Fuse ESB Enterprise

Using the JMS Binding Component

Version 7.1
December 2012

Integration Everywhere

Using the JMS Binding Component

Version 7.1

Updated: 08 Jan 2014

Trademark Disclaimer

Third Party Acknowledgements

One or more products in the Fuse ESB Enterprise release includes third party components covered by licenses that require that the following documentation notices be provided:

- JLine (http://jline.sourceforge.net) jline:jline:jar:1.0

  License: BSD (LICENSE.txt) - Copyright (c) 2002-2006, Marc Prud'hommeaux <mwp1@cornell.edu>

  All rights reserved.

  Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

  - Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
  - Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
  - Neither the name of JLine nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

- Stax2 API (http://woodstox.codehaus.org/StAX2) org.codehaus.woodstox:stax2-api:jar:3.1.1

  License: The BSD License (http://www.opensource.org/licenses/bsd-license.php)

  Copyright (c) <YEAR>, <OWNER> All rights reserved.

  Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

  - Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
  - Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

- jibx-run - JiBX runtime (http://www.jibx.org/main-reactor/jibx-run) org.jibx:jibx-run:bundle:1.2.3

  License: BSD (http://jibx.sourceforge.net/jibx-license.html) Copyright (c) 2003-2010, Dennis M. Sosnoski.

  All rights reserved.

  Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

  - Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
  - Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
  - Neither the name of JiBX nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

- JavaAssist (http://www.jboss.org/javassist) org.jboss.javassist:com.springsource.javassist:jar:3.9.0.GA:compile

  License: MPL (http://www.mozilla.org/MPL/MPL-1.1.html)

- HAPI-OSGI-Base Module (http://hl7api.sourceforge.net/hapi-osgi-base/) ca.uhn.hapi:hapi-osgi-base:bundle:1.2

  License: Mozilla Public License 1.1 (http://www.mozilla.org/MPL/MPL-1.1.txt)

Table of Contents

List of Figures

List of Tables

List of Examples

Chapter 1. Introduction to the Fuse ESB Enterprise JMS Binding Component

The JMS binding component allows you to create endpoints that interact with JMS destinations outside of the Fuse ESB Enterprise's runtime environment. It provides a robust and highly configurable means to interact with JMS systems.

Overview

The Fuse ESB Enterprise JMS binding component is built using the Spring 2.0 JMS framework. It allows you to create two types of endpoints:

Consumer Endpoints
: A *Consumer* endpoint's primary roll is to listen for messages on an external JMS destination and pass them into to the NMR for delivery to endpoints inside of the Fuse ESB Enterprise container. Consumer endpoints can send responses if one is required.

Provider Endpoints
: A *Provider* endpoint's primary roll is to take messages from the NMR and send them to an external JMS destination.

Note

The JMS binding component also supports non-Spring based endpoints. However, the non-Spring based endpoints are deprecated.

In most instances, you do not need to write any Java code to create endpoints. All of the configuration is done using Spring XML that is placed in an `xbean.xml` file. There are some instances where you will need to develop your own Java classes to supplement the basic functionality provided by the binding components default implementations. These cases are discussed at the end of this guide.

Key features

The Fuse ESB Enterprise JMS binding component provides a number of enterprise quality features including:

- Support for JMS 1.0.2 and JMS 1.1
- JMS transactions
- XA transactions

- Support of all MEP patterns
- SOAP support
- MIME support
- Customizable message marshaling

Contents of a JMS service unit

A service unit that configures the JMS binding component will contain two artifacts:

`xbean.xml`

The `xbean.xml` file contains the XML configuration for the endpoint defined by the service unit. The contents of this file are the focus of this guide.

Note

The service unit can define more than one endpoint.

`meta-inf/jbi.xml`

The `jbi.xml` file is the JBI descriptor for the service unit. Example 1.1 on page 14 shows a JBI descriptor for a JMS service unit.

Example 1.1. JBI Descriptor for a JMS Service Unit

```
<jbi xmlns="http://java.sun.com/xml/ns/jbi" version="1.0">
  <services binding-component="false" ❶
            xmlns:b="http://servicemix.apache.org/samples/bridge"> ❷
    <provides service-name="b:jms" ❸
              endpoint-name="endpoint"/> ❹
    <consumes interface-name="b:MyConsumerInterface"/>  ❺
  </services>
</jbi>
```

The elements shown in Example 1.1 on page 14 do the following:

❶ The `service` element is the root element of all service unit descriptors. The value of the `binding-component` attribute is always `false`.

❷ The `service` element contains namespace references for all of the namespaces defined in the `xbean.xml` file's `bean` element.

❸ The `provides` element corresponds to a JMS provider endpoint. The `service-name` attribute derives its value from the `service` attribute in the JMS provider's configuration.

Note

This attribute can also appear on a `consumes` element.

❹ The `endpoint-name` attribute derives its value from the `endpoint` attribute in the JMS provider's configuration.

Note

This attribute can also appear on a `consumes` element.

❺ The `consumes` element corresponds to a JMS consumer endpoint. The `interface-name` attribute derives its value from the `interfaceName` attribute in the JMS consumer's configuration.

Note

This attribute can also appear on a `provides` element.

Using the Maven JBI tooling

The Fuse ESB Enterprise Maven tooling provides two archetypes for seeding a project whose result is a service unit for the JMS binding component:

servicemix-jms-consumer-endpoint

The **servicemix-jms-consumer-endpoint** archetype creates a project that results in a service unit that configures a JMS consumer endpoint.

Tip

You can use the **smx-arch** command to in place of typing the entire Maven command.

```
smx-arch su jms-consumer ["-DgroupId=my.group.id"]
["-DartifactId=my.artifact.id"]
```

servicemix-jms-provider-endpoint

The **servicemix-jms-provider-endpoint** archetype creates a project that results in a service unit that configures a JMS provider endpoint.

Tip

You can use the **smx-arch** command to in place of typing the entire Maven command.

```
smx-arch su jms-provider ["-DgroupId=my.group.id"]
["-DartifactId=my.artifact.id"]
```

The resulting project will contain two generated artifacts:

- a `pom.xml` file containing the metadata needed to generate and package the service unit
- a `src/main/resources/xbean.xml` file containing the configuration for the endpoint

Important

The endpoint configuration generated by the archetype is for the deprecated JMS endpoints. While this configuration will work, it is not recommended for new projects and is not covered in this guide.

If you want to add custom marshalers, custom destination choosers, or other custom Java code, you must add a `java` folder to the generated `src` folder. You also need to modify the generated `pom.xml` file to compile the code and package it with the service unit.

OSGi Packaging

To package JMS endpoints as OSGi bundles you need to make two minor changes:

- include an OSGi bundle manifest in the `META-INF` folder of the bundle

- add the following to your service unit's configuration file:

```
<bean class="org.apache.servicemix.common.osgi.EndpointExporter" />
```

Important

When you deploy JMS endpoints in an OSGi bundle, the resulting endpoints are deployed as a JBI service unit.

For more information on using the OSGi packaging see Appendix D on page 121.

Namespace

The elements used to configure JMS endpoints are defined in the http://servicemix.apache.org/jms/1.0 namespace. You will need to add a namespace declaration similar to the one in Example 1.2 on page 17 to your `xbeans.xml` file's `beans` element.

Example 1.2. Namespace Declaration for Using JMS Endpoints

```
<beans ...
      xmlns:jms="http://servicemix.apache.org/jms/1.0"
      ... >
  ...
</beans>
```

In addition, you need to add the schema location to the Spring `beans` element's `xsi:schemaLocation` as shown in Example 1.3 on page 17.

Example 1.3. Schema Location for Using JMS Endpoints

```
<beans ...
       xsi:schemaLocation="...
http://servicemix.apache.org/jms/1.0 http://servicemix.apache.org/jms/1.0/servicemix-jms.xsd
...">
  ...
</beans>
```

Chapter 2. Configuring the Connection Factory

The JMS binding component needs to have access to your JMS provider's connection factory. This is configured in the XML file and the specifics depend on the JMS provider in use.

When working with a JMS broker, a client application needs a `ConnectionFactory` object to create connections to the broker. The `ConnectionFactory` object is a JMS object that is provided along with the JMS broker. Each JMS provider has a unique `ConnectionFactory` object that uses properties specific to a particular JMS implementation.

When using the Fuse ESB Enterprise JMS binding component, you must configure each service unit with the information it needs to load a `ConnectionFactory` object. Often the `ConnectionFactory` object is looked up through JNDI. However, the information needed depends on the JMS provider you are using.

Commonly used JMS providers include Apache ActiveMQ, Apache ActiveMQ, IBM's WebShere® MQ, BEA's WebLogic®, and Progress Software's SonicMQ®. Apache ActiveMQ and Apache ActiveMQ can be configured using simple Spring XML. Other JMS providers must be configured using either JNDI or using custom Spring beans. This chapter provides basic information for configuring the `ConnectionFactory` objects for each of these platforms.

Using Apache ActiveMQ and Apache ActiveMQ Connection Factories

Overview

The recommended method for creating connections to Apache ActiveMQ, or Apache ActiveMQ, is by using the Jencks AMQPool. It provides support for using a scalable pool of connections for managing overhead. You can download the needed jar from http://repo1.maven.org/maven2/org/jencks/jencks-amqpool/2.0/jencks-amqpool-2.0.jar. Once the jar is downloaded, you need to add it to your classpath. The easiest way to do this is to place the jar into your *InstallDir*\lib folder.

Note

The examples included with Fuse ESB Enterprise use the standard Apache ActiveMQ connection factory. This is fine for testing purposes, but is not robust enough for enterprise deployments.

The Jencks AMQPool supplies three connection factories:

- simple
- XA
- JCA

Namespace

To add the AMQPool configuration elements to your endpoint's configuration, you need to add the following XML namespace declaration to your `beans` element:

```
xmlns:amqpool="http://jencks.org/amqpool/2.0"
```

Simple pool

The simple pooling connection factory supports pooling, but does not support transactions. It is specified using the `amqpool:pool` element. The attributes used to configure the simple pooled connection factory are described in Table 2.1 on page 21.

Table 2.1. Attributes for Configuring the Simple AMQPool Connection Factory

Attribute	Description	Required
`id`	Specifies a unique identifier by which other elements refer to this element.	yes
`url`	Specifies the URL used to connect to the JMS broker.	yes
`maxConnections`	Specifies the maximum number of simultaneous connections to the broker. The default value is `1`, but you can safely increase it to `8` in all conditions.	no
`maximumActive`	Specifies the maximum number of active sessions for a particular connection. The default value is `500`.	no

Example 2.1 on page 21 shows a configuration snippet for configuring the simple AMQPool connection factory.

Example 2.1. Configuring a Simple AMQPool Connection Factory

```
<beans xmlns:amqpool="http://jencks.org/amqpool/2.0"
       ... >
  ...
  <amqpool:pool id="connectionFactory"
                url="tcp://localhost:61616"
                maxConnections="8" />
</beans>
```

XA pool

The XA pooling connection factory supports XA transactions and late enlistment. It is specified using the `amqpool:xa-pool` element. The attributes used to configure the XA pooled connection factory are described in Table 2.2 on page 21.

Table 2.2. Attributes for Configuring the XA AMQPool Connection Factory

Attribute	Description	Required
`id`	Specifies a unique identifier by which other elements refer to this element.	yes

Attribute	Description	Required
`url`	Specifies the URL used to connect to the JMS broker.	yes
`transactionManager`	Specifies a reference to an element that configures an XA transaction manager.	yes
`maxConnections`	Specifies the maximum number of simultaneous connections to the broker. The default value is `1`, but you can safely increase it to `8` in all conditions.	no
`maximumActive`	Specifies the maximum number of active sessions for a particular connection. The default value is `500`.	no

Example 2.2 on page 22 shows a configuration snippet for configuring an XA AMQPool connection factory.

Example 2.2. Configuring an XA AMQPool Connection Factory

```
<beans xmlns:amqpool="http://jencks.org/amqpool/2.0"
       xmlns:jencks="http://jencks.org/2.0"
       ... >
  ...
  <amqpool:xa-pool id="connectionFactory"
                   url="tcp://localhost:61616"
                   maxConnections="8"
                   transactionManager="#transactionManager" />

  <jencks:transactionManager id="transactionManager"
                             transactionLogDir="./data/txlog"
                             defaultTransactionTimeoutSeconds="600" />

</beans>
```

JCA pool

The JCA pooling connection factory is intended to be used inside of J2EE environments or in conjunction with the Jencks JCA environment. It is specified using the `amqpool:jca-pool` element. The attributes used to configure the JCA pooled connection factory are described in Table 2.3 on page 23.

Table 2.3. Attributes for Configuring the JCA AMQPool Connection Factory

Attribute	Description	Required
`id`	Specifies a unique identifier by which other elements refer to this element.	yes
`url`	Specifies the URL used to connect to the JMS broker.	yes
`transactionManager`	Specifies a reference to an element that configures an XA transaction manager.	yes
`name`	Specifies a unique name by which the JMS broker can be identified.	yes
`maxConnections`	Specifies the maximum number of simultaneous connections to the broker. The default value is `1`, but you can safely increase it to `8` in all conditions.	no
`maximumActive`	Specifies the maximum number of active sessions for a particular connection. The default value is `500`.	no

Example 2.3 on page 23 shows a configuration snippet for configuring the JCA AMQPool connection factory.

Example 2.3. Configuring a JCA AMQPool Connection Factory

```
<beans xmlns:amqpool="http://jencks.org/amqpool/2.0"
       xmlns:jencks="http://jencks.org/2.0"
       ... >
  ...
  <amqpool:jca-pool id="connectionFactory"
                    url="tcp://localhost:61616"
                    maxConnections="8"
                    transactionManager="#transactionManager"
                    name="joeFred" />

  <jencks:transactionManager id="transactionManager"
                             transactionLogDir="./data/txlog"
                             defaultTransactionTimeoutSeconds="600" />

</beans>
```

Using JNDI

Overview

Many JMS providers store a reference to their connection factory in a JNDI service to ease retrieval. Fuse ESB Enterprise allows developers to choose between a straight JNDI look-up and using Spring JNDI templates. Which mechanism you choose will depend on your environment.

Spring JEE JNDI lookup

Spring provides a built-in JNDI look-up feature that can be used to retrieve the connection factory for a JMS provider. To use the built-in JNDI look-up do the following:

1. Add the following namespace declaration to your `beans` element in your service unit's configuration.

   ```
   xmlns:jee="http://www.springframework.org/schema/jee"
   ```

2. Add a `jee:jndi-lookup` element to your service unit's configuration.

   The `jee:jndi-lookup` element has two attributes. They are described in Table 2.4 on page 24.

   Table 2.4. Attributes for Using Spring's JEE JNDI Lookup

Attribute	Description
`id`	Specifies a unique identifier by which the JMS endpoints will reference the connection factory.
`jndi-name`	Specifies the JNDI name of the connection factory.

3. Add a `jee:environment` child element to the `jee:jndi-lookup` element.

   The `jee:environment` element contains a collection of Java properties that are used to access the JNDI provider. These properties will be provided by your JNDI provider's documentation.

Example 2.4 on page 25 shows a configuration snippet for using the JNDI look-up with WebLogic.

Example 2.4. Getting the WebLogic Connection Factory Using Spring's JEE JNDI Look-up

```
<beans xmlns:jee="http://www.springframework.org/schema/jee" ... >
  ...
  <jee:jndi-lookup id="connectionFactory" jndi-name="weblogic.jms.XAConnectionFactory">
    <jee:environment>
      java.naming.factory.initial=weblogic.jndi.WLInitialContextFactory
      java.naming.provider.url=t3://localhost:7001
    </jee:environment>
  </jee:jndi-lookup>
  ...
</beans>
```

Spring JNDI Templates

Another approach to using JNDI to get a reference to a JMS connection factory is to use the Spring framework's `JndiTemplate` bean. Using this approach, you configure an instance of the `JndiTemple` bean and then use the bean to perform all of your JNDI look-ups using a `JndiObjectFactoryBean` bean.

To get the JMS connection factory using a Spring JNDI template do the following:

1. Add a `bean` element to your configuration for the JNDI template.
   a. Set the `bean` element's `id` attribute to a unique identifier.
   b. Set the `bean` element's `class` attribute to `org.springframework.jndi.JndiTemplate`.
   c. Add a `property` child element to the `bean` element.

      The `property` element will contain the properties for accessing the JNDI provider.
   d. Set the `property` element's `name` attribute to `environment`.
   e. Add a `props` child to the `property` element.
   f. Add a `prop` child element to the `props` element for each Java property needed to connect to the JNDI provider.

      A `prop` element has a single attribute called `key` whose value is the name of the Java property being set. The value of the element is

the value of the Java property being set. Example 2.5 on page 26 shows a `prop` element for setting the `java.naming.factory.initial` property.

Example 2.5. Setting a Java Property

```
<prop key="java.naming.factory.initial">
  com.sun.jndi.fscontext.RefFSContextFactory
</prop>
```

Note

The properties you need to set will be determined by your JNDI provider. Check its documentation.

2. Add a `bean` element to your configuration to retrieve the JMS connection factory using the JNDI template.
   a. Set the `bean` element's `id` attribute to a unique identifier.
   b. Set the `bean` element's `class` attribute to `org.springframework.jndi.JndiObjectFactoryBean`.
   c. Add a `property` child element to the `bean` element.

      This `property` element loads the JNDI template to be used for the look-up. You must set its `name` attribute to `jndiTemplate`. The value of its `ref` attribute is taken from the `name` attribute of the `bean` element that configured the JNDI template.
   d. Add a second `property` child element to the `bean` element.

      This `property` element specifies the JNDI name of the connection factory. You must set its `name` attribute to `jndiTemplate`.
   e. Add a `value` child element to the `property` element.

      The value of the element is the JNDI name of the connection factory.

Example 2.6 on page 27 shows a configuration fragment for retrieving the WebSphere MQ connection factory using Sun's reference JNDI implementation.

Example 2.6. Using a JNDI Template to Look Up a Connection Factory

```
<beans ... >
  ...
  <bean id="jndiTemplate"
        class="org.springframework.jndi.JndiTemplate">
    <property name="environment">
      <props>
        <prop key="java.naming.factory.initial">
          com.sun.jndi.fscontext.RefFSContextFactory
        </prop>
        <prop key="java.naming.provider.url">
          file:/tmp/
        </prop>
      </props>
    </property>
  </bean>

  <bean id="connectionFactory"
        class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiTemplate"
              ref="jndiTemplate" />
    <property name="jndiName">
      <value>MQConnFactory</value>
     </property>
  </bean>
  ...
</beans>
```

Using a Spring Bean

Overview

You can add your JMS provider's configuration factory directly into the service units configuration as a Spring bean. Configuring the connection factory in this manner requires that you fully specify all of the properties needed to instantiate a `ConnectionFactory` for your JMS provider.

Note

Your JMS provider's documentation will describe the properties needed to instantiate a connection factory and the settings for the properties.

Example

Example 2.7 on page 28 shows an example of a WebSphere MQ connection factory configured as a Spring bean.

Example 2.7. Configuring a Connection Factory with a Spring Bean

```
<bean id="connectionFactory" class="com.ibm.mq.jms.MQQueueConnectionFactory">
  <property name="transportType">
    <util:constant static-field="com.ibm.mq.jms.JMSC.MQJMS_TP_CLIENT_MQ_TCPIP" />
  </property>
  <property name="queueManager" value="my.queue.mgr" />
  <property name="hostName" value="myHost" />
  <property name="channel" value="myChannel" />
  <property name="port" value="12345" />
</bean>
```

Chapter 3. Creating a Consumer Endpoint

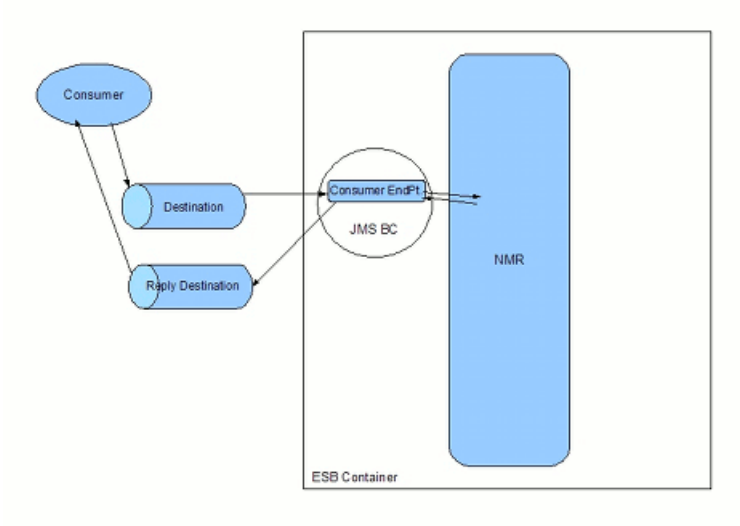
A consumer is an endpoint that listens for messages, passes the messages to the NMR, and sends any response that maybe generated back to the external JMS endpoint. They are built using the Spring framework's JMS `MessageListener` *interface.*

Introduction to Consumer Endpoints

Where does a consumer fit into a solution?

Consumer endpoints play the role of consumer from the vantage point of the other endpoints in the ESB. As shown in Figure 3.1 on page 30,consumer endpoints listen for messages on a JMS destination. When the message is received, the consumer endpoint passes it onto the NMR for delivery. If the JMS message is part of an in-out message exchange, the consumer endpoint will place that message into a reply destination for delivery to the originator of the JMS message.

Figure 3.1. Consumer Endpoint



Types of consumer endpoints

The JMS binding component offers three types of consumer endpoints:

Generic

The generic consumer endpoint can handle any type of message data. It is configured using the `jms:consumer` element.

SOAP

The SOAP consumer endpoint is specifically tailored to receive SOAP messages. It uses a WSDL document to define the structure of the messages. It is configured using the `jms:soap-consumer` element.

Tip

The Apache CXF binding component's JMS transport is better adapted to handling SOAP messages, but offers less control over the JMS connection.

JCA

The JCA consumer endpoint uses JCA to connect to the JMS provider. It is configured using the `jms:jca-consumer` element. For more information on using the JCA consumer endpoint, see "Using the JCA Consumer Endpoint" on page 44.

Using the Generic Endpoint or the SOAP Endpoint

Basic Configuration

Procedure

To configure a generic consumer or a SOAP consumer do the following:

1. Decide what type of consumer endpoint to use.

   See "Types of consumer endpoints" on page 30.

2. Specify the name of the service for which this endpoint is acting as a proxy.

   This is specified using the `service` attribute.

   Tip

   If you are using a SOAP consumer and your WSDL file only has one service defined, you do not need to specify the service name.

3. Specify the name of the endpoint for which this endpoint is acting as a proxy.

   This is specified using the `endpoint` attribute.

   Tip

   If you are using a SOAP consumer and your WSDL file only has one endpoint defined, you do not need to specify the endpoint name.

4. Specify the connection factory the endpoint will use.

   The endpoint's connection factory is configured using the endpoint's `connectionFactory` attribute. The `connectionFactory` attribute's value is a reference to the bean that configures the connection factory. For example, if the connection factory configuration bean is named `widgetConnectionFactory`, the value of the `connectionFactory` attribute would be `#widgetConnectionFactory`.

   For information on configuring a connection factory see "Configuring the Connection Factory" on page 19.

5. Specify the destination onto which the endpoint will place messages.

For more information see "Configuring a destination" on page 34.

6. Specify the ESB endpoint to which incoming messages are targeted.

   For more information see "Specifying the target endpoint" on page 35.

7. If you are using a JMS SOAP consumer, specify the location of the WSDL defining the message exchange using the `wsdl` attribute.

8. If your JMS destination is a topic, set the `pubSubDomaim` attribute to `true`.

9. If your endpoint is interacting with a broker that only supports JMS 1.0.2, set the `jms102` attribute to `true`.

Configuring a destination

A consumer endpoint chooses the destination to use for sending messages with the following algorithm:

1. The endpoint will check to see if you configured the destination explicitly.

   You configure a destination using a Spring bean. You can add the bean directly to the endpoint by wrapping it in a `jms:destination` child element. You can also configure the bean separately and refer the bean using the endpoint's `destination` attribute as shown in Example 3.1 on page 34.

Example 3.1. Configuring a Consumer's Destination

```
<beans xmlns:jms="http://servicemix.apache.org/jms/1.0"
       ... >
  ...
  <jms:consumer service="my:widgetService"
                endpoint="jbiWidget"
                destination="#widgetQueue"
                ... />
  ...
  <jee:jndi-lookup id="widgetQueue" jndi-name="my.widget.queue">
    <jee:environment>
      java.naming.factory.initial=weblogic.jndi.WLInitialContextFactory
      java.naming.provider.url=t3://localhost:7001
    </jee:environment>
  </jee:jndi-lookup>
  ...
</beans>
```

2. If you did not explicitly configure a destination, the endpoint will use the value of the `destinationName` attribute to choose its destination.

   The value of the `destinationName` attribute is a string that will be used as the name for the JMS destination. The binding component's default behavior when you provide a destination name is to resolve the destination using the standard JMS `Session.createTopic()` and `Session.createQueue()` methods.

   Note

   You can override the binding component's default behavior by providing a custom `DestinationResolver` implementation. See "Using a Custom Destination Resolver" on page 87.

Specifying the target endpoint

There are a number of attributes available for configuring the endpoint to which the generated messages are sent. The poller endpoint will determine the target endpoint in the following manner:

1. If you explicitly specify an endpoint using both the `targetService` attribute and the `targetEndpoint` attribute, the ESB will use that endpoint.

   The `targetService` attribute specifies the QName of a service deployed into the ESB. The `targetEndpoint` attribute specifies the name of an endpoint deployed by the service specified by the `targetService` attribute.

2. If you only specify a value for the `targetService` attribute, the ESB will attempt to find an appropriate endpoint on the specified service.

3. If you do not specify a service name or an endpoint name, you must specify an the name of an interface that can accept the message using the `targetInterface` attribute. The ESB will attempt to locate an endpoint that implements the specified interface and direct the messages to it.

   Interface names are specified as QNames. They correspond to the value of the `name` attribute of either a WSDL 1.1 `serviceType` element or a WSDL 2.0 `interface` element.

Important

If you specify values for more than one of the target attributes, the consumer endpoint will use the most specific information.

Examples

Example 3.2 on page 36 shows the basic configuration for a plain JMS provider endpoint.

Example 3.2. Basic Configuration for a Generic Consumer Endpoint

```
<beans xmlns:jms="http://servicemix.apache.org/jms/1.0"
       ... >
  ...
  <jms:consumer service="my:widgetService"
                endpoint="jbiWidget"
                destinationName="widgetQueue"
                connectionFactory="#connectionFactory"
                targetService="my:targetService" />
  ...
</beans>
```

Example 3.3 on page 36 shows the basic configuration for a SOAP JMS provider endpoint.

Example 3.3. Basic Configuration for a SOAP Consumer Endpoint

```
<beans xmlns:jms="http://servicemix.apache.org/jms/1.0"
       ... >
  ...
  <jms:soap-consumer wsdl="classpath:widgets.wsdl"
                     destinationName="widgetQueue"
                     connectionFactory="#connectionFactory"
                     targetService="my:targetService" />
  ...
</beans>
```

Listener Containers

Overview

Both the generic consumer endpoint and the SOAP consumer endpoint use Spring *listener containers* to handle incoming messages. The listener container handles the details of receiving messages from the destination, participating in transactions, and controlling the threads used to dispatch messages to the endpoint.

Types of listener containers

Fuse ESB Enterprise's JMS consumer endpoints support three types of listener containers:

Simple

The simple listener container creates a fixed number of JMS sessions at startup and uses them throughout the lifespan of the container. It cannot dynamically adapt to runtime conditions nor participate in externally managed transactions.

Default

The default listener container provides the best balance between placing requirements on the JMS provider and features. Because of this, it is the default listerner container for Fuse ESB Enterprise JMS consumer endpoints. The default listener container can adapt to changing runtime demands. It is also capable of participating in externally managed transactions.

Server session

The server session listener container leverages the JMS `ServerSessionPool` SPI to allow for dynamic management of JMS sessions. It provides the best runtime scaling and supports externally managed transactions. However, it requires that your JMS provider supports the JMS `ServerSessionPool` SPI.

Specifying an endpoint's listener container

By default, consumer endpoints use the default listener container. If you want to configure the an endpoint to use a different listener container, you specify that using the endpoint's `listenerType` attribute. Table 3.1 on page 38 lists the values for the `listenerType` attribute.

Table 3.1. Values for Configuring a Consumer's Listener Container

Value	Description
`simple`	Specifies that the endpoint will use the simple listener container.
`default`	Specifies that the endpoint will use the default listener container.
`server`	Specifies that the endpoint will use the server session listener container.

Example 3.4 on page 38 shows configuration for SOAP consumer that uses the simple listener container.

Example 3.4. Configuring a SOAP Consumer to Use the Simple Listener Container

```
<beans xmlns:jms="http://servicemix.apache.org/jms/1.0"
       ... >
  ...
  <jms:soap-consumer wsdl="classpath:widgets.wsdl"
                     destinationName="widgetQueue"
                     connectionFactory="#connectionFactory"
                     listenerType="simple" />
  ...
</beans>
```

Performace tuning using the listener container

There are several ways of tuning the performance of a generic consumer endpoint or a SOAP consumer endpoint. They are all controlled by the listener container used by the endpoint.

Table 3.2 on page 38 describes the attributes used to tune endpoint performance.

Table 3.2. Attributes Used to Performance Tune Standard JMS Consumers and SOAP JMS Consumers

Attribute	Type	Listener(s)	Description	Default
`cacheLevel`	int	default	Specifies the level of caching allowed by the listener. Valid values are `0`(CACHE_NONE), `1`(CACHE_CONNECTION),	`0`

Attribute	Type	Listener(s)	Description	Default
			2(CACHE_SESSION), and 3(CACHE_CONSUMER).	
clientId	string	all	Specifies the ID to be used for the shared Connection object used by the listener container.	Uses provider assigned ID
concurrentConsumers	int	default simple	Specifies the number of concurrent consumers created by the listener.	1
maxMessagesPerTask	int	default server	Specifies the number of attempts to receive messages per task.	-1(unlimited)
receiveTimeout	long	default	Specifies the timeout for receiving a message in milliseconds.	1000
recoveryInterval	long	default	Specifies the interval, in milliseconds, between attempts to recover after a failed listener set-up.	5000

Example 3.5 on page 39 shows an example of a generic consumer that allows consumer level message caching and only tries once to receive a message.

Example 3.5. Tuning a Generic Consumer Endpoint

```
<beans xmlns:jms="http://servicemix.apache.org/jms/1.0"
       ... >
  ...
  <jms:consumer service="my:widgetService"
                endpoint="jbiWidget"
                destinationName="widgetQueue"
                connectionFactory="#connectionFactory"
                cacheLevel="3"
                maxMessagesPerTask="1"/>
```

```
  ...
</beans>
```

Configuring the server session listener container's session factory

The server session listener container uses the JMS `ServerSessionPool` SPI to tune an endpoint's performance. In order for the listener container to function,k it uses a `ServerSessionFactory` object. By default, the Fuse ESB Enterprise JMS BC uses the Spring framework's `SimpleServerSessionFactory` object. This server session factory creates a new JMS `ServerSession` object with a new JMS session everytime it is called.

You can configure the endpoint to use a different server session factory using the `serverSessionFactory` attribute. This attribute provides a reference to the bean configuring the `ServerSessionFactory` object.

Note

You can also explicitly configure the endpoint's `ServerSessionFactory` object by adding a `serverSessionFactory` child element to the endpoint's configuration. This element would wrap the `ServerSessionFactory` object's configuration bean.

Example 3.6 on page 40 shows an example of configuring an endpoint to use the Spring framework's `CommonsPoolServerSessionFactory` object as a session factory.

Example 3.6. Configuring a Consumer to Use a Pooled Session Factory

```
<beans xmlns:jms="http://servicemix.apache.org/jms/1.0"
       ... >
  ...
  <jms:consumer service="my:widgetService"
                endpoint="jbiWidget"
                destinationName="widgetQueue"
                connectionFactory="#connectionFactory"
                listenerType="server"
                serverSessionFactory="#pooledSessionFactory"/>

  <bean id="pooledSessionFactory"
        class="org.springframework.jms.listener.serversession.CommonsPoolServerSessionFact
ory" />
  ...
</beans>
```

Advanced Configuration

Using transactions

By default, generic consumers and SOAP consumers do not wrap message exchanges in transactions. If there is a failure during the exchange, you have no guarantee that resending the request will not result in duplicating a task that has already been completed.

If your application requires message exchanges to be wrapped in a transaction, you can use the endpoint's `transacted` attribute to specify the type of transactions to use. Table 3.3 on page 41 describes the possible values for the `transacted` attribute.

Table 3.3. Consumer Transaction Support

Value	Description
`none`	Specifies that message exchanges are not wrapped in a transaction. This is the default setting.
`jms`	Specifies that message exchanges are wrapped in local JMS transactions.
`xa`	Specifies that message exchanges will be wrapped in an externally managed XA transaction. You must also provide a transaction manager when using XA transactions.

Important

Only the default listener container can support XA transactions.

Using message selectors

If you want to configure your consumer to use a JMS message selector, you can set the optional `messageSelector` attribute. The value of the attribute is the string value of the selector. For more information on the syntax used to specify message selectors, see the JMS 1.1 specification.

Using durable subscriptions

If you want to configure your server to use durable subscriptions, you need to set values for two attributes. To indicate that the consumer uses a durable subscription you set the `subscriptionDurable` attribute to `true`. You specify

the name used to register the durable subscription using the `durableSubscriberName` attribute.

Example 3.7 on page 42 shows a configuration snipit for a consumer that registers for a durable subscription.

Example 3.7. Consumer using a Durable Subscription

```
<beans xmlns:jms="http://servicemix.apache.org/jms/1.0"
       ... >
  ...
  <jms:soap-consumer wsdl="classpath:widgets.wsdl"
                     destinationName="widgetQueue"
                     connectionFactory="#connectionFactory"
                     subscriptionDurable="true"
                     durableSubscriberName="widgetSubscriber" />
  ...
</beans>
```

SOAP Specific Configuration

Overview

The SOAP consumer has two specialized configuration properties. One controls if the endpoint needs to use the JBI wrapper to make messages consumable. The other determines if the endpoint checks its WSDL for compliance with the WS-I basic profile.

Using the JBI wrapper

There are instances when a JBI component cannot consume a native SOAP message. For instance, SOAP headers pose difficulty for JBI components. The JBI specification defines a JBI wrapper that can be used to make SOAP messages, or any message defined in WSDL 1.1, conform to the expectations of a JBI component.

To configure a SOAP consumer to wrap messages in the JBI wrapper you set its `useJbiWrapper` attribute to `true`.

Example 3.8 on page 43 shows a configuration fragment for configuring a SOAP consumer to use the JBI wrapper.

Example 3.8. Configuring a SOAP Consumer to Use the JBI Wrapper

```
<beans xmlns:jms="http://servicemix.apache.org/jms/1.0"
       ... >
  ...
  <jms:soap-consumer wsdl="classpath:widgets.wsdl"
                     destinationName="widgetQueue"
                     connectionFactory="#connectionFactory"
                     useJbiWrapper="true" />
  ...
</beans>
```

WSDL verification

The WS-I basic profile is a specification describing the minimum set of requirements for a Web service to be considered interoperable. The requirement of the specification mostly constrain the binding of messages into SOAP containers.

By default, SOAP consumers will verify that their WSDL complies to the WS-I basic profile before starting up. If the WSDL does not comply, the endpoint will not start up.

If you want to skip the WS-I basic profile verification, you can set the consumer's `validateWsdl` attribute to `false`.

Using the JCA Consumer Endpoint

Procedure

To configure a JCA consumer endpoint do the following:

1. Specify the name of the service for which this endpoint is acting as a proxy.

   This is specified using the `service` attribute.

2. Specify the name of the endpoint for which this endpoint is acting as a proxy.

   This is specified using the `endpoint` attribute.

3. Specify the connection factory the endpoint will use.

   The endpoint's connection factory is configured using the endpoint's `connectionFactory` attribute. The `connectionFactory` attribute's value is a reference to the bean that configures the connection factory. For example if the connection factory configuration bean is named `widgetConnectionFactory`, the value of the `connectionFactory` attribute would be `#widgetConnectionFactory`.

   For information on configuring a connection factory see "Configuring the Connection Factory" on page 19.

4. Specify the destination onto which the endpoint will place messages.

   For more information see "Configuring a destination" on page 45.

5. Configure the JCA resource adapter that the consumer will use.

   You configure the endpoint's resource adapter using the `resourceAdapter` attribute. The attribute's value is a reference to the bean that configures the resource adapter.

6. Configure the `ActivationSpec` object that will be used by the endpoint.

   You configure the endpoint's resource adapter using the `activationSpec` attribute. The attribute's value is a reference to the bean that configures the `ActivationSpec` object.

7. Specify the ESB endpoint to which incoming messages are targeted.

For more information see "Specifying the target endpoint" on page 46.

8. If your JMS destination is a topic, set the `pubSubDomaim` attribute to `true`.

Configuring a destination

A consumer endpoint chooses the destination to use for sending messages with the following algorithm:

1. The endpoint will check to see if you configured the destination explicitly.

   You configure a destination using a Spring bean. You can add the bean directly to the endpoint by wrapping it in a `jms:destination` child element. You can also configure the bean separately and refer the bean using the endpoint's `destination` attribute as shown in Example 3.9 on page 45.

Example 3.9. Configuring a JCA Consumer's Destination

```
<beans xmlns:jms="http://servicemix.apache.org/jms/1.0"
       ... >
  ...
  <jms:jca-consumer service="my:widgetService"
                    endpoint="jbiWidget"
                    destination="#widgetQueue"
                    ...  />
  ...
  <jee:jndi-lookup id="widgetQueue" jndi-name="my.widget.queue">
    <jee:environment>
      java.naming.factory.initial=weblogic.jndi.WLInitialContextFactory
      java.naming.provider.url=t3://localhost:7001
    </jee:environment>
  </jee:jndi-lookup>
  ...
</beans>
```

2. If you did not explicitly configure a destination, the endpoint will use the value of the `destinationName` attribute to choose its destination.

   The value of the `destinationName` attribute is a string that corresponds to the name of the JMS destination. The binding component's default behavior when you provide a destination name is to resolve the destination using the standard JMS `Session.createTopic()` and `Session.createQueue()` methods.

Note

You can override the binding component's default behavior by providing a custom `DestinationResolver` implementation. See "Using a Custom Destination Resolver" on page 87.

Specifying the target endpoint

There are a number of attributes available for configuring the endpoint to which the generated messages are sent. The poller endpoint will determine the target endpoint in the following manner:

1. If you explicitly specify an endpoint using both the `targetService` attribute and the `targetEndpoint` attribute, the ESB will use that endpoint.

   The `targetService` attribute specifies the QName of a service deployed into the ESB. The `targetEndpoint` attribute specifies the name of an endpoint deployed by the service specified by the `targetService` attribute.

2. If you only specify a value for the `targetService` attribute, the ESB will attempt to find an appropriate endpoint on the specified service.

3. If you do not specify a service name or an endpoint name, you must specify an the name of an interface that can accept the message using the `targetInterface` attribute. The ESB will attempt to locate an endpoint that implements the specified interface and direct the messages to it.

   Interface names are specified as QNames. They correspond to the value of the `name` attribute of either a WSDL 1.1 `serviceType` element or a WSDL 2.0 `interface` element.

Important

If you specify values for more than one of the target attributes, the consumer endpoint will use the most specific information.

Example

Example 3.10 on page 47 shows the configuration for a JCA consumer endpoint.

Example 3.10. Basic Configuration for a JCA Consumer Endpoint

```
<beans xmlns:jms="http://servicemix.apache.org/jms/1.0"
       ... >
  ...
  <jms:jca-consumer service="my:widgetService"
                    endpoint="jbi"
                    destinationName="widgetQueue"
                    connectionFactory="#connectionFactory"
                    resourceAdapter="#ra"
                    activationSpec="#as"
                     targetService="my:targetService" />

  <bean id="ra"
        class="org.activemq.ra.ActiveMQConnectionFactory">
    ...
  </bean>

  <bean id="as"
        class="org.apache.activemq.ra.ActiveMQActivationSpec">
    ...
  </bean>
  ...
</beans>
```

Configuring How Replies are Sent

If your endpoint is participating in in/out message exchanges, or exceptions need to be returned to the external endpoint, you need to configure how your endpoint will handle the reply messages. You can configure the JMS destination used to send the reply and how the endpoint specifies the reply message's correlation ID. In addition, you can specify a number of QoS settings including:

- the reply message's priority
- the reply message's persistence
- the reply message's lifespan

You can also specify a number of custom properties to place in a reply message's JMS header.

Configuring the Reply Destination

Overview

Fuse ESB Enterprise JMS consumers determine destination of reply messages and exceptions uses a straightforward algorithm. By default, the reply destination is supplied by the message that started the exchange. If the reply destination cannot be determined from the request message, the endpoint will use a number of strategies to determine the reply destination.

You can customize how the endpoint determines the reply destination using the endpoint's configuration. You can also supply fall back values for the endpoint to use.

Determining the reply destination

Consumer endpoints use the following algorithm to determine the reply destination for a message exchange:

1. If the in message of the exchange includes a value for the `JMSReplyTo` property, that value is used as the reply destination.
2. If the `JMSReplyTo` is not specified, the endpoint looks for a destination chooser implementation to use.

   If you have configured your endpoint with a destination chooser, the endpoint will use the destination chooser to select the reply destination.

   For more information on using destination choosers see "Using a Custom Destination Chooser" on page 84.
3. If the `JMSReplyTo` is not specified and there is no configured destination chooser, the endpoint checks its `replyDestination` attribute for a destination.

   You configure a destination using a Spring bean. The recommend method to configure the destination is to configure the bean separately and refer the bean using the endpoint's `replyDestination` attribute as shown in Example 3.11 on page 50. You can also add the bean directly to the endpoint by wrapping it in a `jms:replyDestination` child element.
4. As a last resort, the endpoint will use the value of the `replyDestinationName` attribute to determine the reply destination.

   The `replyDestinationName` attribute takes a string that is used as the name of the destination to use. The binding component's default behavior when you provide a destination name is to resolve the destination using

the standard JMS `Session.createTopic()` and `Session.createTopic()` methods to resolve the JMS destination.

Note

You can override the binding component's default behavior by providing a custom `DestinationResolver` implementation. See "Using a Custom Destination Resolver" on page 87.

Example

Example 3.11 on page 50 shows an example of configuring a consumer endpoint to use a dedicated JMS destination.

Example 3.11. Configuring a Consumer's Reply Destination

```
<beans xmlns:jms="http://servicemix.apache.org/jms/1.0"
       ... >
  ...
  <jms:consumer service="my:widgetService"
                endpoint="jbiWidget"
                destinationName="my.widgetQueue"
                connectionFactory="#connectionFactory"
                replyDestination="#widgetReplyQueue" />
  ...
  <jee:jndi-lookup id="widgetReplyQueue" jndi-name="my.widget.reply.queue">
    <jee:environment>
      java.naming.factory.initial=weblogic.jndi.WLInitialContextFactory
      java.naming.provider.url=t3://localhost:7001
    </jee:environment>
  </jee:jndi-lookup>
  ...
</beans>
```

Configuring the Qualities of Service

Overview

You can specify a number of the reply message's QoS settings including:

- the reply message's priority
- the reply message's persistence
- the reply message's lifespan

These properties are stored in the JMS message header. By default, the JMS broker automatically populates their values. You can, however, configure an endpoint to override the broker's default.

Setting the reply message's priority

JMS uses a priority system to determine the relative importance of delivering a message. Messages with higher priority are delivered before messages with a lower priority.

You configure the priority of the reply message messages by setting the consumer's `replyPriority` attribute. The value is used to set the reply message's `JMSPriority` property.

JMS supports priority values between 0 and 9. The lowest priority is 0 and the highest priority is 9. The default priority for a message is 4.

Setting the reply message's persistence

JMS uses a message's delivery mode to determine its persistence in the system. You can set the delivery mode for the reply messages sent by an endpoint by setting the endpoint's `replyDeliveryMode` attribute. The value you provide for the `replyDeliveryMode` attribute is used to set the reply message's `JMSDeliveryMode` property.

JMS implementations support two delivery modes: persistent and non-persistent.

Persistent messages can survive a shutdown of the JMS broker. This is the default setting for JMS messages. You can specify persistence by setting the endpoint's `deliveryMode` attribute to `2`. This setting corresponds to `DeliveryMode.PERSISTENT`.

Non-persistent messages are lost if the JMS broker is shutdown before they are delivered. You can specify non-persistence by setting the endpoint's

`deliveryMode` attribute to `1`. This setting corresponds to `DeliveryMode.NON_PERSISTENT`.

Setting a reply message's lifespan

You can control how long reply messages live before the JMS broker reap them by setting the endpoint's `replyTimeToLive` attribute. The value is the number of milliseconds you want the message to be available from the time it is sent.

The value of the `replyTimeToLive` attribute is used to compute the value for the reply message's `JMSExpirary` property. The value is computed by adding the specified number of milliseconds to the time the message is created.

The default behavior is to allow messages to persist forever.

Enforcing the configured values

By default, the consumer ignores these settings and allows the JMS provider to insert its own default values for the reply message's QoS settings. To force your settings to be used, you need to set the endpoint's `replyExplicitQosEnabled` to `true`. Doing so instructs the consumer to always use the values provided in the configuration.

Example

Example 3.12 on page 52 shows the configuration for a consumer whose reply messages are set to have low priority and to be non-persistent.

Example 3.12. Consumer with Reply QoS Properties

```
<beans xmlns:jms="http://servicemix.apache.org/jms/1.0"
       ... >
  ...
  <jms:jca-consumer service="my:widgetService"
                endpoint="jbiWidget"
                connectionFactory="#connectionFactory"
                destinationName="widgetQueue"
                resourceAdapter="#ra"
                activationSpec="#as"
                replyExplicitQosEnabled="true"
                replyDeliveryMode="1"
                replyPriority="0" />
  ...
</beans>
```

Setting Custom JMS Properties

Overview

The JMS specification allows for the placing of custom properties into a message's header. These custom properties are specified as a set of name/value pairs that can store both simple types and Java objects. The properties can be used for a number of tasks including message selection.

When using the Fuse ESB Enterprise JMS binding component, you define the custom properties added to the reply messages as property map. This is done using the Spring `map` element. You can configure one static map that will be applied to every reply message generated by the consumer.

Setting custom JMS header properties

You can configure a consumer to add custom properties to reply messages in one of two ways:

1. Use the endpoint's `replyProperties` attribute to refer to the property map defining the custom properties.
2. Add a `jms:replyProperties` child element to the endpoint. The `jms:replyProperties` element wraps the property map.

Defining the property map

The property map containing the custom properties you want added to the reply messages is stored in a `java.util.Map` object. You define that map object using the Spring `util:map` element.

The `util:map` element is defined in the http://www.springframework.org/schema/util namespace. In order to use the element you will need to add the following namespace alias to your `beans` element:

```
xmlns:util="http://www.springframework.org/schema/util"
```

The entries in the map are defined by adding `entry` child element's to the `util:map` element. Each `entry` element takes two attributes. The `key` entry is the map key and corresponds to the properties name. The `value` attribute is the value of the property.

Tip

If you want the value of a property to be complex type that is stored in a Java object, you can use the `entry` element's `ref` attribute

instead of the `value` attribute. The `ref` attribute points to another `bean` element that defines a Java object.

Example

Example 3.13 on page 54 shows an example of a SOAP consumer whose reply messages have a set of custom properties added to their header.

Example 3.13. Adding Custom Properties to a Reply Message

```
<beans xmlns:jms="http://servicemix.apache.org/jms/1.0"
       xmlns:util="http://www.springframework.org/schema/util"
       ... >
  ...
  <jms:consumer service="my:widgetService"
                endpoint="jbiWidget"
                destinationName="my.widgetQueue"
                connectionFactory="#connectionFactory"
                replyDestination="#widgetReplyQueue"
                replyProperties="#jmsProps" />
  ...
  <util:map id="jmsProps">
    <entry key="location" value="San Jose"/>
    <entry key="orig_code" value="sjwf"/>
    <entry key="client_code" value="widget010"/>
  </util:map>
  ...
</beans>
```

Chapter 4. Creating a Provider Endpoint

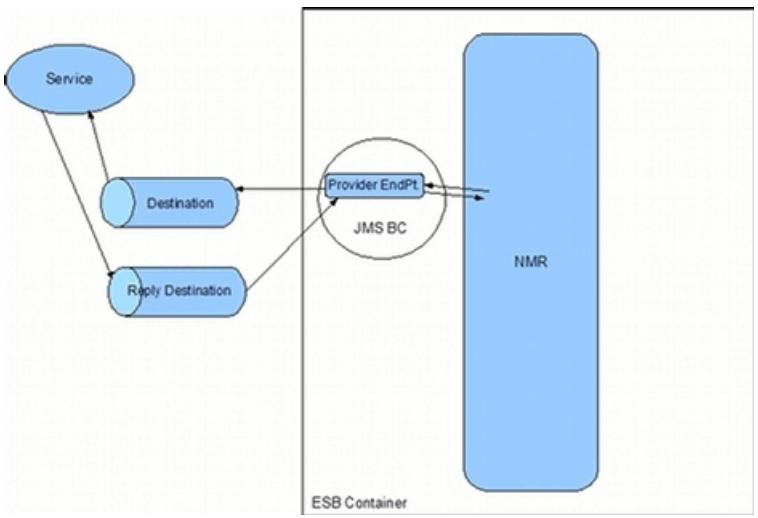
A provider is an endpoint that sends messages to remotes endpoints and, depending on the message exchange pattern, waits for a response. They use the Spring framework's `JMSTemplate` interface.

Introduction to Provider Endpoints

Where does a provider fit into a solution?

A provider endpoint plays the role of a provider from the vantage point of other endpoints inside of the ESB. As shown in Figure 4.1 on page 56, a provider endpoint receives messages from the NMR and places them onto a JMS destination. If the NMR message is part of an in-out message exchange, the endpoint will listen for the response on a reply destination.

Figure 4.1. Provider Endpoint



Types of providers

The JMS binding component has two types of provider endpoints:

Generic

The generic provider endpoint can handle any type of message data. It is configured using the `jms:provider` element.

SOAP

The SOAP provider endpoint is specifically tailored to receive SOAP messages. It uses a WSDL document to define the structure of the messages. It is configured using the `jms:soap-provider` element.

Tip

The Apache CXF binding component's JMS transport is better adapted to handling SOAP messages, but offers less control over the JMS connection.

Basic Configuration

Procedure

To configure a provider endpoint do the following:

1. Decide what type of provider endpoint to use.

   See "Types of providers" on page 56.

2. Specify the name of the service for which this endpoint is acting as a proxy.

   This is specified using the `service` attribute.

   Tip

   If you are using a SOAP provider and your WSDL file only has one service defined, you do not need to specify the service name.

3. Specify the name of the endpoint for which this endpoint is acting as a proxy.

   This is specified using the `endpoint` attribute.

   Tip

   If you are using a SOAP provider and your WSDL file only has one endpoint defined, you do not need to specify the endpoint name.

4. Specify the connection factory the endpoint will use.

   The endpoint's connection factory is configured using the endpoint's `connectionFactory` attribute. The `connectionFactory` attribute's value is a reference to the bean that configures the connection factory. For example, if the connection factory configuration bean is named `widgetConnectionFactory`, the value of the `connectionFactory` attribute would be `#widgetConnectionFactory`.

   For information on configuring a connection factory see "Configuring the Connection Factory" on page 19.

5. Specify the destination onto which the endpoint will place messages.

For more information see "Configuring a destination" on page 59.

6. If you are using a JMS SOAP provider, specify the location of the WSDL defining the message exchange using the `wsdl` attribute.
7. If your JMS destination is a topic, set the `pubSubDomaim` attribute to `true`.
8. If your endpoint is interacting with a broker that only supports JMS 1.0.2, set the `jms102` attribute to `true`.

Configuring a destination

A provider endpoint chooses the destination to use for sending messages with the following algorithm:

1. If you provided a custom `DestinationChooser` implementation, the endpoint will use that to choose it's endpoint.

   For more information about providing custom `DestinationChooser` implementations see "Using a Custom Destination Chooser" on page 84.

2. If you did not provide a custom `DestinationChooser` implementation, the endpoint will use its default `DestinationChooser` implementation to choose an endpoint.

   The default destination chooser checks the message exchange received from the NMR for a `DESTINATION_KEY` property. If the message exchange has that property set, it returns that destination.

3. If the destination chooser does not return a destination, the endpoint will check to see if you configured the destination explicitly.

   You configure a destination using a Spring bean. The recommend way to configure the destination is to configure the bean separately and refer the bean using the endpoint's `destination` attribute as shown in Example 4.1 on page 59. You can also add the bean directly to the endpoint by wrapping it in a `jms:destination` child element.

Example 4.1. Configuring a Provider's Destination

```
<beans xmlns:jms="http://servicemix.apache.org/jms/1.0"
       ... >
  ...
  <jms:provider service="my:widgetService"
                endpoint="jbiWidget"
```

```
                destination="#widgetQueue"
                connectionFactory="#connectionFactory" />
  ...
  <jee:jndi-lookup id="widgetQueue" jndi-name="my.widget.queue">
    <jee:environment>
      java.naming.factory.initial=weblogic.jndi.WLInitialContextFactory
      java.naming.provider.url=t3://localhost:7001
    </jee:environment>
  </jee:jndi-lookup>
  ...
</beans>
```

4. If the destination chooser does not return a destination and you did not explicitly configure a destination, the endpoint will use the value of the `destinationName` attribute to choose its destination.

   The `destinationName` attribute takes a string that is used as the name of the destination to use. The binding component's default behavior when you provide a destination name is to resolve the destination using the standard JMS `Session.createTopic()` and `Session.createQueue()` methods to resolve the JMS destination.

Note

You can override the binding component's default behavior by providing a custom `DestinationResolver` implementation. See "Using a Custom Destination Resolver" on page 87.

Examples

Example 4.2 on page 60 shows the basic configuration for a plain JMS provider endpoint.

Example 4.2. Basic Configuration for a Generic Provider Endpoint

```
<beans xmlns:jms="http://servicemix.apache.org/jms/1.0"
       ... >
  ...
  <jms:provider service="my:widgetService"
                endpoint="jbiWidget"
                destinationName="widgetQueue"
                connectionFactory="#connectionFactory" />
  ...
</beans>
```

Example 4.3 on page 61 shows the basic configuration for a SOAP JMS provider endpoint.

Example 4.3. Basic Configuration for a SOAP Provider Endpoint

```
<beans xmlns:jms="http://servicemix.apache.org/jms/1.0"
       ... >
  ...
  <jms:soap-provider wsdl="classpath:widgets.wsdl"
                     destinationName="widgetQueue"
                     connectionFactory="#connectionFactory"
/>
  ...
</beans>
```

Configuring How Responses are Received

Overview

If your provider endpoint participates in in/out message exchanges, it will wait for a response from receiving endpoint. You can configure the JMS destination on which the endpoint listens for the response. You can also configure the amount of time the endpoint will wait for a response before it times out.

Configuring the response destination

An endpoint chooses the destination to use for receiving responses with the following algorithm:

1. If you provided a custom `DestinationChooser` implementation, the endpoint will use that to choose it's endpoint.

   For more information about providing custom `DestinationChooser` implementations see "Using a Custom Destination Chooser" on page 84.

2. If you did not provide a custom `DestinationChooser` implementation, the endpoint will use its default `DestinationChooser` implementation to choose an endpoint.

   The default destination chooser checks the message exchange received from the NMR for a `DESTINATION_KEY` property. If the message exchange has that property set, it returns that destination.

3. If the destination chooser does not return a destination, the endpoint will check to see if you configured the destination explicitly.

   You configure a response destination using a Spring bean. The recommend way to configure the destination is to configure the bean separately and refer the bean using the endpoint's `replyDestination` attribute as shown in Example 4.1 on page 59. You can also add the bean directly to the endpoint by wrapping it in a `jms:replyDestination` child element.

4. If the destination chooser does not return a destination and you did not explicitly configure a destination, the endpoint will use the value of the `replyDestinationName` attribute to choose its destination.

   The `replyDestinationName` attribute takes a string that is used as the name of the destination to use. The binding component's default behavior when you provide a destination name is to resolve the destination using

the standard JMS `Session.createTopic()` and `Session.createTopic()` methods to resolve the JMS destination.

Note

You can override the binding component's default behavior by providing a custom `DestinationResolver` implementation. See "Using a Custom Destination Resolver" on page 87.

Configuring the timeout interval

By default, a provider endpoint will wait an unlimited amount of time for a response. Since the provider blocks while it is waiting for a response, your application may hang indefinitely if a response does not arrive.

You can configure the endpoint to timeout using the `recieveTimeout` attribute. The `recieveTimeout` attribute specifies the number of milliseconds the provider endpoint will wait for a response before timing out.

Example

Example 4.4 on page 63 shows a JMS provider endpoint that will wait for a response for one minute.

Example 4.4. JMS Provider Endpoint with a Response Destination

```
<beans xmlns:jms="http://servicemix.apache.org/jms/1.0"
       ... >
  ...
  <jms:soap-provider wsdl="classpath:widgets.wsdl"
                     destinationName="widgetQueue"
                     connectionFactory="#connectionFactory"
                     recieveTimeout="60000"
                    replyDestinationName="widgetResponse" />

  ...
</beans>
```

Advanced Provider Configuration

JMS Message Qualities of Service

Overview

JMS messages have a number of quality of service properties that can be set. These QoS properties include the following:

- the message's relative priority
- the message's persistence
- the message's lifespan

These properties are stored in the JMS message header. By default, the JMS broker automatically populates their values. You can, however, configure an endpoint to override the broker's default.

Setting a message's priority

You configure the endpoint to set the priority for all out going JMS messages using the `priority` attribute. The value you provide for the `priority` attribute is used to set the JMS message's `JMSPriority` property.

JMS priority values can range from 0 to 9. The lowest priority is 0 and the highest priority is 9. If you do not provide a value, the JMS provider will use the default priority value of 4. The default priority is considered normal.

Setting a message's persistence

In JMS a message's persistence is controlled by its delivery mode property. You configure the delivery mode of the messages produced by a JMS provider by setting its `deliveryMode` attribute. The value you provide for the `deliveryMode` attribute is used to set the JMS message's `JMSDeliveryMode` property.

JMS implementations support two delivery modes: persistent and non-persistent.

Persistent messages can survive a shutdown of the JMS broker. This is the default setting for JMS messages. You can specify persistence by setting the endpoint's `deliveryMode` attribute to `2`. This setting corresponds to `DeliveryMode.PERSISTENT`.

Non-persistent messages are lost if the JMS broker is shutdown before they are delivered. You can specify non-persistence by setting the endpoint's

`deliveryMode` attribute to `1`. This setting corresponds to `DeliveryMode.NON_PERSISTENT`.

Setting a message's life span

You can control how long messages persists before the JMS broker reaps them by setting the endpoint's `timeToLive` attribute. The value is the number of milliseconds you want the message to be available from the time it is sent. The default behavior is to allow messages to persist forever.

The value of the `timeToLive` attribute is used to compute the value for the message's `JMSExpirary` property. The value is computed by adding the specified number of milliseconds to the time the message is created.

Enforcing configured values

By default, a JMS provider endpoint will allow the JMS provider to set these values to default values and ignore any values set through the configuration. To override this behavior, you need to set the endpoint's `explicitQosEnabled` attribute to `true`.

Example

Example 4.5 on page 66 shows configuration for a JMS SOAP provider whose messages have a priority of 1.

Example 4.5. Setting JMS Provider Endpoint Message Properties

```
<beans xmlns:jms="http://servicemix.apache.org/jms/1.0"
       ... >
  ...
  <jms:soap-provider wsdl="classpath:widgets.wsdl"
                     destinationName="widgetQueue"
                     connectionFactory="#connectionFactory"
                     priority="1"
                     explicitQosEnabled="true" />
  ...
</beans>
```

JMS Message Optimization

Overview

JMS message producers are able to provide hints to the JMS broker about possible message optimizations. These hints include whether or not JMS message IDs are required and whether or not timestamps are needed.

By default, Fuse ESB Enterprise JMS provider endpoints require that messages have IDs and timestamps. However, if your application does not require them you can instruct the endpoint to inform the JMS provider that it can skip the creation of IDs and time stamps. The JMS provider is not required to take the hint.

Message IDs

By default, a JMS message broker generates a unique identifiers for each message that it manages and places the ID in the message's header. These IDs can be used by JMS applications for a number of purposes. One reason to use them is to correlate request and reply messages.

Message IDs take time to create and increase the size of a message. If your application does not require message IDs, you can optimize it by configuring the endpoint to disable message ID generation by setting the `messageIdEnabled` attribute to `false`.

Setting the `messageIdEnabled` attribute to `false` causes the endpoint to call its message producer's `setDisableMessageID()` method with a value of `true`. The JMS broker is then given a hint that it does not need to generate message IDs or add them to the messages from the endpoint. The JMS broker can choose to accept the hint or ignore it.

Time stamps

By default, a JMS message broker places time stamp representing the time the message is processed into each message's header.

Time stamps increase the size of a message. If your application does not use the timestamps, you can optimize it by configuring the endpoint to disable time stamp generation by setting the `messageTimeStampEnabled` attribute to `false`.

Setting the `messageTimeStampEnabled` attribute to `false` causes the endpoint to call its message producer's `setDisableMessageTimestamp()` method with a value of `true`. The JMS broker is then given a hint that it does not need to generate message IDs or add them to the messages from the endpoint. The JMS broker can choose to accept the hint or ignore it.

SOAP Specific Configuration

Overview

The SOAP provider has two specialized configuration properties. One controls if the endpoint needs to use the JBI wrapper to make messages consumable. The other determines if the endpoint checks its WSDL for compliance with the WS-I basic profile.

Using the JBI wrapper

There are instances when a JBI component cannot consume a native SOAP message. For instance, SOAP headers pose difficulty for JBI components. The JBI specification defines a JBI wrapper that can be used to make SOAP messages, or any message defined in WSDL 1.1, conform to the expectations of a JBI component.

To configure a SOAP provider to wrap messages in the JBI wrapper, you set its `useJbiWrapper` attribute to `true`.

Example 4.6 on page 68 shows a configuration fragment for configuring a SOAP provider to use the JBI wrapper.

Example 4.6. Configuring a SOAP Provider to Use the JBI Wrapper

```
<beans xmlns:jms="http://servicemix.apache.org/jms/1.0"
       ... >
  ...
  <jms:soap-provider wsdl="classpath:widgets.wsdl"
                     destinationName="widgetQueue"
                     connectionFactory="#connectionFactory"
                     useJbiWrapper="true" />
  ...
</beans>
```

WSDL verification

The WS-I basic profile is a specification describing the minimum set of requirements for a Web service to be considered interoperable. The requirement of the specification mostly constrain the binding of messages into SOAP containers.

By default, SOAP providers will verify that their WSDL complies to the WS-I basic profile before starting up. If the WSDL does not comply, the endpoint will not start up.

If you want to skip the WS-I basic profile verification, you can set the provider's `validateWsdl` attribute to `false`.

Chapter 5. Making Endpoints Stateful

You can configure JMS endpoints to store a copy of the current message exchange in a persistent datastore. This helps in cases where you need to recover from failures.

Overview

Fuse ESB Enterprise JMS endpoints typically do not store any state information. You can, however, configure them to store a copy of the current JMS message being sent. The message can be stored either in memory or in a JDBC configured database.

Having the endpoint store a copy of the current JMS message can aid in recovery from failures. For example, if your application is deployed in a cluster of Fuse ESB Enterprise containers you can configure your endpoints to fail over if one of the containers crashes. If your endpoints are configured to store state in a JDBC database, they can then resend any request that was in process.

Activating statefullness

You configure an endpoint to save a copy of the current message by setting its `stateless` attribute to `false`.

Configuring the datastore

By default, JMS endpoints uses a memory based message store. The memory based message store is a simple hash map that is stored in active memory. It cannot persist in the event of a failure, does not support transactions, or access by multiple members of a cluster.

If you need to use a more robust message store, you can configure a provider endpoint to use a JDBC accessible database as a message store. A JDBC message store can be shared among a cluster of endpoints, can be persisted in the event of a failure, and, depending on the database, be enlisted in transactions.

To configure an endpoint to use a JDBC accessible datastore, you configure its `storeFactory` attribute to reference a bean configuring an instance of the `org.apache.servicemix.store.jdbc.JdbcStoreFactory` class. Table 5.1 on page 70 list the properties you can set for the JDBC store factory.

Table 5.1. Properties Used to Configure a JDBC Store Factory

Name	Description
`clustered`	Specifies if a datastore can be accessed by the members of an endpoint cluster.
`transactional`	Specifies if the datastore can be enlisted in transactions.
`dataSource`	Specifies the configuration for the data source to be used when creating the store.
`adapter`	Specifies the configuration for the JDBC adapter used to connect to the data source.

Note

The values for `dataSource` and `adapter` will depend on the database you are using and the JDBC adapter you are using.

Example

The fragment in Example 5.1 on page 70 shows the configuration needed for a stateful JMS provider endpoint using MySQL as a JDBC accessible datastore.

Example 5.1. Configuring a Statefull JMS Provider Endpoint

```
<jms:provider service="tns:widgetServer"
              endpoint="widgetPort"
              storeFactory="#storeFactory"> ❶
              stateless="false" /> ❷

<bean id="storeFactory" ❸
      class="org.apache.servicemix.store.jdbc.JdbcStoreFactory">
  <property name="clustered" value="true"/>
  <property name="dataSource">
    <ref local="mysql-ds"/>
  </property>
</bean>

<bean id="mysql-ds" ❹
      class="com.mchange.v2.c3p0.ComboPooledDataSource"
      destroy-method="close">
```

```
  <property name="driverClass" value="com.mysql.jdbc.Driver"/>
  <property name="jdbcUrl"
            value="jdbc:mysql://localhost:3306/activemq?relaxAutoCommit=true"/>
  <property name="user" value="activemq"/>
  <property name="password" value="activemq"/>
  <property name="minPoolSize" value="5"/>
  <property name="maxPoolSize" value="10"/>
  <property name="acquireIncrement" value="3"/>
  <property name="autoCommitOnClose" value="false"/>
</bean>
```

The fragment in Example 5.1 on page 70 does the following:

❶ Configures the endpoint's store factory by providing a reference to the bean configuring the factory.
❷ Configures the endpoint to store a copy of the current message in the datastore.
❸ Configures the JDBC factory store to create a datastore that can be accessed by a cluster of endpoints.
❹ Configures the MySQL JDBC driver.

Chapter 6. Working with Message Marshalers

When using JMS endpoints, you may want to customize how messages are processed as they are passed into and out of the ESB. The Fuse ESB Enterprise JMS binding component allows you to write custom marshalers for your JMS endpoints.

Consumer Marshalers

Overview

Consumer endpoints use an implementation of the `org.apache.servicemix.jms.endpoints.JmsConsumerMarshaler` interface to process the incoming JMS messages and convert them into normalized messages. Consumer marshalers also convert fault messages and response messages into JMS messages that can be returned to the remote endpoint. The JMS binding component comes with two consumer marshaler implementations:

`DefaultConsumerMarshaler`

The `DefaultConsumerMarshaler` class provides the marshaler used by generic consumer endpoints and the JCA consumer endpoints.

`JmsSoapConsumerMarshaler`

The `JmsSoapConsumerMarshaler` class provides the marshaler used by SOAP consumer endpoints.

Note

The default SOAP marshaler does not support the full range of SOAP messages nor does it support marshaling map based messages into JMS messages.

When the default consumer marshaler does not suffice for your application you can provide a custom implementation of the `JmsConsumerMarshaler` interface.

Implementing the marshaler

To create a custom consumer marshaler, you implement the `org.apache.servicemix.jms.endpoints.JmsConsumerMarshaler` interface. The `JmsConsumerMarshaler` interface, shown in Example 6.1 on page 74, has five methods that need implementing:

Example 6.1. The Consumer Marshaler Interface

```
public interface JmsConsumerMarshaler
 {
    public interface JmsContext
    {
        Message getMessage();
```

```
    }

    JmsContext createContext(Message message) throws Exception;

    MessageExchange createExchange(JmsContext jmsContext, ComponentContext jbiContext)
throws Exception;

    Message createOut(MessageExchange exchange,
                      NormalizedMessage outMsg,
                      Session session,
                      JmsContext context) throws Exception;

    Message createFault(MessageExchange exchange,
                        Fault fault,
                        Session session,
                        JmsContext context) throws Exception;

    Message createError(MessageExchange exchange,
                        Exception error,
                        Session session,
                        JmsContext context) throws Exception;
}
```

`createContext()`

The `createContext()` method takes the JMS message and returns an object that implements the `JmsContext` interface.

`createExchange()`

The `createExchange()` creates a message exchange using the JMS message and the JBI context. Creating a message exchange entails the creation of the exchange, populating the exchange's in message, specifying the message exchange pattern to use, and setting any other required properties.

`createOut()`

The `createOut()` method takes the response message from the message exchange and converts it into a JMS message. The method takes the message exchange, the outgoing message, the active JMS session, and the JMS context.

`createFault()`

The `createFault()` method is called if a fault message is returned. It takes the message exchange, the fault message, the active JMS session,

and the JMS context and returns a JMS message that encapsulates the fault message.

`createError()`

The `createError()` method is called if an exception is thrown while the message exchange is being processed. It takes the message exchange, the exception, the active JMS session, and the JMS context and returns a JMS message that encapsulates the exception.

In addition to implementing the methods, you need to provide an implementation of the `JmsContext` interface. The `JmsContext` interface has a single method called `getMessage()` which returns the JMS message contained in the context.

Example 6.2 on page 76 shows a simple consumer marshaler implementation.

Example 6.2. Consumer Marshaler Implementation

```
package com.widgetVendor.example;

import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

import javax.jbi.component.ComponentContext;
import javax.jbi.messaging.Fault;
import javax.jbi.messaging.MessageExchange;
import javax.jbi.messaging.NormalizedMessage;
import javax.jms.Message;
import javax.jms.Session;
import javax.jms.TextMessage;
import javax.xml.transform.Source;

import org.apache.servicemix.jbi.jaxp.SourceTransformer;
import org.apache.servicemix.jbi.jaxp.StringSource;
import org.apache.servicemix.jbi.messaging.MessageExchangeSupport;

public class widgetConsumerMarshaler implements JmsConsumerMarshaler
{
    public JmsContext createContext(Message message) throws Exception
    {
        return new Context(message);
    }

   public MessageExchange createExchange(JmsContext jmsContext, ComponentContext jbiContext)
 throws Exception
```

```
    {
        Context ctx = (Context) jmsContext;
       MessageExchange exchange = jbiContext.getDeliveryChannel().createExchangeFactory().cre
ateExchange(MessageExchangeSupport.IN_ONLY);
        NormalizedMessage inMessage = exchange.createMessage();
        TextMessage textMessage = (TextMessage) ctx.message;
        Source source = new StringSource(textMessage.getText());
        inMessage.setContent(source);
        exchange.setMessage(inMessage, "in");
        return exchange;
    }

    public Message createOut(MessageExchange exchange, NormalizedMessage outMsg, Session
session, JmsContext context) throws Exception
    {
        String text = new SourceTransformer().contentToString(outMsg);
        return session.createTextMessage(text);
    }

    public Message createFault(MessageExchange exchange, Fault fault, Session session, Jm
sContext context) throws Exception
    {
        String text = new SourceTransformer().contentToString(fault);
        return session.createTextMessage(text);
    }

    public Message createError(MessageExchange exchange, Exception error, Session session,
 JmsContext context) throws Exception
    {
        throw error;
    }

    protected static class Context implements JmsContext
    {
        Message message;

        Context(Message message)
        {
          this.message = message;
        }

        public Message getMessage()
        {
            return this.message;
        }
    }
```

```
}
```

Configuring the consumer

You configure a consumer to use a custom marshaler using its `marshaler` attribute. The `marshaler` attribute's value is a reference to a `bean` element specifying the class of your custom marshaler implementation.

Example 6.3 on page 78 shows configuration for a consumer that uses a custom marshaler.

Example 6.3. Configuring a Consumer to Use a Customer Marshaler

```
<beans xmlns:jms="http://servicemix.apache.org/jms/1.0"
       ... >
  ...
  <jms:soap-consumer wsdl="classpath:widgets.wsdl"
                     destinationName="widgetQueue"
                     connectionFactory="#connectionFactory"
                     marshaler="#myConsumerMarshaler" />

  <bean id="myConsumerMarshaler" class="com.widgetVendor.example.widgetConsumerMarshaler"
/>
  ...
</beans>
```

Note

You can also configure a consumer to use a custom marshaler by adding a child `marshaler` element to the consumer's configuration. The `marshaler` element simply wraps the `bean` element that configures the marshaler.

Provider Marshalers

Overview

Providers use an implementation of the `org.apache.servicemix.jms.endpoints.JmsProviderMarshaler` interface to convert normalized messages into JMS messages. The marshaler also converts the incoming reply from a JMS message into a normalized message. The JMS binding component comes with two provider marshaler implementations:

`DefaultProviderMarshaler`

The `DefaultProviderMarshaler` class provides the marshaler used by generic provider endpoints.

`JmsSoapProviderMarshaler`

The `JmsSoapProviderMarshaler` class provides the marshaler used by SOAP provider endpoints.

Note

The default SOAP marshaler does not support the full range of SOAP messages nor does it support marshaling map based messages into JMS messages.

When the default provider marshalers do not suffice for your application, you can provide a custom implementation of the `JmsProviderMarshaler` interface.

Implementing the marshaler

To create a custom provider marshaler, you implement the `org.apache.servicemix.jms.endpoints.JmsProviderMarshaler` interface. The `JmsProviderMarshaler` interface, shown in Example 6.4 on page 79, has two methods you need to implement:

Example 6.4. The Provider Marshaler Interface

```
public interface JmsProviderMarshaler
{
  Message createMessage(MessageExchange exchange, NormalizedMessage in, Session session)
throws Exception;

  void populateMessage(Message message, MessageExchange exchange, NormalizedMessage normal
```

```
izedMessage) throws Exception;
}
```

`createMessage()`

The `createMessage()` method uses information from the Fuse ESB Enterprise core to generate a JMS message. Its parameters include the message exchange, the normalized message that is received by the provider, and the active JMS session.

`populateMessage()`

The `populateMessage()` method takes a JMS message and adds it to a message exchange for use by the Fuse ESB Enterprise core.

Example 6.5 on page 80 shows a simple provider marshaler implementation.

Example 6.5. Provider Marshaler Implementation

```
package com.widgetVendor.example;

import javax.jbi.messaging.MessageExchange;
import javax.jbi.messaging.NormalizedMessage;
import javax.jms.Message;
import javax.jms.Session;
import javax.jms.TextMessage;
import javax.xml.transform.Source;

import org.apache.servicemix.jbi.jaxp.SourceTransformer;
import org.apache.servicemix.jbi.jaxp.StringSource;
import org.apache.servicemix.jms.endpoints.JmsProviderMarshaler;

public class widgetProviderMarshaler implements JmsProviderMarshaler
{
    private SourceTransformer transformer = new SourceTransformer();

    public Message createMessage(MessageExchange exchange, NormalizedMessage in, Session
session) throws Exception
    {
       TextMessage text = session.createTextMessage();
       text.setText(transformer.contentToString(in));
       return text;
    }

   public void populateMessage(Message message, MessageExchange exchange, NormalizedMessage
 normalizedMessage) throws Exception
    {
      TextMessage textMessage = (TextMessage) message;
      Source source = new StringSource(textMessage.getText());
```

```
      normalizedMessage.setContent(source);
    }
}
```

Configuring the provider

You configure a provider to use a custom marshaler using its `marshaler` attribute. The `marshaler` attribute's value is a reference to a `bean` element specifying the class of your custom marshaler implementation.

Example 6.6 on page 81 shows configuration for a provider that uses a custom marshaler.

Example 6.6. Configuring a Provider to Use a Customer Marshaler

```
<beans xmlns:jms="http://servicemix.apache.org/jms/1.0"
       ... >
  ...
  <jms:soap-provider wsdl="classpath:widgets.wsdl"
                     destinationName="widgetQueue"
                     connectionFactory="#connectionFactory"
                     marshaler="#myProviderMarshaler" />

  <bean id="myProviderMarshaler" class="com.widgetVendor.example.widgetProviderMarshaler"
/>
  ...
</beans>
```

Note

You can also configure a provider to use a custom marshaler by adding a child `marshaler` element to the provider's configuration. The `marshaler` element simply wraps the `bean` element that configures the marshaler.

Chapter 7. Implementing Destination Resolving Logic

You can provide logic that allows your JMS endpoints to resolve destinations at run time. This is done by providing an implementation of the `DestinationChooser` *interface or the* `DestinationResolver` *interface.*

It may not always be appropriate to hard code destinations into applications. Instead, you may want to allow the endpoints to dynamically discover the JMS destinations. The Fuse ESB Enterprise JMS binding component provides two mechanisms for endpoints to dynamically discover destinations:

destination choosers
: Destination choosers are specific to the Fuse ESB Enterprise JMS binding component. They are the first mechanism used by an endpoint when it trys to pick a JMS destination.

  Destination choosers implement the `org.apache.servicemix.jms.endpoints.DestinationChooser` interface.

destination resolvers
: Destination resolvers are part of the Spring JMS framework. They are used when the JMS destination is specified using a string. This can happen if either the destination chooser returns a string or if the endpoint's destination is configured using the `destinationName` attribute.

  Destination resolvers implement the `org.springframework.jms.support.destination.DestinationResolver` interface.

Using a Custom Destination Chooser

Overview

Provider endpoints use a destination chooser to determine the JMS destination on which to send requests and receive replies. They have a default destination chooser that queries the message exchange for a property that specifies the destination to use. Consumer endpoints use destination choosers to determine where to send reply messages. In both cases, the destination chooser is the first method employed by an endpoint when looking for a JMS destination. If the destination chooser returns a destination, or a destination name, the endpoint will use the returned value.

To customize the logic used in choosing a destination, you can provide an implementation of the `org.apache.servicemix.jms.endpoints.DestinationChooser` interface and configure the endpoint to load it. The configured destination chooser will be used in place of the default destination chooser.

Implementing a destination chooser

Destination choosers implement the `org.apache.servicemix.jms.endpoints.DestinationChooser` interface. This interface has a single method: `chooseDestination()`.

`chooseDestination()`, whose signature is shown in Example 7.1 on page 84, takes the JBI message exchange and a copy of the message. It returns either a JMS `Destination` object or a string representing the destination name.

Note

If the destination chooser returns a string, the endpoint will use a destination resolver to convert the string into a JMS destination. See "Using a Custom Destination Resolver" on page 87.

Example 7.1. Destination Chooser Method

```
Object chooseDestination(MessageExchange exchange,
                         Object message);
```

The *`message`* parameter can be either of the following type of object:

- `javax.jbi.messaging.NormalizedMessage`
- `javax.jbi.messaging.Fault`

- `Exception`

Example 7.2 on page 85 shows a simple destination chooser implementation. It checks the message for a property that represents the JMS destination on which the request is to be placed.

Example 7.2. Simple Destination Chooser

```
package com.widgetVendor.example;

import package org.apache.servicemix.jms.endpoints.DestinationChooser;
import javax.jbi.messaging.MessageExchange;
import javax.jbi.messaging.NormalizedMessage;
import javax.jms.Destination;

public class widgetDestinationChooser implements DestinationChooser {

    public static final String DESTINATION_KEY = "org.apache.servicemix.jms.destination";

    public SimpleDestinationChooser() {
    }

    public Object chooseDestination(MessageExchange exchange, Object message) {
        Object property = null;
        if (message instanceof NormalizedMessage) {
            property = ((NormalizedMessage) message).getProperty(DESTINATION_KEY);
        }
        if (property instanceof Destination) {
            return (Destination) property;
        }
        if (property instanceof String) {
            return (String) property;
        }
        return new String("widgetDest");
    }
}
```

Configuring an endpoint to use a destination chooser

You can configure an endpoint to use a custom destination chooser in one of two ways. The recommended way is to configure the destination chooser as a bean and have the endpoint reference the destination chooser's bean. The other way is to explicitly include the destination chooser's configuration as a child of the endpoint.

As shown in Example 7.3 on page 86, configuring an endpoint's destination chooser using a bean reference is a two step process:

1. Configure a `bean` element for your destination chooser.
2. Add a `destinationChooser` attribute that references the destination chooser's bean to your endpoint.

Example 7.3. Configuring a Destination Chooser with a Bean Reference

```
<beans xmlns:jms="http://servicemix.apache.org/jms/1.0"
       ... >
  ...
  <jms:provider service="my:widgetService"
                endpoint="jbiWidget"
                destinationName="widgetQueue"
                connectionFactory="#connectionFactory"
                destinationChooser="#widgetDestinationChooser" />
  <bean id="widgetDestinationChooser"
        class="com.widgetVendor.example.widgetDestinationChooser" />
  ...
</beans>
```

Example 7.4 on page 86 shows an example configuration using the `jms:destinationChooser` element. This method is less flexible than the recommended method because other endpoints cannot reuse the destination chooser's configuration.

Example 7.4. Explicitly Configuring a Destination Chooser

```
<beans xmlns:jms="http://servicemix.apache.org/jms/1.0"
       ... >
  ...
  <jms:provider service="my:widgetService"
                endpoint="jbiWidget"
                destinationName="widgetQueue"
                connectionFactory="#connectionFactory">
    <jms:destinationChooser>
      <bean id="widgetDestinationChooser"
            class="com.widgetVendor.example.widgetDestinationChooser" />
    </jms:destinationChooser>
  </jms:provider>
  ...
</beans>
```

Using a Custom Destination Resolver

Overview

Destination resolvers are a part of the JMS technology Fuse ESB Enterprise inherits from the Spring Framework. They convert string destination names into JMS `Destination` objects. For example, if you specify an endpoint's destination using the `destinationName` attribute, the endpoint will use a destination resolver to get the appropriate JMS `Destination` object. Destination resolvers are also used if a destination chooser returns a string and not a JMS `Destination` object.

Fuse ESB Enterprise JMS endpoints default to using the `DynamicDestinationResolver` destination resolver provided by the Spring Framework. This destination resolver uses the standard JMS `Session.createTopic()` and `Session.createQueue()` methods to resolve destination names.

Fuse ESB Enterprise JMS endpoints can also use the Spring Framework's `JndiDestinationResolver` destination resolver. This destination resolver uses the string destination name to perform a JNDI lookup for the JMS destination. If JMS destination is not returned from the JNDI lookup, the resolver resorts to dynamically resolving the destination name. For information on configuring and endpoint to use the `JndiDestinationResolver` destination resolver. See "Configuring an endpoint to use a destination resolver" on page 88.

Implementing a destination resolver

Destination resolvers implement the

`org.springframework.jms.support.destination.DestinationResolver` interface. The interface has a single method: `resolveDestinationName()`.

The `resolveDestinationName()` method, whose signature shown in Example 7.5 on page 87, takes three parameters: a JMS session, a destination name, and a boolean specifying if the destination is a JMS topic.[1] It returns a JMS destination that correlates to the provided destination name.

Example 7.5. Destination Resolver Method

```
Destination resolveDestinationName(Session session,
                                   String destinationName,
```

[1] If the value is `false`, a JMS queue will be returned.

```
                                                    boolean pubSubDomain)
                                    throws JMSException;
```

Example 7.6 on page 88 shows a simple destination resolver implementation.

Example 7.6. Simple Destination Resolver

```
package com.widgetVendor.example;

import org.springframework.jms.support.destination.DestinationResolver;
import javax.jms.Destination;
import javax.jms.JMSException;
import javax.jms.Session;

public class widgetDestinationResolver implements DestinationResolver
{

  public Destination resolveDestinationName(Session session,
                                            String destinationName,
                                            boolean pubSubDomain)
  throws JMSException
  {
    if (pubSubDomain)
    {
      return session.createTopic(destinationName);
    }
    else
    {
      return session.createQueue(destinationName);
    }
  }
}
```

Configuring an endpoint to use a destination resolver

You can configure an endpoint to use a custom destination resolver in one of two ways. The recommended way is to configure the destination resolver as a bean and have the endpoint reference the destination resolver's bean. The other way is to explicitly include the destination resolver's configuration as a child of the endpoint.

As shown in Example 7.7 on page 89, configuring an endpoint's destination resolver using a bean reference is a two step process:

1. Configure a `bean` element for your destination resolver.
2. Add a `destinationResolver` attribute that references the destination resolver's bean to your endpoint.

Example 7.7. Configuring a Destination Resolver with a Bean Reference

```
<beans xmlns:jms="http://servicemix.apache.org/jms/1.0"
       ... >
  ...
  <jms:consumer service="my:widgetService"
                endpoint="jbiWidget"
                destinationName="widgetQueue"
                connectionFactory="#connectionFactory"
                destinationResolver="#widgetDestinationResolver" />
  <bean id="widgetDestinationResolver"
        class="com.widgetVendor.example.widgetDestinationResolver" />
  ...
</beans>
```

Example 7.8 on page 89 shows an example configuration using the `jms:destinationResolver` element. This method is less flexible than the recommended method because other endpoints cannot reuse the destination resolver's configuration.

Example 7.8. Explicitly Configuring a Destination Resolver

```
<beans xmlns:jms="http://servicemix.apache.org/jms/1.0"
       ... >
  ...
  <jms:consumer service="my:widgetService"
                endpoint="jbiWidget"
                destinationName="widgetQueue"
                connectionFactory="#connectionFactory">
    <jms:destinationResolver>
      <bean id="widgetDestinationResolver"
            class="com.widgetVendor.example.widgetDestinationResolver" />
    </jms:destinationChooser>
  </jms:consumer>
  ...
</beans>
```

Appendix A. Consumer Endpoint Properties

Common Properties

Attributes

The attributes described in Table A.1 on page 92 can be used on all elements used to configure a consumer endpoint.

Table A.1. Common Consumer Endpoint Property Attributes

Name	Type	Description	Required
`connectionFactory`	string	Specifies a reference to the bean configuring the connection factory which is to be used by the endpoint.	yes
`service`	QName	Specifies the service name of the proxied endpoint.	yes
`endpoint`	string	Specifies the endpoint name of the proxied endpoint.	yes
`interfaceName`	QName	Specifies the interface name of the proxied endpoint.	no
`jms102`	boolean	Specifies if the consumer uses JMS 1.0.2 compliant APIs.	no (defaults to `false`)
`pubSubDomaim`	boolean	Specifies if the destination is a topic.	no
`replyDeliveryMode`	int	Specifies the JMS delivery mode used for the reply.	no (defaults to `PERSISTENT`(2))
`replyDestinationName`	string	Specifies the name of the JMS destination to use for the reply.	no (if not set `replyDestination` or `destinationChooser` is used)
`replyExplicitQosEnabled`	boolean	Specifies if the QoS values specified for the endpoint are explicitly used when the reply is sent.	no (default is false)
`replyPriority`	int	Specifies the JMS message priority of the reply.	no (defaults to 4)

Name	Type	Description	Required
replyTimeToLive	long	Specifies the number of milliseconds the reply message is valid.	no (defaults to unlimited)
stateless	boolean	Specifies if the consumer retains state information about the message exchange while it is in process.	no
synchronous	boolean	Specifies if the consumer will block while waiting for a response. This means the consumer can only process one message at a time.	no (defaults to true)
targetEndpoint	string	Specifies the endpoint name of the target endpoint.	no (defaults to the endpoint attribute)
targetInterface	QName	Specifies the interface name of the target endpoint.	no
targetService	QName	Specifies the service name of the target endpoint.	no (defaults to the service attribute)
targetUri	string	Specifies the URI of the target endpoint.	no
useMessageIdInResponse	boolean	Specifies if the request message's ID is used as the reply's correlation ID.	no (defaults to false meaning the request's correlation ID is used)

Beans

The elements described in Table A.2 on page 93 can be used on all elements used to configure a consumer endpoint.

Table A.2. Common Consumer Endpoint Property Beans

Name	Type	Description	Required
marshaler	JmsConsumerMarshaler	Specifies the class implementing the message marshaler.	no (defaults to DefaultConsumerMarshaler)

Name	Type	Description	Required
destinationChooser	DestinationChooser	Specifies a class implementing logic for choosing reply destinations.	no
destinationResolver	DestinationResolver	Specifies the class implementing logic for converting strings into destination IDs.	no (defaults to DynamicDestinationResolver)
replyDestination	Destination	Specifies the JMS destination for the replies.	no (if not set either the replyDestinationName or the destinationChooser is used)
replyProperties	Map	Specifies custom properties to be placed in the reply's JMS header.	no
storeFactory	StoreFactory	Specifies the factory class used to create the data store for state information.	no (defaults to MemoryStoreFactory)
store	Store	Specifies the data store used to store state information.	no

Properties Specific to Generic Consumers and SOAP Consumers

Common Attributes

The attributes described in Table A.3 on page 95 are specific to the `jms:consumer` element and the `jms:soap-consumer` elements.

Table A.3. Attributes Uses to Configure Standard JMS Consumers and SOAP JMS Consumers

Attribute	Type	Listener(s)	Description	Required
`listenerType`	string	all	Specifies the type of Spring JMS message listener to use. Valid values are `default`, `simple`, and `server`.	no (defaults to `default`)
`transacted`	string	all	Specifies the type of transaction used to wrap the message exchanges. Valid values are `none`, `xa`, and `jms`.	no (defaults to `none`)
`clientId`	string	all	Specifies the JMS client id for a shared `Connection` created and used by this listener.	no
`destinationName`	string	all	Specifies the name of the destination used to receive messages.	no
`durableSubscriptionName`	string	all	Specifies the name used to register the durable subscription.	no
`messageSelector`	string	all	Specifies the message selector string to use.	no
`sessionAcknowlegeMode`	int	all	Specifies the acknowledgment mode that is used	no (defaults to `Session.AUTO_ACKNOWLEDGE`)

Attribute	Type	Listener(s)	Description	Required
			when creating a `Session` to send a message.	
`subscriptionDurable`	boolean	all	Specifies if the listener uses a durable subscription to listen form messages.	no (defaults to `false`)
`pubSubNoLocal`	boolean	default simple	Specifies if messages published by the listener's `Connection` are suppressed.	no (defaults to `false`)
`concurrentConsumers`	int	default simple	Specifies the number of concurrent consumers created by the listener.	no (defaults to `1`)
`cacheLevel`	int	default	Specifies the level of caching allowed by the listener.	no (defaults to `0`)
`receiveTimeout`	long	default	Specifies the timeout for receiving a message in milliseconds.	no (default is `1000`)
`recoveryInterval`	long	default	Specifies the interval, in milliseconds, between attempts to recover after a failed listener set-up.	no (defaults to `5000`)
`maxMessagesPerTask`	int	default server	Specifies the number of attempts to receive messages per task.	no (defaults to `-1`)

Common Beans

The elements described in Table A.4 on page 97 are specific to the `jms:consumer` element and the `jms:soap-consumer` elements.

Table A.4. Elements Uses to Configure Standard JMS Consumers and SOAP JMS Consumers

Element	Type	Listener(s)	Description	Required
`destination`	`Destination`	all	Specifies the destination used to receive messages.	no
`exceptionListener`	`ExceptionListener`	all	Specifies an `ExceptionListener` to notify in case of a `JMSException` is thrown by the registered message listener or the invocation infrastructure.	no
`serverSessionFactory`	`ServerSessionFactory`	server	Specifies the `ServerSessionFactory` to use.	no (defaults to `SimpleServerSessionFactory`)

SOAP consumer specific attributes

The attributes described in Table A.5 on page 97 are specific to the `jms:soap-consumer` element.

Table A.5. Attributes for the JMS SOAP Consumer

Attribute	Type	Description	Required
`wsdl`	string	Specifies the WSDL describing the service.	yes
`useJbiWrapper`	boolean	Specifies if the JBI wrapper is sent in the body of the message.	no (defaults to `true`)
`validateWsdl`	boolean	Specifies if the WSDL is checked WSI-BP compliance.	no (defaults to `true`)
`policies`	`Policy[]`	Specifies a list of interceptors used to process the message.	no

Properties Specific to a JCA Consumer

The elements described in Table A.6 on page 98 are specific to the `jms:jca-consumer` element.

Table A.6. Elements Used to Configure a JCA Consumer

Element	Type	Description	Required
`resourceAdapter`	`ResourceAdapter`	Specifies the resource adapter used for the endpoint.	yes
`activationSpec`	`ActivationSpec`	Specifies the activation information needed by the endpoint.	yes
`bootstrapContext`	`BootstrapContext`	Specifies the bootstrap context used when starting the resource adapter.	no (a default one will be created)

Appendix B. Provider Endpoint Properties

Common Properties

Attributes

The attributes described in Table B.1 on page 100 can be used on all elements used to configure a provider endpoint.

Table B.1. Common Provider Endpoint Property Attributes

Attribute	Type	Description	Required
`connectionFactory`	string	Specifies a reference to the bean which configure the connection factory to be used by the endpoint.	yes
`deliveryMode`	int	Specifies the JMS delivery mode.	no (defaults to persistent)
`destinationName`	string	Specifies the JNDI name of the destination used to send messages.	no
`endpoint`	string	Specifies the endpoint name of the proxied endpoint.	yes
`explicitQosEnabled`	boolean	Specifies if the JMS messages have the specified properties explicitly applied.	no (defaults to `false`)
`interfaceName`	QName	Specifies the interface name of the proxied endpoint.	no
`jms102`	boolean	Specifies if the provider is to be JMS 1.0.2 compatible.	no (defaults to `false`)
`messageIdEnabled`	boolean	Specifies if JMS message IDs are enabled.	no (defaults to `true`)
`messageTimeStampEnabled`	boolean	Specifies if JMS messages are time stamped.	no (defaults to `true`)
`priority`	int	Specifies the priority assigned to the JMS messages.	no (defaults to `4`)
`pubSubDomain`	boolean	Specifies if the destination is a topic.	no (defaults to `false`

Attribute	Type	Description	Required
pubSubNoLocal	boolean	Specifies if messages published by the listener's Connection are suppressed.	no (defaults to false)
recieveTimeout	long	Specifies the timeout for receiving a message in milliseconds.	no (defaults to unlimited)
replyDestinationName	string	Specifies the JNDI name of the destination used to receive messages.	no
service	QName	Specifies the service name of the proxied endpoint.	yes
stateless	boolean	Specifies if the consumer retains state information about the message exchange while it is in process.	no (defaults to false)
timeToLive	long	Specifies the number of milliseconds the message is valid.	no (defaults to unlimited)

Beans

The elements described in Table B.2 on page 101 can be used on all elements used to configure a JMS provider endpoint.

Table B.2. Common Provider Endpoint Property Beans

Element	Type	Description	Required
destination	Destination	Specifies the JMS destination used to send messages.	no
destinationChooser	DestinationChooser	Specifies a class implementing logic for choosing the JMS destinations.	no (defaults to SimpleDestinationChooser)
destinationResolver	DestinationResolver	Specifies a class implementing logic for converting strings into destination IDs.	no (defaults to DynamicDestinationResolver)

Element	Type	Description	Required
marshaler	JmsProviderMarshaler	Specifies the class implementing the message marshaler.	no (defaults to DefaultProviderMarshaler or JmsSoapProviderMarshaler)
replyDestination	Destination	Specifies the JMS destination used to receive messages.	no
replyDestinationChooser	DestinationChooser	Specifies a class implementing logic for choosing the destination used to receive replies.	no (defaults to SimpleDestinationChooser)
storeFactory	StoreFactory	Specifies the factory class used to create the data store for state information.	no (defaults to MemoryStoreFactory)
store	Store	Specifies the data store used to store state information.	no

Properties Specific to SOAP Providers

Attributes

The attributes described in Table B.3 on page 103 are specific to `jms:soap-provider` elements.

Table B.3. Attributes Used to Configure SOAP JMS Providers

Attribute	Type	Description	Required
`useJbiWrapper`	boolean	Specifies if the JBI wrapper is sent in the body of the message.	no (defaults to `true`)
`validateWsdl`	boolean	Specifies if the WSDL is checked for WSI-BP compliance.	no (defaults to `true`)
`wsdl`	string	Specifies the location of the WSDL describing the service.	yes

Beans

The elements described in Table B.4 on page 103 are specific to `jms:soap-provider` elements.

Table B.4. Elements Used to Configure SOAP JMS Providers

Element	Type	Description	Required
`policies`	`Policy[]`	Specifies a list of interceptors that will process the message.	no

Appendix C. Using the Maven JBI Tooling

Packaging application components so that they conform the JBI specification is a cumbersome job. Fuse ESB Enterprise includes tooling that automates the process of packaging you applications and creating the required JBI descriptors.

Fuse ESB Enterprise provides a Maven plug-in and a number of Maven archetypes that make developing, packaging, and deploying JBI artifacts easier. The tooling provides you with a number of benefits including:

- automatic generation of JBI descriptors
- dependency checking

Because Fuse ESB Enterprise only allows you to deploy service assemblies, you will need to do the following when using the Maven JBI tooling:

1. Set up a top-level project on page 106 to build all of the service units and the final service assembly.
2. Create a project for each of your service units. on page 111.
3. Create a project for the service assembly on page 117.

Setting up a Fuse ESB Enterprise JBI project

Overview

When working with the Fuse ESB Enterprise JBI Maven tooling, you create a top-level project that can build all of the service units and then package them into a service assembly. Using a top-level project for this purpose has several advantages:

- It allows you to control the dependencies for all of the parts of an application in a central location.
- It limits the number of times you need to specify the proper repositories to load.
- It provides you a central location from which to build and deploy the application.

The top-level project is responsible for assembling the application. It uses the Maven assembly plug-in and lists your service units and the service assembly as modules of the project.

Directory structure

Your top-level project contains the following directories:

- A source directory containing the information required for the Maven assembly plug-in
- A directory to store the service assembly project
- At least one directory containing a service unit project

Tip

You will need a project folder for each service unit that is to be included in the generated service assembly.

Setting up the Maven tools

To use the Fuse ESB Enterprise JBI Maven tooling, add the elements shown in Example C.1 to your top-level POM file.

Example C.1. POM elements for using Fuse ESB Enterprise Maven tooling

```
...
<pluginRepositories>
```

```
  <pluginRepository>
    <id>fusesource.m2</id>
    <name> Open Source Community Release Repository</name>
    <url>http://repo.fusesource.com/maven2</url>
    <snapshots>
      <enabled>false</enabled>
    </snapshots>
    <releases>
      <enabled>true</enabled>
    </releases>
  </pluginRepository>
</pluginRepositories>
<repositories>
  <repository>
    <id>fusesource.m2</id>
    <name> Open Source Community Release Repository</name>
    <url>http://repo.fusesource.com/maven2</url>
    <snapshots>
       <enabled>false</enabled>
    </snapshots>
    <releases>
      <enabled>true</enabled>
    </releases>
  </repository>
  <repository>
    <id>fusesource.m2-snapshot</id>
    <name> Open Source Community Snapshot Repository</name>
    <url>http://repo.fusesource.com/maven2-snapshot</url>
    <snapshots>
      <enabled>true</enabled>
    </snapshots>
    <releases>
      <enabled>false</enabled>
    </releases>
  </repository>
</repositories>
  ...
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.servicemix.tooling</groupId>
      <artifactId>jbi-maven-plugin</artifactId>
      <version>servicemix-version</version>
      <extensions>true</extensions>
    </plugin>
  </plugins>
</build>
  ...
```

These elements point Maven to the correct repositories to download the Fuse ESB Enterprise Maven tooling and to load the plug-in that implements the tooling.

Listing the sub-projects

The top-level POM lists all of the service units and the service assembly that is generated as modules. The modules are contained in a `modules` element. The `modules` element contains one `module` element for each service unit in the assembly. You also need a `module` element for the service assembly.

The modules are listed in the order in which they are built. This means that the service assembly module is listed after all of the service unit modules.

Example JBI project pOM

Example C.2 shows a top-level POM for a project that contains a single service unit.

Example C.2. Top-level POM for a Fuse ESB Enterprise JBI project

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                             http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <parent>
    <groupId>com.widgets</groupId>
    <artifactId>demos</artifactId>
    <version>1.0</version>
  </parent>

  <groupId>com.widgets.demo</groupId>
  <artifactId>cxf-wsdl-first</artifactId>
  <name>CXF WSDL Fisrt Demo</name>
  <packaging>pom</packaging>

  <pluginRepositories> ❶
    <pluginRepository>
      <id>fusesource.m2</id>
      <name> Open Source Community Release Repository</name>
      <url>http://repo.fusesource.com/maven2</url>
      <snapshots>
        <enabled>false</enabled>
      </snapshots>
      <releases>
        <enabled>true</enabled>
      </releases>
    </pluginRepository>
```

```
</pluginRepositories>
<repositories>
  <repository>
    <id>fusesource.m2</id>
    <name> Open Source Community Release Repository</name>
    <url>http://repo.fusesource.com/maven2</url>
    <snapshots>
       <enabled>false</enabled>
    </snapshots>
    <releases>
      <enabled>true</enabled>
    </releases>
  </repository>
  <repository>
    <id>fusesource.m2-snapshot</id>
    <name> Open Source Community Snapshot Repository</name>
    <url>http://repo.fusesource.com/maven2-snapshot</url>
    <snapshots>
      <enabled>true</enabled>
    </snapshots>
    <releases>
      <enabled>false</enabled>
    </releases>
  </repository>
</repositories>

<modules> ❷
  <module>wsdl-first-cxfse-su</module>
   <module>wsdl-first-cxf-sa</module>
</modules>

<build>
  <plugins>
    <plugin> ❸
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-assembly-plugin</artifactId>
       <version>2.1</version>
       <inherited>false</inherited>
         <executions>
           <execution>
              <id>src</id>
              <phase>package</phase>
              <goals>
                <goal>single</goal>
              </goals>
              <configuration>
                <descriptors>
                  <descriptor>src/main/assembly/src.xml</descriptor>
                </descriptors>
```

```
                </configuration>
              </execution>
            </executions>
          </plugin>
          <plugin> ❹
            <groupId>org.apache.servicemix.tooling</groupId>
            <artifactId>jbi-maven-plugin</artifactId>
            <extensions>true</extensions>
          </plugin>
    </plugins>
  </build>
</project>
```

The top-level POM shown in Example C.2 on page 108 does the following:

❶ Configures Maven to use the repositories for loading the Fuse ESB Enterprise plug-ins.

❷ Lists the sub-projects used for this application. The `wsdl-first-cxfse-su` module is the module for the service unit. The `wsdl-first-cxf-sa` module is the module for the service assembly

❸ Configures the Maven assembly plug-in.

❹ Loads the Fuse ESB Enterprise JBI plug-in.

A service unit project

Overview

Each service unit in the service assembly must be its own project. These projects are placed at the same level as the service assembly project. The contents of a service unit's project depends on the component at which the service unit is targeted. At the minimum, a service unit project contains a POM and an XML configuration file.

Seeding a project using a Maven artifact

Fuse ESB Enterprise provides Maven artifacts for a number of service unit types. They can be used to seed a project with the **smx-arch** command. As shown in Example C.3, the **smx-arch** command takes three arguments. The `groupId` value and the `artifactId` values correspond to the project's group ID and artifact ID.

Example C.3. Maven archetype command for service units

```
smx-arch su suArchetypeName ["-DgroupId=my.group.id"]
["-DartifactId=my.artifact.id"]
```

Important

The double quotes(") are required when using the `-DgroupId` argument and the `-DartifactId` argument.

The *suArchetypeName* specifies the type of service unit to seed. Table C.1 lists the possible values and describes what type of project is seeded.

Table C.1. Service unit archetypes

Name	Description
camel	Creates a project for using the Apache Camel service engine
cxf-se	Creates a project for developing a Java-first service using the Apache CXF service engine
cxf-se-wsdl-first	Creates a project for developing a WSDL-first service using the Apache CXF service engine
cxf-bc	Creates an endpoint project targeted at the Apache CXF binding component

Name	Description
http-consumer	Creates a consumer endpoint project targeted at the HTTP binding component
http-provider	Creates a provider endpoint project targeted at the HTTP binding component
jms-consumer	Creates a consumer endpoint project targeted at the JMS binding component (see *Using the JMS Binding Component* on page 3)
jms-provider	Creates a provider endpoint project targeted at the JMS binding component (see *Using the JMS Binding Component* on page 3)
file-poller	Creates a polling (consumer) endpoint project targeted at the file binding component (see "Using Poller Endpoints" in *Using the File Binding Component*)
file-sender	Creates a sender (provider) endpoint project targeted at the file binding component (see "Using Sender Endpoints" in *Using the File Binding Component*)
ftp-poller	Creates a polling (consumer) endpoint project targeted at the FTP binding component
ftp-sender	Creates a sender (provider) endpoint project targeted at the FTP binding component
jsr181-annotated	Creates a project for developing an annotated Java service to be run by the JSR181 service engine [a]
jsr181-wsdl-first	Creates a project for developing a WSDL generated Java service to be run by the JSR181 service engine [a]

Name	Description
saxon-xquery	Creates a project for executing xquery statements using the Saxon service engine
saxon-xslt	Creates a project for executing XSLT scripts using the Saxon service engine
eip	Creates a project for using the EIP service engine. [b]
lwcontainer	Creates a project for deploying functionality into the lightweight container [c]
bean	Creates a project for deploying a POJO to be executed by the bean service engine
ode	Create a project for deploying a BPEL process into the ODE service engine

[a]The JSR181 has been deprecated. The Apache CXF service engine has superseded it.

[b]The EIP service engine has been deprecated. The Apache Camel service engine has superseded it.

[c]The lightweight container has been deprecated.

Contents of a project

The contents of your service unit project change from service unit to service unit. Different components require different configuration. Some components, such as the Apache CXF service engine, require that you include Java classes.

At a minimum, a service unit project will contain two things:

- a POM file that configures the JBI plug-in to create a service unit
- an XML configuration file stored in `src/main/resources`

  For many of the components, the XML configuration file is called `xbean.xml`. The Apache Camel component uses a file called `camel-context.xml`.

Configuring the Maven plug-in

You configure the Maven plug-in to package the results of the project build as a service unit by changing the value of the project's `packaging` element to `jbi-service-unit` as shown in Example C.4.

Example C.4. Configuring the maven plug-in to build a service unit

```
<project ...>
  <modelVersion>4.0.0</modelVersion>

  ...
  <groupId>com.widgets.demo.cxf-wsdl-first</groupId>
  <artifactId>cxfse-wsdl-first-su</artifactId>
  <name>CXF WSDL Fisrt Demo :: SE Service Unit</name>
  <packaging>jbi-service-unit</packaging>
  ...
</project>
```

Specifying the target components

To correctly fill in the metadata required for packaging a service unit, the Maven plug-in must be told what component (or components) the service unit is targeting. If your service unit only has a single component dependency, you can specify it in one of two ways:

- List the targeted component as a dependency
- Add a `componentName` property specifying the targeted component

If your service unit has more than one component dependency, you must configure the project as follows:

1. Add a `componentName` property specifying the targeted component.
2. Add the remaining components to the list dependencies.

Example C.5 shows the configuration for a service unit targeting the Apache CXF binding component.

Example C.5. Specifying the target components for a service unit

```
...
<dependencies>
  <dependency>
    <groupId>org.apache.servicemix</groupId>
    <artifactId>servicemix-cxf-bc</artifactId>
    <version>3.3.1.0-fuse</version>[1]
  </dependency>
>/dependencies>
...
```

[1] You replace this with the version of Apache CXF you are using.

The advantage of using the Maven dependency mechanism is that it allows Maven to verify if the targeted component is deployed in the container. If one of the components is not deployed, Fuse ESB Enterprise will not hold off deploying the service unit until all of the required components are deployed.

Tip

Typically, a message identifying the missing component(s) is written to the log.

If your service unit's targeted component is not available as a Maven artifact, you can specify the targeted component using the `componentName` element. This element is added to the standard Maven properties block and it specifies the name of a targeted component, as specified in Example C.6.

Example C.6. Specifying a target component for a service unit

```
...
<properties>
  <componentName>servicemix-bean</componentName>
</properties>
...
```

When you use the `componentName` element, Maven does not check to see if the component is installed, nor does it download the required component.

Example

Example C.7 shows the POM file for a project that is building a service unit targeted to the Apache CXF binding component.

Example C.7. POM file for a service unit project

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                             http://maven.apache.org/maven-v4_0_0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <parent> ❶
        <groupId>com.widgets.demo</groupId>
        <artifactId>cxf-wsdl-first</artifactId>
        <version>1.0</version>
    </parent>

  <groupId>com.widgets.demo.cxf-wsdl-first</groupId>
  <artifactId>cxfse-wsdl-first-su</artifactId>
```

```
  <name>CXF WSDL Fisrt Demo :: SE Service Unit</name>
  <packaging>jbi-service-unit</packaging> ❷

  <dependencies> ❸
    <dependency>
      <groupId>org.apache.servicemix</groupId>
      <artifactId>servicemix-cxf-bc</artifactId>
      <version>3.3.1.0-fuse</version>
    </dependency>
  >/dependencies>

  <build>
    <plugins>
      <plugin> ❹
        <groupId>org.apache.servicemix.tooling</groupId>
        <artifactId>jbi-maven-plugin</artifactId>
        <extensions>true</extensions>
      </plugin>
    </plugins>
  </build>
</project>
```

The POM file in Example C.7 on page 115 does the following:

❶ Specifies that it is a part of the top-level project shown in Example C.2 on page 108

❷ Specifies that this project builds a service unit

❸ Specifies that the service unit targets the Apache CXF binding component

❹ Specifies to use the Fuse ESB Enterprise Maven plug-in

A service assembly project

Overview

Fuse ESB Enterprise requires that all service units are bundled into a service assembly before they can be deployed to a container. The Fuse ESB Enterprise Maven plug-in collects all of the service units to be bundled and the metadata necessary for packaging. It will then build a service assembly containing the service units.

Seeding a project using a Maven artifact

Fuse ESB Enterprise provides a Maven artifact for seeding a service assembly project. You can seed a project with the **smx-arch** command. As shown in Example C.8, the **smx-arch** command takes two arguments: the `groupId` value and the `artifactId` values, which correspond to the project's group ID and artifact ID.

Example C.8. Maven archetype command for service assemblies

`smx-arch` sa ["-DgroupId=*my.group.id*"] ["-DartifactId=*my.artifact.id*"]

Important

The double quotes(") are required when using the `-DgroupId` argument and the `-DartifactId` argument.

Contents of a project

A service assembly project typically only contains the POM file used by Maven.

Configuring the Maven plug-in

T configure the Maven plug-in to package the results of the project build as a service assembly, change the value of the project's `packaging` element to `jbi-service-assembly`, as shown in Example C.9.

Example C.9. Configuring the Maven plug-in to build a service assembly

```
<project ...>
  <modelVersion>4.0.0</modelVersion>

  ...
  <groupId>com.widgets.demo.cxf-wsdl-first</groupId>
  <artifactId>cxf-wsdl-first-sa</artifactId>
  <name>CXF WSDL Fisrt Demo :: Service Assembly</name>
  <packaging>jbi-service-assembly</packaging>
```

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

Specifying the target components

The Maven plug-in must know what service units are being bundled into the service assembly. This is done by specifying the service units as dependencies, using the standard Maven `dependencies` element. Add a `dependency` child element for each service unit. Example C.10 shows the configuration for a service assembly that bundles two service units.

Example C.10. Specifying the target components for a service unit

```
...
<dependencies>
  <dependency>
    <groupId>com.widgets.demo.cxf-wsdl-first</groupId>
    <artifactId>cxfse-wsdl-first-su</artifactId>
    <version>1.0</version>
  </dependency>
  <dependency>
    <groupId>com.widgets.demo.cxf-wsdl-first</groupId>
    <artifactId>cxfbc-wsdl-first-su</artifactId>
    <version>1.0</version>
  </dependency>
</dependencies>
...
```

Example

Example C.11 shows a POM file for a project that is building a service assembly.

Example C.11. POM for a service assembly project

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                             http://maven.apache.org/maven-v4_0_0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <parent> ❶
        <groupId>com.widgets.demo</groupId>
        <artifactId>cxf-wsdl-first</artifactId>
        <version>1.0</version>
    </parent>

  <groupId>com.widgets.demo.cxf-wsdl-first</groupId>
```

```
  <artifactId>cxf-wsdl-first-sa</artifactId>
  <name>CXF WSDL Fisrt Demo ::  Service Assemby</name>
  <packaging>jbi-service-assembly</packaging> ❷

  <dependencies> ❸
    <dependency>
      <groupId>com.widgets.demo.cxf-wsdl-first</groupId>
      <artifactId>cxfse-wsdl-first-su</artifactId>
      <version>1.0</version>
    </dependency>
    <dependency>
      <groupId>com.widgets.demo.cxf-wsdl-first</groupId>
      <artifactId>cxfbc-wsdl-first-su</artifactId>
      <version>1.0</version>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin> ❹
        <groupId>org.apache.servicemix.tooling</groupId>
        <artifactId>jbi-maven-plugin</artifactId>
        <extensions>true</extensions>
      </plugin>
    </plugins>
  </build>
</project>
```

The POM in Example C.11 on page 118 does the following:

❶ Specifies that it is a part of the top-level project shown in Example C.2 on page 108

❷ Specifies that this project builds a service assembly

❸ Specifies the service units being bundled by the service assembly

❹ Specifies to use the Fuse ESB Enterprise Maven plug-in

Appendix D. Using the Maven OSGi Tooling

Manually creating a bundle, or a collection of bundles, for a large project can be cumbersome. The Maven bundle plug-in makes the job easier by automating the process and providing a number of shortcuts for specifying the contents of the bundle manifest.

The Fuse ESB Enterprise OSGi tooling uses the Maven bundle plug-in[1] from Apache Felix. The bundle plug-in is based on the **bnd**[2] tool from Peter Kriens. It automates the construction of OSGi bundle manifests by introspecting the contents of the classes being packaged in the bundle. Using the knowledge of the classes contained in the bundle, the plug-in can calculate the proper values to populate the `Import-Packages` and the `Export-Package` properties in the bundle manifest. The plug-in also has default values that are used for other required properties in the bundle manifest.

To use the bundle plug-in, do the following:

1. Add the bundle plug-in to your project's POM file.
2. Configure the plug-in to correctly populate your bundle's manifest.

[1] http://cwiki.apache.org/FELIX/apache-felix-maven-bundle-plugin-bnd.html
[2] http://www.aqute.biz/Code/Bnd

Setting up a Fuse ESB Enterprise OSGi project

Overview

A Maven project for building an OSGi bundle can be a simple single level project. It does not require any sub-projects. However, it does require that you do the following:

1. Add the bundle plug-in to your POM.
2. Instruct Maven to package the results as an OSGi bundle.

Tip

There are several Maven archetypes you can use to set up your project with the appropriate settings.

Directory structure

A project that constructs an OSGi bundle can be a single level project. It only requires that you have a top-level POM file and a `src` folder. As in all Maven projects, you place all Java source code in the `src/java` folder, and you place any non-Java resources in the `src/resources` folder.

Non-Java resources include Spring configuration files, JBI endpoint configuration files, and WSDL contracts.

Note

Fuse ESB Enterprise OSGi projects that use Apache CXF, Apache Camel, or another Spring configured bean also include a `beans.xml` file located in the `src/resources/META-INF/spring` folder.

Adding a bundle plug-in

Before you can use the bundle plug-in you must add a dependency on Apache Felix. After you add the dependency, you can add the bundle plug-in to the plug-in portion of the POM.

Example D.1 on page 123 shows the POM entries required to add the bundle plug-in to your project.

Example D.1. Adding an OSGi bundle plug-in to a POM

```
...
<dependencies>
  <dependency> ❶
    <groupId>org.apache.felix</groupId>
    <artifactId>org.osgi.core</artifactId>
    <version>1.0.0</version>
  </dependency>
...
</dependencies>
...
<build>
  <plugins>
    <plugin> ❷
      <groupId>org.apache.felix</groupId>
      <artifactId>maven-bundle-plugin</artifactId>
      <configuration>
        <instructions>
          <Bundle-SymbolicName>${pom.artifactId}</Bundle-SymbolicName> ❸
          <Import-Package>*,org.apache.camel.osgi</Import-Package> ❹
          <Private-Package>org.apache.servicemix.examples.camel</Private-Package> ❺
        </instructions>
      </configuration>
    </plugin>
  </plugins>
</build>
...
```

The entries in Example D.1 on page 123 do the following:

❶ Adds the dependency on Apache Felix

❷ Adds the bundle plug-in to your project

❸ Configures the plug-in to use the project's artifact ID as the bundle's symbolic name

❹ Configures the plug-in to include all Java packages imported by the bundled classes; also imports the `org.apache.camel.osgi` package

❺ Configures the plug-in to bundle the listed class, but not to include them in the list of exported packages

Note

Edit the configuration to meet the requirements of your project.

For more information on configuring the bundle plug-in, see "Configuring the Bundle Plug-In" on page 127.

Activating a bundle plug-in

To have Maven use the bundle plug-in, instruct it to package the results of the project as a bundle. Do this by setting the POM file's `packaging` element to `bundle`.

Useful Maven archetypes

There are several Maven archetypes to generate a project that is preconfigured to use the bundle plug-in:

- "Spring OSGi archetype"
- "Apache CXF code-first archetype"
- "Apache CXF wsdl-first archetype"
- "Apache Camel archetype"

Spring OSGi archetype

The Spring OSGi archetype creates a generic project for building an OSGi project using Spring DM, as shown:

```
org.springframework.osgi/spring-bundle-osgi-archetype/1.1.2
```

You invoke the archetype using the following command:

```
mvn archetype:create
-DarchetypeGroupId=org.springframework.osgi
-DarchetypeArtifactId=spring-osgi-bundle-archetype
-DarchetypeVersion=1.12
-DgroupId=groupId
-DartifactId=artifactId
-Dversion=version
```

Apache CXF code-first archetype

The Apache CXF code-first archetype creates a project for building a service from Java, as shown:

```
org.apache.servicemix.tooling/servicemix-osgi-cxf-code-first-archetype/2008.01.0.3-fuse
```

You invoke the archetype using the following command:

```
mvn archetype:create
-DarchetypeGroupId=org.apache.servicemix.tooling
-DarchetypeArtifactId=spring-osgi-bundle-archetype
-DarchetypeVersion=2008.01.0.3-fuse
-DgroupId=groupId
-DartifactId=artifactId
-Dversion=version
```

Apache CXF wsdl-first archetype

The Apache CXF wsdl-first archetype creates a project for creating a service from WSDL, as shown:

```
org.apache.servicemix.tooling/servicemix-osgi-cxf-wsdl-first-archetype/2008.01.0.3-fuse
```

You invoke the archetype using the following command:

```
mvn archetype:create
-DarchetypeGroupId=org.apache.servicemix.tooling
-DarchetypeArtifactId=servicemix-osgi-cxf-wsdl-first-archetype
-DarchetypeVersion=2008.01.0.3-fuse
-DgroupId=groupId
-DartifactId=artifactId
-Dversion=version
```

Apache Camel archetype

The Apache Camel archetype creates a project for building a route that is deployed into Fuse ESB Enterprise, as shown:

```
org.apache.servicemix.tooling/servicemix-osgi-camel-archetype/2008.01.0.3-fuse
```

You invoke the archetype using the following command:

```
mvn archetype:create
-DarchetypeGroupId=org.apache.servicemix.tooling
-DarchetypeArtifactId=servicemix-osgi-camel-archetype
-DarchetypeVersion=2008.01.0.3-fuse
-DgroupId=groupId
-DartifactId=artifactId
-Dversion=version
```

Configuring the Bundle Plug-In

Overview

A bundle plug-in requires very little information to function. All of the required properties use default settings to generate a valid OSGi bundle.

While you can create a valid bundle using just the default values, you will probably want to modify some of the values. You can specify most of the properties inside the plug-in's `instructions` element.

Configuration properties

Some of the commonly used configuration properties are:

- Bundle-SymbolicName
- Bundle-Name
- Bundle-Version
- Export-Package
- Private-Package
- Import-Package

Setting a bundle's symbolic name

By default, the bundle plug-in sets the value for the `Bundle-SymbolicName` property to *`groupId`* `+ "." +` *`artifactId`*, with the following exceptions:

- If *`groupId`* has only one section (no dots), the first package name with classes is returned.

  For example, if the group Id is `commons-logging:commons-logging`, the bundle's symbolic name is `org.apache.commons.logging`.

- If *`artifactId`* is equal to the last section of *`groupId`*, then *`groupId`* is used.

  For example, if the POM specifies the group ID and artifact ID as `org.apache.maven:maven`, the bundle's symbolic name is `org.apache.maven`.

- If *`artifactId`* starts with the last section of *`groupId`*, that portion is removed.

  For example, if the POM specifies the group ID and artifact ID as `org.apache.maven:maven-core`, the bundle's symbolic name is `org.apache.maven.core`.

To specify your own value for the bundle's symbolic name, add a `Bundle-SymbolicName` child in the plug-in's `instructions` element, as shown in Example D.2.

Example D.2. Setting a bundle's symbolic name

```
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <configuration>
   <instructions>
     <Bundle-SymbolicName>${project.artifactId}</Bundle-SymbolicName>
     ...
    </instructions>
  </configuration>
</plugin>
```

Setting a bundle's name

By default, a bundle's name is set to `${project.name}`.

To specify your own value for the bundle's name, add a `Bundle-Name` child to the plug-in's `instructions` element, as shown in Example D.3.

Example D.3. Setting a bundle's name

```
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <configuration>
   <instructions>
     <Bundle-Name>JoeFred</Bundle-Name>
     ...
    </instructions>
  </configuration>
</plugin>
```

Setting a bundle's version

By default, a bundle's version is set to `${project.version}`. Any dashes (`-`) are replaced with dots (`.`) and the number is padded up to four digits. For example, `4.2-SNAPSHOT` becomes `4.2.0.SNAPSHOT`.

To specify your own value for the bundle's version, add a `Bundle-Version` child to the plug-in's `instructions` element, as shown in Example D.4.

Example D.4. Setting a bundle's version

```
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <configuration>
   <instructions>
     <Bundle-Version>1.0.3.1</Bundle-Version>
     ...
    </instructions>
  </configuration>
</plugin>
```

Specifying exported packages

By default, the OSGi manifest's `Export-Package` list is populated by all of the packages in your local Java source code (under `src/main/java`), *except* for the deault package, `.`, and any packages containing `.impl` or `.internal`.

Important

If you use a `Private-Package` element in your plug-in configuration and you do not specify a list of packages to export, the default behavior includes only the packages listed in the `Private-Package` element in the bundle. No packages are exported.

The default behavior can result in very large packages and in exporting packages that should be kept private. To change the list of exported packages you can add an `Export-Package` child to the plug-in's `instructions` element.

The `Export-Package` element specifies a list of packages that are to be included in the bundle and that are to be exported. The package names can be specified using the `*` wildcard symbol. For example, the entry `com.fuse.demo.*` includes all packages on the project's classpath that start with `com.fuse.demo`.

You can specify packages to be excluded be prefixing the entry with `!`. For example, the entry `!com.fuse.demo.private` excludes the package `com.fuse.demo.private`.

When excluding packages, the order of entries in the list is important. The list is processed in order from the beginning and any subsequent contradicting entries are ignored.

For example, to include all packages starting with `com.fuse.demo` except the package `com.fuse.demo.private`, list the packages using:

```
!com.fuse.demo.private,com.fuse.demo.*
```

However, if you list the packages using `com.fuse.demo.*,!com.fuse.demo.private`, then `com.fuse.demo.private` is included in the bundle because it matches the first pattern.

Specifying private packages

If you want to specify a list of packages to include in a bundle *without* exporting them, you can add a `Private-Package` instruction to the bundle plug-in configuration. By default, if you do not specify a `Private-Package` instruction, all packages in your local Java source are included in the bundle.

Important

If a package matches an entry in both the `Private-Package` element and the `Export-Package` element, the `Export-Package` element takes precedence. The package is added to the bundle and exported.

The `Private-Package` element works similarly to the `Export-Package` element in that you specify a list of packages to be included in the bundle. The bundle plug-in uses the list to find all classes on the project's classpath that are to be included in the bundle. These packages are packaged in the bundle, but not exported (unless they are also selected by the `Export-Package` instruction).

Example D.5 shows the configuration for including a private package in a bundle

Example D.5. Including a private package in a bundle

```
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <configuration>
   <instructions>
     <Private-Package>org.apache.cxf.wsdlFirst.impl</Private-Package>
     ...
    </instructions>
  </configuration>
</plugin>
```

Specifying imported packages

By default, the bundle plug-in populates the OSGi manifest's `Import-Package` property with a list of all the packages referred to by the contents of the bundle.

While the default behavior is typically sufficient for most projects, you might find instances where you want to import packages that are not automatically added to the list. The default behavior can also result in unwanted packages being imported.

To specify a list of packages to be imported by the bundle, add an `Import-Package` child to the plug-in's `instructions` element. The syntax for the package list is the same as for the `Export-Package` element and the `Private-Package` element.

Important

When you use the `Import-Package` element, the plug-in does not automatically scan the bundle's contents to determine if there are any required imports. To ensure that the contents of the bundle are scanned, you must place an `*` as the last entry in the package list.

Example D.6 shows the configuration for specifying the packages imported by a bundle

Example D.6. Specifying the packages imported by a bundle

```
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <configuration>
   <instructions>
     <Import-Package>javax.jws,
         javax.wsdl,
         org.apache.cxf.bus,
         org.apache.cxf.bus.spring,
         org.apache.cxf.bus.resource,
         org.apache.cxf.configuration.spring,
         org.apache.cxf.resource,
         org.springframework.beans.factory.config,
         *
     </Import-Package>
     ...
   </instructions>
  </configuration>
</plugin>
```

More information

For more information on configuring a bundle plug-in, see:

- *Managing OSGi Dependencies*
- Apache Felix documentation[3]
- Peter Kriens' aQute Software Consultancy web site[4]

[3] http://felix.apache.org/site/apache-felix-maven-bundle-plugin-bnd.html
[4] http://www.aqute.biz/Code/Bnd

Index

T

U

W

X