Red Hat JBoss Fuse 6.3

Security on JBoss EAP

Security Guide for JBoss Fuse on JBoss EAP

Last Updated: 2020-10-27

Red Hat JBoss Fuse 6.3 Security on JBoss EAP

Security Guide for JBoss Fuse on JBoss EAP

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

Legal Notice

Abstract

Use this guide as a guide to securing JBoss Fuse on JBoss EAP

Table of Contents

CHAPTER 1. SWITCHYARD SECURITY

SwitchYard services can be secured by:

- Specifying a list of security policies that are required for that service.
- Configuring application-level security processing details for the services within a domain.
- Configuring system-level security processing details.
- Storing sensitive information, such as passwords, in the JBoss AS password vault.

For information on SAML (Security Assertion Markup Language) and Java Security Manager, refer JBoss Enterprise Application Platform 6.1.1 Security Guide.

See Also:

- Section 6.6, “Security Policy”
- Section 1.3, “SwitchYard Security Configuration”
- Section 1.7, “Login Modules”

1.1. ABOUT SWITCHYARD SECURITY

SOA architecture involves applications to be exposed as services. These services must be protected against security vulnerabilities such as a SQL injection attack, XML entity expansion, and denial of service attack. The security implementation covers these security concerns and also provides the ability to monitor usage of services in SOA. However, you need to address the security concerns as an application developer if you are building your application on top of the product. For more information on such security concerns, refer https://www.owasp.org/index.php/Top_10_2013-Top_10.

SwitchYard services are secured in the following ways:

- Specify a list of security policies that are required for that service in the SwitchYard application descriptor (**switchyard.xml**). Edit the **switchyard.xml** file using the SwitchYard editor plug-in and specify the security policy by using the requires attribute of a component service definition as shown below:

  ```
  <service name="WorkService" requires="authorization clientAuthentication confidentiality">
  ```

- You can configure the security processing details for the services within a domain in the following ways:
  - Select the Service for a component and view the **Properties View** in the SwitchYard editor.
  - Hover over the Service for a component. A list of tools including the **Property Sheet** appears. It contains the security information.
- Ensure **Authorization**, **Client Authentication** and **Confidentiality** are checked.

This guide provides information on Red Hat JBoss Fuse security. For information on the security of underlying application platform, refer JBoss Enterprise Application Platform 6.1.1 Security Guide.

1.2. CONFIGURING SECURITY FOR SWITCHYARD

All services within a domain can define or share their own security configuration, which is specified in **META-INF/switchyard.xml** file.

```
<sy:switchyard>
    <sca:composite ...>
        <component ...>
            ...
            <service ... sy:security="security-name">
            ...
            </service>
            <reference ... sy:security="default">
            ...
            </reference>
        </component>
    <sca:composite>
    <domain>
        <securities>
            <security callbackHandler="callback-handler-class-name" name="security-name"
rolesAllowed="users, administrators" runAs="leaders" securityDomain="jaas-domain-name">
                <properties>
                    <property name="property-name" value="property-value"/>
                </properties>
            </security>
        </securities>
    </domain>
</sy:switchyard>
```

1.3. SWITCHYARD SECURITY CONFIGURATION

All services within a domain share the same security configuration, which is specified in **META-INF/switchyard.xml**:

```
<switchyard>
    <domain>
        <security callbackHandler="callback-handler-class-name" moduleName="jaas-domain-name"
rolesAllowed="users, administrators" runAs="leaders">
            <properties>
                <property name="property-name" value="property-value"/>
            </properties>
        </security>
    </domain>
</switchyard>
```

The <security> element

This is an optional element. If not specified, the *callbackHandler* and *moduleName* attributes described below will fallback to their default values.

The callbackHandler attribute

This is an optional attribute. If not specified, a default value of **org.switchyard.security.callback.NamePasswordCallbackHandler** will be used.

The moduleName attribute

This is an optional attribute. If not specified, a default value of other will be used. The value maps to a JAAS security domain name.

The rolesAllowed attribute

This is an optional attribute. If specified, and if a Service has an authorization security policy requirement, the authenticated user must be in one of the roles listed. The value is a comma-separated list of roles (whitespace gets trimmed).

The runAs attribute

This is an optional attribute. If specified, the value of this attribute will be added as a role to the authenticated user.

The <properties> and <property> elements

A <security> element can optionally specify a <properties> element. This can be adjusted to specify zero to many (0..*) <property> elements. Each <property> element requires a name and a value attribute.

NOTE

The list of specified name/value properties is made available to the SwitchYard Security configuration, as well as the configured callbackHandler. Some CallbackHandlers require configuration information beyond what can be assumed in a no-argument constructor. See the individual CallbackHandler implementations for details.

1.4. USING SECURITY ELEMENTS AND ATTRIBUTES

The <component><service> and <component><reference> security Attribute

Component Services and Component References can specify an optional sy:security attribute. This attribute points to a named <security> element in the domain section. If not defined, use the default value.

The callbackHandler Attribute

This is an optional attribute. If not specified, use the default value of **org.switchyard.security.callback.NamePasswordCallbackHandler**.

The name Attribute

This is an optional attribute. If not specified, use the default value of *default*. Component Services and Component References point to this name.

The rolesAllowed Attribute

This is an optional attribute. If specified, and if a Service has an authorization security policy requirement, the authenticated user must be in one of the roles listed.

The runAs Attribute

This is an optional attribute. If specified, add the value of this attribute as a role to the authenticated user.

The securityDomain Attribute

This is an optional attribute. If not specified, use the default value.

The <properties> and <property> Elements

A <security> element can optionally specify a <properties> element, which can optionally specify zero to many <property> elements. Each <property> element has two required attributes: name and value. The list of specified name/value properties are made available to the SwitchYard Security configuration, as well as the configured callbackHandler.

The <securities> Element

This is an optional element. Contains any number of <security> elements. If not defined, use the default security configuration.

The <security> Element

This is an optional element. If not specified, the callbackHandler, name, and securityDomain attributes will fallback to their default values.

1.5. SECURITY

SwitchYard allows you to make services secure by specifying a list of security properties for a service. You may also configure the security processing details for services within a domain.

1.6. CALLBACK HANDLERS

The following is a list of available CallbackHandlers within the **org.switchyard.security.callback** Java package:

NamePasswordCallbackHandler

Provides name and password credentials to a configured LoginModule stack.

STSTokenCallbackHandler

Provides assertion credentials to a configured LoginModule stack.

STSIssueCallbackHandler

Utilizes the NamePasswordCallbackHandler and the STSTokenCallbackHandler to provide name, password and assertion credentials to a configured LoginModule stack.

CertificateCallbackHandler

Provides Certificate credentials to a configured LoginModule stack.

1.7. LOGIN MODULES

Login modules provide user information including logins, passwords and roles within JBoss Fuse. They provide services with customizable user authentication and the ability to reuse existing login modules defined for the platform.

JBoss Fuse bundles various PicketBox (underlying security capability) LoginModules, as well as various PicketLink (federated trust security capability) LoginModules. They can be stacked underneath a single security domain.

For more information on Login Module examples, refer JBoss Enterprise Application Platform 6.1 Security Guide.

1.8. SWITCHYARD SECURITY QUICKSTARTS

The SwitchYard distribution contains security examples in the form of quickstart demos. The following is a brief description of each of the quickstarts:

NOTE

All the security quickstarts for SwitchYard are located in the **EAP_HOME/quickstarts/switchyard/demos** directory. You can find out more information from the **Readme.md** file packaged with each.

policy-security-basic

This quickstart exposes a bean service through a soap binding. Confidentiality is provided via SSL, and client authentication via a HTTP Basic Authorization header.

policy-security-basic-propagate

This quickstart is similar to policy-security-basic, however the bean service additionally invokes a different back end bean service which also has security policy requirements. The client's security context (authenticated subject and credentials) is propagated to this secondary service.

policy-security-cert

This quickstart exposes a bean service through a soap binding. Confidentiality is provided via SSL, and client authentication via an X509 Certificate.

policy-security-saml

This quickstart exposes a bean service through a soap binding. Confidentiality is provided via SSL, and client authentication via a SAML assertion in the form of a token retrieved from PicketLink STS.

policy-security-wss-signencrypt

This quickstart exposes a bean service through a soap binding. Proper Signature and Encryption are enforced by JBossWS-CXF.

policy-security-wss-username

This quickstart exposes a bean service through a soap binding. Confidentiality is provided via SSL, and client authentication via a WS-Security UsernameToken which is handled by JBossWS-CXF.

1.9. SECURE WAYS OF RUNNING RED HAT JBOSS FUSE

Enabling the Java Security Manager (JSM) to sandbox the evaluation of MVEL may introduce a performance hit in high load environments. Following are some secure ways of running Red Hat JBoss Fuse:

- If you run Red Hat JBoss Fuse without Runtime Governance, you can disable JSM as it does not introduce MVEL security risks.
- If you need Runtime Governance in high performance environment, Red Hat recommends running Runtime Governance in a separate JVM. The JVM instance running Runtime Governance must have JSM enabled, whereas other application server instances can run without JSM.

- If you are working on testing and development environments without high loads, it is okay to run one JVM with the server, Runtime Governance, and JSM enabled as the performance hit is not dramatic.

WARNING

Red Hat does not recommend running the server with Runtime Governance enabled and JSM disabled in one JVM instance, as this is not secure.

CHAPTER 2. PATCH INSTALLATION

2.1. ABOUT PATCHING MECHANISMS

JBoss patches are released in two forms.

- Asynchronous updates: one-off patches which are released outside the normal update cycle of the existing product. These may include security patches, as well as other one-off patches provided by Red Hat Global Support Services (GSS) to fix specific issues.

- Planned updates: These include cumulative patches, as well as micro, minor or major upgrades of an existing product. Cumulative patches include all previously developed asynchronous updates for that version of the product.

Deciding whether a patch is released as part of a planned update or an asynchronous update depends on the severity of the issue being fixed. An issue of low impact is typically deferred, and is resolved in the next cumulative patch or minor release of the affected product. Issues of moderate or higher impact are typically addressed in order of importance as an asynchronous update to the affected product, and contain a fix for only a specific issue.

Cumulative and security patches for JBoss products are distributed in two forms: zip (for all products) and RPM (for a subset of products).

IMPORTANT

A JBoss product installation must always only be updated using one patch method: either zip or RPM patches. Not all patches will be available via RPM, therefore customers using the RPM installation will not be able to update via the zip patch, and will not have the option to patch their instance.

Security updates for JBoss products are provided by an erratum (for both zip and RPM methods). The erratum encapsulates a list of the resolved flaws, their severity ratings, the affected products, textual description of the flaws, and a reference to the patches. Bug fix updates are not announced via an erratum.

For more information on how Red Hat rates JBoss security flaws, refer to: Section 2.4, “Severity and Impact Rating of JBoss Security Patches”

Red Hat maintains a mailing list for notifying subscribers about security related flaws. See Section 2.2, “Subscribe to Patch Mailing Lists”

2.2. SUBSCRIBE TO PATCH MAILING LISTS

Summary

The JBoss team at Red Hat maintains a mailing list for security announcements for Red Hat JBoss Enterprise Middleware products. This section covers what you need to do to subscribe to this list.

Prerequisites

- None

Procedure 2.1. Subscribe to the JBoss Watch List

1. Click the following link to go to the JBoss Watch mailing list page: JBoss Watch Mailing List.
2. Enter your email address in the **Subscribing to Jboss-watch-list** section.
3. [You may also wish to enter your name and select a password. Doing so is optional but recommended.]
4. Press the **Subscribe** button to start the subscription process.
5. You can browse the archives of the mailing list by going to: JBoss Watch Mailing List Archives.

Result

After confirmation of your email address, you will be subscribed to receive security related announcements from the JBoss patch mailing list.

2.3. INSTALL PATCHES IN ZIP FORM

Summary

JBoss bug fix patches are distributed in zip format. This task describes the steps you need to take to install the patches (security or bug fixes) via the zip format.

Prerequisites

- Valid access and subscription to the Red Hat Customer Portal.
- A current subscription to a JBoss product installed in a zip format.

Procedure 2.2. Apply a patch to a JBoss product via the zip method

Security updates for JBoss products are provided by an erratum (for both zip and RPM methods). The erratum encapsulates a list of the resolved flaws, their severity ratings, the affected products, textual description of the flaws, and a reference to the patches. Bug fix updates are not announced via an erratum.

For zip distributions of JBoss products, the ERRATA includes a link to a URL on the Customer Portal where the patch zip can be downloaded. This download contains the patched versions of existing JBoss products and only includes the files that have been changed from the previous install.

> **WARNING**
>
> Before installing a patch, you must backup your JBoss product along with all customized configuration files.

1. Get notified about the security patch either via being a subscriber to the JBoss watch mailing list or by browsing the JBoss watch mailing list archives.

   > **NOTE**
   >
   > Only security patches are announced on the JBoss watch mailing list.

2. Read the ERRATA for the security patch and confirm that it applies to a JBoss product in your environment.
3. If the security patch applies to a JBoss product in your environment, then follow the link to download the patch from the Red Hat Customer Portal.
4. The downloadable zip file from the customer portal will contain all the files required to fix the security issue or bug. Download this patch zip file in the same location as your JBoss product.
5. Unzip the patch file in the same location where the JBoss product is installed. The patched versions overwrite the existing files.

Result

The JBoss product is patched with the latest update using the zip format.

2.4. SEVERITY AND IMPACT RATING OF JBOSS SECURITY PATCHES

To communicate the risk of each JBoss security flaw, Red Hat uses a four-point severity scale of low, moderate, important and critical, in addition to Common Vulnerability Scoring System (CVSS) version 2 base scores which can be used to identify the impact of the flaw.

Table 2.1. Severity Ratings of JBoss Security Patches

Severity	Description
Critical	This rating is given to flaws that could be easily exploited by a remote unauthenticated attacker and lead to system compromise (arbitrary code execution) without requiring user interaction. These are the types of vulnerabilities that can be exploited by worms. Flaws that require an authenticated remote user, a local user, or an unlikely configuration are not classed as critical impact.
Important	This rating is given to flaws that can easily compromise the confidentiality, integrity, or availability of resources. These are the types of vulnerabilities that allow local users to gain privileges, allow unauthenticated remote users to view resources that should otherwise be protected by authentication, allow authenticated remote users to execute arbitrary code, or allow local or remote users to cause a denial of service.
Moderate	This rating is given to flaws that may be more difficult to exploit but could still lead to some compromise of the confidentiality, integrity, or availability of resources, under certain circumstances. These are the types of vulnerabilities that could have had a critical impact or important impact but are less easily exploited based on a technical evaluation of the flaw, or affect unlikely configurations.

Severity	Description
Low	This rating is given to all other issues that have a security impact. These are the types of vulnerabilities that are believed to require unlikely circumstances to be able to be exploited, or where a successful exploit would give minimal consequences.

The impact component of a CVSS v2 score is based on a combined assessment of three potential impacts: Confidentiality (C), Integrity (I) and Availability (A). Each of these can be rated as None (N), Partial (P) or Complete (C).

Because the JBoss server process runs as an unprivileged user and is isolated from the host operating system, JBoss security flaws are only rated as having impacts of either None (N) or Partial (P).

Example 2.1. CVSS v2 Impact Score

The example below shows a CVSS v2 impact score, where exploiting the flaw would have no impact on system confidentiality, partial impact on system integrity and complete impact on system availability (that is, the system would become completely unavailable for any use, for example, via a kernel crash).

```
C:N/I:P/A:C
```

Combined with the severity rating and the CVSS score, organizations can make informed decisions on the risk each issue places on their unique environment and schedule upgrades accordingly.

For more information about CVSS2, please see: CVSS2 Guide.

CHAPTER 3. KNOWN SECURITY ISSUES

3.1. THE POODLE ISSUE AND JBOSS FUSE

The Poodle SSLv3 vulnerability is an issue with SSLv3 which could allow man-in-the-middle attacks. Red Hat has provided a description of the issue and its effect on some Red Hat proucts in this article *POODLE: SSLv3 vulnerability (CVE-2014-3566)* .

The Poodle SSLv3 vulnerability will affect some of the components of Red Hat JBoss Fuse. The structure of the product offers some protection by providing a layer of abstraction. Red Hat JBoss Fuse will have the same protections in place as Red Hat JBoss Enterprise Application Platform. See the Red Hat JBoss Enterprise Application Platform section of the article mentioned above for links to how to disable SSLv3 for various products.

NOTE

Please note that some of the instructions in the article links are not available for Red Hat JBoss Fuse. The layers of abstraction in the product remove the ability to directly interact with some of the components. Contact the Red Hat helpdesk with specific queries regarding any Red Hat JBoss Fuse components that your company uses.

A best practice for maximum security is to adapt new product releases and product patches soon after they are made available. Please work with your operations team to implement this best practice.

CHAPTER 4. WS-SECURITY

4.1. WS-SECURITY OVERVIEW

WS-Security standards allow policies to be applied to SOA for controlled service usage and monitoring. WS-Security provides the means to secure your services beyond transport level protocols such as HTTPS. Through a number of standards such as XML-Encryption, and headers defined in the WS-Security standard, it allows you to:

- Pass authentication tokens between services
- Encrypt messages or parts of messages
- Sign messages
- Timestamp messages

WS-Security makes heavy use of public and private key cryptography. It is helpful to understand these basics to really understand how to configure WS-Security. With public key cryptography, a user has a pair of public and private keys. These are generated using a large prime number and a key function. The keys are related mathematically, but cannot be derived from one another. With these keys we can encrypt messages. For example, if Bob wants to send a message to Alice, he can encrypt a message using her public key. Alice can then decrypt this message using her private key. Only Alice can decrypt this message as she is the only one with the private key. Messages can also be signed. This allows you to ensure the authenticity of the message. If Alice wants to send a message to Bob, and Bob wants to be sure that it is from Alice, Alice can sign the message using her private key. Bob can then verify that the message is from Alice by using her public key.

4.2. JBOSS WS-SECURITY SUPPORT

JBoss Web Services supports many real world scenarios requiring WS-Security functionalities. This includes signature and encryption support through X509 certificates, authentication and authorization through username tokens as well as all ws-security configurations covered by WS-SecurityPolicy specification. The core of WS-Security functionalities is provided through the Apache CXF engine. On top of that the JBossWS integration adds few configuration enhancements to simplify the setup of WS-Security enabled endpoints.

4.3. APACHE CXF WS-SECURITY IMPLEMENTATION

Apache CXF features a WS-Security module that supports multiple configurations and is easily extendible. The system is based on interceptors that delegate to Apache WSS4J for the low level security operations. Interceptors can be configured in different ways, either through Spring configuration files or directly using Apache CXF client API. For more details, refer to the Apache CXF documentation.

Recent versions of Apache CXF introduced support for WS-Security Policy, which aims at moving most of the security configuration into the service contract (through policies), so that clients can easily be configured almost completely automatically from that. This way users do not need to manually deal with configuring or installing the required interceptors. The Apache CXF WS-Policy engine internally takes care of that.

4.3.1. WS-Security Policy Support

WS-SecurityPolicy describes the actions that are required to securely communicate with a service advertised in a given WSDL contract. The WSDL bindings and operations reference WS-Policy fragments with the security requirements to interact with the service. The WS-SecurityPolicy specification allows for specifying things like asymmetric and symmetric keys, using transports (https) for encryption, which parts or headers to encrypt or sign, whether to sign then encrypt or encrypt then sign, whether to include timestamps, and whether to use derived keys.

Some mandatory configuration elements are not covered by WS-SecurityPolicy because they are not meant to be public or part of the published endpoint contract. These include things such as keystore locations, usernames and passwords. Apache CXF allows configuring these elements either through Spring xml descriptors or using the client API or annotations. Below is the list of supported configuration properties:

Table 4.1. Supported Configuration Properties

Property	Description
ws-security.username	The username used for UsernameToken policy assertions.
ws-security.password	The password used for UsernameToken policy assertions. If not specified, the callback handler is called.
ws-security.callback-handler	The WSS4J security CallbackHandler that is used to retrieve passwords for keystores and UsernameTokens.
ws-security.signature.properties	The properties file or object that contains the WSS4J properties for configuring the signature keystore and crypto objects.
ws-security.encryption.properties	The properties file or object that contains the WSS4J properties for configuring the encryption keystore and crypto objects.
ws-security.signature.username	The username or alias for the key in the signature keystore. If not specified, it uses the default alias set in the properties file. If that is also not set, and the keystore only contains a single key, that key is used.
ws-security.encryption.username	The username or alias for the key in the encryption keystore. If not specified, it uses the default alias set in the properties file. If that is also not set, and the keystore only contains a single key, that key is used. For the web service provider, the useReqSigCert keyword can be used to accept (encrypt to) any client whose public key is in the service's truststore (defined in **ws-security.encryption.properties**.)
ws-security.signature.crypto	Instead of specifying the signature properties, this can point to the full WSS4J Crypto object. This can allow easier programmatic configuration of the Crypto information.

Property	Description
ws-security.encryption.crypto	Instead of specifying the encryption properties, this can point to the full WSS4J Crypto object. This can allow easier programmatic configuration of the Crypto information.

Here is an example of configuration using the client API:

```
    Map<String, Object> ctx = ((BindingProvider)port).getRequestContext();
ctx.put("ws-security.encryption.properties", properties);
port.echoString("hello");
```

For additional configuration details, refer to the Apache CXF documentation.

4.3.2. JBossWS Configuration Additions

In order for removing the need of Spring on server side for setting up WS-Security configuration properties not covered by policies, the JBossWS integration allows for getting those pieces of information from a defined endpoint configuration. Endpoint configurations can include property declarations and endpoint implementations can be associated with a given endpoint configuration using the @EndpointConfig annotation.

```
<?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</property-
value>
```

```
    </property>
  </endpoint-config>
</jaxws-config>
```

```
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!";
  }
}
```

4.3.3. Apache CXF Annotations

The JBossWS configuration additions allow for a descriptor approach to the WS-Security Policy engine configuration. If you prefer to provide the same information through an annotation approach, you can leverage the Apache CXF @org.apache.cxf.annotations.EndpointProperties annotation:

```
@WebService(
  ...
)
@EndpointProperties(value = {
    @EndpointProperty(key = "ws-security.signature.properties", value = "bob.properties"),
    @EndpointProperty(key = "ws-security.encryption.properties", value = "bob.properties"),
    @EndpointProperty(key = "ws-security.signature.username", value = "bob"),
    @EndpointProperty(key = "ws-security.encryption.username", value = "alice"),
    @EndpointProperty(key = "ws-security.callback-handler", value =
"org.jboss.test.ws.jaxws.samples.wsse.policy.basic.KeystorePasswordCallback")
    }
)
public class ServiceImpl implements ServiceIface {
  ...
}
```

4.4. ENABLE WS-SECURITY

Procedure 4.1. 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 <securityAction>.

   This is so that JBossWS-CXF knows which tokens to respect within incoming SOAP requests.

3. Include a **WEB-INF/jboss-web.xml** file in your application with a <security-domain> specified.

   This is so that JBossWS-CXF knows which modules to use for authentication and rolemapping.

4.5. 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.

4.6. SIGNATURE AND ENCRYPTION SUPPORT

To add the support for WS-Security Signature and Encryption, ensure the following:

- Include the added requirements to the Policy in your WSDL. See Section 4.7, “Sample Endpoint Configurations”, Section 4.8, “Sample Client Configurations”, and Section 4.9, “Endpoint Serving Multiple Clients”.
- Add a CXF Interceptor which sets certain CXF security properties. See Section 4.10, “Sample CXF Interceptor Configurations”.

4.7. 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
```

4.8. 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.

4.9. 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.

4.10. 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
```

CHAPTER 5. S-RAMP SECURITY

5.1. S-RAMP BROWSER

S-RAMP comes with a user interface that allows you to browse all of the artifacts in the S-RAMP repository. This UI is capable of viewing and manipulating all S-RAMP artifacts in a very generic way, supporting all aspects of the S-RAMP specification such as properties, classifiers, and relationships.

The browser is a web based application built using Google Web Toolkit (GWT) and Errai projects, and is compatible with all modern web browsers. Additionally, it is capable of scaling the interface down to a size that is useful on a smart phone.

5.1.1. S-RAMP Browser Authentication

The installer handles the following S-RAMP Browser Authentication configurations automatically:

- Security domain configuration (for both S-RAMP Browser and S-RAMP Server)
- Web based single-sign-on configuration
- IDP web application configuration
- Managing Admin user

5.1.2. S-RAMP Browser Authorization

The S-RAMP Browser UI does not support any sort of fine grained authorization. You only need to have the following role in order to log in and use the UI:

```
overlorduser
```

5.2. S-RAMP SERVER

The S-RAMP implementation is a fully compliant reference implementation of the S-RAMP specification. It provides a Java based client library that you can use to integrate your own applications with an S-RAMP compliant server. The server implementation is a conventional Java web application (WAR). The following technologies are used to provide the various components that make up the server implementation:

- JCR (ModeShape): Used as the persistence engine, where all S-RAMP data is stored. Artifacts and ontologies are both stored as nodes in a JCR tree. All S-RAMP queries are mapped to JCRSQL2 queries for processing by the JCR API. The ModeShape JCR implementation is used by default. However, the persistence layer is pluggable allowing alternative providers to be implemented in the future.
- JAX-RS (RESTEasy): Used to provide the S-RAMP Atom based REST API. The S-RAMP specification documents an Atom based REST API that implementations should make available. The S-RAMP implementation uses JAX-RS (specifically RESTEasy) to expose all of the REST endpoints defined by the specification.
- JAXB: Used to expose a Java data model based on the S-RAMP data structures defined by the specification (S-RAMP XSD schemas).

For more information on the underlying runtime, refer JBoss Enterprise Application Platform 6.1 Security Guide.

5.2.1. S-RAMP Server Authorization

When accessing the S-RAMP Atom API, the authenticated user must have certain roles. As the implementation leverages ModeShape as its persistence store by default, the authenticated user must have the following JAAS role, which is required by ModeShape:

```
admin.sramp
```

Additionally, the S-RAMP Atom API web application requires the user to have the following role:

```
overlorduser
```

CHAPTER 6. POLICY

6.1. ABOUT POLICY

Policy enables you to verify the runtime behavior of a service in a declarative manner, free of the service implementation and binding details. For example, if you require a service to always participate in a global transaction only, you can add logic to your service implementation which checks the current transaction state, associates with an active global transaction, and handles error cases. Also, ensure that the gateway used to expose the service is transactional and that it propagates the transaction to the service implementation.

The SwitchYard runtime reads these policy definitions during deployment and enforces them on a per-message basis as services are invoked.

6.2. CONFIGURING POLICY

The Policy definition is comprised of two aspects:

- Policy that the service provider requires
- Policy support that the service consumer provides

You can define the policy that a service requires by annotating the service's configuration in the SwitchYard application descriptor. You can also define how a service is consumed through a gateway binding, which effectively determines how the policy requirements are satisfied or provided. The SwitchYard runtime takes care of determining whether the consumer satisfies the policy requirements by evaluating the configuration of the application and the runtime state of the messages exchanged between the consumer and provider. Here is an example configuration:

```
<composite name="policy-transaction">
  <service name="WorkService" promote="WorkService">
    <camel:binding.camel configURI="jms://policyQSTransacted?
connectionFactory=%23JmsXA&transactionManager=%23jtaTransactionManager&transacted=true"/>

    <camel:binding.camel configURI="jms://policyQSNonTransacted?
connectionFactory=#ConnectionFactory"/>
  </service>
  <component name="WorkService">
    <implementation.bean
class="org.switchyard.quickstarts.demo.policy.transaction.WorkServiceBean"
requires="managedTransaction.Global"/>
    <service name="WorkService" requires="propagatesTransaction">
      <interface.java interface="org.switchyard.quickstarts.demo.policy.transaction.WorkService"/>
    </service>
    <reference name="TaskAService" requires="propagatesTransaction">
      <interface.java interface="org.switchyard.quickstarts.demo.policy.transaction.TaskAService"/>
    </reference>
  </component>
</composite>
```

6.3. INTERACTION POLICY

In SwitchYard, the component service and component reference both are marked by policies using

requires attribute. You can use the Interaction policy on component service and component reference and is not allowed to be marked on component implementation. This policy enables you to manage the communication between the service provider and consumer.

6.4. IMPLEMENTATION POLICY

In SwitchYard, the component service and component reference both are marked by policies using ***requires*** attribute. You can use the Implementation policy on component implementation and is not allowed to be marked on component service nor component reference.

6.5. TRANSACTION POLICY

Transaction Policy defines implementation and interaction policies that relate to transactional quality of service in components and their interactions. The transaction policies are specified as intents which represent the transaction quality of service behavior offered by specific component implementations or bindings.

6.5.1. Transaction Interaction Policy

You can specify the Transaction Interaction Policy by using a component service or component reference definition's ***requires*** attribute. Here is an example:

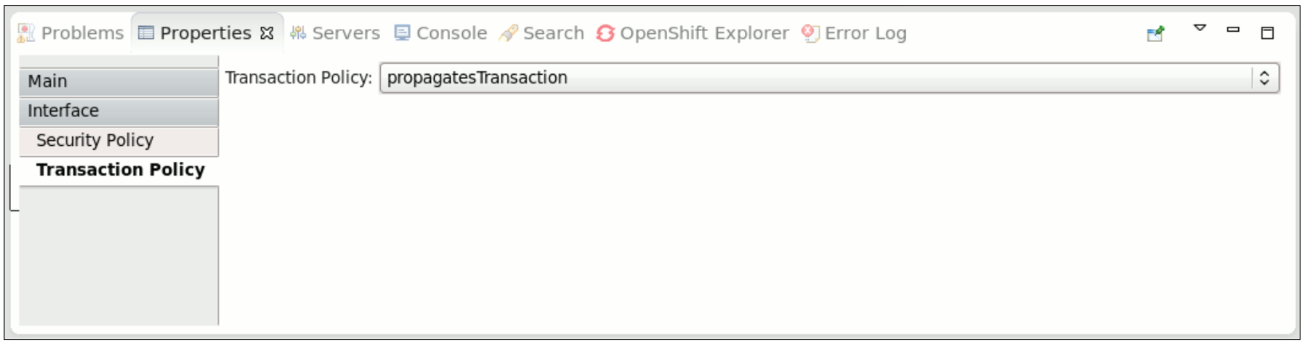
```
<service name="WorkService" requires="propagatesTransaction">
```

Valid values for the Transaction Interaction Policy intents are:

- ***propagatesTransaction***: Indicates that a global transaction is required when a service is invoked. If no transaction is present, the SwitchYard generates an error at runtime.
- ***suspendsTransaction***: Indicates that if a transaction exists, the SwitchYard runtime suspends it before the service implementation is invoked and resumes it after the service invocation. This setting allows you to separate a gateway binding's transactional context from the transactional context of the service implementation.

You can attach the mutually exclusive ***propagatesTransaction*** and ***suspendsTransaction*** intents either to an interface or explicitly to a service and reference XML element in order to describe how any client transaction context is made available and used by the target service component.

Figure 6.1. Transaction Interaction Policy



Transaction Policy: propagatesTransaction

6.5.2. Transaction Implementation Policy

You can specify the Implementation Interaction Policy by using a component service or component reference definition's ***requires*** attribute. Here is an example:

```
<implementation.bean class="org.switchyard.quickstarts.demo.policy.transaction.WorkServiceBean"
requires="managedTransaction.Global">
```

Valid values for Transaction Implementation Policy intents are:

- ***managedTransaction.Global***: Indicates that this service implementation runs under a global transaction. If no transaction is present, the SwitchYard runtime creates a new JTA transaction before the execution. The SwitchYard runtime commits this newly created transaction at the end of service execution.
- ***managedTransaction.Local***: Indicates that this service implementation runs under a local transaction. If a transaction exists, the SwitchYard runtime suspends it. SwitchYard always creates a new JTA transaction before execution. The SwitchYard runtime commits this newly created transaction and resumes the suspended transcation after the service invocation.

  NOTE

  As the local transaction does not propagate its transaction through the reference, you must mark all of the component reference as ***suspendsTransaction***. If not, the SwitchYard runtime generates an error.

- ***noManagedTransaction***: Indicates that this service implementation does not run under any managed transaction. If a transaction exists, the SwitchYard runtime suspends it before the service implementation is invoked and resumes it after the service invocation.

You can use the mutually exclusive ***managedTransaction*** and ***noManagedTransaction*** intents to describe the transactional environment required by a service component.

Scope of Support

Currently, the following gateways are transaction aware:

- Camel JMS Gateway (binding.jms)
- Camel JPA Gateway (binding.jpa)
- Camel SQL Gateway (binding.sql)
- JCA Gateway (binding.jca)
- SCA Gateway (binding.sca)

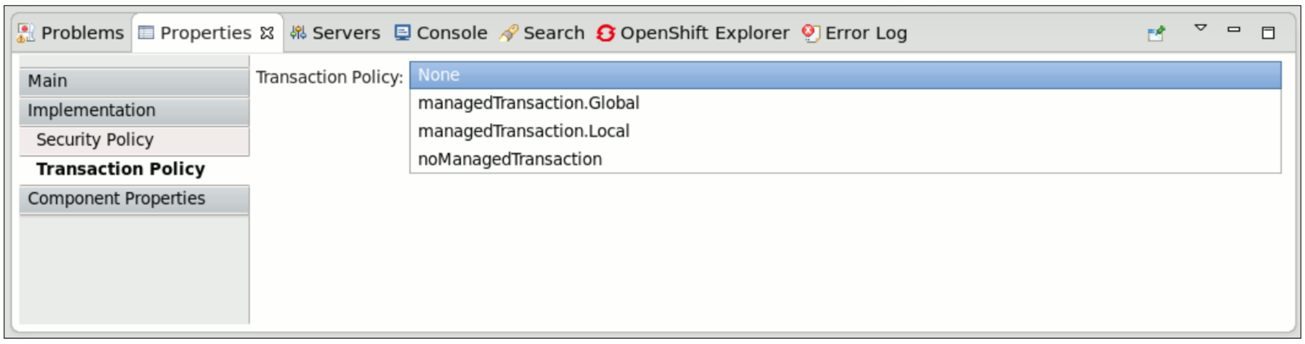
NOTE

BPM service (implementation.bpm) always need a JTA transaction if the persistence is enabled. It synchronizes with incoming transaction if exists, otherwise it begins a new JTA Transaction and commit/rollback by itself.

NOTE

If your application has multiple camel-jms bindings which are bound to the same JMS provider, you need to define distinct connection factory for each binding to get Transaction Policy working. otherwise, transaction can't be suspended as expected. Similarly, distinct xa-datasource is needed to get Transaction Policy working on camel-jpa and camel-sql. (https://issues.jboss.org/browse/SWITCHYARD-1285)

Figure 6.2. Transaction Implementation Policy



Transaction Policy: None
managedTransaction.Global
managedTransaction.Local
noManagedTransaction

6.6. SECURITY POLICY

SwitchYard services can be secured by specifying a list of security policies that are required for that service and configuring the security processing details for the services within a domain.

NOTE

The security manager gets enabled with additional security policies if you select the Runtime Governance component during installation. Following are the security policies included in JBoss Fuse:

- security.policy- It defines a JVM level permitAll policy.
- kie.policy- It is used by Rule-based services.
- rtgov.policy- It is used for the Runtime Governance REST API.

NOTE

Support for security policy is limited to bean services (implementation.bean), SOAP endpoints via the SOAP gateway (binding.soap), and HTTP endpoints via the HTTP gateway (binding.http).

6.6.1. Security Interaction Policy

The Security Interaction Policy is defined using the ***requires*** attribute of a component service definition.

```
<service name="WorkService" requires="authorization clientAuthentication confidentiality">
```

Valid values for Security Interaction policy are:

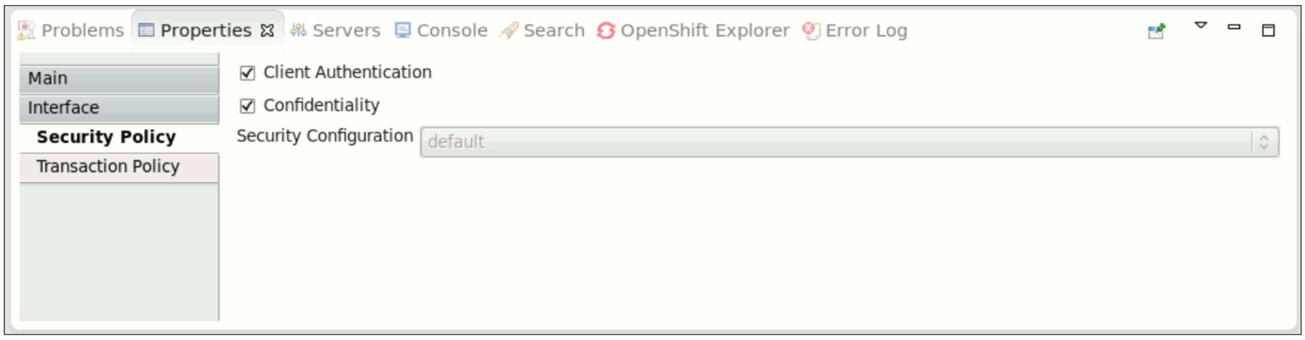
- *clientAuthentication*: indicates that the client has been authenticated when a service is invoked. If the associated authenticated user principal is not available, SwitchYard runtime generates an error.

  NOTE

  There are multiple reasons for why the clientAuthentication policy may not be fulfilled, including incorrect username, incorrect password, or configuration issues. Please check the credentials again. If they seem to be in order engage the support team for further analysis.

- *confidentiality*: indicates that the request has been made over a secure channel. When a SOAP request is made over SSL and its confidentiality is not verified, SwitchYard runtime generates an error,

Figure 6.3. Security Interaction Policy



☑ Client Authentication
☑ Confidentiality
Security Configuration: default

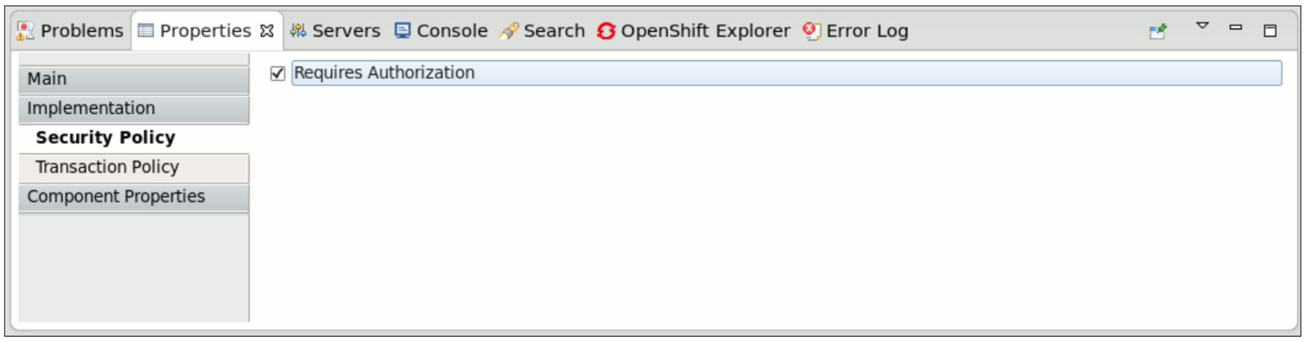
6.6.2. Security Implementation Policy

Security Implementation Policy is specified using the requires attribute of a component implementation definition.

Valid values for Security Implementation policy are:

- *authorization*: indicates that the client is authorized to invoke the service. If the associated authenticated subject does not have an allowed role, SwitchYard runtime generates an error.

Figure 6.4. Security Implementation Policy



☑ Requires Authorization

6.6.3. Setting Security Policy

You can define the Security Policy in the following ways:

- Edit the SwitchYard application descriptor (**switchyard.xml**) and add the ***requires*** attribute to a service definition.

- Use the ***@Requires*** attribute in your service implementation to declare security policy for the service. When the application project is built, the SwitchYard application finds ***@Requires*** annotations and automatically generates the required configuration.

6.6.4. Security Processing

When the container does not automatically provide certain security policies, the SwitchYard application can be configured to process security credentials extracted from the binding-specific data, then provide certain security policies itself (like clientAuthentication). All services within a domain share the same security configuration, which is specified in the **switchyard.xml**.

```
<switchyard>
  <domain>
    <security callbackHandler="callback-handler-class-name" moduleName"="jaas-domain-name"
rolesAllowed="users, administrators" runAs="leaders">
       <properties>
          <property name="property-name" value="property-value"/>
       </properties>
    </security>
  </domain>
</switchyard>
```

NOTE

Support for security policy is limited to bean services (implementation.bean), SOAP endpoints via the SOAP gateway (binding.soap), and HTTP endpoints via the HTTP gateway (binding.http).