")
    public String transform(Element from) {
        String value = null;
        NodeList nodes = from.getElementsByTagName("return");
        if(nodes.getLength() > 0 ){
            value = nodes.item(0).getChildNodes().item(0).getNodeValue();
        }
        return value;
    }
}
```

The second transformer then takes care of transforming from the element to the Statistics Java model using JAXB.

JBoss Fuse Service Works 6.0 knows, from the contracts on the reference definitions, that it needs to go from **{http://web.service.com/}GetStatisticsResponse** to **java:com.sample.model.Statistics** and will automatically invoke both transformers to make that happen.

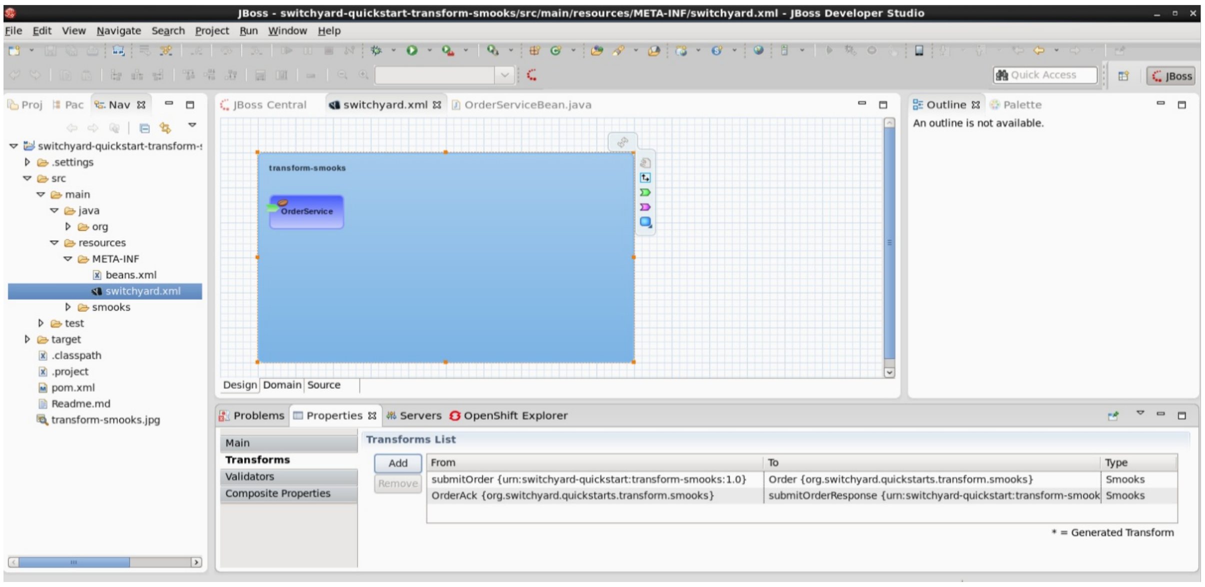
See Unable to transform from SOAP response with CDATA block to a POJO in FSW 6 for more information.

10.5. ADDING A TRANSFORMER USING SWITCHYARD EDITOR

This section describes how to add a transformer to an application in the SwitchYard editor. The example in this section shows how to add a Smooks transformer.

1. Select the main object (**transform-smooks**) in the SwitchYard editor.

Figure 10.1. Smooks Transformations in SwitchYard Editor



Transforms List

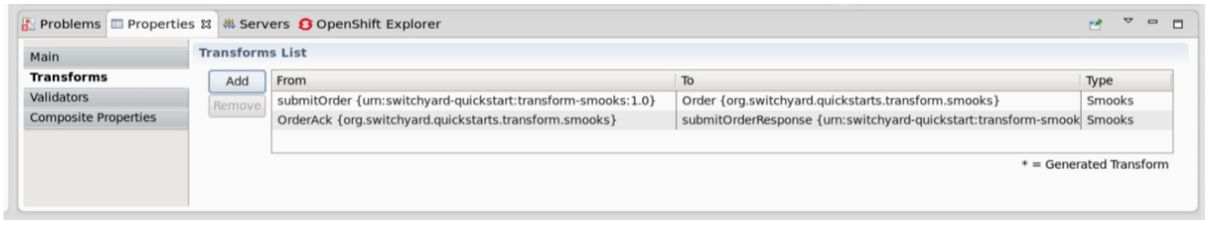
Main / Transforms / Validators / Composite Properties	From	To	Type
	submitOrder {urn:switchyard-quickstart:transform-smooks:1.0}	Order {org.switchyard.quickstarts.transform.smooks}	Smooks
	OrderAck {org.switchyard.quickstarts.transform.smooks}	submitOrderResponse {urn:switchyard-quickstart:transform-smook	Smooks

* = Generated Transform

2. Select the **Transforms** tab in the **Properties** view. The existing transformers are listed.

3. Select **Add** to add new transformers.

Figure 10.2. List of Transformers Under Properties View



Transforms List

From	To	Type
submitOrder {urn:switchyard-quickstart:transform-smooks:1.0}	Order {org.switchyard.quickstarts.transform.smooks}	Smooks
OrderAck {org.switchyard.quickstarts.transform.smooks}	submitOrderResponse {urn:switchyard-quickstart:transform-smook	Smooks

* = Generated Transform

NOTE

- Users are not able to add transformers unless they are required.
- If no transformers are visible in the properties page, try updating the Maven project configuration by right-clicking the project and selecting **Maven → Update Project**.

10.6. MESSAGE CONTENT TYPE NAMES

Since transformations occur between named types (for example, from type A to type B), it is important to understand how the type names are derived. The type of the message is determined based on the service contract, which can be WSDL or Java.

- WSDL

  For WSDL interfaces, the message name is determined based on the fully-qualified element name of a WSDL message. Here is an example of a WSDL definition:

```
<definitions xmlns:tns="urn:switchyard-quickstart:bean-service:1.0">
  <message name="submitOrder">
    <part name="parameters" element="tns:submitOrder"/>
  </message>
  <portType name="OrderService">
```

```
    <operation name="submitOrder">
      <input message="tns:submitOrder"/>
    </operation>
  </portType>
</definitions>
```

This yields the following message type name based on the message element name defined in the WSDL:

```
{urn:switchyard-quickstart:bean-service:1.0}submitOrder
```

- Java

  For Java interfaces, the message name consists of the full package name and the class name prefixed with "***java:***". Here is an example of a Java definition:

  ```
  package org.switchyard.example;
  public interface OrderService {
      void submitOrder(Order order);
  }
  ```

  This yields the following message type name:

  ```
  java:org.switchyard.example.Order
  ```

  You may override the default operation name generated for a Java interface. The @OperationTypes annotation provides this capability by allowing you to specify the input, output, and fault type names used for a Java service interface. For example, if you want to accept XML input content without any need for transformation to a Java object model, you can change the **OrderService** interface as shown below:

  ```
  package org.switchyard.example;
  public interface OrderService {
      @OperationTypes(in = "{urn:switchyard-quickstart:bean-service:1.0}submitOrder")
      void submitOrder(String orderXML);
  }
  ```

  This annotation can be useful if you want to maintain a tight control over the names used for message content.

10.7. SUPPORTED TRANSFORMATIONS

10.7.1. Java

10.7.1.1. Java Transformer

You can create a Java-based transformer in SwitchYard using any of the following methods:

- Implement the **org.switchyard.transform.Transformer** interface and add a <transform.java> definition to your **switchyard.xml**.

- Annotate one or more methods on your Java class with @Transformer annotation.

When using the @Transformer annotation, the SwitchYard maven plug-in automatically generates the <transform.java> definition and adds them to the **switchyard.xml** packaged in your application. Here is an example of a Java class that produces the <transform.java> definition:

```
@Named("MyTransformerBean")
public class MyTransformer {
   @Transformer(from = "{urn:switchyard-quickstart-demo:orders:1.0}submitOrder")
   public Order transform(Element from) {
      // handle transformation here
   }
}
```

Here, the ***from*** and ***to*** elements of the @Transformer annotation are optional. You can use them to specify the qualified type name used during transformer registration. If not specified, the full class name of the method parameter is used as the ***from*** type and the full class name of the return type is used as the ***to*** type.

The CDI bean name specified by @Named annotation is used to resolve transformer class. If not specified, the class name of the transformer is used instead. Here is an example:

```
<transforms>
   <transform.java class="org.switchyard.quickstarts.demos.orders.MyTransformer"
                   from="{urn:switchyard-quickstart-demo:orders:1.0}submitOrder"
                   to="java:org.switchyard.quickstarts.demos.orders.Order"/>
</transforms>
```

NOTE

The bean attribute and class attribute are mutually exclusive.

10.7.2. JAXB

10.7.2.1. JAXB Transformer

SwitchYard automatically detects and adds the JAXB Transformations for your application deployment. For example, if you develop a CDI Bean Service and use JAXB generated types in the Service Interface, you do not need to configure any of the transformations. SwitchYard automatically detects their availability for your application and automatically applies them at the appropriate time during service invocation.

The JAXB transformer allows you to perform Java to XML and XML to Java transformations using JAXB (XML marshalling and unmarshalling). The JAXB Transformer is similar to the JSON Transformer configuration. It also requires a ***to*** and ***from*** configuration with one Java type and one QNamed XML type.

For example in the source file **ObjectFactory.java**, the factory methods represent the available marshallings/unmarshallings as shown below:

```
@XmlElementDecl(namespace = "http://com.acme/orders", name = "create")
public JAXBElement<CreateOrder> createOrder(CreateOrder value) {
   return new JAXBElement<Order>(_CreateOrder_QNAME, CreateOrder.class, null, value);
}
```

Here, the @XmlElementDecl annotation implies that the XML QName associated with the **com.acme.orders.CreateOrder** type is "{http://com.acme/orders}create". This means that you can have the following SwitchYard JAXB Transformer configurations:

```
<transform.jaxb from="{http://com.acme/orders}create" to="java:com.acme.orders.CreateOrder" />
<transform.jaxb from="java:com.acme.orders.CreateOrder" to="{http://com.acme/orders}create" />
```

10.7.3. JSON

10.7.3.1. JSON Transformer

The JSON transformer provides a basic mapping facility between POJOs and JSON (JSON marshalling and unmarshalling). Just like the JAXB Transformer, specification of the transformer requires a ***to*** and ***from*** specification with one Java type and one QNamed JSON type, depending on whether you are performing a Java to JSON or JSON to Java transformation.

Here is an example configuration of JSON to Java transformation:

```
<trfm:transform.json
        from="{urn:switchyard-quickstart:transform-json:1.0}order"
        to="java:org.switchyard.quickstarts.transform.json.Order"/>
```

Here is a sample configuration of Java to JSON transformation:

```
<trfm:transform.json
        from="java:org.switchyard.quickstarts.transform.json.Order"
        to="{urn:switchyard-quickstart:transform-json:1.0}order"/>
```

10.7.4. Smooks

10.7.4.1. Smooks Transformer

SwitchYard provides three distinct transformation models for Smooks:

- XML to Java : Based on a standard Smooks Java Binding configuration.
- Java to XML: Based on a standard Smooks Java Binding configuration.
- Smooks : This is a normal Smooks transformation in which you must define which Smooks filtering result is to be exported back to the SwitchYard Message as the transformation result.

You can declare a Smooks transformation by including a <transform.smooks> definition in your **switchyard.xml**. Here is an example:

```
<transform.smooks config="/smooks/OrderAck_XML.xml"
            from="java:org.switchyard.quickstarts.transform.smooks.OrderAck"
            to="{urn:switchyard-quickstart:transform-smooks:1.0}submitOrderResponse"
            type="JAVA2XML"/>
```

The Smooks and XSLT translators require an external configuration file that tells the translator how to go about its actions. In the example above, the ***config*** attribute points to a Smooks resource containing the mapping definition. The ***type*** attribute can be one of SMOOKS, XML2JAVA, or JAVA2XML.

For more information on Smooks and XSLT configuration files, see Red Hat JBoss Fuse *Development Guide Volume 2: Smooks*.

For more information on the elements that extend the abstract transform, see the **/docs/schema/soa/org/switchyard/transform/config/model/v1/transform_<latest_version>.xsd**.

10.7.5. XSLT

10.7.5.1. XSLT Transformer

You can perform transformations with the XSLT transformer using an XSLT. You can configure it by specifying the ***to*** and ***from*** QNames, and the path to the XSLT. Here is an example of XSLT Transformer configuration:

```
<transform.xslt from="{http://acme/}A" to="{http://acme/}B" xsltFile="com/acme/xslt/A2B.xslt"/>
```

CHAPTER 11. VALIDATORS

11.1. WHAT IS A VALIDATOR

A Validator provides message content checking functionality. SwitchYard allows you to declare the validation logic outside the service logic and inject it into the mediation layer at runtime.

11.2. MESSAGE VALIDATION

Here is a sample that depicts message validation.

Message Content:

```
<MyBook xmlns="example">
  <Chapter1/>
  <Chapter2/>
</MyBook>
```

This is the associated XML Schema definition:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        targetNamespace="example"
        xmlns:orders="example">
        <element name="MyBook" type="example:MyBook"/>
        <complexType name="MyBook">
            <sequence>
                <element name="Chapter1" type="string"/>
            </sequence>
        </complexType>
</schema>
```

The XML content is still well-formed, but it has a **Chapter2** element that is not declared as the child of **MyBook** element in the XML schema. Hence the content cannot be validated against the schema.

11.3. ADD VALIDATION TO YOUR APPLICATION

Procedure 11.1. Add Validation to Your Application

- Specify message validation in the descriptor of your SwitchYard application by editing the **switchyard.xml** file using the JBoss Developer Studio SwitchYard Editor.

  The qualified name of the type being validated is defined along with the validator implementation. This allows validation to be a declarative aspect of a SwitchYard application, as the runtime will automatically register and execute validators in the course of a message exchange.

  ```
  <validates>
     <validate.xml schemaType="XML_SCHEMA"
                   name="{urn:example}MyBook">
        <schemaFiles><entry file="/xsd/orders.xsd"/></schemaFiles>
  ```

```
  </validate.xml>
</validates>
```

11.4. SUPPORTED VALIDATORS

11.4.1. Java

11.4.1.1. Use a Java Validator

Procedure 11.2. Create a Java Validator

1. Implement the **org.switchyard.validate.Validator** interface.
2. Add a <validate.java> definition to your **switchyard.xml** file.
3. Alternatively, annotate one or more methods to your Java class with @Validator:

   ```
   @Named("MyValidatorBean")
   public class MyValidator {
      @Validator(name = "{urn:switchyard-quickstart-demo:orders:1.0}submitOrder")
      public ValidationResult validate(Element from) {
         // handle validation here
      }
   }
   ```

   When using the @Validator annotation, the SwitchYard Maven plug-in automatically generates the <validate.java> definitions for you and add them to the **switchyard.xml** file packaged in your application.

   The Java class above produces this <validate.java> definition:

   ```
   <validate.java bean="MyValidatorBean"
                  name="{urn:switchyard-quickstart-demo:orders:1.0}submitOrder"/>
   ```

11.4.1.2. Java Validator Reference

The optional name element of the @Validator annotation can be used to specify the qualified type name used during validator registration. If not supplied, the full class name of the method parameter is used as the type.

The CDI bean name specified by @Named annotation is used to resolve validator class. If you fail to specify it, then the validator's class name is used instead:

```
<validates>
  <validate.java class="org.switchyard.quickstarts.demos.orders.MyValidator"
```

```
                    name="{urn:switchyard-quickstart-demo:orders:1.0}submitOrder"/>
</transforms>
```

NOTE

Each of the <validate.java> definitions have either a bean attribute or a class attribute. These two attributes are mutually exclusive.

11.4.1.3. ValidationResult

ValidationResult is a simple interface which represents the result of validation. It has two methods, **isValid()** and **getDetail()**.

11.4.1.4. Java Validation Failure

If Java validation fails, the message exchange process stops immediately and a **HandlerException** is thrown along with a validation failure message which is returned by **ValidationResult.getDetail()**. Make sure that you return a **ValidationResult** with failure detail message when you want to indicate a validation failure in your Java Validator, instead of throwing a **RuntimeException**.

11.4.1.5. ValidationResult Properties

isValid() returns whether the validation succeeded or not. **_getDetail()** returns an error message if validation fails.

```
package org.switchyard.validate;
public interface ValidationResult {
    boolean isValid();
    String getDetail();
}
```

There are three convenience methods available for **org.switchyard.validate.BaseValidator**. These are **validResult()**, **invalidResult()** and **invalidResult(String)** Use them to generate ValidationResult objects.

11.4.2. XML

11.4.2.1. Use an XML Validator

The XML validator allows you to perform a validation against its schema definition. It support DTD, XML_SCHEMA, and RELAX_NG schema types. You can configure an XML validator by specifying the schema Type, the name QName, and the path to the schema file. Here is an example:

```
<validate.xml schemaType="XML_SCHEMA" name="{urn:switchyard-quickstart:validate-
xml:0.1.0}order" failOnWarning="true" namespaceAware="true">
   <schemaFiles>
      <entry file="/xsd/orders.xsd"/>
   </schemaFiles>
   <schemaCatalogs>
      <entry file="/xsd/catalog.xml"/>
   </schemaCatalogs>
</validate.xml>
```

If you specify the ***failOnWarning*** attribute as true, the validation fails if any warning is detected during validation. If the XML content to be validated has namespace prefix, then you need to specify ***namespaceAware*** as true.

11.4.2.2. XML Catalog

You can use XML catalog to decouple the schema file location from schema definition itself. This schema is **orders.xsd** which has a import element. It refers to logical name orders.base by the ***schemaLocation*** attribute:

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        targetNamespace="urn:switchyard-quickstart:validate-xml:0.1.0"
        xmlns:base="urn:switchyard-quickstart:validate-xml-base:0.1.0"
        xmlns:orders="urn:switchyard-quickstart:validate-xml:0.1.0">
        <import namespace="urn:switchyard-quickstart:validate-xml-base:0.1.0"
schemaLocation="orders.base"/>
...
```

Here is the **catalog.xml** file that resolves actual schema location from logical name orders.base:

```
<catalog xmlns="urn:oasis:names:tc:entity:xmlns:xml:catalog">
    <system systemId="orders.base" uri="orders-base.xsd"/>
</catalog>
```

11.4.2.3. XML Validation Failure

If XML validation fails, the message exchange process stops immediately and a **HandlerException** is thrown along with a validation failure message. XMLValidator collects a set of validation failures through the SAX **ErrorHandler**, and uses the **getMessage()** of each received **SAXParseException** as a failure detail with extracting root cause, if exists.

CHAPTER 12. CONFIGURATION PROPERTIES

SwitchYard supports configuration properties that can be injected into configuration or into service implementations.

12.1. SWITCHYARD MODEL CONFIGURATION

SwitchYard allows you to replace any attribute or element value in the **switchyard.xml** file with a property from the runtime environment. The syntax for a replaced token is:

```
${varname}
```

Here, varname is the property name. The configuration layer in SwitchYard is configured with instances of PropertyResolver, which are used to resolve the value of a property based on its name.

Property values are resolved from the following locations:

- System properties passed by **-D** option of Java VM. For example:

  ```
  -Dproperty.name=property.value
  ```

- System environment variables, referenced with an **env.** prefix. For example:

  ```
  env.PATH
  ```

- Unit test properties
- JBoss EAP properties, including access into the SecurityVault
- Domain properties in **switchyard.xml** file
- SCA property definitions in the composite or component

The priority in resolving a property is from top to bottom. So a property defined as a System property always takes precedence over a property defined at domain or composite, and a property at domain level always takes precedence over a property defined at component level.

12.2. INJECTING PROPERTIES INTO SERVICE IMPLEMENTATION

Implementation properties allow you to inject one or more property values into a service implementation. This is based on the property support in the SCA assembly specification. Since the property is injected into service implementation logic, the injection mechanism itself is unique to each implementation type. Here are the details for each implementation type:

- Java: Injected using @Property into a CDI bean
- Camel: Wired into Camel properties component and accessible in a Camel route using Camel's own varName property notation
- BPEL: Mapped into process variables using <assign> with using **resolveProperty()** XPath custom function
- BPMN 2: Inserted into process variables by data input associations

- Drools: Available in a global map

12.2.1. Injecting Properties in Java Bean Implementations

Implementation properties represent environmental properties that you have defined in the SwitchYard application descriptor (**switchyard.xml**) for your bean implementation. Implementation properties in SwitchYard are the properties that you can configure on a specific service implementation. That is, you can make the property value available to service logic executing inside an implementation container. Here is an example:

```
<sca:component name="SimpleServiceBean">
     <bean:implementation.bean
class="com.example.switchyard.switchyard_example.SimpleServiceBean"/>
     <sca:service name="SimpleService">
       <sca:interface.java interface="com.example.switchyard.switchyard_example.SimpleService">
         <properties>
           <property name="userName" value="${user.name}"/>
         </properties>
       </sca:interface.java>
     </sca:service>
</sca:component>
```

To access the Implementation Properties, add an @Property annotation to your bean class identifying the property you want to inject:

```
@Service(SimpleService.class)
public class SimpleServiceBean implements SimpleService {

  @Property(name="userName")
  private String name;

  @Override
  public String sayHello(String message) {
       return "Hello " + name + ", I got a message: " + message;
    }

}
```

12.2.2. Injecting Implementation Properties in Camel Routes

SwitchYard integrates with the Properties Component in Camel to make system and application properties available inside your route definitions. You can inject properties into your camel route using **{{propertyname}}** expression, where propertyName is the name of the property. For example, the following camel route expects the user.name property to be injected in the last <Log> statement:

```
<route xmlns="http://camel.apache.org/schema/spring" id="CamelTestRoute">
    <log message="ItemId [${body}]"/>
    <to uri="switchyard://WarehouseService?operationName=hasItem"/>
    <log message="Title Name [${body}]"/>
    <log message="Properties [{{user.name}}]"/>
</route>
```

12.2.3. Injecting Implementation Properties in BPEL

You can inject properties into your BPEL process definition with using **SwitchYardPropertyFunction.resolveProperty()** XPath custom function. In the example below, **bpel:copy** section copies Greeting property value into the **ReplySayHelloVar** variable:

```
<bpel:copy>
 <bpel:from
xmlns:property="java:org.switchyard.component.bpel.riftsaw.SwitchYardPropertyFunction"
                expressionLanguage="urn:oasis:names:tc:wsbpel:2.0:sublang:xpath2.0">
      <![CDATA[concat(property:resolveProperty('Greeting'),
$ReceiveSayHelloVar.parameters/tns:input)]]>
      <bpel:from>
          <bpel:to part="parameters" variable="ReplySayHelloVar">
           <bpel:query queryLanguage="urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0">
            <![CDATA[tns:result]]>
      </bpel:query>
 </bpel:to>
</bpel:copy>
```

12.3. INVOCATION PROPERTIES

While it is a best practice to write your service logic to the data that is defined in the contract (the input and output message types), there can be situations where you need to access contextual information like message headers (such as, received file name in your implementation). To facilitate this, the Bean component allows you to access the SwitchYard Exchange Context instance associated with a given Bean Service Operation invocation. Invocation properties represent the contextual information (like message headers) in your bean implementation.

To enable access to the invocation properties, add a Context property to your bean and annotate it with the CDI @Inject annotation:

```
@Service(SimpleService.class)
public class SimpleServiceBean implements SimpleService {

@Inject
private Context context;

public String sayHello(String message) {
        System.out.println("*** Funky Context Property Value: " +
context.getPropertyValue("funkyContextProperty"));
        return "Hi there!!";
    }
}
```

Here, the Context interface allows your bean logic to get and set properties in the context.

NOTE

You can invoke the Context instance only within the scope of one of the Service Operation methods. If you invoke it outside this scope, it results in an **UnsupportedOperationException** error.

12.4. CONFIGURATION TIPS AND TRICKS

12.4.1. Defining Default Value for a Property

When you define a property, you can provide a default value for it. To define this default value, append the default value to the property name, separated by a colon (:). For example:

```
${server.port:8080}
```

12.4.2. Defining Environment Properties

12.4.2.1. Environment Properties as Component Properties

You can define properties as component properties. This way of defining properties is not dynamic, but you can override each property defined here, by properties defined in a prioritized scope.

```
<sy:switchyard ...>
  <sca:composite ...>
    <sca:component ...>
      ...
      <sca:property value="test" name="MY_PROPERTY"/>
    </sca:component>
    <sca:service...>
      ...
    </sca:service>
    <sca:reference ...>
      ...
    </sca:reference>

  </sca:composite>
  ...
</sy:switchyard>
```

12.4.2.2. Environment Properties as Composite Properties

You can define properties as composite properties. This way of defining properties is not dynamic, but you can override every property defined here, by properties defined in a prioritized scope.

```
<sy:switchyard ...>
  <sca:composite ...>
    <sca:component ...>
      ...
    </sca:component>
    <sca:service...>
      ...
    </sca:service>
    <sca:reference ...>
      <sca:interface.java .../>
      <file:binding.file name="FileBinding">
        <file:directory>/tmp</file:directory>
        <file:fileName>${MY_FILENAME}</file:fileName>
        <file:produce/>
      </file:binding.file>
    </sca:reference>
    <sca:property value="test.txt" name="MY_FILENAME"/>
  </sca:composite>
  ...
</sy:switchyard>
```

12.4.2.3. Environment Properties as Domain Properties

You can define properties as composite properties. This way of defining properties is not dynamic, but you can override each property defined here, by properties defined in a prioritized scope.

```
<sy:switchyard ...>
  <sca:composite ...>
    <sca:component ...>
      ...
    </sca:component>
    <sca:service...>
      ...
    </sca:service>
    <sca:reference ...>
      ...
    </sca:reference>
  </sca:composite>
  ...
  <sca:domain>
    <sca:property value="test.txt" name="MY_FILENAME"/>
  </sca:domain>
 </sy:switchyard>
```

12.4.2.4. Environment Properties as OS Environment Properties

You can load properties from OS environment properties. Every environment property is accessible by prefixing it with env. For example, in bash, you can use a property defined as **export MY_PROPERTY=test**, in your SwitchYard application as shown below:

```
<ftp:binding.sftp>
    <ftp:host>${env.MY_PROPERTY}</ftp:host>
   ....
</ftp:binding.sftp>
```

12.4.2.5. Environment Properties as Application Server Properties from AS Configuration

Application server has the ability to define properties directly in its configuration either by file or with the console. This configuration is dynamically updated and persisted.

Add the configuration to the server definition as shown below:

```

<server name="xyz.home" xmlns="urn:jboss:domain:1.0">
    <extensions>
        <extension module="org.jboss.as.clustering.infinispan"/>
        <extension module="org.jboss.as.clustering.jgroups"/>
        <extension module="org.jboss.as.connector"/>
        ....
    </extensions>
    <system-properties>
        <property name="MY_PROPERTY" value="test"/>
    </system-properties>
```

This property is used in your SwitchYard application as:

```
<ftp:binding.sftp>
    <ftp:host>${MY_PROPERTY}</ftp:host>
   ....
</ftp:binding.sftp>
```

12.4.2.6. Environment Properties as Application Server Properties from File

You can pass a properties file as an argument to JBoss Application Server (AS) startup script to load all the properties in the file and make them accessible. You can start the AS as shown below:

```
$./standalone.sh -P file:///data/production.properties
```

Here are some alternatives:

- -P=<url>: Load system properties from the given URL
- -P<url>: Load system properties from the given URL
- --properties=<url>: Load system properties from the given URL

In your SwitchYard application, you can use these properties as:

```
<ftp:binding.sftp>
    <ftp:host>${env.MY_PROPERTY}</ftp:host>
   ....
</ftp:binding.sftp>
```

12.4.3. Loading Properties for Test

In tests, you can add and resolve properties at the top level. The PropertyMixIn eases working with properties:

```
private PropertyMixIn pmi;

...
pmi.set("test.property.name", "test");
pmi.set("test.property.name", Integer.valueOf(100));
...
pmi.get("test.property.name");
...
```

In case you need access to the PropertyResolver for tests, where a MixIn is not applicable, you can use TestPropertyResolver.INSTANCE and avoid setting command line parameters as shown below:

```
TestPropertyResolver.INSTANCE.getMap().put("name","value");
```

CHAPTER 13. TESTING SUPPORT IN SWITCHYARD

SwitchYard uses JUnit, which is a unit testing framework for Java. SwitchYard provides comprehensive unit test support for testing your applications. There are three primary elements to test support in SwitchYard:

- **SwitchYardRunner**
- **SwitchYardTestKit**
- **SwitchYardTestCaseConfig**

13.1. SWITCHYARDRUNNER CLASS

The SwitchYardRunner Class is a JUnit test Runner class that takes care of bootstrapping an embedded SwitchYard runtime and deploying a SwitchYard application for the test instance.

In order to take advantage of the test support in SwitchYard, ensure that your unit test is annotated with the **SwitchYardRunner** JUnit test Runner class. The **SwitchYardRunner** creates and starts an embedded runtime for each test method. After the embedded runtime starts, the project containing the test is packaged as a SwitchYard application and deployed to it. An instance of the **SwitchYardTestKit** class is injected into the test when a property of type SwitchYardTestKit is declared in the test. This instance represents the runtime state of the deployed SwitchYard test application.

```
@RunWith(SwitchYardRunner.class)
public class MyServiceTest  {

    private SwitchYardTestKit testKit;

    @Test
    public void testOperation() {
        MyTestServiceHandler service = new MyTestServiceHandler();

        // register the service...
        testKit.registerInOutService("MyService", service);

        // invoke the service and capture the response...
        Message response = newInvoker("MyService")
        .sendInOut("<create>A1234</create>");

        // test the response content by doing an XML comparison with a
        // file resource on the classpath...
        testKit.compareXMLToResource(response.getContent(String.class), "/myservice/expected-
create-response.xml");
    }

    private class MyTestServiceHandler implements ExchangeHandler {
        // implement methods....
    }
}
```

The **SwitchYardTestKit** class provides a set of utility methods for performing all sorts of deployment configuration and test operations.

13.2. SWITCHYARDTESTKIT CLASS

The **SwitchYardRunner** Class represents the runtime state of the deployed SwitchYard application instance deployed by **SwitchYardRunner**. It also provides access to a set of test utility methods for the test (For example, assertion methods). If a property of type **SwitchYardTestKit** is declared in the test, the **SwitchYardTestKit** is injected into the test instance.

13.3. SWITCHYARDTESTCASECONFIG

The SwitchYardTestCaseConfig annotation is optional. You can use it control the behavior of the SwitchYardRunner:

- ***config***: Enables you to specify a SwitchYard XML configuration file (**switchyard.xml**) for the test. The SwitchYardRunner attempts to load the specified configuration from the classpath. If it fails to locate the config on the classpath, it attempts to locate it on the file system (For example, within the project structure).

- ***mixins***: Enables you to add specific testing tools to your test case. Each TestMixIn is a composition-based method that provides customized testing tools for service implementations, gateway bindings, and transformers. When a TestMixIn is annotated on a test class, the SwitchYardRunner handles all the initialization and cleanup (life cycle) of the TestMixIn instances. It is also possible to manually create and manage TestMixIn instances within your test class if you are not using the SwitchYardRunner.

- ***scanners***: Enables you to add classpath scanning as part of the test life cycle. This adds the same Scanner behavior as the one available with the SwitchYard maven build plug-in. However, it allows the scanning to take place as part of the test life cycle. You may need to add Scanners if you want your test to run inside your IDE. This is because running your test inside your IDE bypasses the whole maven build process, which means the build plug-in does not perform any scanning.

Here is how you can use the SwitchYardTestCaseConfig annotation:

```
@RunWith(SwitchYardRunner.class)
@SwitchYardTestCaseConfig(config = "testconfigs/switchyard-01.xml", mixins = {CDIMixIn.class},
scanners = {BeanSwitchYardScanner.class, TransformSwitchYardScanner.class})
public class MyServiceTest  {

    @Test
    public void testOperation() {
        newInvoker("OrderService")
        .operation("createOrder")
        .sendInOnly("<order><product>AAA</product><quantity>2</quantity></order>");
    }
}
```

13.4. ADD TEST SUPPORT TO A SWITCHYARD APPLICATION

You can add test support to your SwitchYard application by adding a dependency to the switchyard-test module in your application's **pom.xml** as shown below:

```
<dependency>
    <groupId>org.switchyard</groupId>
    <artifactId>switchyard-test</artifactId>
    <version>[release-version]</version> <!-- e.g. "1.1.1-p5-redhat-1" -->
    <scope>test</scope>
</dependency>
```

NOTE

Note: camel dependency version is 2.10.0.redhat-60024.

In addition to a dependency on the core test framework, you can also use the MixIns in your test classes.

13.5. THE TESTMIXIN FEATURE

The TestMixIn feature allows you to selectively enable additional test functionality based on the capabilities of your application. To include MixIn support in your application, you must include a Maven dependency in your application's **pom.xml**:

```
<dependency>
    <groupId>org.switchyard.components</groupId>
    <artifactId>switchyard-component-test-mixin-name</artifactId>
    <version>release-version</version> <!-- e.g. "1.0" -->
    <scope>test</scope>
</dependency>
```

13.6. TESTMIXIN CLASSES

- **CDIMixIn** (switchyard-component-test-mixin-cdi): Boostraps a stand-alone CDI environment, automatically discovers CDI beans, registers bean services, and injects references to SwitchYard services.
- **HTTPMixIn** (switchyard-component-test-mixin-http): Client methods for testing HTTP-based services.
- **SmooksMixIn** (switchyard-component-test-mixin-smooks): Stand-alone testing of any Smoooks transformers in your application.
- **HornetQMixIn** (switchyard-component-test-mixin-hornetq): Bootstraps a stand-alone HornetQ server and provides utility methods to interact with it for testing purpose. It can be also used to interact with remote HornetQ server.
- **NamingMixIn** (switchyard-component-test-mixin-naming): Provides access to naming and JNDI services within an application.
- **PropertyMixIn** (switchyard-test): Provides ability to set test values to properties that are used within the configuration of the application.

13.7. SCANNERS

Scanners add classpath scanning as part of the test life cycle. This adds the same Scanner behavior as is available with the SwitchYard maven build plug-in, but allows the scanning to take place as part of the test life cycle. SwitchYard provides the following Scanners:

- BeanSwitchYardScanner: Scans for CDI Bean Service implementations.
- TransformSwitchYardScanner: Scans for Transformer
- BpmSwitchYardScanner: Scans for @Process, @StartProcess, @SignalEvent and @AbortProcessInstance annotations.

- RouteScanner: Scans for Camel Routes
- RulesSwitchYardScanner: Scans for @Rule annotations

13.8. METADATA AND SUPPORT CLASS INJECTIONS

13.8.1. TestKit Injection

- You can inject the SwitchYardTestKit instance into the test at runtime by declaring a property of that type in the test class, as shown below:

```
@RunWith(SwitchYardRunner.class)
public class MyServiceTest  {

    private SwitchYardTestKit testKit;

    // implement test methods...
}
```

13.8.2. Deployment Injection

You can inject the deployment instance by declaring a property of the type Deployment, as shown below:

```
@RunWith(SwitchYardRunner.class)
public class MyServiceTest  {

    private Deployment deployment;

    // implement test methods...
}
```

13.8.3. SwitchYardModel Injection

You can inject the SwitchYardModel instance by declaring a property of the type SwitchYardModel, as shown below:

```
@RunWith(SwitchYardRunner.class)
public class MyServiceTest  {

    private SwitchYardModel model;

    // implement test methods...
}
```

13.8.4. ServiceDomain Injection

You can inject the ServiceDomain instance by declaring a property of the type ServiceDomain, as shown below:

–

```
@RunWith(SwitchYardRunner.class)
public class MyServiceTest  {

    private ServiceDomain serviceDomain;

    // implement test methods...
}
```

13.8.5. TransformerRegistry Injection

You can inject the TransformerRegistry instance by declaring a property of the type TransformerRegistry, as shown below:

```
@RunWith(SwitchYardRunner.class)
public class MyServiceTest  {

    private TransformerRegistry transformRegistry;

    // implement test methods...
}
```

13.8.6. TestMixIn Injection

You can inject the TestMixIn Injection instance by declaring a property of the type TestMixIn Injection, as shown below:

```
@RunWith(SwitchYardRunner.class)
@SwitchYardTestCaseConfig(mixins = {CDIMixIn.class, HTTPMixIn.class})
public class MyServiceTest  {

    private CDIMixIn cdiMixIn;
    private HTTPMixIn httpIn;

    // implement test methods...
}
```

13.8.7. PropertyMixIn Injection

PropertyMixIn instances are injected like any other TestMixIn type, however you must set any properties you wish to use on the MixIn before deployment in order for them to be used. To do this, use the @BeforeDeploy annotation:

```
@RunWith(SwitchYardRunner.class)
@SwitchYardTestCaseConfig(mixins = {CDIMixIn.class, PropertyMixIn.class, HTTPMixIn.class})
public class MyServiceTest  {

    private PropertyMixIn propMixIn;
    private HTTPMixIn httpMixIn;

    @BeforeDeploy
    public void setTestProperties() {
        propMixIn.set("soapPort", Integer.valueOf(18002));
    }

    // implement test methods...
}
```

13.8.8. Invoker Injection

To inject Service Invoker instances, declare properties of the type Invoker and annotate them with @ServiceOperation. (Note the annotation value is a dot-delimited Service Operation name of the form **[service-name].[operation-name]**.)

```

@RunWith(SwitchYardRunner.class)
@SwitchYardTestCaseConfig(config = "testconfigs/switchyard-01.xml")
public class MyServiceTest  {

    @ServiceOperation("OrderService.createOrder")
    private Invoker createOrderInvoker;

    @Test
    public void test_createOrder() {
        createOrderInvoker.sendInOnly("<order><product>AAA</product><quantity>2</quantity>
</order>");
    }
}
```

13.9. SELECTIVELY ENABLING ACTIVATORS FOR A TEST

The test framework defaults to a mode where the entire application descriptor is processed during a test run. This means all gateway bindings and service implementations are activated during each test. There are times when this may not be appropriate, so we allow activators to be selectively enabled or disabled based on your test configuration.

In this example, SOAP bindings are excluded from all tests. (This means that SOAP gateway bindings will not be activated when the test framework loads the application.)

```
@RunWith(SwitchYardRunner.class)
@SwitchYardTestCaseConfig(config = "testconfigs/switchyard-01.xml" exclude="soap")
public class NoSOAPTest  {
   ...
}
```

This example includes only CDI bean services as defined in the application descriptor:

```
@RunWith(SwitchYardRunner.class)
@SwitchYardTestCaseConfig(config = "testconfigs/switchyard-02.xml" include="bean")
public class BeanServicesOnlyTest  {
...
}
```

Sometimes you will need to add some procedures before you perform the test. The JUnit @Before operation is invoked immediately after the application is deployed. You cannot, however, use it if you expect something to happen before deployment.

13.10. USEFUL TIPS FOR CREATING JUNIT TESTS

13.10.1. Using Harmcrest to Assert

Hamcrest is a framework for writing matcher objects. It allows you to define match rules declaratively. Use Hamcrest's **assertThat** construct and the standard set of matchers, both of which you can statically import:

```
import static org.hamcrest. MatcherAssert .assertThat;
import static org.hamcrest. Matchers .*;
```

Hamcrest comes with a library of useful matchers, such as:

- Core
  - anything: Always matches, useful if you do not want to know what the object under test is
  - describedAs: Decorator for adding custom failure description
  - is: Decorator to improve readability
- Logical
  - allOf: Matches if all matchers match, short circuits (like && in Java)
  - anyOf: Matches if any matchers match, short circuits (like || in Java)
  - not: Matches if the wrapped matcher does not match and vice versa
- Object
  - equalTo: Test object equality using Object.equals
  - hasToString: Test Object.toString
  - instanceOf, isCompatibleType: Test type
  - notNullValue, nullValue: Test for null
  - sameInstance: Test object identity
- Beans
  - hasProperty: Test JavaBeans properties
- Collections
  - array: Test an array's elements against an array of matchers
  - hasEntry, hasKey, hasValue: Test a map contains an entry, key or value
  - hasItem, hasItems: Test a collection contains elements

    - hasItemInArray: Test an array contains an element
- Number
    - closeTo: Test floating point values are close to a given value
    - greaterThan, greaterThanOrEqualTo, lessThan, lessThanOrEqualTo: Test ordering
- Text
    - equalToIgnoringCase: Test string equality ignoring case
    - equalToIgnoringWhiteSpace: Test string equality ignoring differences in runs of whitespace
    - containsString, endsWith, startsWith: Test string matching

13.10.2. Invoking a Component Service

In order to invoke a component service, you must inject an invoker for certain ServiceOperation. When injecting a service operation, specify it in ***[service_name].[operation_name]*** notation.

```
import org.switchyard.test.Invoker;
...

@RunWith(SwitchYardRunner.class)
@SwitchYardTestCaseConfig(mixins = CDIMixIn.class)
public class ExampleServiceTest {

   @ServiceOperation("ExampleService.submitOperation")
   private Invoker submitOperation;

   @Test
   public void testOK() throws Exception {
      ParamIn testParam = new ParamIn()
         .set...(...);

      ParamOut result = submitOperation
         .sendInOut(testParam)
         .getContent(ParamOut.class);

      Assert....
   }

   @Test
   public void testForFault() throws Exception {
      ParamIn testParam = new ParamIn()
         .set...(...);

      try{
         // This method invocation should throw a fault
         ParamOut result = submitOperation
            .sendInOut(testParam)
            .getContent(ParamOut.class);

         Assert.fail
      } catch (InvocationFaultException ifex){
        Assert.... // Assert for correct type of exception
    }
}
```

An invocation to a service operation can throw a **InvocationFaultException** whenever the method throws a fault. So catching this exception is similar to validating for the fault being thrown.

You can:

- Check against original exception by checking the type of the InvocationFaultException:

  ```
  ifex.isType(MyOriginalException.class)
  ```

- Use the JUnit functionality of setting the expected exception in the test:

  ```
  @Test(expected=org.switchyard.test.InvocationFaultException.class)
  ```

13.10.3. SwitchYardTestKit Utility Methods

TestKit provides the following set of utility methods to ease validations and some common operations that are performed on test classes:

- Access to underlying
  - getTestInstance
  - getActivators
  - getDeployment
  - getServiceDomain
  - createQName
- Service manipulation
  - registerInOutService
  - registerInOnlyService
  - removeService
  - replaceService
- Invocation
  - newInvoker
- Transformations
  - addTransformer
  - newTransformer
  - registerTransformer

- MixIns
  - getMixIns
  - getMixIn
- Dependencies
  - getRequiredDependencies
  - getOptionalDependencies
- Resources
  - getResourceAsStream
  - readResourceBytes
  - readResourceString: Reads a resource (file) form the classpath
  - readResourceDocument
- Configuration
  - loadSwitchYardModel
  - loadConfigModel
- XML Comparison
  - compareXMLToResource: Compares a XML in string format with a XML file in the classpath.
  - compareXMLToString
- Tracing
  - traceMessages: enables message tracing for the application under test.

13.10.4. Testing Transformations in Component Service

While testing a component invocation, you can test for the appropriate transformation with additional methods on the invocation. You can do this for the input transformation, as well as for the output transformation as shown below:

```
...

    @ServiceOperation("ExampleService.submitOperation")
    private Invoker serviceOperationInvocation;

    @Test
    public void testForInputTransformation() throws Exception {
        ParamOut result =  serviceOperationInvocation
                      .inputType(QName.valueOf("{urn:com.examaple:service:1.0"}submitOperation))
                      .sendInOut(....)
                      .getContent(ParamOut.class);
        Assert....  // Assert that result is OK, so transformation was OK
    }
    @Test
    public void testForOutputXMLTransformation() throws Exception {
        ParamIn testParam = new ParamIn()
            .set...(...);

        ParamOut result =  serviceOperationInvocation
                         .expectedOutputType(QName.valueOf("
{urn:com.examaple:service:1.0"}submitOperationResponse))
                         .sendInOut(testParam)
                         .getContent(Element.class); // Expect Element as transformation is for XML

        XMLAssert....  // Assert that result is what is expected
    }
```

You can use XMLUnit and XMLAssert from **org.custommonkey.xmlunit** to ease validations.

13.10.5. Mocking a Service, Component, or Reference

Mocking a component may be useful, so it is never invoked for the sake of a test. For this, SwitchYardTestKit provides with the ability of adding, replacing, or removing services.

```
// replace existing implementation for testing purposes
    testKit.removeService("MyService");s
    final MockHandler myService = testKit.registerInOnlyService("MyService");

    .... // Invoke the service under test

    // Assert what has arrived ath the mocked service
    final LinkedBlockingQueue<Exchange> recievedMessages = myService.getMessages();
    assertThat(recievedMessages, is(notNullValue()));

    final Exchange recievedExchange = recievedMessages.iterator().next();
    assertThat(recievedExchange.getMessage().getContent(String.class), is(equalTo(...)));
```

- If you want to assert what has arrived or produced in the MockHandler, you can use the following options:
    - getMessages(): This provides with the list of received messages.
    - getFaults(): This provides with the list of prodced faults.
- If the service is InOut, you may need to mock a response. You can use the following options:
    - forwardInToOut()
    - forwardInToFault()
    - replyWithOut(Object)
    - replyWithFault(Object)

      For example:

      ```
      final MockHandler mockHandler = testKit.registerInOutService("MyService");
          mockHandler.forwardInToOut();
      ```

-

- If you want to instruct the MockHandler to wait for certain message, you can use the following options:
    - waitForOkMessage()
    - waitForFaultMessage()

      The MockHandler waits for 5 seconds by default, unless instructed to wait for a different period with setWaitTimeout(milis).

13.10.6. Mocking a Service For More Than One Method Invocation

In some cases, the service you are mocking may be called

- Twice in the context of a single unit test, or
- Multiple times for the same method, or
- Multiple times for different methods

In this case, you can register an **ExchangeHandler** with the mock, while registering and replacing the original service. The **ExchangeHandler** gets the message, and contains the logic that you need to put to deal with this scenario, as shown below:

```
testKit.replaceService(qname, new ExchangeHandler() {

        @Override
        public void handleMessage(Exchange arg0) throws HandlerException {
            // Here logic to handle with messages
        }

        @Override
        public void handleFault(Exchange arg0) throws HandlerException {
            // Here logic to handle with faults
        }
    });
```

You can reuse this **ExchangeHandler** by making it a named class (not anonymous).

13.10.6.1. Multiple Invocations of a Single Method

In the case of multiple invocation of the same method, the ExchangeHandler keeps track of the invocation number, in case it has to answer with different messages:

```
testKit.replaceService(qname, new ExchangeHandler() {
        int call=1;

        @Override
        public void handleMessage(Exchange exchange) throws HandlerException {
            if (call++ == 1){ // First call
                // Do whatever wants to be done as result of this operation call, and return the expected
output
                Result result = ...; / Result is return type for operation store
                exchange.send(exchange.createMessage().setContent(result));
            }else if (call++ == 2){ // Second call
                // Do whatever wants to be done as result of this operation call, and return the expected
output
                Result result = ...; / Result is return type for operation store
                exchange.send(exchange.createMessage().setContent(result));
            }else{
                throw new HandlerException("This mock should not be called more than 2 times");
            }
        }

        @Override
        public void handleFault(Exchange exchange) throws HandlerException {
            // Here logic to handle with faults
        }
    });
```

13.10.6.2. Multiple Invocations of Different Methods

In the case of multiple invocation of different methods, the ExchangeHandler checks for operation name, to know which method is being invoked:

```
testKit.replaceService(qname, new ExchangeHandler() {

        @Override
        public void handleMessage(Exchange exchange) throws HandlerException {
            if (exchange.getContract().getProviderOperation().getName().equals("store")){
                // Do whatever wants to be done as result of this operation call, and return the expected
output
                Result result = ...; / Result is return type for operation store
                exchange.send(exchange.createMessage().setContent(result));
            }else if (exchange.getContract().getProviderOperation().getName().equals("getId")){
                // Do whatever wants to be done as result of this operation call, and return the expected
output
                exchange.send(exchange.createMessage().setContent(1)); // This operation returns a Int
            }else{
                throw new HandlerException("No operation with that name should be executed");
            }
        }

        @Override
        public void handleFault(Exchange exchange) throws HandlerException {
            // Here logic to handle with faults
        }
    });
```

13.10.7. Setting Properties For a Test

You can use the PropertyMixIn property to set test properties in configurations, as shown below:

```
    private PropertyMixIn pmi;

    ...
    pmi.set("test.property.name", "test");
    pmi.set("test.property.name", Integer.valueOf(100));
```

```
...
pmi.get("test.property.name");
...
```

13.10.8. Testing a Deployed Service with HTTPMixin

Use HTTPMixin to test a deployed service, as shown below:

```
@RunWith(SwitchYardRunner.class)
@SwitchYardTestCaseConfig(
        scanners = TransformSwitchYardScanner.class,
        mixins = {CDIMixIn.class, HTTPMixIn.class})
public class WebServiceTest {

    private HTTPMixIn httpMixIn;

    @Test
    public void invokeWebService() throws Exception {
        // Use the HttpMixIn to invoke the SOAP binding endpoint with a SOAP input (from the test
classpath)
        // and compare the SOAP response to a SOAP response resource (from the test classpath)...
        httpMixIn.setContentType("application/soap+xml");
        httpMixIn.postResourceAndTestXML("http://localhost:18001/service-context/ServiceName",
"/xml/soap-request.xml", "/xml/soap-response.xml");
    }
}
```

You can also use HTTPMixin from a main class, as shown below:

```
    /**
     * Only execution point for this application.
     * @param ignored not used.
     * @throws Exception if something goes wrong.
     */
    public static void main(final String[] ignored) throws Exception {

        HTTPMixIn soapMixIn = new HTTPMixIn();
        soapMixIn.initialize();

        try {
            String result = soapMixIn.postFile(URL, XML);
            System.out.println("SOAP Reply:\n" + result);
        } finally {
            soapMixIn.uninitialize();
        }
    }
```

13.10.9. Creating an Embedded WebService to Test a Component

In situations where you wish to only test a single component, you can expose it dynamically as a WebService and invoke it, as shown below:

```
import javax.xml.ws.Endpoint;
...
```

```
@RunWith(SwitchYardRunner.class)
@SwitchYardTestCaseConfig(
        config = SwitchYardTestCaseConfig.SWITCHYARD_XML,
        scanners = {TransformSwitchYardScanner.class},
        mixins = {HTTPMixIn.class})
public class CamelSOAPProxyTest {

    private static final String WEB_SERVICE = "http://localhost:8081/MyService";

    private HTTPMixIn _http;
    private Endpoint _endpoint;

    @BeforeDeploy
    public void setProperties() {
        System.setProperty("org.switchyard.component.http.standalone.port", "8081");
    }

    @Before
    public void startWebService() throws Exception {
        _endpoint = Endpoint.publish(WEB_SERVICE, new ReverseService());
    }

    @After
    public void stopWebService() throws Exception {
        _endpoint.stop();
    }

    @Test
    public void testWebService() throws Exception {
        _http.postResourceAndTestXML(WEB_SERVICE, "/xml/soap-request.xml", "/xml/soap-
response.xml");
    }
}
```

13.10.10. Testing a Deployed Service with HornetQMixIn

When you need to test an application that has a JMS binding, you may want to test with the binding itself. In such cases, you can use HornetQMixIn. HornetQMixIn gets its configuration from the following two files, which must be present on the classpath for the test:

- **hornetq-configuration.xml**: This file contains the configuration for the HornetQ server.

```
<configuration xmlns="urn:hornetq">

        <paging-directory>target/data/paging</paging-directory>
        <bindings-directory>target/data/bindings</bindings-directory>
        <persistence-enabled>false</persistence-enabled>
        <journal-directory>target/data/journal</journal-directory>
        <journal-min-files>10</journal-min-files>
        <large-messages-directory>target/data/large-messages</large-messages-directory>
        <security-enabled>false</security-enabled>

        <connectors>
                <connector name="invm-connector">
```

```
                    <factory-
class>org.hornetq.core.remoting.impl.invm.InVMConnectorFactory</factory-class>
                </connector>
                <connector name="netty-connector">
                 <factory-
class>org.hornetq.core.remoting.impl.netty.NettyConnectorFactory</factory-class>
                 <param key="port" value="5545"/>
      </connector>
        </connectors>

        <acceptors>
                <acceptor name="invm-acceptor">
                        <factory-
class>org.hornetq.core.remoting.impl.invm.InVMAcceptorFactory</factory-class>
                </acceptor>
                <acceptor name="netty-acceptor">
                        <factory-
class>org.hornetq.core.remoting.impl.netty.NettyAcceptorFactory</factory-class>
                        <param key="port" value="5545"/>
                </acceptor>
        </acceptors>

</configuration>
```

IMPORTANT

The **camel-netty** component is deprecated since JBoss Fuse 6.3 and will be replaced by the **camel-netty4** component in a future release of JBoss Fuse.

- **hornetq-configuration.xml**: This file contains the definition of the connection factories, queues, and topics.

```
<configuration xmlns="urn:hornetq">

   <connection-factory name="ConnectionFactory">
      <connectors>
        <connector-ref connector-name="invm-connector"/>
      </connectors>

      <entries>
         <entry name="ConnectionFactory"/>
      </entries>
   </connection-factory>

   <queue name="TestRequestQueue">
      <entry name="TestRequestQueue"/>
   </queue>
   <queue name="TestReplyQueue">
      <entry name="TestReplyQueue"/>
   </queue>

</configuration>
```

To use HornetQMixIn in a test, you need to get a reference to the MixIn and use the appropriate mixin methods, as shown below:

```
@RunWith(SwitchYardRunner.class)
@SwitchYardTestCaseConfig(
        config = SwitchYardTestCaseConfig.SWITCHYARD_XML,
        mixins = {CDIMixIn.class, HornetQMixIn.class}
)

public class JmsBindingTest {

    private HornetQMixIn _hqMixIn;


    @Test
    public void testHelloService() throws Exception {
        Session session = _hqMixIn.getJMSSession();
        MessageProducer producer =
session.createProducer(HornetQMixIn.getJMSQueue(REQUEST_NAME));
        Message message = _hqMixIn.createJMSMessage(createPayload(NAME));
        producer.send(message);

        MessageConsumer consumer =
session.createConsumer(HornetQMixIn.getJMSQueue(REPLY_NAME));
        message = consumer.receive(3000);
        String reply = _hqMixIn.readStringFromJMSMessage(message);
        SwitchYardTestKit.compareXMLToString(reply, createExpectedReply(NAME));
    }

    @Before
    public void getHornetQMixIn() {
        _hqMixIn = _testKit.getMixIn(HornetQMixIn.class);
    }
```

You can also test from a standalone client, as shown below:

```
public static void main(final String[] args) throws Exception {

        HornetQMixIn hqMixIn = new HornetQMixIn(false)
                                    .setUser(USER)
                                    .setPassword(PASSWD);
        hqMixIn.initialize();

        try {
            Session session = hqMixIn.getJMSSession();
            final MessageProducer producer =
session.createProducer(HornetQMixIn.getJMSQueue(REQUEST_NAME));
            producer.send(hqMixIn.createJMSMessage("<....>"));
            System.out.println("Message sent. Waiting for reply ...");

            final MessageConsumer consumer =
session.createConsumer(HornetQMixIn.getJMSQueue(REPLY_NAME));
            Message message = consumer.receive(3000);
            String reply = hqMixIn.readStringFromJMSMessage(message);
            System.out.println("REPLY: \n" + reply);
        } finally {
            hqMixIn.uninitialize();
        }

    }
```

13.10.11. Testing a Deployed Service with TransactionMixIn

You can use TransactionMixIn to test your required services with a transaction. TransactionMixIn with combination of CDIMixIn injects a UserTransaction object when required. If you need explicit access, you can use @Inject in the UserTransaction object. otherwise, it is injected in SwitchYard's functionalities. This MixIn introduces NamingMixIn, as it is a required dependency.

```
@SwitchYardTestCaseConfig(
    config = SwitchYardTestCaseConfig.SWITCHYARD_XML,
    mixins = {CDIMixIn.class, TransactionMixIn.class}
)
public YourClass{
    ....
}
```

This binds the following objects into the JNDI tree:

- TransactionManager: java:jboss/TransactionManager
- UserTransaction: java:jboss/UserTransaction
- TransactionSynchronizationRegistry: java:jboss/TransactionSynchronizationRegistry

If you need access to the provided objects, you can use the MixIn to get a reference, as shown below:

```
    private TransactionMixIn transaction;
    ....
    transaction.getUserTransaction();
    transaction.getTransactionManager();
    transaction.getSynchronizationRegistry();
```

This mixin creates transactional logs in **target/tx-store** and uses Arjuna Transactions Provider (com.arjuna.ats.jta).

13.10.12. Testing With a Different SwitchYard Configuration File

You can use the following annotation on the test class and create your reduced <**switchyard-XXXX.xml**> within the **test/resources** folder at the same package level as your test class:

```
@SwitchYardTestCaseConfig(config = "switchyard-XXXXX.xml", mixins = {.....})
```

13.10.13. Selectively Enabling Activators for a Test

The test framework defaults to a mode where the entire application descriptor is processed during a test run. This means all gateway bindings and service implementations are activated during each test. There are times when this may not be appropriate. So you must allow activators to be selectively enabled or disabled based on your test configuration. In the example below, SOAP bindings are excluded from all tests. This means that SOAP gateway bindings are not activated when the test framework loads the application.

```
@RunWith(SwitchYardRunner.class)
@SwitchYardTestCaseConfig(config = "testconfigs/switchyard-01.xml" exclude="soap")
public class NoSOAPTest  {
   ...
}
```

The example below includes only CDI bean services as defined in the application descriptor:

```

@RunWith(SwitchYardRunner.class)
@SwitchYardTestCaseConfig(config = "testconfigs/switchyard-02.xml" include="bean")
public class BeanServicesOnlyTest  {
...
}
```

You may need to add some procedures before you perform the test. The JUnit @Before operation is invoked immediately after the application is deployed. However, you can not use it if you expect something to happen before deployment.

13.10.14. Preparing Procedure for Test

JUnit @Before operation is invoked right after the application is deployed. So, you can not use @Before operation if you expect something before deployment. Use @BeforeDeploy annotation when you need to add some procedures before a test is performed.

13.10.15. Testing a Camel Binding

If you are exposing services with a camel binding, you can test it by getting the **CamelContext** and then creating a **ProducerTemplate** as shown below:

```
@RunWith(SwitchYardRunner.class)
@SwitchYardTestCaseConfig(
        config = SwitchYardTestCaseConfig.SWITCHYARD_XML,
        mixins = { CDIMixIn.class })
public class ExampleTest {

    private SwitchYardTestKit testKit;

    @Test
    public void testIntake() throws Exception {
        ServiceDomain domain = testKit.getServiceDomain();
        CamelContext ctx = (CamelContext)domain.getProperty("CamelContextProperty");
        ProducerTemplate producer = ctx.createProducerTemplate();
        producer.sendBody("direct://HelloService", "Message content");
    }
}
```

You can test a service like the one defined below that has a camel binding:

```
<sca:service name="Hello/HelloService" promote="Hello/HelloService">
```

```
    <sca:interface.java interface="org.jboss.example.ExampleService"/>
    <camel_1:binding.uri name="camel1" configURI="direct://HelloService"/>
</sca:service>
```

CHAPTER 14. REMOTE INVOKER

14.1. SWITCHYARD REMOTE INVOKER

The RemoteInvoker serves as a remote invocation client for SwitchYard services. It allows non-SwitchYard applications to invoke any service in SwitchYard which uses a <binding.sca> binding. It is also used by the internal clustering channel to facilitate intra-cluster communication between instances.

14.2. USE A REMOTEINVOKER

Procedure 14.1. Task

- Include a RemoteInvoker and supporting classes in your application with the following Maven dependency:

```
<dependency>
  <groupId>org.switchyard</groupId>
  <artifactId>switchyard-remote</artifactId>
  <version> <!-- SY version goes here (e.g. 1.0) --> </version>
</dependency>
```

Each instance of SwitchYard includes a special context path called switchyard-remote, which is bound to the default HTTP listener in Red Hat JBoss Fuse. The initial version of RemoteInvoker supports communication with this endpoint directly. Here is an example of invoking an in-out service in SwitchYard using the HttpInvoker:

```
public class MyClient {
    public static void main(String[] args) throws Exception {
        RemoteInvoker invoker = new HttpInvoker("http://localhost:8080/switchyard-remote");
        Offer offer = new Offer();
        offer.setAmount(100);
        offer.setItem("honda");

        RemoteMessage msg = invoker.invoke(new RemoteMessage()
            .setContext(new DefaultContext())
            .setService(new QName("urn:com.example.switchyard:remote", "Dealer"))
            .setContent(offer));

        Deal deal = (Deal)msg.getContent();
        System.out.println("It's a deal? " + deal.isAccepted());
    }
}
```

CHAPTER 15. SERIALIZATION

15.1. SERIALIZATION AND DESERIALIZATION IN SWITCHYARD

Serialization is a process of converting an object into a binary format, which can be persisted in a database or transmitted over a network. Deserialization is a process of creating an object from binary format. In SwitchYard, the process of serialization is a concern in case of clustering and execution from a client invoker. In such cases, SwitchYard uses objects that represent the content of a SwitchYard Message or objects that are placed within a SwitchYard Exchange's Context, for serialization and deserialization.

15.2. CUSTOM OBJECTS

Custom (application-defined) objects that are stored as the content of the message or property of the exchange context, are not required to implement **java.io.Serializable**. This is because SwitchYard does not use the JDK's default object serialization mechanism. It traverses the object graph, extracting the properties along the way, and stores the values in its own graph representation, which by default is serialized to/from the JSON format. For this to work, custom objects must follow one of the following two rules:

- Adhere to the JavaBeans specification. This means a public, no arg constructor, and public getter/setter pairs for every property you want serialized. If you follow this rule, your custom objects do not require any compilation dependencies on SwitchYard.
- Use SwitchYard annotations to define your serialization strategy.

15.3. SWITCHYARD ANNOTATIONS FOR SERIALIZATION

The org.switchyard.serial.graph package provides the following annotations, enums, interface and class:

@Strategy

Here you can define the serialization strategy, including access type, coverage type, and factory for your class. All these are optional.

- access=AccessType: BEAN (default) for getter/setter property access, FIELD for member variable access.
- coverage=CoverageType: INCLUSIVE (default) for serializing all properties, EXCLUSIVE for ignoring all properties.
- factory=Factory: Interface for how the class gets instantiated.
  - DefaultFactory: Creates an instance of the class using the default constructor.

@Include

You can place this on individual getter methods or fields to override CoverageType.EXCLUSIVE.

@Exclude

You can place this on individual getter methods or fields to override CoverageType.INCLUSIVE.

15.4. SWITCHYARD SERIALIZATION API USAGE

You can implement Serialization using Serializers from the SerializerFactory. Although it is not recommended to use SwitchYard's serialization API directly, however, if you need to use it, here is an example of Serializer.create invocation:

```
// See SerializerFactory for overloaded "create" methods.
Serializer ser = SerializerFactory.create(FormatType.JSON, CompressionType.GZIP, true);


// to and from a byte array
Foo fooBefore = new Foo();
byte[] bytes = ser.serialize(fooBefore, Foo.class);
Foo fooAfter = ser.deserialize(bytes, Foo.class);


// to and from output/input streams
Bar barBefore = new Bar();
OutputStream out = ...
ser.serialize(barBefore, Bar.class, out);
InputStream in = ...
Bar barAfter = ser.deserialize(in, Bar.class);
```

Out of the box, the available FormatTypes are SER_OBJECT, XML_BEAN and JSON, and the available CompressionTypes are GZIP, ZIP, or null (for no compression).

CHAPTER 16. CONTEXT MAPPING

16.1. CONTEXT MAPPER

A ContextMapper moves native binding message headers and/or properties to and from SwitchYard's canonical context. The default ContextMapper implementations are used by their associated bindings but can be overridden by the user.

16.2. CREATE A CUSTOM CONTEXT MAPPER

Procedure 16.1. Create a Custom Context Mapper

1. Implement the **org.switchyard.component.common.composer.ContextMapper** interface:

```
public interface ContextMapper<T> {
    void mapFrom(T source, Context context) throws Exception;
    void mapTo(Context context, T target) throws Exception;
}
```

2. Specify your implementation in your **switchyard.xml** file:

```
<binding.xyz ...>
    <contextMapper class="com.example.MyContextMapper"/>
</binding.xyz>
```

3. Alternatively, you can implement the **org.switchyard.component.common.composer.RegexContextMapper** interface, which adds regular expression support:

```
public interface RegexContextMapper<T> extends ContextMapper<T> {
    ContextMapper<T> setIncludes(String includes);
    ContextMapper<T> setExcludes(String excludes);
    ContextMapper<T> setIncludeNamespaces(String includeNamespaces);
    ContextMapper<T> setExcludeNamespaces(String excludeNamespaces);
    boolean matches(String name);
    boolean matches(QName qname);
}
```

16.3. CUSTOM CONTEXT MAPPER PROPERTIES

- Your **mapFrom()** method needs to map a source native message's properties to the SwitchYard Message's Context.
- Your **mapTo()** method needs to map a SwitchYard Message's Context properties into the target native message.
- If you extend **BaseContextMapper**, these methods are canceled with "no-op" implementations so you only have to implement what you wish.

If you are adding regular expression, support, note the following:

- The **setIncludes()**, **setExcludes()**, **setIncludeNamespaces()** and **setExcludeNamespaces()** methods are just bean properties. The **matches()** methods use those bean properties to determine if the specified name or qualified name passes the collective regular expressions.

- If you extend the **BaseRegexContextMapper**, all of these are implemented for you. Then, in your implementation's mapFrom and mapTo methods, you only need to first check if the property matches before you map it.

Also for **RegexContextMapper**, the following additional attributes of the <contextMapper/> element are used:

includes
: This indicates which context property names to include.

excludes
: This indicates which context property names to exclude.

includeNamespaces
: This indicates which context property namespaces to include (if the property name is a qualified name).

excludeNamespaces
: This indicates which context property namespaces to exclude (if the property name is a qualified name).

16.4. CONTEXT MAPPER IMPLEMENTATIONS AVAILABLE OUT-OF-THE-BOX

The following four out-of-the-box implementations extend **BaseRegexContextMapper**, thus each of them can be configured to use the additional regular expression attributes.

SOAPContextMapper
: When processing an incoming SOAPMessage, the SOAPContextMapper takes the MIME (in most cases, HTTP) headers from a SOAP envelope and maps them into the SwitchYard Context as Scope.IN properties with the SOAPComposition.SOAP_MESSAGE_MIME_HEADER label, and takes the SOAP header elements from the soap envelope and maps them into the SwitchYard Context as Scope.EXCHANGE properties with the SOAPComposition.SOAP_MESSAGE_HEADER label. When processing an outgoing SOAPMessage, it takes the SwitchYard Scope.OUT Context properties and maps them into mime (in most cases, HTTP) headers, and takes the SwitchYard Scope.EXCHANGE Context properties and maps them into the SOAP envelope as soap header elements.

  You can map these properties with a namespace to the soap header.

```
response.getContext().setProperty("{urn:test:1.0}soapProperty", "soapPropertyValue",
Scope.EXCHANGE);
```

NOTE

The SOAPContextMapper has an additional attribute that the other OOTB ContextMappers do not use, namely soapHeadersType:

```
<binding.soap>
   <contextMapper includes=".*" soapHeadersType="VALUE"/>
</binding.soap>
```

The value of soapHeadersType can be CONFIG, DOM, VALUE or XML (and correspond to the enum SOAPHeadersType.CONFIG, DOM, VALUE or XML). With CONFIG, each soap header element is mapped into an org.switchyard.config.Configuration object, with DOM, each soap header element is left as is (a DOM element), with VALUE, only the String value of each soap header element is stored, and with XML, each soap header element is transformed into an XML String.

CamelContextMapper

When processing an incoming CamelMessage, the CamelContextMapper takes the CamelMessage headers and maps them into the SwitchYard Context as Scope.IN properties with the CamelComposition.CAMEL_MESSAGE_HEADER label, and takes the Camel Exchange properties and maps them into the SwitchYardContext as Scope.EXCHANGE properties with the CamelComposition.CAMEL_EXCHANGE_PROPERTY label. When processing an outgoing CamelMessage, it takes the SwitchYard Scope.OUT Context properties and maps them into the CamelMessage as headers, and takes the SwitchYard Scope.EXCHANGE Context properties and maps them into the Camel Exchange as properties.

HttpContextMapper

When processing an incoming HTTP request, the HttpContextMapper takes the incoming request headers and maps them into the SwitchYard Context as Scope.IN with the HttpComposition.HTTTP_HEADER label. When processing an outgoing HTTP response, it takes the SwitchYard Scope.OUT Context properties and maps them into the response headers.

RESTEasyContextMapper

When processing an incoming HTTP request, the RESTEasyContextMapper takes the incoming request headers and maps them into the SwitchYard Context as Scope.IN with the RESTEasyComposition.HTTP_HEADER label. When processing an outgoing HTTP response, it takes the SwitchYard Scope.OUT Context properties and maps them into the response headers.

The JCA Component provides three different **ContextMappers**:

IndexedRecordContextMapper

When processing an incoming IndexedRecord, the IndexedRecordContextMapper takes the record name and record short description and maps them into the SwitchYardContext as Scope.EXCHANGE properties with the JCAComposition.JCA_MESSAGE_PROPERTY label. When processing an outgoing IndexedRecord, it looks for those properties specifically in the SwitchYard.EXCHANGE Context properties by key and sets them on the IndexedRecord.

MappedRecordContextMapper

When processing an incoming MappedRecord, the MappedRecordContextMapper takes the record name and record short description and maps them into the SwitchYardContext as Scope.EXCHANGE properties with the JCAComposition.JCA_MESSAGE_PROPERTY label. When processing an outgoing MappedRecord, it looks for those properties specifically in the SwitchYard.EXCHANGE Context properties by key and sets them on the MappedRecord.

JMSContextMapper

When processing an incoming (JMS) Message, the JMSContextMapper takes the Message properties and maps them into the SwitchYardContext as Scope.EXCHANGE properties with the JCAComposition.JCA_MESSAGE_PROPERTY label. When processing an outgoing (JMS) Message, it takes the SwitchYard.EXCHANGE Context properties and maps them into the Message as Object properties.

CHAPTER 17. AUDITING

17.1. SWITCHYARD AUDITING

SwitchYard possesses auditing functionality. It traces exchanges through their various mediation states. The auditing functionality requires the CDI environment (the **META-INF/beans.xml** file) to run. The auditing functionality also works in a test environment.

17.2. ENABLE CUSTOM AUDITORS

Audit mechanism requires CDI runtime environment to run.

To enable custom auditors, define the Auditor implementations with the **@Named** annotation. It helps Apache Camel component to recognize all the auditor implementations. Camel Exchange Bus, a default implementation used by SwitchYard, look up for bean definitions with **@Audit** annotation.

```
@Audit
@Named("custom auditor")
public class SimpleAuditor implements Auditor
{
@Override
public void beforeCall(Processors processor, Exchange exchange)
{
System.out.println("Before " + processor.name());
}

@Override
public void afterCall(Processors processor, Exchange exchange)
{
System.out.println("After " + processor.name());
}

}
```

NOTE

Do not include any state inside the Custom Auditor's field. Red Hat recommends you to use exchange properties or message headers to store values.

17.3. MEDIATION STATE

Mediation state is the term used to described the interim states a SwitchYard exchange goes through as it is sent from a service consumer to a service provider.

17.4. LIST OF MEDIATION STATES

Domain handlers

In this state all the handlers defined in **switchyard.xml** are executed. This is an early phase of mediation where you can either implement own logic or choose the service provider logic to use.

Addressing

If this is not specified by the domain handlers then the addressing handler will determine what to do by using the consumer contract.

Transaction

If the service is required to run a transaction this handler starts it.

Security

This state verifies constraints related to authentication and authorization.

General policy

This executes checks other than those for security and transactions.

Validation

This executes custom validators.

Transformation

This prepares the payload by calling a provider.

Validation (2)

This validates the transformed payload.

Provider call

This calls the provisional service.

Transaction (2)

This commits or, if necessary, rolls back the transaction.

If the service consumer is synchronous and the exchange pattern is set to in-out, then some of these handlers may be called once again:

Domain handlers

These are called when a response is generated by a provider service.

Validation

This verifies the output generated by the provider.

Transformation

This converts the payload to the structure required by the consumer.

Validation

This checks the output after the transformation has occurred.

Consumer callback

This returns the exchange to the service consumer.

17.5. CREATE A CUSTOM AUDITOR

Prerequisities

- CDI Runtime
- Annotate your auditor implementations with @Named in order to have Camel recognize them.

NOTE

The Camel Exchange Bus looks for bean definitions with the @Audit annotation.

Here is code that shows what a very simple auditor would look like:

```
@Audit
@Named("custom auditor")
public class SimpleAuditor implements Auditor {

    @Override
    public void beforeCall(Processors processor, Exchange exchange) {
        System.out.println("Before " + processor.name());
    }

    @Override
    public void afterCall(Processors processor, Exchange exchange) {
        System.out.println("After " + processor.name());
    }

}
```

IMPORTANT

Be aware that the afterCall method is not called if the step it surrounds throws an exception. If this happens, afterCall will be skipped.

Result

You can see many statements like 'Before DOMAIN_HANDLERS' and 'Before ADDRESSING' appearing in the server console. This is because every step of mediation is surrounded by this SimpleAuditor class.

17.6. DETERMINE LOCATION FOR AUDIT ASSIGNMENT

In SwitchYard, you may choose to provide an argument to the @Audit annotation. The accepted values for this comes from org.switchyard.bus.camel.processors.Processors enumeration. For example, the following combination can handle only validation occurrences:

```
@Audit(Processors.VALIDATION)
```

Note the following important facts about validation, transformation, and transaction in SwitchYard:

- The validation is executed twice for in-only exchanges and four times for in-out exchanges.

- The validation occurs before and after transformation of inbound messages.
- When SwitchYard sends outgoing messages, the validation occurs before and after transformation of outbound messages.
- Transformation is executed once for in-only exchanges and twice for in-out exchanges.
- Transaction phase is always executed twice.

If you want to implement only one execution of your auditor, use the following combination:

```
@Audit(Processors.PROVIDER_CALLBACK).
```

Here, the auditor is executed just before sending exchange to service implementation. You can also implement one auditor instance with few mediation steps. For example, a bean with annotation following:

```
@Audit({Processors.PROVIDER_CALLBACK, Processors.CONSUMER_CALLBACK})
```

This bean is executed twice. One pair of before or after call for provider service and second pair for outgoing response.

17.7. USE EXCHANGE PROPERTIES

As only one instance of auditor is created by default and there is no guarantee for dispatching order, ensure that the custom auditors do not preserve state inside any of the fields. If you want to store values, use exchange properties or message headers. Here is an example of how to count processing time using Exchange properties as temporary storage:

```
@Named("custom auditor")
public class SimpleAuditor implements Auditor {

    private Logger _logger = Logger.getLogger(SimpleAuditor.class);

    @Override
    public void beforeCall(Processors processor, Exchange exchange) {
        exchange.setProperty("time", System.currentTimeMillis());
    }

    @Override
    public void afterCall(Processors processor, Exchange exchange) {
        long time = System.currentTimeMillis() - exchange.getProperty("time", 0, Long.class);
        _logger.info("Step " + processor.name() + " took " + time + "ms");
    }

}
```

CHAPTER 18. EXTENSIONS

An extension is a module that extends the core capabilities of JBoss EAP.

You can extend SwitchYard functionality by creating extension modules for JBoss EAP. Use this feature to:

- Support additional binding types through Camel components not included in the distribution.
- Add data formats and other Camel libraries for use within Camel routing services.
- Implement custom gateway bindings as Camel components.

18.1. CREATE SWITCHYARD EXTENSION MODULE IN JBOSS EAP

Procedure 18.1. Create, Register and Build a SwitchYard Extension Module in JBoss EAP

1. An extension is deployed in the **modules** folder of JBoss EAP.

   Extension modules are placed under the **modules/system/layers/soa/org/** directory of your JBoss EAP installation. The jar files that make up the module are placed in the **modules/system/layers/soa/org/*product*/*subsystem*/*modulename*/main** directory. The **module.xml** file contains definition information. For example, in directory **modules/system/layers/soa/org/apache/camel/saxon/main**, the **module.xml** file looks like this:

   ```
   <?xml version="1.0" encoding="UTF-8"?>
    <module xmlns="urn:jboss:module:1.0" name="org.apache.camel.saxon">

      <resources>
         <resource-root path="camel-saxon-2.10.0.redhat-60024.jar"/>
         <resource-root path="saxon9he-9.3.0.11.jar"/>
      </resources>

      <dependencies>
         <module name="javax.api"/>
         <module name="org.slf4j"/>
         <module name="org.apache.camel.core"/>
      </dependencies>

   </module>
   ```

   The **module.xml** file contains the following information:

   - The module name. The module name is comprised of the directory names for the module underneath the ***EAP-Home*/modules/system/layers/soa** directory.
   - Resources required. Notice that the files mentioned in the **<resources>** section are in the same directory as the **module.xml** file.
   - Dependencies for the module.

2. In order to make SwitchYard aware of the extension module, add the module name to the list of extensions defined in the SwitchYard subsystem in **standalone.xml**:

```
<subsystem xmlns="urn:jboss:domain:switchyard:1.0">
  <modules>
    <module identifier="org.switchyard.component.bean"
implClass="org.switchyard.component.bean.deploy.BeanComponent"/>
    <module identifier="org.switchyard.component.soap"
implClass="org.switchyard.component.soap.deploy.SOAPComponent">
    <!-- snip -->
  </modules>
  <extensions>
    <extension identifier="org.apache.camel.mvel"/>
    <extension identifier="org.apache.camel.ognl"/>
    <extension identifier="org.apache.camel.jaxb"/>
    <extension identifier="org.apache.camel.soap"/>
    <extension identifier="org.apache.camel.saxon"/>
  </extensions>
</subsystem>
```

3. Build the application that will be using the module using the **mvn clean install** command. If there are problems running the JVM tests step locally, use the **-DskipTests** argument.

   Update the **pom.xml** file for the application. Add the module as a dependency and mark it as **provided**.

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-saxon</artifactId>
    <version>2.10.0.redhat-60024</version>
    <scope>provided</scope>
</dependency>
```

4. If the module is created from a jar not supplied with JBoss Fuse Service Works, you must ensure that a **jboss-deployment-structure.xml** file exists in the ***application-name*/src/main/resources/META-INF** folder of the application that will run it. The **jboss-deployment-structure.xml** file contains information in the following format:

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss-deployment-structure>
    <deployment>
        <dependencies>
            <module name="org.apache.camel.saxon" services="import" export="true">
                <imports>
                    <include path="META-INF/services/org/apache/camel/component" />
                    <include path="META-INF/services/org/apache/camel/language" />
                </imports>
                <exports>
                    <include path="META-INF/services/org/apache/camel/component" />
                    <include path="META-INF/services/org/apache/camel/language" />
                </exports>
            </module>
        </dependencies>
    </deployment>
</jboss-deployment-structure>
```

   The module name must be the same as the module name defined in step 1 with the same naming rules.

For an example of an extension module and an application, see https://access.redhat.com/solutions/653823.

18.2. EXTENSION TYPES AND USAGE

How you use an extension in your application depends on the type of extension.

Table 18.1. Extensions

Type	Usage
Camel DataFormat	Once a data format is packaged as a module and added as an extension to the SwitchYard subsystem, no further configuration is required in your application. You can refer to the data format directly in your route as if it was packaged directly within your application.
Camel Gateway Components	If the extension is to be used as a gateway binding for services or references, then use the Camel URI binding in your application to configure the endpoint details.

CHAPTER 19. REFERENCE

19.1. CONFIGURATION DESCRIPTORS

Each SwitchYard application must include a configuration descriptor named **switchyard.xml** in the **/META-INF** directory of its archive. The basic structure of this descriptor is:

- A parent *<switchyard>* element, which contains all other configuration.
- One child *<composite>* element, which contains the SCA description of the application.
- No more than one *<transforms>* element, which can contain one or more transform definitions.
- No more than one *<validates>* element, which can contain one or more validate definitions.

19.2. DESCRIPTOR CONFIGURATION EXAMPLE

Here's an example of what a SwitchYard descriptor looks like:

```
<switchyard xmlns="urn:switchyard-config:switchyard:1.0"
            xmlns:sca="http://docs.oasis-open.org/ns/opencsa/sca/200912"
            xmlns:bean="urn:switchyard-component-bean:config:1.0"
            xmlns:soap="urn:switchyard-component-soap:config:1.0">
    <sca:composite name="orders" targetNamespace="urn:switchyard-quickstart-demo:orders:0.1.0">
        <sca:service name="OrderService" promote="OrderService">
            <soap:binding.soap>
                <soap:socketAddr>:18001</soap:socketAddr>
                <soap:wsdl>wsdl/OrderService.wsdl</soap:wsdl>
            </soap:binding.soap>
        </sca:service>
        <sca:component name="InventoryService">
            <bean:implementation.bean
class="org.switchyard.quickstarts.demos.orders.InventoryServiceBean"/>
            <sca:service name="InventoryService">
                <sca:interface.java interface="org.switchyard.quickstarts.demos.orders.InventoryService"/>
            </sca:service>
        </sca:component>
        <sca:component name="OrderService">
            <bean:implementation.bean
class="org.switchyard.quickstarts.demos.orders.OrderServiceBean"/>
            <sca:service name="OrderService">
                <sca:interface.java interface="org.switchyard.quickstarts.demos.orders.OrderService"/>
            </sca:service>
            <sca:reference name="InventoryService">
                <sca:interface.java interface="org.switchyard.quickstarts.demos.orders.InventoryService"/>
            </sca:reference>
        </sca:component>
    </sca:composite>
    <transforms xmlns="urn:switchyard-config:transform:1.0">
        <transform.java bean="Transformers"
                        from="{urn:switchyard-quickstart-demo:orders:1.0}submitOrder"
                        to="java:org.switchyard.quickstarts.demos.orders.Order"/>
        <transform.java bean="Transformers"
                        from="java:org.switchyard.quickstarts.demos.orders.OrderAck"
```

```
                    to="{urn:switchyard-quickstart-demo:orders:1.0}submitOrderResponse"/>
    </transforms>
    <validates>
        <validate.java bean="Validators"
                    name="java:org.switchyard.quickstarts.demos.orders.Order"/>
        <validate.java bean="Validators"
                    name="java:org.switchyard.quickstarts.demos.orders.OrderAck"/>
    </validates>
</switchyard>
```

NOTE

Do not edit the **SwitchYard.xml** file directly. Red Hat recommends using the JBoss Developer Studio SwitchYard Editor to edit the **SwitchYard.xml** file.

19.3. COMPOSITE

The composition of a SwitchYard application is defined using the Service Component Architecture Assembly Model, an open specification undergoing standardization in OASIS. Through this, a number of elements can be added to a service.

19.4. COMPOSITE ELEMENTS

Table 19.1. Composite Elements

Name	Description
<composite>	Single, top-level application element which defines a set of services, the relationships and dependencies of those services, and the linkage (if any) to services available outside the application.
<component>	Contains the implementation of a service and the dependencies of that service.
<service>	Defines the name and interface for a service. This element is used inside a component definition to declare a service and can also appear inside a composite definition to indicate that a service is visible to outside applications.
<interface.XXX>	The contract for a service. The type of the interface replaces 'XXX' in the definition. Supported interface types are Java and WSDL.
<binding.XXX>	The binding of a service. The type of the binding replaces 'XXX' in the definition. Example bindings include binding.soap and binding.camel.
<implementation.XXX>	The implementation of a service. The type of the implementation replaces 'XXX' in the definition. Example implementations include implementation.bean and implementation.camel.

19.5. THE TRANSFORMS DEFINITION

The <transforms> definition is used to define transformers local to your application. Much like interfaces, bindings, and implementations, each transformer definition includes a type in its definition (e.g. transform.java, transform.smooks). The *from* and *to* definitions on each transformer correspond to the message type used on a service and/or reference interface used within SwitchYard.

19.6. THE VALIDATES DEFINITION

The <validates> definition is used to set validators locally to your application. Similar to transformers, each validator definition includes a type in its definition (e.g. validate.java, validate.xml). The name definition on each validator corresponds to the message type used on a service and/or reference interface used within SwitchYard.

19.7. GENERATED CONFIGURATION

SwitchYard is capable of generating configurations from annotations in an application code. This negates the need to edit the XML configuration manually. The generated configuration is packaged with the application as part of the Maven build life cycle. It is located in **target/classes/META-INF/switchyard.xml**. (You may also place a descriptor in **src/main/resources/META-INF/switchyard.xml**.)

This version of the configuration can be edited by the user directly and is also used by Forge tooling to specify configuration in response to SwitchYard forge commands. During the build process, the user-editable **switchyard.xml** is merged with any generated configuration to produce the final **switchyard.xml** in the **target/** directory.

19.8. SERVICE OPERATIONS

All SwitchYard services, no matter their implementation type, are composed of one or more *service operations*. In the case of a Bean service, the service operations are the set of Java methods exposed by the service interface.

A Bean service operation does the following:

- Declares a maximum of one Input type. The Java method signature must have a maximum of one Java parameter.
- Declares a maximum of one Output type. This is enforced by the Java language. You can only define one return type on a Java method.
- Declares a maximum of one Fault (exception) type.

19.9. SERVICE OPERATION TYPES

All service operations on a SwitchYard service can define *Input*, *Output* and *Fault* messages. These messages have a type associated with them, which is defined as a *QName*. This type is used by the data transformation layer, when trying to work out which transformers to apply to a message payload.

For Bean services, the default type QName for Input (input param), Output (return value) and Fault (Exception) are derived from the Java class name in each case (param, return, throws). For some types however (such as org.w3c.dom.Element), the Java type name alone does not tell you the real type of the data being held by that Java Object instance. For this reason, Bean service operations (methods) can be annotated with the @OperationTypes annotation.

19.10. THE @DEFAULTTYPE ANNOTATION

You can set the default data type for a Java type using the *@DefaultType* type level annotation. This is useful for setting the type for a hierarchy of Java types, as shown below:

```
public interface OrderService {

    @OperationTypes(
        in = "{http://acme.com/orders}createOrder",
        out = "{http://acme.com/orders}createOrderResult",
        fault = "java:com.acme.exceptions.OrderManagementException"
    )
    Element createOrder(Element order) throws OrderCreateFailureException;

}
```

In this example, the type for OrderCreateFailureException has been changed to *java:com.acme.exceptions.OrderManagementException* by defining a fault type on the @OperationTypes. Its type would otherwise default to *java:com.acme.exceptions.OrderCreateFailureException*. It could also be done by annotating the base OrderManagementException class with the @DefaultType annotation. This would set the default type for the OrderManagementException class and all its sub-classes, including OrderCreateFailureException, which would mean not having to defining a fault type on the @OperationTypes wherever one of these exceptions is used on a Bean Service Operation.

19.11. BEAN SERVICES IN A JAVA EE WEB APPLICATION CONTAINER

The JEE Web Application Container can be used to deploy applications (for example, Tomcat or Jetty). This means that you do not have to use the SwitchYard Application Servers for deployment.You can use it to configure the SwitchYard WebApplicationDeployer servlet listener in your web application, and configure Weld into a servlet container.

19.12. JAVASERVER FACES

The JavaServer Faces (JSF) technology provides a server-side component framework that is designed to simplify the development of user interfaces (UIs) for Java EE applications. JSF integrates with CDI to provide the Object Model behind the JSF user interface components. The fact that SwitchYard Bean Services are based on CDI means it's possible to have a smooth integration between SwitchYard Bean services and a JSF-based user interface.

19.13. INDIRECT SERVICE INVOCATION

SwitchYard services can be indirectly invoked through the SwitchYard Exchange mechanism. This reduces the coupling between JSF components and SwitchYard Service implementations. This is done by invoking a SwitchYard CDI Bean Service and the SwitchYard Exchange mechanism through a @Reference injected Service reference. This provides a client side proxy bean that handles all the SwitchYard Exchange invocation details for all the operations exposed by the service.

NOTE

The proxy beans for these invocations are not available (through CDI) to the JSF components. To work around this, users can create a standard named (@Named) CDI bean. It must contain a @Reference between the JSF components and the SwitchYard CDI Bean services.

19.14. INDIRECT SERVICE INVOCATION EXAMPLE

This invocation uses an **OrderService** example:

```
public interface OrderService {

    OrderAck submitOrder(Order order);

}

public class Order implements Serializable {

    private String orderId;
    private String itemId;
    private int quantity = 1;

    public String getOrderId() {
        return orderId;
    }

    public void setOrderId(String orderId) {
        this.orderId = orderId;
    }

    public String getItemId() {
        return itemId;
    }

    public void setItemId(String itemId) {
        this.itemId = itemId;
    }

    public int getQuantity() {
        return quantity;
    }

    public void setQuantity(int quantity) {
        this.quantity = quantity;
    }
}
```

19.15. INDIRECT SERVICE INVOCATION WITH JSF COMPONENTS EXAMPLE

JSF components have a more fluid interaction with the Order bean than with the **OrderService**:

```
<div id="content">
```

```
    <h1>New Order</h1>

    <div style="color: red">
       <h:messages id="messages" globalOnly="false" />
    </div>

    <h:form id="newOrder">
        <div>
            Order ID:
            <h:inputText id="orderID" value="#{order.orderId}" required="true"/>
            <br/>
            Item ID:
            <h:inputText id="itemID" value="#{order.itemId}" required="true"/>
            <br/>
            Quantity:
            <h:inputText id="quantity" value="#{order.quantity}" required="true"/>
            <p/>
            <h:commandButton id="createOrder" value="Create" action="#{order.create}"/>
        </div>
    </h:form>

</div>
```

19.16. INDIRECT SERVICE INVOCATION WITH ANNOTATIONS

In the example below, annotations have been added to the bean called **Order** to manage invocation:

```
@Named
@RequestScoped
public class Order implements Serializable {

    @Inject
    @Reference
    private OrderService orderService;

    private String orderId;
    private String itemId;
    private int quantity = 1;

    public String getOrderId() {
        return orderId;
    }

    public void setOrderId(String orderId) {
        this.orderId = orderId;
    }

    public String getItemId() {
        return itemId;
    }

    public void setItemId(String itemId) {
        this.itemId = itemId;
    }
    public int getQuantity() {
        return quantity;
    }

    public void setQuantity(int quantity) {
        this.quantity = quantity;
    }

    public void create() {
        OrderAck serviceAck = orderService.submitOrder(this);
        FacesContext.getCurrentInstance().addMessage(null, new
FacesMessage(serviceAck.toString()));
    }
}
```

- Annotations @Named and @RequestScoped have been added.
- The **OrderService** property has a @Reference annotation. Because of this, its instance is not a reference to the actual service implementation. Instead, it is a SwitchYard Exchange proxy to that service implementation. By using @Reference injected service references, backend service implementations can be non-CDI Bean service implementations.
- Implementing the create method invokes the **OrderService** reference (exchange proxy).

19.17. CORE API ANNOTATIONS

Table 19.2. Core API Annotations

Feature	Feature's Annotations
Policy	@Requires
Transformation	@Transformer
Validation	@Validator
Service Interface	@OperationTypes
Serialization	@Strategy, @Include, @Exclude

19.18. COMPONENT ANNOTATIONS

Table 19.3. Component Annotations

Component	Annotations
Bean	@DefaultType, @Inject, @Service, @Reference, @Property
Camel	@Route

Component	Annotations
Knowledge Services	@Channel, @Listener, @Logger, @Manifest, @Container, @Resource, @Property
BPM	@BPM, @WorkItemHandler, @StartProcess, @SignalEvent, @AbortProcessInstance
Rules	@Rules, @Execute, @Insert, @FireAllRules, @FireUntilHalt

19.19. TESTING ANNOTATIONS

Table 19.4. Testing Annotations

Action	Annotations
Testing	@RunWith, @Test, @SwitchYardTestCaseConfig, @ServiceOperation, @BeforeDeploy

19.20. JBOSS RULES

19.20.1. JBoss Rules

JBoss Rules is the name of the business rule engine provided as part of the Red Hat JBoss Fuse product.

19.20.2. The JBoss Rules Engine

The JBoss Rules engine is the computer program that applies rules and delivers Knowledge Representation and Reasoning (KRR) functionality to the developer.

19.20.3. Production Rules

A *production rule* is a two-part structure that uses first order logic to represent knowledge. It takes the following form:

```
when
   <conditions>
then
   <actions>
```

19.20.4. The Inference Engine

The *inference engine* is the part of the BRMS engine which matches production facts and data to rules. It performs the actions based on what it infers from the information. A production rules system's inference engine is *stateful* and is responsible for *truth maintenance*.

19.20.5. ReteOO

The Rete implementation used in JBoss Rules is called *ReteOO*. It is an enhanced and optimized implementation of the Rete algorithm specifically for object-oriented systems. The Rete Algorithm has now been deprecated, and PHREAK is an enhancement of Rete. This section merely describes how the Rete Algorithm functions.

19.20.6. Using JBoss Rules

To learn how to use JBoss Rules, please refer to the *BRMS Development Guide*.

19.21. APACHE CAMEL

19.21.1. Apache Camel

Camel is an open source rules-based router developed by the Apache Project.

19.21.2. Using Apache Camel

To learn how to use Apache Camel, refer to the Red Hat JBoss Fuse *Web Services and Routing with Camel CXF Guide*.

19.21.3. Using Camel Routes Directly

You can deploy Camel routes directly in your user application, without needing to use SwitchYard. For example, you can deploy the Camel routes as a WAR file, fronted by a servlet or CXF.

19.21.4. Supported Camel Components

The following Apache Camel components are supported by JBoss Fuse:

- camel-amqp
- camel-atom
- camel-atom
- camel-bindy
- camel-cdi
- camel-cxf
- camel-dozer
- camel-file
- camel-ftp
- camel-hl7
- camel-http
- camel-jaxb

- camel-jms
- camel-jpa
- camel-mail
- camel-mqtt
- camel-netty
- camel-quartz
- camel-rss
- camel-sap
- camel-saxon
- camel-sql

19.21.5. Running Camel Examples

The Camel CXF (code first) example uses code-first to expose a web service in Camel running on JBoss EAP. It can be run using Maven.

Procedure 19.1. Task

1. To build the example, navigate to its directory and run the following command:

   ```
   mvn clean install
   ```

2. To run the example, deploy it in JBoss EAP by copying the **camel-example-cxf-tomcat.war** located in the target directory to the **standalone/deployments/** folder.

   The webservice is available in *http://localhost:8080/camel-example-cxf-tomcat/webservices/incident?wsdl*.

19.21.6. Sample Client With Camel CXF

This is what a sample client made with Camel CXF looks like:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
   <soap:Body>
      <ns1:reportIncident xmlns:ns1="http://incident.cxf.example.camel.apache.org/">
         <arg0>
            <details>Accounting Department</details>
            <email>johncitizen@acompany.com</email>
            <familyName>Citizen</familyName>
            <givenName>John</givenName>
            <incidentId>12345</incidentId>
            <summary>Incident</summary>
         </arg0>
      </ns1:reportIncident>
   </soap:Body>
</soap:Envelope>
```

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
    <soap:Body>
        <ns1:statusIncident xmlns:ns1="http://incident.cxf.example.camel.apache.org/">
            <arg0>
                <incidentId>12345</incidentId>
            </arg0>
        </ns1:statusIncident>
    </soap:Body>
</soap:Envelope>
```

19.21.7. Camel Servlet Without Spring Example

Procedure 19.2. Task

1. To build the example, navigate to the Apache Tomcat directory and enter the following command:

   ```
   mvn package
   ```

2. To deploy the example, copy the resulting **.war** file into Apache Tomcat's **Deploy** folder.

3. To access the example, open your browser and navigate to http://localhost:8080/camel-example-servlet-tomcat-no-spring-2.10.0

4. To access the servlet, go to http://localhost:8080/camel-example-servlet-tomcat-no-spring-2.10.0/camel

CHAPTER 20. DEBUGGING

20.1. MESSAGE TRACING

Message tracing provides a view of the content and context of a message exchange on the SwitchYard bus. Message tracing prints exchange information to the log. An exchange interceptor that generates the trace, is triggered at the following points:

- Immediately after the consumer sends the request message. For example, In the case of a service which is invoked from a service binding, this is the point at which the gateway binding puts the message onto the bus.
- Immediately before the service provider is invoked.
- Immediately after the service provider is invoked.
- At completion of the exchange before the message is returned to the consumer.

20.1.1. Trace Output Example

Trace output includes details on the metadata, context properties, payload, and attachments for a message exchange. Here is an example of a trace entry:

```
12:48:25,038 INFO  [org.switchyard.handlers.MessageTraceHandler]
------- Begin Message Trace -------
Consumer -> {urn:switchyard-quickstart:bean-service:0.1.0}OrderService
Provider -> [unassigned]
Operation -> submitOrder
MEP -> IN_OUT
Phase -> IN
State -> OK
Exchange Context ->
    org.switchyard.bus.camel.consumer : ServiceReference [name={urn:switchyard-quickstart:bean-
service:0.1.0}OrderService, interface=BaseServiceInterface [type=wsdl, operations=[submitOrder :
IN_OUT : [{urn:switchyard-quickstart:bean-service:1.0}submitOrder, {urn:switchyard-quickstart:bean-
service:1.0}submitOrderResponse, null]]], domain=ServiceDomain
[name=org.switchyard.domains.root]]
    org.switchyard.exchangeGatewayName : _OrderService_soap_1
    org.switchyard.bus.camel.securityContext : SecurityContext[credentials=[ConfidentialityCredential
[confidential=false]], securityDomainsToSubjects={}]
    org.switchyard.exchangeInitiatedNS : 1375980505021790000
    CamelCreatedTimestamp : Thu Aug 08 12:48:25 EDT 2013
    org.switchyard.bus.camel.phase : IN
    org.switchyard.bus.camel.dispatcher : org.switchyard.bus.camel.ExchangeDispatcher@b4aa453
    org.switchyard.bus.camel.labels : {org.switchyard.exchangeGatewayName=
[org.switchyard.label.behavior.transient]}
    CamelToEndpoint : direct://%7Burn:switchyard-quickstart:bean-service:0.1.0%7DOrderService
    org.switchyard.bus.camel.contract : org.switchyard.metadata.BaseExchangeContract@2176feaf
    org.switchyard.bus.camel.replyHandler :
org.switchyard.component.common.SynchronousInOutHandler@516a4aef
    CamelFilterMatched : false
Message Context ->
    org.switchyard.bus.camel.labels : {org.switchyard.contentType=
[org.switchyard.label.behavior.transient], org.switchyard.bus.camel.messageSent=
[org.switchyard.label.behavior.transient]}
```

```
    org.switchyard.messageId : ID-kookaburra-local-49858-1375980502093-0-1
    org.switchyard.bus.camel.messageSent : true
    org.switchyard.soap.messageName : submitOrder
    org.switchyard.contentType : {urn:switchyard-quickstart:bean-service:1.0}submitOrder
    breadcrumbId : ID-kookaburra-local-49858-1375980502093-0-1
Message Content ->
<?xml version="1.0" encoding="UTF-8"?><orders:submitOrder xmlns:orders="urn:switchyard-
quickstart:bean-service:1.0">
            <order>
                <orderId>PO-19838-XYZ</orderId>
                <itemId>BUTTER</itemId>
                <quantity>200</quantity>
            </order>
        </orders:submitOrder>
------ End Message Trace -------
```

20.1.2. Enabling Message Tracing

You can enable message tracing within JBoss Developer Studio.

- When using the SwitchYard visual editor to view the **switchyard.xml** file, select the **Domain** tab to view the **Domain Settings**, then select the **Enable Message Trace** check box.

This sets the value of the org.switchyard.handlers.messageTrace.enabled property to **true** in your application domain. This is captured by the <sy:domain> element in the **switchyard.xml** source file.

20.2. EXCHANGE INTERCEPTORS

Exchange Interceptors provide a mechanism for injecting logic into the message path of the SwitchYard exchange bus. You can use an interceptor to read or update message content and context properties, which makes interceptors useful for debugging and for applying logic outside a traditional service implementation in SwitchYard.

20.2.1. Implementing an Exchange Interceptor

The Java class **ExchangeInterceptor** has the following properties:

- Implements the org.switchyard.ExchangeInterceptor interface.
- Is annotated with @Named so that it can be discovered as a CDI bean.

The **ExchangeInterceptor** interface looks like this:

```
public interface ExchangeInterceptor {
    String CONSUMER = "Consumer";
    String PROVIDER = "Provider";

    void before(String target, Exchange exchange) throws HandlerException;
    void after(String target, Exchange exchange) throws HandlerException;
    List<String> getTargets();
}
```

An interceptor is invoked for all message exchanges in an application, so if you only care about a specific service you will want to add a conditional to before() and after() to check for service name. You can restrict the interception points used through the getTargets() method. The CONSUMER and PROVIDER string constants are provided for use with getTargets() to restrict interception to the consumer, provider, or both. The CONSUMER target maps to an injection point just after the consumer sends a request and just before the reply is handed back. The PROVIDER target maps to an injection point just before the provider is called with a request and just after it produces a response.

Here is an example ExchangeInterceptor implementation from the bean-service quickstart:

```
package org.switchyard.quickstarts.bean.service;

import java.util.Arrays;
import java.util.List;

import javax.inject.Named;

import org.switchyard.Exchange;
import org.switchyard.ExchangeInterceptor;
import org.switchyard.ExchangeState;
import org.switchyard.HandlerException;

/**
 * This is an example of an exchange interceptor which can be used to inject code
 * around targets during a message exchange.  This example updates the content of
 * OrderAck after the provider has generated a response.
 */
@Named("UpdateStatus")
public class OrderInterceptor implements ExchangeInterceptor {

    @Override
    public void before(String target, Exchange exchange) throws HandlerException {
        // Not interested in doing anything before the provider is invoked
    }

    @Override
    public void after(String target, Exchange exchange) throws HandlerException {
        // We only want to intercept successful replies from OrderService
        if (exchange.getProvider().getName().getLocalPart().equals("OrderService")
                && ExchangeState.OK.equals(exchange.getState())) {
            OrderAck orderAck = exchange.getMessage().getContent(OrderAck.class);
            orderAck.setStatus(orderAck.getStatus() + " [intercepted]");
        }
    }

    @Override
    public List<String> getTargets() {
       return Arrays.asList(PROVIDER);
    }

}
```