Red Hat AMQ 6.3

WS-Notification Guide

Accessing topic subscriptions through the WS-Notification standard

Last Updated: 2020-10-23

Red Hat AMQ 6.3 WS-Notification Guide

Accessing topic subscriptions through the WS-Notification standard

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

Legal Notice

Abstract

WS-Notification is implemented as a wrapper around the JBoss A-MQ broker, enabling you to access topic subscriptions through a standardised Web service interface.

Table of Contents

CHAPTER 1. INTRODUCTION TO WS-NOTIFICATION

1.1. WS-NOTIFICATION STANDARD

Overview

WS-Notification is a standard that describes a publish/subscribe messaging model implemented over Web services. The functionality is quite similar to the JMS publish/subscribe model, but the interfaces and the protocols are defined in terms of Web services standards (based on Apache CXF, in the context of JBoss A-MQ).

The WS-Notification standard is defined by combining the following OASIS specifications:

- WS-Topics
- WS-BaseNotification
- WS-BrokeredNotification

WS-Topics

The WS-Topics standard describes how to organize and define the topics used in a notification broker. In particular, the following aspects of a topic are described:

- *Topic hierarchy*—a hierarchical naming scheme, which can be defined using an XML document associated with the notification broker.
- *Topic set*—a standardized XML schema, which can optionally be used to define the hierarchy of topic names.
- *Topic dialect*—a particular type of name expression that is used to specify one topic or to select multiple topics. The following dialects are defined by the WS-Topics specification:
  - Simple
  - Concrete
  - Full
  - XPath

NOTE

Topic hierarchies are not supported in JBoss A-MQ. Only Simple topic names can be defined.

WS-BaseNotification

The WS-BaseNotification standard describes a simple point-to-point model of message notification. The base standard can be useful, if you want to send notifications through a standardized interface, without deploying a fully-fledged broker to mediate the messages. The following WSDL interfaces are defined in this standard:

NotificationPublisher

Must be implemented by the entity that wants to publish messages. This interfaces exposes the **subscribe** operation, which enables consumers to register their interest in receiving notifications from this publisher.

NotificationConsumer

Must be implemented by the entity that wants to receive messages. This interface exposes the **notify** operation, which enables the consumer to receive message notifications directly from the publisher.

In addition to the two preceding interfaces for point-to-point communication, WS-BaseNotification defines another pair of interfaces for supporting pull-style notification, as follows:

CreatePullPoint

Exposes the **createPullPoint** operation, which creates a **PullPoint** object that can be used to accumulates messages.

PullPoint

Exposes the **notify** operation, which enables the pull-point to accumulate notification messages, and the **getMessages** operation, which enables a pull-style consumer to retrieve the accumulated messages when it is ready.

WS-BrokeredNotification

The WS-BrokeredNotification standard describes a brokered model of message notification, where a central broker (or network of brokers) can be used to route messages between publishers and consumers. This architecture scales much better than point-to-point, because each consumer requires only a single connection to the broker in order to monitor notifications from *all* publishers. The following additional interfaces are defined in this specification:

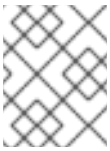
NotificationBroker

Combines the **NotificationPubisher**, **NotificationConsumer**, and **CreatePullPoint** interfaces, enabling you to provide the full range of notification services in a single application.

The **NotificationBroker** interface defines one additional operation, the **registerPublisher** operation, which can optionally be used to register publishers with the broker. In particular, this operation can be useful when constructing a federation of brokers.

RegisterPublisher

The notification broker also implements the **RegisterPublisher** interface, which defines one additional operation, **registerPublisher**. A publisher can optionally use the **registerPublisher** operation its **NotificationPublisher** object with the broker.



NOTE

It is also possible for publishers to send messages to the broker straightaway, by invoking **notify**, without needing to register in advance.

PublisherRegistrationManager

The return value of the **registerPublisher** operation is a reference to a **PublisherRegistrationManager** object, which can be used to destroy a registration.

References

For more information about the WS-Notification standards, see the following references:

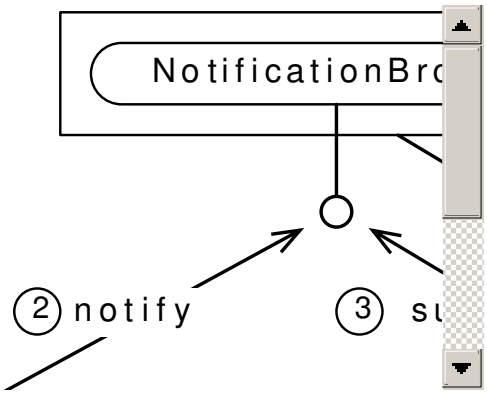
- WS-Topics 1.3 OASIS Standard
- WS-BaseNotification 1.3 OASIS Standard
- WS-BrokeredNotification 1.3 OASIS Standard

1.2. CONSUMER CLIENT SCENARIO

Overview

In the consumer client scenario, the consumer client receives messages *directly* from the broker, as soon as they become available. This approach requires the consumer client to implement a callback object, which exposes a Web service endpoint. Figure 1.1, "A Consumer Client Scenario" provides an overview of this scenario.

Figure 1.1. A Consumer Client Scenario



Clients in this scenario

There are two clients involved in this scenario:

- *Publisher client*—generates notification messages and publishes the messages on a specific topic, by sending them to the notification broker.
- *Consumer client*—a client that implements a consumer callback object (exposing a Web service endpoint of **NotificationConsumer** type), which is capable of receiving notifications directly from the notification broker.

Scenario steps

In this scenario, a consumer client receives notification messages from the broker as follows:

1. The consumer client instantiates a consumer callback object, which implements the **NotificationConsumer** interface and is capable of receiving notifications from the broker.
2. The consumer client creates a subscription by invoking the **subscribe** operation on the broker, passing the following operation arguments:
    - *Topic name*—specifies the topic that the client wants to subscribe to.

- *Callback reference*—a reference to the consumer callback object that will receive the notifications, where the service reference has the format of a WS-Addressing Endpoint Reference (EPR).

3. A publisher client sends a notification message on a specific topic, by invoking the **notify** operation on the broker.

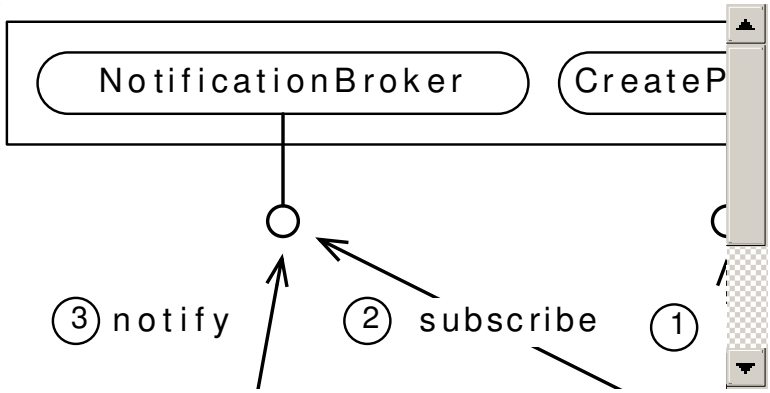
4. If the message topic matches the consumer client's subscription, the broker will forward the message to the consumer client by invoking the **notify** operation on the consumer callback service.

1.3. PULLPOINT CLIENT SCENARIO

Overview

In the pull-point client scenario, the pull-point client does *not* receive messages directly from the broker. Instead, the pull-point client allows messages to accumulate in a remote **PullPoint** object (which acts as a message drop-box) and retrieves the messages from time to time by invoking the **getMessages** operation on the **PullPoint**. Figure 1.2, "A PullPoint Client Scenario" provides an overview of this scenario.

Figure 1.2. A PullPoint Client Scenario



Clients in this scenario

There are two clients involved in this scenario:

- *Publisher client*—generates notification messages and publishes the messages on a specific topic, by sending them to the notification broker.

- *PullPoint client*—a client that uses a polling strategy to get notification messages. Instead of receiving notification messages directly from the broker, this client creates a remote **PullPoint** instance. Messages that accumulate in the **PullPoint** can be retrieved at any time by invoking the **getMessages** operation on the **PullPoint**.

Scenario steps

In this scenario, a pull-point client polls for notification messages as follows:

1. The pull-point client creates a remote **PullPoint** instance by invoking the **create** operation on the **CreatePullPoint** interface in the broker. The return value from this operation contains a WS-Addressing reference to the remote pull-point.

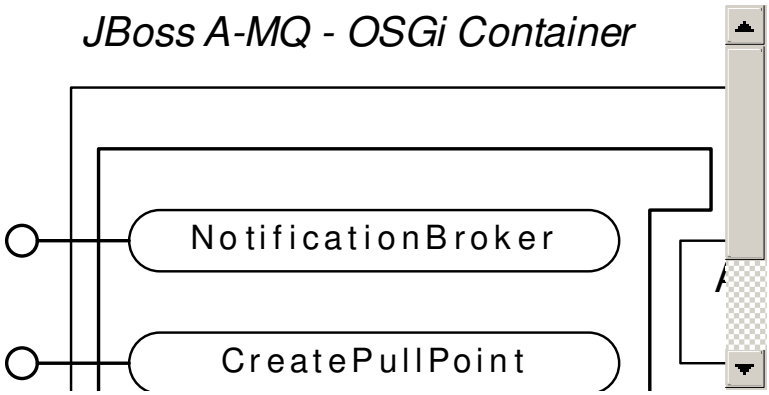
2. The pull-point client creates a subscription by invoking the **subscribe** operation on the broker, passing the following operation arguments:
   - *Topic name*—specifies the topic that the client wants to subscribe to.
   - *Callback reference*—a reference to the remote **PullPoint** instance that will receive the notifications on behalf of the client.
3. A publisher client sends a notification message on a specific topic, by invoking the **notify** operation on the broker.
4. At any time, the pull-point client can retrieve messages that have accumulated in the **PullPoint** instance by invoking the **getMessages** operation on the **PullPoint**.

1.4. IMPLEMENTATION OF WS-NOTIFICATION

Overview

Figure 1.3, "Notification Broker Architecture" shows an overview of how the WS-Notification standard is implemented in JBoss A-MQ, where the notification broker supports both the WS-BaseNotification standard and the WS-BrokeredNotification standard.

Figure 1.3. Notification Broker Architecture



Notification broker as wrapper around ActiveMQ broker

The JBoss A-MQ notification broker is implemented essentially as a wrapper around the Apache ActiveMQ broker. This is possible, because the topic-based messaging model at the heart of WS-Notification is essentially the same as the JMS publish/subscribe model. The notification broker wrapper layer provides the SOAP/HTTP protocol, implements the standard WSDL interfaces, and implements the integration layer; the ActiveMQ broker component provides persistence, message routing, and JMX support (amongst other things).

The notification broker wrapper and the ActiveMQ broker are connected together using a normal client-broker connection. In theory, you could use any ActiveMQ supported protocol for this connection, but it makes the most sense to embed both components in the same JVM and to use the VM protocol. This embedded coupling ensures optimum efficiency and performance.

OSGi container deployment

In theory, the notification broker can be deployed standalone or into various containers. The normal deployment model in JBoss A-MQ, however, is the OSGi container deployment. To simplify OSGi deployment, the notification broker can be installed as the Karaf feature, **cxf-wsn**.

Supported WS-Notification interfaces

The notification broker service supports the following two WS-Notification interfaces:

NotificationBroker

The main notification broker interface enables you to create subscriptions (**subscribe** operation), send notification messages (**notify** operation), and register Publisher services (**registerPublisher** operation).

CreatePullPoint

The create pull-point interface enables you to create new pull-point endpoints on the notification broker, which are used to accumulate messages until a consumer client is ready to retrieve them.

Qualities of service

Most of the options to configure qualities of service are provided by the underlying ActiveMQ broker. All of the usual topic-oriented features and qualities of service can be configured on the underlying broker. In particular, you can turn on persistence in the broker, so that subscriptions and messages are persisted.

Topics

Notification messages are organized by topic, so that messages sent on a particular topic will be received by those consumers that are subscribed to that topic. In JBoss A-MQ, the notification topics are mapped to the underlying ActiveMQ topics, as follows:

- Only the SIMPLE dialect is supported (of the dialects described in the WS-Notification specification).
- In a WS-Notification client, you can specify a topic name as the **String** type or as the **QName** type.
- A notification topic name maps *directly* to an ActiveMQ topic name.
- Topic hierarchies are *not* supported in JBoss A-MQ, but something very similar is supported by the underlying ActiveMQ broker. In Apache ActiveMQ, you can define a topic to have a segmented structure, where each segment is delimited by the **.** character—for example, **STOCKS.NYSE.REDHAT**. Within the ActiveMQ configuration, you can exploit this structure to match multiple topics—for example, **STOCKS.NYSE.>** matches all topics starting with **STOCKS.NYSE.**.
- Topics are ad-hoc—in other words, there is no need to pre-define any topic hierarchy in XML. Topics are created dynamically: if you use them, they are automatically created in the broker. This is the standard approach supported in the underlying ActiveMQ broker.

Configuration of the notification broker

The notification broker is configured mainly by the following OSGi Config Admin configuration files:

etc/org.apache.cxf.wsn.cfg

Configures the wrapper component of the notification broker. For details about the properties you can set in this file, see the section called “org.apache.cxf.wsn.cfg settings”.

etc/io.fabric8.mq.fabric.server-default.cfg

Customizes the OSGi deployment of the Apache ActiveMQ broker. A couple of important properties can be set in this file—for example, the broker name.

etc/activemq.xml

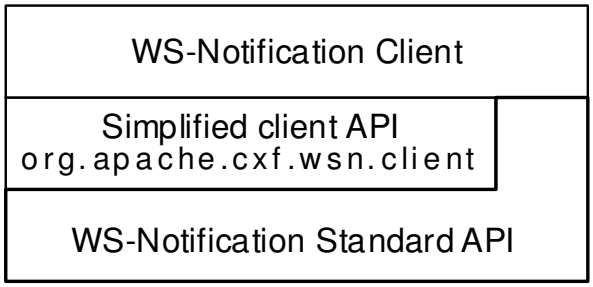
Configures the Apache ActiveMQ broker. Most of the broker features and properties can be configured in this file. For example, you can configure message persistence and fine tune broker performance in this file.

1.5. CLIENT API

Overview

Figure 1.4, "Client APIs" gives an overview of the available APIs for programing WS-Notification clients.

Figure 1.4. Client APIs



WS-Notification standard API

Clients can be implemented using the standard WS-Notification API, which is obtained by mapping the standard WSDL interfaces to Java the JAX-WS and JAX-B. This has the advantage that you can use standard client code to access the notification broker (ensuring code portability), but it has the disadvantage that the standard API is relatively complicated to program with.

Simplified client API

To simplify working with the notification broker, JBoss A-MQ offers a simplified (non-standard) client API for accessing the notification broker. This API automatically takes care of tedious manipulation of JAX-B data types. Using this API, you typically require just a few method calls to implement a basic WS-Notification client.

For example, see the client code samples in Chapter 2, *WS-Notification Tutorial*.

API reference

The full API reference for the simplified client API is provided in the *Apache CXF API Reference*, which is available from the *API Reference* in the JBoss Fuse library. All of the relevant classes can be found in the following Java package:

```
org.apache.cxf.wsn.client
```

CHAPTER 2. WS-NOTIFICATION TUTORIAL

2.1. INSTALL AND CONFIGURE THE NOTIFICATION BROKER

Overview

This section of the tutorial describes how to install and configure the notification broker as a Web service in the JBoss A-MQ standalone container. For convenient OSGi deployment, the notification broker is packaged as an Apache Karaf feature.

Prerequisites

This tutorial assumes that you are starting from a plain standalone container, in the initial configuration you find it in after installing JBoss A-MQ (and, in particular, that the container is *not* configured as part of a Fuse fabric).

Steps to install the notification broker

To install and configure the notification broker in the JBoss A-MQ container, perform the following steps:

1. Make sure you have already configured some user accounts in the **etc/users.properties** file. If necessary, create a user account by adding lines in the following format:

   ```
   Username=Password[,Role1][,Role2]...
   ```

   For example, this tutorial assumes that the following **admin** user account is defined (which has privileges defined by the **Administrator** role):

   ```
   admin=admin,Administrator
   ```

2. Create the notification broker configuration file, ***InstallDir*/etc/org.apache.cxf.wsn.cfg**, and use a text editor to add the following property settings:

   ```
   cxf.wsn.activemq=vm://amq?create=false&waitForStart=10000
   cxf.wsn.activemq.username=admin
   cxf.wsn.activemq.password=admin
   ```

   The following aspects of the notification broker are configured in this file:

   - *Connection to the ActiveMQ broker*—the **vm://amq** URL connects through the Java Virtual Machine to access the broker named **amq** (where the broker's name is defined by the **broker-name** setting in the **etc/io.fabric8.mq.fabric.server-default.cfg** file). The following options are specified on this URL:

     create

     By setting **create=false**, you can ensure that the notification broker does not try to create its own (embedded) instance of a broker, but always tries to connect to the existing broker instance named **amq**.

     waitForStart

To compensate for any delays that might occur during the container's start-up sequence, this endpoint defines a grace period, during which it waits for the broker to start.

- *Credentials for the connection*—because authentication is enabled by default in the broker, you must provide credentials (username and password) for connecting to the broker. The credentials must refer to one of the user accounts defined in **etc/users.properties**.

3. Start up the JBoss A-MQ container, by entering the following command from the ***InstallDir*/bin** directory:

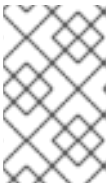
```
./amq
```

4. Install and start up the notification broker using the **features:install** console command, as follows:

```
JBossA-MQ:karaf@root> features:install cxf-wsn
```

5. Check that broker has started up by navigating to the following URL in your Web browser (querying the WSDL contract from the Web service endpoint):

```
http://localhost:8182/wsn/NotificationBroker?wsdl
```



NOTE

Your browser should display the NotificationBroker WSDL contract in response to this URL, but this does not work in all browsers. For example, the Safari browser just displays a blank page.

Troubleshooting

If you are not sure whether the notification broker is running properly, you can get some diagnostic information using the following commands:

osgi:list

If you run **osgi:list** at the console prompt, you should see some output like the following:

```
JBossA-MQ:karaf@root> osgi:list
...
[ 149] [Active     ] [Created     ] [   40] Apache CXF API (2.6.0.redhat-60024)
[ 150] [Active     ] [Created     ] [   40] Apache CXF Runtime Core (2.6.0.redhat-60024)
[ 151] [Active     ] [            ] [   40] Apache CXF Runtime Management (2.6.0.redhat-60024)
[ 152] [Active     ] [Created     ] [   40] Apache CXF Karaf Commands (2.6.0.redhat-60024)
[ 153] [Active     ] [            ] [   30] Apache Neethi (3.0.2)
[ 154] [Active     ] [Created     ] [   40] Apache CXF Runtime WS Policy (2.6.0.redhat-60024)
[ 155] [Active     ] [            ] [   40] Apache CXF Runtime XML Binding (2.6.0.redhat-60024)
[ 156] [Active     ] [Created     ] [   40] Apache CXF Runtime SOAP Binding (2.6.0.redhat-60024)
[ 157] [Active     ] [Created     ] [   40] Apache CXF Runtime WS Addressing (2.6.0.redhat-60024)
[ 158] [Active     ] [            ] [   40] Apache CXF Runtime JAXB DataBinding (2.6.0.redhat-60024)
[ 159] [Active     ] [Created     ] [   40] Apache CXF Runtime HTTP Transport (2.6.0.redhat-60024)
[ 160] [Active     ] [Created     ] [   40] Apache CXF Runtime Simple Frontend (2.6.0.redhat-60024)
[ 161] [Active     ] [Created     ] [   40] Apache CXF Runtime JAX-WS Frontend (2.6.0.redhat-
```

```
60024)
[ 162] [Active     ] [            ] [   60] Apache CXF WSN API (2.6.0.redhat-60024)
[ 163] [Active     ] [Created     ] [   40] Apache CXF Runtime HTTP Jetty Transport (2.6.0.redhat-
60024)
[ 166] [Active     ] [Created     ] [   60] Apache CXF WSN Core (2.6.0.redhat-60024)
```

In particular, the **Apache CXF WSN Core** bundle (which deploys the notification broker server) must have the status **Active** and **Created**.

log:display

Run the **log:display** command at the console prompt to search the container log for errors and warnings.

org.apache.cxf.wsn.cfg settings

You can set the following properties in the **etc/org.apache.cxf.wsn.cfg** configuration file:

cxf.wsn.activemq

Specifies the URI for connecting to the ActiveMQ broker (must be an ActiveMQ client URL). Default is **vm:localhost**.

cxf.wsn.activemq.username

Specifies the username credentials for logging on to the ActiveMQ broker. Default is **user**.

cxf.wsn.activemq.password

Specifies the password credentials for logging on to the ActiveMQ broker. Default is **password**.

cxf.wsn.rootUrl

Specifies the host and IP port of the notification broker's Web service endpoints. Default is **http://0.0.0.0:8182**.

cxf.wsn.context

Defines the servlet context for notification broker's Web service endpoints. Default is **/wsn**.

By default, the notification broker constructs its **NotificationBroker** endpoint address and its **CreatePullPoint** endpoint address as follows:

```
${cxf.wsn.rootUrl}${cxf.wsn.context}/NotificationBroker
${cxf.wsn.rootUrl}${cxf.wsn.context}/CreatePullPoint
```

Advanced configuration

Because the ActiveMQ broker provides the core functionality of the notification broker, most of the configuration options are available in the **etc/activemq.xml** file. For example, through the settings in this file you can configure persistent storage and you can optimize the broker for optimum performance.

For more details, see *Managing and Monitoring a Broker* and *Configuring Broker Persistence*.

2.2. CREATE A PUBLISHER CLIENT

Overview

This section describes how to create a publisher client of the notification broker. The publisher client is capable of sending messages on a specific topic to the notification broker.

Prerequisites

In order to access artifacts from the Maven repository, you need to add the **fusesource** repository to Maven's **settings.xml** file. Maven looks for your **settings.xml** file in the following standard location:

- UNIX: **home/*User*/.m2/settings.xml**
- Windows: **Documents and Settings*User*\.m2\settings.xml**

If there is currently no **settings.xml** file at this location, you need to create a new **settings.xml** file. Modify the **settings.xml** file by adding the **repository** element for **fusesource**, as highlighted in the following example:

```
<settings>
  <profiles>
    <profile>
      <id>my-profile</id>
      <activation>
        <activeByDefault>true</activeByDefault>
      </activation>
      <repositories>
        <repository>
          <id>redhat-ga-repository</id>
          <url>https://maven.repository.redhat.com/ga</url>
          <releases>
            <enabled>true</enabled>
          </releases>
          <snapshots>
            <enabled>false</enabled>
          </snapshots>
        </repository>
        ...
      </repositories>
    </profile>
  </profiles>
  ...
</settings>
```

Sample publisher client code

Example 2.1, “Publisher Client Code” shows the code for a sample publisher client that pushes a simple **Hello World!** message to the **MyTopic** topic on the notification broker.

Example 2.1. Publisher Client Code

```
// Java
package org.jboss.fuse.example.wsn.publisher;
```

```
import javax.xml.bind.JAXBElement;
import javax.xml.namespace.QName;

import org.w3c.dom.Element;

import org.apache.cxf.wsn.client.Consumer;
import org.apache.cxf.wsn.client.NotificationBroker;
import org.apache.cxf.wsn.client.Subscription;
import org.oasis_open.docs.wsn.b_2.NotificationMessageHolderType;

/**
 *
 */
public final class Client {
    private Client() {
        //not constructed
    }

    /**
     * @param args
     */
    public static void main(String[] args) throws Exception {
        String wsnPort = "8182";
        if (args.length > 0) {
            wsnPort = args[0];
        }

        // Create a NotificationBroker proxy
        NotificationBroker notificationBroker
            = new NotificationBroker("http://localhost:" + wsnPort + "/wsn/NotificationBroker");


        for (int i=0; i<120; i++) {
            // Send notifications on the Topic
            notificationBroker.notify(
                "MyTopic",
                new JAXBElement<String>(
                    new QName("urn:test:org", "foo"),
                    String.class,
                    "Hello World!"
                )
            );

            // Sleep for 1s between notifications
            Thread.sleep(1000);
        }

        // Cleanup and exit
        System.exit(0);
    }

}
```

NotificationBroker proxy class

The client code from Example 2.1, "Publisher Client Code" uses the **NotificationBroker** proxy class to connect to the remote notification broker and to publish notifications to the broker. In this example, the following **NotificationBroker** methods are invoked:

NotificationBroker(String address, Class<?>... cls)

The **NotificationBroker** constructor normally takes a single argument, which is the URL of the remote notification broker Web service.

notify(String topic, Object msg)

Sends a message, **msg**, on the topic, **topic**, to the notification broker, where the format of the **msg** argument is an XML document. For example, you can use the JAX-B API to create a single XML element containing a string for the message, as shown in Example 2.1, "Publisher Client Code".

NOTE

This **NotificationBroker** proxy class belongs to the simplified client API provided by the Apache CXF implementation of WS-Notification; it is *not* an instance of the standard **NotificationBroker** SEI defined by JAX-WS (although the standard SEI is also available and could be used instead).

Steps to create a publisher client

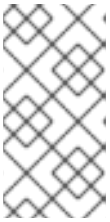
Perform the following steps to create a publisher client:

1. You can create a Maven project directly from the command line, by invoking the **archetype:generate** goal. First of all, create a directory to hold the WS-Notification client projects. Open a command prompt, navigate to a convenient location in your file system, and create the **wsn** directory, as follows:

```
mkdir wsn
cd wsn
```

You can now use the **archetype:generate** goal to invoke the **karaf-soap-archetype** archetype, which generates a simple Apache CXF demonstration, as follows:

```
mvn archetype:generate \
  -DarchetypeGroupId=io.fabric8.archetypes \
  -DarchetypeArtifactId=karaf-soap-archetype \
  -DarchetypeVersion=1.2.0.redhat-630187 \
  -DgroupId=org.jboss.fuse.example \
  -DartifactId=wsn-publisher \
  -Dversion=1.0-SNAPSHOT \
  -Dpackage=org.jboss.fuse.example.wsn.publisher \
  -Dfabric8-profile=wsn-publisher-profile
```



NOTE

The backslash characters at the end of each line are effective as line-continuation characters on UNIX and LINUX platforms. If you are entering the command on a Windows platform, however, you must enter the entire command on a single line.

You will be prompted to confirm the project settings, with a message similar to this one:

```
Confirm properties configuration:
groupId: org.jboss.fuse.example
artifactId: wsn-publisher
version: 1.0-SNAPSHOT
package: org.jboss.fuse.example.wsn.publisher
fabric8-profile: wsn-publisher-profile
 Y: :
```

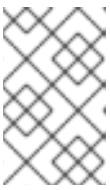
Type **Return** to accept the settings and generate the project. When the command finishes, you should find a new Maven project in the **wsn/wsn-publisher** directory.

2. Some of the generated project files are not needed for this tutorial. Under the **wsn/wsn-publisher** directory, delete the following files and directories:

```
src/main/resources/OSGI-INF/blueprint/blueprint.xml
src/main/java/org/jboss/fuse/example/wsn/publisher/HelloWorld.java
src/main/java/org/jboss/fuse/example/wsn/publisher/HelloWorldImpl.java
```

3. Edit the **pom.xml** file in the **wsn-publisher** directory, and add the dependency required for WS-Notification clients:

```
<?xml version="1.0" encoding="UTF-8"?>
<project ...>
    ...
    <dependencies>
        ...
        <!-- Needed for WS-Notification -->
        <dependency>
            <groupId>org.apache.cxf.services.wsn</groupId>
            <artifactId>cxf-services-wsn-api</artifactId>
        </dependency>
    </dependencies>
    ...
</project>
```



NOTE

There is no need to specify the version of this artifact, because the version is provided by the Fabric8 BOM, which uses Maven dependency management to specify the artifact versions.

4. Delete the **cxf-java2ws-plugin** plug-in configuration from the **wsn-publisher/pom.xml** file. That is, open the **pom.xml** file and delete the **cxf-java2ws-plugin** plug-in configuration as highlighted in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<project ...>
  ...
  <build>
   <plugins>
    ...
    <!-- DELETE THE FOLLOWING LINES! -->
    <plugin>
     <groupId>org.apache.cxf</groupId>
     <artifactId>cxf-java2ws-plugin</artifactId>
     <version>${cxf-version}</version>
     <executions>
      <execution>
       <id>process-classes</id>
       <phase>process-classes</phase>
       <goals>
        <goal>java2ws</goal>
       </goals>
       <configuration>
        <className>org.jboss.fuse.example.wsn.publisher.HelloWorld</className>
        <genWsdl>true</genWsdl>
        <attachWsdl>false</attachWsdl>
        <verbose>true</verbose>
       </configuration>
      </execution>
     </executions>
    </plugin>
    ...
   </plugins>
  </build>
  ...
</project>
```

5. Add a **client** profile to the POM file, which provides an easy way to run the publisher client code. Edit the **wsn-publisher/pom.xml** file and add the new **profile** element, as highlighted in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<project ...>
  ...
  <profiles>
    ...
    <profile>
      <id>client</id>
      <build>
        <defaultGoal>test</defaultGoal>
        <plugins>
          <plugin>
            <groupId>org.codehaus.mojo</groupId>
            <artifactId>exec-maven-plugin</artifactId>
            <executions>
              <execution>
                <phase>test</phase>
                <goals>
                  <goal>java</goal>
```

```
                    </goals>
                    <configuration>

<mainClass>org.jboss.fuse.example.wsn.publisher.Client</mainClass>
                    </configuration>
                  </execution>
                </executions>
              </plugin>
            </plugins>
          </build>
        </profile>
    </profiles>
    ...
</project>
```

6. Create a new **Client.java** file in the **wsn-publisher/src/main/java/org/jboss/fuse/example/wsn/publisher/** directory, and add the code from Example 2.1, "Publisher Client Code" to this file.

7. You can now run the publisher client from the **wsn-publisher** directory by entering the following command:

   ```
   mvn -Pclient
   ```

   In the command window, you should see some output like the following:

   ```
   [INFO] --- exec-maven-plugin:1.4.0:java (default) @ wsn-publisher ---
   ```

   Notification messages are now accumulating in the broker, but you will not be able to receive the messages until you create a consumer client.

2.3. CREATE A CONSUMER CLIENT

Overview

This section describes how to create a consumer client of the notification broker. The consumer client subscribes to a particular topic and creates a callback service, which is capable of receiving messages directly from the broker.

Sample consumer client code

Example 2.2, "Consumer Client Code" shows the code for a sample consumer client that subscribes to messages published on the **MyTopic** topic.

Example 2.2. Consumer Client Code

```
// Java
package org.jboss.fuse.example.wsn.consumer;

import javax.xml.bind.JAXBElement;
import javax.xml.namespace.QName;

import org.w3c.dom.Element;
```

```
import org.apache.cxf.wsn.client.Consumer;
import org.apache.cxf.wsn.client.NotificationBroker;
import org.apache.cxf.wsn.client.Subscription;
import org.oasis_open.docs.wsn.b_2.NotificationMessageHolderType;

/**
 *
 */
public final class Client {
    private Client() {
        //not constructed
    }

    /**
     * @param args
     */
    public static void main(String[] args) throws Exception {
        String wsnPort = "8182";
        if (args.length > 0) {
            wsnPort = args[0];
        }

        // Start a consumer that will listen for notification messages
        // We'll just print the text content out for now.
        Consumer consumer = new Consumer(new Consumer.Callback() {
            public void notify(NotificationMessageHolderType message) {
                Object o = message.getMessage().getAny();
                System.out.println(message.getMessage().getAny());
                if (o instanceof Element) {
                    System.out.println(((Element)o).getTextContent());
                }
            }
        }, "http://localhost:9001/MyConsumer");


        // Create a subscription for a Topic on the broker
        NotificationBroker notificationBroker
            = new NotificationBroker("http://localhost:" + wsnPort + "/wsn/NotificationBroker");
        Subscription subscription = notificationBroker.subscribe(consumer, "MyTopic");


        // Just sleep for a bit to pick up some incoming messages
        Thread.sleep(60000);

        // Cleanup and exit
        subscription.unsubscribe();
        consumer.stop();
        System.exit(0);
    }

}
```

Creating a consumer callback object

In order to receive notification messages from the notification broker, you must create a consumer callback object to receive the messages. The consumer callback object is in fact a Web service which is embedded in your client. The easiest way to create the consumer callback is to use the **org.apache.cxf.wsn.client.Consumer** class from the simplified client API, which enables you to define a callback as follows:

```
// Start a consumer that will listen for notification messages
// We'll just print the text content out for now.
Consumer consumer = new Consumer(new Consumer.Callback() {
    public void notify(NotificationMessageHolderType message) {
        Object o = message.getMessage().getAny();
        System.out.println(message.getMessage().getAny());
        if (o instanceof Element) {
            System.out.println(((Element)o).getTextContent());
        }
    }
}, "http://localhost:9001/MyConsumer");
```

The first argument to the **Consumer** constructor is a reference to the consumer callback object, which is defined inline. The second argument specifies the URL of the consumer callback endpoint, which can receive messages from the notification broker.

Subscribing to a topic

To start receiving messages, you must subscribe the consumer to a topic in the notification broker. To create a subscription, invoke the following **subscribe** method on the **NotificationBroker** proxy object:

```
Subscription subscribe(Referencable consumer, String topic)
```

The first argument is a reference to a **Consumer** object (which is capable of returning a WS-Addressing endpoint reference to the consumer callback through the **Referencable.getEpr()** method). The second argument is the name of the topic you want to subscribe to.

The return value is a reference to a **Subscription** object, which you can use to manage the subscription (for example, pause, resume, or unsubscribe).

Threading in the consumer client

Because the consumer client has an embedded Web service (the consumer callback object), which automatically starts in a background thread, it is necessary to manage threading in this sample client. In particular, after creating the subscription, you need to put the main thread to sleep (by calling **Thread.sleep(60000)**), so that the thread context can switch to the background thread, where the callback Web service is running. This makes it possible for the consumer callback to receive some messages.

Steps to create a consumer client

Perform the following steps to create a consumer client:

1. Use the **archetype:generate** goal to invoke the **servicemix-cxf-code-first-osgi-bundle** archetype. Under the **wsn** directory, invoke the Maven archetype as follows:

```
mvn archetype:generate \
  -DarchetypeGroupId=io.fabric8.archetypes \
```

```
-DarchetypeArtifactId=karaf-soap-archetype \
-DarchetypeVersion=1.2.0.redhat-630187 \
-DgroupId=org.jboss.fuse.example \
-DartifactId=wsn-consumer \
-Dversion=1.0-SNAPSHOT \
-Dpackage=org.jboss.fuse.example.wsn.consumer \
-Dfabric8-profile=wsn-consumer-profile
```

NOTE

The backslash characters at the end of each line are effective as line-continuation characters on UNIX and LINUX platforms. If you are entering the command on a Windows platform, however, you must enter the entire command on a single line.

You will be prompted to confirm the project settings, with a message similar to this one:

```
Confirm properties configuration:
groupId: org.jboss.fuse.example
artifactId: wsn-consumer
version: 1.0-SNAPSHOT
package: org.jboss.fuse.example.wsn.consumer
fabric8-profile: wsn-consumer-profile
 Y: :
```

Type **Return** to accept the settings and generate the project. When the command finishes, you should find a new Maven project in the **wsn/wsn-consumer** directory.

2. Some of the generated project files are not needed for this tutorial. Under the **wsn/wsn-consumer** directory, delete the following files and directories:

```
src/main/resources/OSGI-INF/blueprint/blueprint.xml
src/main/java/org/jboss/fuse/example/wsn/consumer/HelloWorld.java
src/main/java/org/jboss/fuse/example/wsn/consumer/HelloWorldImpl.java
```

3. Edit the **pom.xml** file in the **wsn-consumer** directory, and add the following dependencies, as required by the consumer client:

```
<?xml version="1.0" encoding="UTF-8"?>
<project ...>
    ...
    <dependencies>
        ...
        <!-- Needed for HTTP callback endpoint -->
        <dependency>
            <groupId>org.apache.cxf</groupId>
            <artifactId>cxf-rt-transports-http-jetty</artifactId>
        </dependency>

        <!-- Needed for WS-Notification -->
        <dependency>
            <groupId>org.apache.cxf.services.wsn</groupId>
            <artifactId>cxf-services-wsn-api</artifactId>
        </dependency>
```

```
    </dependencies>
    ...
</project>
```

4. Delete the **cxf-java2ws-plugin** plug-in configuration from the **wsn-consumer/pom.xml** file. That is, open the **pom.xml** file and delete the **cxf-java2ws-plugin** plug-in configuration as highlighted in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<project ...>
    ...
    <build>
      <plugins>
        ...
       <!-- DELETE THE FOLLOWING LINES! -->
        <plugin>
          <groupId>org.apache.cxf</groupId>
          <artifactId>cxf-java2ws-plugin</artifactId>
          <version>${cxf-version}</version>
          <executions>
            <execution>
              <id>process-classes</id>
              <phase>process-classes</phase>
              <goals>
                <goal>java2ws</goal>
              </goals>
              <configuration>

<className>org.jboss.fuse.example.wsn.consumer.HelloWorld</className>
                <genWsdl>true</genWsdl>
                <attachWsdl>false</attachWsdl>
                <verbose>true</verbose>
              </configuration>
            </execution>
          </executions>
        </plugin>
        ...
      </plugins>
    </build>
    ...
</project>
```

5. Add a **client** profile to the POM file, which provides an easy way to run the publisher client code. Edit the **wsn-consumer/pom.xml** file and add the new **profile** element, as highlighted in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<project ...>
    ...
    <profiles>
       ...
       <profile>
          <id>client</id>
          <build>
             <defaultGoal>test</defaultGoal>
```

```
            <plugins>
              <plugin>
                <groupId>org.codehaus.mojo</groupId>
                <artifactId>exec-maven-plugin</artifactId>
                <executions>
                  <execution>
                    <phase>test</phase>
                    <goals>
                      <goal>java</goal>
                    </goals>
                    <configuration>

<mainClass>org.jboss.fuse.example.wsn.consumer.Client</mainClass>
                    </configuration>
                  </execution>
                </executions>
              </plugin>
            </plugins>
          </build>
        </profile>
      </profiles>
      ...
</project>
```

6. Create a new **Client.java** file in the**wsn-consumer/src/main/java/org/jboss/fuse/example/wsn/consumer/** directory, and add the code from Example 2.2, "Consumer Client Code" to this file.

Test the consumer client

Test the consumer client as follows:

1. If the JBoss A-MQ container is not already running (with the notification broker installed), start it up now:

   ```
   ./amq
   ```

2. Run the publisher client at the command line. Open a new command prompt, and enter the following commands:

   ```
   cd wsn/wsn-publisher
   mvn -Pclient
   ```

   In the command window, you should see some output like the following:

   ```
   ...
   INFO: Creating Service {http://cxf.apache.org/wsn/jaxws}NotificationBrokerService
   from WSDL: jar:file:/Users/fbolton/.m2/repository/org/apache/cxf/services/wsn/
   cxf-services-wsn-api/2.6.0.redhat-60024/cxf-services-wsn-api-2.6.0.redhat-60024.jar
   !/org/apache/cxf/wsn/wsdl/wsn.wsdl
   ```

   You now have approximately two minutes before the publisher client times out.

3. Run the consumer client at the command line. Open a new command prompt and enter the following commands:

```
cd wsn/wsn-consumer
mvn -Pclient
```

In the command window, you should see some output like the following:

```
...
[INFO] --- exec-maven-plugin:1.4.0:java (default) @ wsn-consumer ---
[foo: null]
Hello World!
...
```

4. To inspect the state of the notification broker, you can connect to the JMX port of the ActiveMQ broker. Start up a JMX console by entering the following command at the command line:

```
jconsole
```

In the **JConsole: New Connection** dialog, select **Remote Process** and enter the following URL in the accompanying text field:

```
service:jmx:rmi:///jndi/rmi://localhost:1099/karaf-root
```

In the **Username** and **Password** fields, enter one of the user credentials you created at the start of this tutorial. When you are connected to the JMX port, you can inspect the state of the broker by clicking on the **MBeans** tab and drilling down the object tree in the JConsole.

2.4. CREATE A PULLPOINT CLIENT

Overview

This section describes how to create a pull-point client of the notification broker. The pull-point client first creates a remote pull-point (which is used to accumulate messages), then subscribes the pull-point to a particular topic. Finally, the pull-point client retrieves the accumulated messages from the pull-point.

Sample PullPoint client code

Example 2.3, "PullPoint Client Code" shows the code for a sample pull-point client that subscribes to messages published on the **MyTopic** topic.

Example 2.3. PullPoint Client Code

```
// Java
package org.jboss.fuse.example.wsn.pullpoint;

import java.util.List;
import java.util.Iterator;

import javax.xml.bind.JAXBElement;
import javax.xml.namespace.QName;
```

```
import org.w3c.dom.Element;

import org.apache.cxf.wsn.client.PullPoint;
import org.apache.cxf.wsn.client.NotificationBroker;
import org.apache.cxf.wsn.client.CreatePullPoint;
import org.apache.cxf.wsn.client.Subscription;
import org.oasis_open.docs.wsn.b_2.NotificationMessageHolderType;

/**
 *
 */
public final class Client {
    private Client() {
        //not constructed
    }

    /**
     * @param args
     */
    public static void main(String[] args) throws Exception {
        String wsnPort = "8182";
        if (args.length > 0) {
            wsnPort = args[0];
        }

        // Create a PullPoint
        CreatePullPoint createPullPoint
            = new CreatePullPoint("http://localhost:" + wsnPort + "/wsn/CreatePullPoint");
        PullPoint pullPoint = createPullPoint.create();

        // Create a PullPoint style subscription
        NotificationBroker notificationBroker
            = new NotificationBroker("http://localhost:" + wsnPort + "/wsn/NotificationBroker");
        Subscription subscription = notificationBroker.subscribe(pullPoint, "MyTopic");

        // Wait for some messages to accumulate in the pull point
        Thread.sleep(10000);

        // Now retrieve messages from the pull point
        List<NotificationMessageHolderType> messages = pullPoint.getMessages(10);

        if (!messages.isEmpty()) {
            Iterator<NotificationMessageHolderType> messageIterator = messages.iterator();
            while (messageIterator.hasNext()) {
                NotificationMessageHolderType messageH = messageIterator.next();
                Object o = messageH.getMessage().getAny();
                System.out.println(messageH.getMessage().getAny());
                if (o instanceof Element) {
                    System.out.println(((Element)o).getTextContent());
                }
            }
        }
        else {
            System.out.println("Warn: message list is empty!");
        }
```

```
        subscription.unsubscribe();
        pullPoint.destroy();

        System.exit(0);
    }

}
```

Steps to create a pullpoint client

Perform the following steps to create a PullPoint client:

1. Use the **archetype:generate** goal to invoke the**servicemix-cxf-code-first-osgi-bundle** archetype. Under the **wsn** directory, invoke the Maven archetype as follows:

```
mvn archetype:generate \
  -DarchetypeGroupId=io.fabric8.archetypes \
  -DarchetypeArtifactId=karaf-soap-archetype \
  -DarchetypeVersion=1.2.0.redhat-630187 \
  -DgroupId=org.jboss.fuse.example \
  -DartifactId=wsn-pullpoint \
  -Dversion=1.0-SNAPSHOT \
  -Dpackage=org.jboss.fuse.example.wsn.pullpoint \
  -Dfabric8-profile=wsn-pullpoint-profile
```

NOTE

The backslash characters at the end of each line are effective as line-continuation characters on UNIX and LINUX platforms. If you are entering the command on a Windows platform, however, you must enter the entire command on a single line.

You will be prompted to confirm the project settings, with a message similar to this one:

```
Confirm properties configuration:
groupId: org.jboss.fuse.example
artifactId: wsn-pullpoint
version: 1.0-SNAPSHOT
package: org.jboss.fuse.example.wsn.pullpoint
fabric8-profile: wsn-pullpoint-profile
 Y: :
```

Type **Return** to accept the settings and generate the project. When the command finishes, you should find a new Maven project in the **wsn**/**wsn-pullpoint** directory.

2. Some of the generated project files are not needed for this tutorial. Under the **wsn**/**wsn-pullpoint** directory, delete the following files and directories:

```
src/main/resources/OSGI-INF/blueprint/blueprint.xml
src/main/java/org/jboss/fuse/example/wsn/pullpoint/HelloWorld.java
src/main/java/org/jboss/fuse/example/wsn/pullpoint/HelloWorldImpl.java
```

3. Edit the **pom.xml** file in the **wsn-pullpoint** directory, and add the following dependency required for WS-Notification clients:

```
<?xml version="1.0" encoding="UTF-8"?>
<project ...>
  ...
  <dependencies>
    ...
    <!-- Needed for WS-Notification -->
    <dependency>
      <groupId>org.apache.cxf.services.wsn</groupId>
      <artifactId>cxf-services-wsn-api</artifactId>
    </dependency>
  </dependencies>
  ...
</project>
```

4. Delete the **cxf-java2ws-plugin** plug-in configuration from the **wsn-pullpoint/pom.xml** file. That is, open the **pom.xml** file and delete the **cxf-java2ws-plugin** plug-in configuration as highlighted in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<project ...>
  ...
  <build>
   <plugins>
    ...
   <!-- DELETE THE FOLLOWING LINES! -->
    <plugin>
     <groupId>org.apache.cxf</groupId>
     <artifactId>cxf-java2ws-plugin</artifactId>
     <version>${cxf-version}</version>
     <executions>
       <execution>
         <id>process-classes</id>
         <phase>process-classes</phase>
         <goals>
           <goal>java2ws</goal>
         </goals>
         <configuration>
           <className>org.jboss.fuse.example.wsn.pullpoint.HelloWorld</className>
           <genWsdl>true</genWsdl>
           <attachWsdl>false</attachWsdl>
           <verbose>true</verbose>
         </configuration>
       </execution>
     </executions>
    </plugin>
    ...
   </plugins>
  </build>
  ...
</project>
```

5. Add a **client** profile to the POM file, which provides an easy way to run the publisher client code. Edit the **wsn-pullpoint/pom.xml** file and add the new**profile** element, as highlighted in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<project ...>
  ...
  <profiles>
    ...
    <profile>
      <id>client</id>
      <build>
        <defaultGoal>test</defaultGoal>
        <plugins>
          <plugin>
            <groupId>org.codehaus.mojo</groupId>
            <artifactId>exec-maven-plugin</artifactId>
            <executions>
              <execution>
                <phase>test</phase>
                <goals>
                  <goal>java</goal>
                </goals>
                <configuration>

<mainClass>org.jboss.fuse.example.wsn.pullpoint.Client</mainClass>
                </configuration>
              </execution>
            </executions>
          </plugin>
        </plugins>
      </build>
    </profile>
  </profiles>
  ...
</project>
```

6. Create a new **Client.java** file in the**wsn-pullpoint/src/main/java/org/jboss/fuse/example/wsn/pullpoint/** directory, and add the code from Example 2.3, "PullPoint Client Code" to this file.

Test the PullPoint client

Test the PullPoint client as follows:

1. If the JBoss A-MQ container is not already running (with the notification broker installed), start it up now:

```
./amq
```

2. Run the publisher client at the command line. Open a new command prompt, and enter the following commands:

```
cd wsn/wsn-publisher
mvn -Pclient
```

In the command window, you should see some output like the following:

```
...
[INFO] Tests are skipped.
[INFO]
[INFO] --- exec-maven-plugin:1.4.0:java (default) @ wsn-publisher ---
```

You now have approximately two minutes before the publisher client times out.

3. Run the PullPoint client at the command line. Open a new command prompt and enter the following commands:

```
cd wsn/wsn-pullpoint
mvn -Pclient
```

After a ten second delay, you should see some output like the following:

```
...
[INFO] Tests are skipped.
[INFO]
[INFO] --- exec-maven-plugin:1.4.0:java (default) @ wsn-pullpoint ---
[foo: null]
Hello World!
[foo: null]
Hello World!
...
```

4. To inspect the state of the notification broker, you can connect to the JMX port of the ActiveMQ broker. Start up a JMX console by entering the following command at the command line:

```
jconsole
```

In the **JConsole: New Connection** dialog, select **Remote Process** and enter the following URL in the accompanying text field:

```
service:jmx:rmi:///jndi/rmi://localhost:1099/karaf-root
```

In the **Username** and **Password** fields, enter one of the user credentials you created at the start of this tutorial. When you are connected to the JMX port, you can inspect the state of the broker by clicking on the **MBeans** tab and drilling down the object tree in the JConsole.