Red Hat JBoss Fuse 6.1

JBI Development Guide

Using the legacy Java Business Integration framework

Last Updated: 2017-10-12

Red Hat JBoss Fuse 6.1 JBI Development Guide

Using the legacy Java Business Integration framework

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

Legal Notice

Abstract

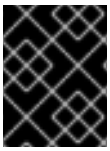
Java Business Integration is a legacy container framework which uses the Normalized Message Router (NMR) as a message bus for transferring normalized XML messages between application components.

Table of Contents

PART I. INTRODUCING JAVA BUSINESS INTEGRATION

Abstract

Provides an overview of JBI, introducing the JBI framework and management structure; describes how to deploy JBI artifacts into the Red Hat JBoss Fuse runtime; and how to use the JBI console commands.



IMPORTANT

The Java Business Integration components of Red Hat JBoss Fuse are considered deprecated. You should consider migrating any JBI applications to OSGi.

CHAPTER 1. INTRODUCTION TO JBI

Abstract

Java Business Integration (JBI) defines an architecture for integrating systems through components that interoperate by exchanging normalized messages through a router.

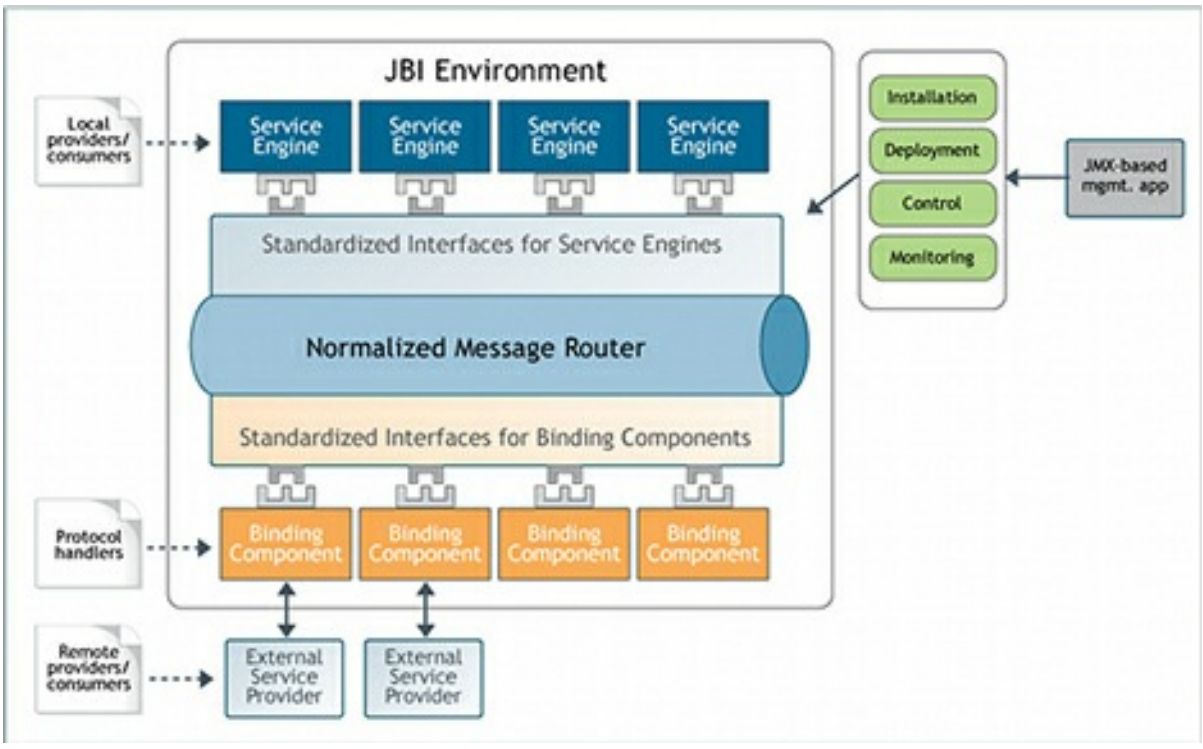
IMPORTANT

The Java Business Integration components of Red Hat JBoss Fuse are considered deprecated. You should consider migrating any JBI applications to OSGi.

The Java Business Integration (JBI) specification defines an integration architecture based on service-oriented concepts. Applications are divided into decoupled functional units. The functional units are deployed into JBI components that are hosted within the JBI environment. The JBI environment provides message normalization and message mediation among the JBI components.

The JBI environment is made up of the following parts, as shown in Figure 1.1, “The JBI architecture”.

Figure 1.1. The JBI architecture



- The JBI component framework hosts and manages the JBI components. For more information see Chapter 2, *The Component Framework*.
- The normalized message router provides message mediation among the JBI components. For more information see Chapter 3, *The Normalized Message Router*.
- The management structure controls the life-cycle of the JBI components and the functional units deployed into the JBI components. It also provides mechanisms for monitoring the artifacts that are deployed into the JBI environment. For more information see Chapter 4, *Management Structure*.

CHAPTER 2. THE COMPONENT FRAMEWORK

Abstract

The JBI component framework is the structure into which JBI components plug into the ESB.

IMPORTANT

The Java Business Integration components of Red Hat JBoss Fuse are considered deprecated. You should consider migrating any JBI applications to OSGi.

OVERVIEW

The JBI component framework provides a pluggable interface between the functional units installed into the JBI environment and the infrastructure services offered by the JBI environment. The framework divides JBI components into two types based on their functionality. The framework also defines a packaging mechanism for deploying functional units into JBI components.

COMPONENT TYPES

JBI defines two types of components:

- **Service Engine** — Component that provides some of the logic required to provide services inside of the JBI environment. For example:
  - message transformation
  - orchestration
  - advanced message routing

  A service engine can communicate only with other components inside of the JBI environment. Service engines act as containers for the functional units deployed into the Red Hat JBoss Fuse.
- **Binding Component** — Provides access to services outside the JBI environment using a particular protocol. Binding components implement the logic required to connect to a transport, and consume the messages received over that transport. Binding components are also responsible for the normalization of messages as they enter the JBI environment.

The distinction between the two types of components is a matter of convention, and this distinction makes the decoupling of business logic and integration logic more explicit.

PACKAGING

JBI defines a common packaging model for all of the artifacts that can be deployed into the JBI environment. Each type of package is a ZIP archive that includes a JBI descriptor in the file **`META-INF/jbi.xml`**. The packages differ based on the root element of the JBI descriptor and the contents of the package. The JBI environment uses four types of packaging to install and deploy functionality. The two most common types used by an application developer are:

- **Service Assembly** — A collection of service units. The root element of the JBI descriptor is a **`service-assembly`** element. The contents of the package is a collection of ZIP archives

containing service units. The JBI descriptor specifies the target JBI component for each of the bundled service units.

- **Service Unit** — A package that contains functionality to be deployed into a JBI component. For example, a service unit intended for a routing service engine contains the definition for one or more routes. Note that service units are packaged as a ZIP file. The root element of the JBI descriptor is a **`service-unit`** element. The contents of the package are specific to the service engine for which the service unit is intended.

IMPORTANT

Service units cannot be installed without being bundled into a service assembly.

COMPONENT ROLES

Once configured by one or more service units, a JBI component implements the functionality described in the service unit. The JBI component then takes on one of the following roles:

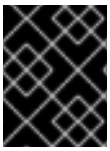
- **Service Provider** — Receives request messages and returns response messages, when required.
- **Service Consumer** — Initiates message exchanges by sending requests to a service provider.

Depending on both the number and the type of service units deployed into a JBI component, a single component can play one or both roles. For example, the HTTP binding component could host a service unit that acts as a proxy to consumers running outside of the Red Hat JBoss Fuse. In this instance, the HTTP component is playing the role of a service provider because it is receiving requests from the external consumer, and passing the responses back to the external consumer. If the service unit also configures the HTTP component to forward the requests to another process running inside of the JBI environment, then the HTTP component also plays the role of a service consumer because it is making requests on another service unit.

CHAPTER 3. THE NORMALIZED MESSAGE ROUTER

Abstract

The normalized message router is a bus that shuttles messages between the endpoints deployed on the ESB.



IMPORTANT

The Java Business Integration components of Red Hat JBoss Fuse are considered deprecated. You should consider migrating any JBI applications to OSGi.

OVERVIEW

The *normalized message router*(NMR) is the part of the JBI environment that is responsible for mediating messages between JBI components. The JBI components never send messages directly to each other; instead, they pass messages to the NMR, which is responsible for delivering the messages to the correct JBI endpoints. This allows the JBI components, and the functionality they expose, to be location independent. It also frees the application developer from concerns about the connection details between the different parts of an application.

MESSAGE EXCHANGE PATTERNS

The NMR uses a WSDL-based messaging model to mediate the message exchanges between JBI components. Using a WSDL-based model provides the necessary level of abstraction to ensure that the JBI components are fully decoupled. The WSDL-based model defines operations as a message exchange between a service provider and a service consumer. The message exchanges are defined from the point of view of the service provider and fit into one of four message exchange patterns:

in-out

A consumer sends a request message to a provider, which then responds to the request with a response message. The provider might also respond with a fault message if an error occured during processing.

in-optional-out

A consumer sends a request message to a provider. The provider might send a response message back to the consumer, but the consumer does not require a response. The provider might also respond with a fault message if an error occurred during processing. The consumer can also send a fault message to the provider.

in-only

A consumer sends a message to a provider, but the provider does not send a response, and, if an error occurs, the provider does not send fault messages back to the consumer.

robust-in-only

A consumer sends a message to a provider. The provider does not respond to the consumer except to send a fault message back to the consumer to signal an error condition.

NORMALIZED MESSAGES

To completely decouple the entities involved in message exchanges, JBI uses *normalized messages*. A normalized message is a genericized format used to represent all of the message data passed through the NMR and consists of the following three parts:

meta-data, properties

Holds information about the message. This information can include transaction contexts, security information, or other QoS information. The meta-data can also hold transport headers.

payload

An XML document that conforms to the XML Schema definition in the WSDL document that defines the message exchange. The XML document holds the substance of the message.

attachments

Hold any binary data associated with the message. For example, an attachment can be an image file sent as an attachment to a SOAP message.

security `Subject`

Holds security information associated with the message, such as authentication credentials. For more information about the security **`Sublect`**, see Sun's API documentation.

JBI binding components are responsible for normalizing all of the messages placed into the NMR. Binding components normalize messages received from external sources before passing them to the NMR. The binding component also denormalizes the message so that it is in the appropriate format for the external source.

CHAPTER 4. MANAGEMENT STRUCTURE

Abstract

The JBI specification mandates that most parts of the environment are managed through JMX.

IMPORTANT

The Java Business Integration components of Red Hat JBoss Fuse are considered deprecated. You should consider migrating any JBI applications to OSGi.

OVERVIEW

The JBI environment is managed using JMX (Java Management Extensions). The internal components of the JBI environment provide a set of MBeans that facilitate the management of the JBI environment and the deployed components. The JBI environment also supplies a number of Apache Ant tasks to manage the JBI environment.

The management of the JBI environment largely consists of:

- Installing and uninstalling artifacts into the JBI container
- Managing the life-cycle of JBI components
- Managing the life-cycle of service units

In addition to the JMX interface, all JBI environments provide a number of Ant tasks, which make it possible to automate many of the common management tasks.

JMX

Java Management Extensions (JMX) is a standard technology for monitoring and managing Java applications. The foundations for using JMX are provided as part of the standard Java 5 JVM, and can be used by any Java application. JMX provides a lightweight way of providing monitoring and management capabilities to any Java application that implements the **`MBean`** interface.

JBI implementations provide MBeans that can be used to manage the components installed into the container and the service units deployed into the components. In addition, application developers can add MBeans to their service units to add additional management touch points.

The MBeans can be accessed using any management console that uses JMX. **JConsole**, the JMX console provided with the Java 5 JRE, is an easy to use, free tool for managing a JBI environment. JBoss ON (JON), available through the Red Hat Customer Portal at access.redhat.com, provides a more robust management console.

INSTALLING AND UNINSTALLING ARTIFACTS INTO THE JBI ENVIRONMENT

There are four basic types of artifacts that can be installed into a JBI environment:

- JBI components
- Shared libraries

- Service assemblies
- Service units

JBI components and shared libraries are installed using the **`InstallationService`** MBean that is exposed through the JMX console. In addition, the following Ant tasks are provided for installing and uninstalling JBI components and shared libraries:

- **`InstallComponentTask`**
- **`UninstallComponentTask`**
- **`InstallSharedLibraryTask`**
- **`UninstallSharedLibraryTask`**

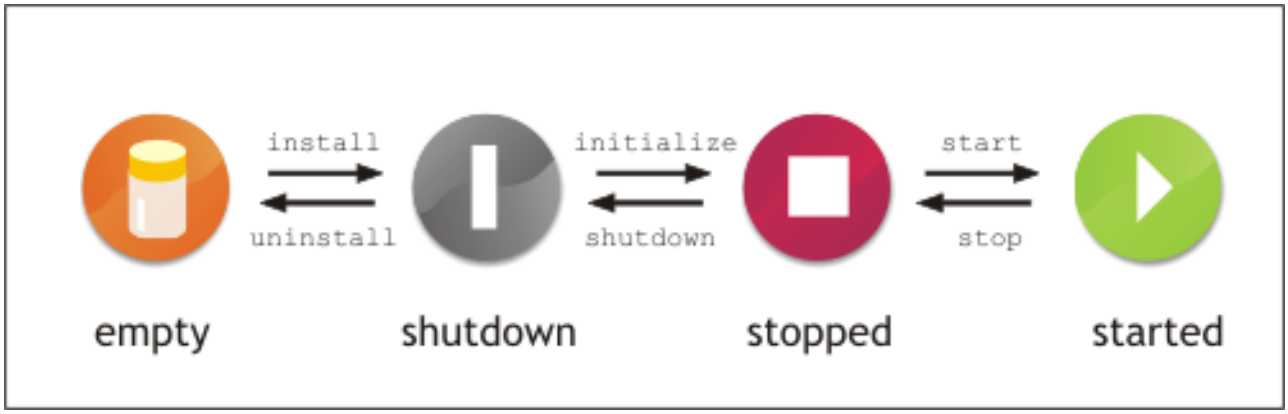
When a service assembly is installed into a JBI environment, all service units contained within the assembly are deployed to their respective JBI components. Service assemblies and service units are installed using the **`DeploymentService`** MBean that is exposed through the JMX console. In addition to the MBean, the following Ant tasks are provided for installing service assemblies and service units:

- **`DeployServiceAssemblyTask`**
- **`UndeployServiceAssemblyTask`**

MANAGING JBI COMPONENTS

Figure 4.1 shows the life-cycle of a JBI component.

Figure 4.1. JBI component life-cycle



Components begin life in an *empty* state. The component and the JBI environment have no knowledge of each other. Once the component is installed into the JBI environment, the component enters the *shutdown* state. In this state, the JBI environment initializes any resources required by the component. From the shutdown state a component can be initialized and moved into the *stopped* state. In the stopped state, a component is fully initialized and all of its resources are loaded into the JBI environment. When a component is ready to process messages, it is moved into the *started* state. In this state the component, and any service units deployed into the component, can participate in message exchanges.

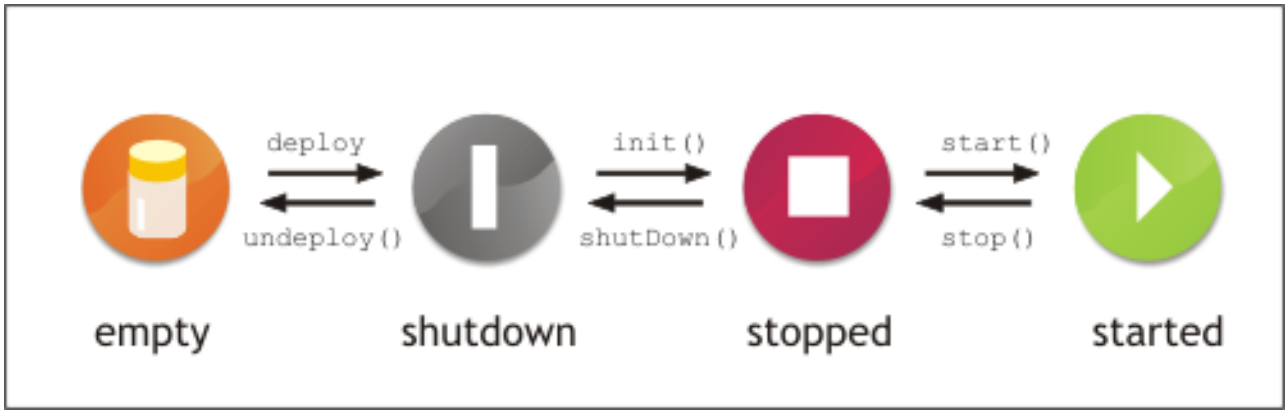
Components can be moved back and forth through the shutdown, stopped, and started states without being uninstalled. You can manage the lifecycle of an installed JBI component using the **`InstallationService`** MBean and the component's **`ComponentLifeCycle`** MBean. In addition, you can manage a component's lifecycle using the following Ant tasks:

- **`StartComponentTask`**
- **`StopComponentTask`**
- **`ShutDownComponentTask`**

MANAGING SERVICE UNITS

Figure 4.2 shows the life-cycle of a service unit.

Figure 4.2. Service unit life-cycle



Service units must first be deployed into the appropriate JBI component. The JBI component is the container that will provide the runtime resources necessary to implement the functionality defined by the service unit. When a service unit is in the *shutdown* state, the JBI component has not provisioned any resources for the service unit. When a service unit is moved into the *stopped* state, the JBI component has provisioned the resources for the service unit but the service unit cannot use any of the provisioned resources. When a service unit is in the *started* state, it is using the resources provisioned for it by the JBI container. In the started state, the functionality defined by the service unit is accessible.

A service can be moved through the different states while deployed. You manage the lifecycle of a service unit using the JBI environment's **`DeploymentService`** MBean. In addition, you can manage service units using the following Ant tasks:

- **`DeployServiceAssemblyTask`**
- **`UndeployServiceAssemblyTask`**
- **`StartServiceAssemblyTask`**
- **`StopServiceAssemblyTask`**
- **`ShutDownServiceAssemblyTask`**
- **`ListServiceAssembliesTask`**

CHAPTER 5. CLUSTERING JBI ENDPOINTS

IMPORTANT

The Java Business Integration components of Red Hat JBoss Fuse are considered deprecated. You should consider migrating any JBI applications to OSGi.

OVERVIEW

Red Hat JBoss Fuse provides a clustering engine that enables you to use Apache ActiveMQ, or any other JMS broker, to specify the endpoints to cluster in a JBI application. The Red Hat JBoss Fuse clustering engine works in conjunction with the normalized message router (NMR), and uses Apache ActiveMQ and specifically configured JBI endpoints to build clusters.

A cluster is defined as two or more JBI containers networked together. Implementing clustering between JBI containers gives you access to features including load balancing and high availability, rollback and redelivery, and remote container awareness.

FEATURES

Clustering provides the following features that can be implemented in your applications:

- Connect JBI containers to form a network, and dynamically add and remove the containers from the network.
- Enable rollback and redelivery when a JBI exchange fails.
- Implement load balancing among JBI containers capable of handling a given exchange. For example:
  - Install the same component in multiple JBI containers to provide increased capacity and high availability (if one container fails, the same component in another container can service the request).
  - Partition the workload among multiple JBI container instances to enable different containers to handle different tasks, spreading the workload across multiple containers.
- Remote component awareness means each clustered JBI container is aware of the components in its peer containers. Networked containers listen for remote component registration/deregistration events and can route requests to those components.

STEPS TO SET UP CLUSTERING

Complete the following steps to set up JBI endpoint clustering:

1. Install the jbi-cluster feature included in Red Hat JBoss Fuse. See the section called “Installing the clustering feature”.
2. Optionally, configure the clustering engine with a JMS broker other than the Red Hat JBoss A-MQ. See the section called “Changing the JMS broker”.
3. Optionally, change the default clustering engine configuration to specify different cluster and destination names. See the section called “Changing the default configuration”.

4. Add endpoints and register the endpoint definition in the Spring configuration. See the section called “Using clustering in an application”.

See the following sections for additional information:

- the section called “Establishing network connections between containers”
- the section called “High availability”
- the section called “Cluster configuration conventions”

INSTALLING THE CLUSTERING FEATURE

To install the jbi-cluster feature, use the install command from the command console:

1. Start Red Hat JBoss Fuse.
2. At the **JBossFuse:karaf@root>** prompt, type:

   ```
   features:install jbi-cluster
   ```

3. Type **featuresL:list** to list the existing features and their installation state. Verify that the jbi-cluster feature is installed.

The cluster configuration bundle is automatically installed when you install the jbi-cluster feature.

DEFAULT CLUSTERING ENGINE CONFIGURATION

Red Hat JBoss Fuse has a pre-installed clustering engine that is configured to use the included Red Hat JBoss A-MQ. The default configuration for the Red Hat JBoss Fuse cluster engine is defined in the **jbi-cluster.xml** file in the **org.apache.servicemix.jbi.cluster.config** bundle. This bundle is located in the installation directory in **\system\org\apache\servicemix\jbi\cluster**.

The default cluster engine configuration, shown in Example 5.1, is designed to meet most basic requirements.

Example 5.1. Default cluster engine configuration

```
<bean id="clusterEngine"
class="org.apache.servicemix.jbi.cluster.engine.ClusterEngine">
  <property name="pool">
    <bean
class="org.apache.servicemix.jbi.cluster.requestor.ActiveMQJmsRequestorP
ool">
     <property name="connectionFactory" ref="connectionFactory" />
     <property name="destinationName" value="${destinationName}" />
    </bean>
  </property>
  <property name="name" value="${clusterName}" />
</bean>
<osgi:list id="clusterRegistrations"

interface="org.apache.servicemix.jbi.cluster.engine.ClusterRegistration"
      cardinality="0..N">
  <osgi:listener ref="clusterEngine" bind-method="register" unbind-
```

```
method="unregister" />
</osgi:list>
<osgi:reference id="connectionFactory"
interface="javax.jms.ConnectionFactory" />
<osgi:service ref="clusterEngine">
  <osgi:interfaces>
    <value>org.apache.servicemix.nmr.api.Endpoint</value>
    <value>org.apache.servicemix.nmr.api.event.Listener</value>
    <value>org.apache.servicemix.nmr.api.event.EndpointListener</value>
    <value>org.apache.servicemix.nmr.api.event.ExchangeListener</value>
  </osgi:interfaces>
  <osgi:service-properties>
    <entry key="NAME" value="${clusterName}" />
  </osgi:service-properties>
</osgi:service>
<osgix:cm-properties id="clusterProps"
        persistent-id="org.apache.servicemix.jbi.cluster.config">
  <prop key="clusterName">${servicemix.name}</prop>
  <prop key="destinationName">org.apache.servicemix.jbi.cluster</prop>
</osgix:cm-properties>
<ctx:property-placeholder properties-ref="clusterProps" />
</beans>
```

Red Hat JBoss Fuse has a preconfigured Red Hat JBoss A-MQ instance that automatically starts when the container is started. This means you do not have to start a broker instance for the clustering engine to work.

CHANGING THE DEFAULT CONFIGURATION

You can alter the default configuration by adding a configuration file to the bundle **`org.apache.servicemix.jbi.cluster.config`**. This added configuration file enables you to change both the **`clusterName`** and the **`destinationName`**.

CHANGING THE JMS BROKER

You can configure the cluster engine with another JMS broker by adding a Spring XML file containing the full configuration to the ***`InstallDir`*`\deploy`** directory.

USING CLUSTERING IN AN APPLICATION

When using an OSGi packaged JBI service assembly, you can include the clustered endpoints definitions directly in the Spring configuration. In addition to the endpoint definition, you must add a bean that registers the endpoint with the clustering engine.

Example 5.2 shows an OSGi packaged HTTP consumer endpoint that is part of a cluster.

Example 5.2. OSGi packaged JBI endpoint

```
<http:consumer id="myHttpConsumer" service="test:myService"
endpoint="myEndpoint" />
<bean
class="org.apache.servicemix.jbi.cluster.engine.OsgiSimpleClusterRegistr
ation">
```

```
  <property name="endpoint" ref="myHttpConsumer" />
</bean>
```

When using a JBI packaged service assembly, you must create a Spring application to register the endpoint as a clustered endpoint. This configuration requires that you provide additional information about the endpoint.

Example 5.3 shows a JBI packaged HTTP consumer endpoint that is part of a cluster.

Example 5.3. JBI packaged endpoint

```
<http:consumer id="myHttpConsumer" service="test:myService"
endpoint="myEndpoint" />
<bean
class="org.apache.servicemix.jbi.cluster.engine.OsgiSimpleClusterRegistr
ation">
  <property name="serviceName" value="test:myService" />
  <property name="endpointName" value="myEndpoint" />
</bean>
```

ESTABLISHING NETWORK CONNECTIONS BETWEEN CONTAINERS

To create a network of JBI containers, you must establish network connections between each of the containers in the network, and then establish a network connection between the active containers. You can configure these network connections as either **static** or **multicast** connections.

- **Static network connections** — Configure each **`networkConnector`** in the cluster in the broker configuration file **`install_dir/conf/activemq.xml`**.

  Example 5.4 shows an example of a static **`networkConnector`** discovery configuration.

  Example 5.4. Static configuration

```
<!-- Set the brokerName to be unique for this container -->
<amq:broker id="broker" brokerName="host1_broker1" depends-
on="jmxServer">

              ....

  <networkConnectors>
            <networkConnector name="host1_to_host2"
uri="static://(tcp://host2:61616)"/>

    <!-- A three container network would look like this -->
    <!-- (Note it is not necessary to list the hostname in the uri
list) -->
    <!-- networkConnector name="host1_to_host2_host3"
          uri="static://(tcp://host2:61616,tcp://host3:61616)"/ -
->

  </networkConnectors>
```

```
</amq:broker>
```

- **Multicast network connections** — Enable multicast on your network and configure multicast in the broker configuration file **`installation_directory/conf/activemq.xml`** for each container in the network. When the containers start they detect each other and transparently connect to one another.

  Example 5.5 shows an example of a multicast **`networkConnector`** discovery configuration.

Example 5.5. Multicast configuration

```
<networkConnectors>
  <!-- by default just auto discover the other brokers -->
            <networkConnector name="default-nc"
uri="multicast://default"/>
    </networkConnectors>
```

When a network connection is established, each container discovers the other containers' remote components and can route to them.

HIGH AVAILABILITY

You can cluster JBI containers to implement high availability by configuring two distinct Red Hat JBoss Fuse container instances in a master-slave configuration. In all cases, the master is in **`ACTIVE`** mode and the slave is in **`STANDBY`** mode waiting for a failover event to trigger the slave to take over.

You can configure the master and the slave one of the following ways:

- **Shared file system master-slave** — In a shared database master-slave configuration, two containers use the same physical data store for the container state. You should ensure that the file system supports file level locking, as this is the mechanism used to elect the master. If the master process exits, the database lock is released and the slave acquires it. The slave then becomes the master.

- **JDBC master-slave** — In a JDBC master-slave configuration, the master locks a table in the backend database. The failover event in this case is that the lock is released from the database.

- **Pure master-slave** — A pure master-slave configuration can use either a shared database or a shared file system. The master replicates all state changes to the slave so additional overhead is incurred. The failover trigger in a pure master-slave configuration is that the slave loses its network connection to its master. Because of the additional overhead and maintenance involved, this option is less desirable than the other two options.

CLUSTER CONFIGURATION CONVENTIONS

The following conventions apply to configuring clustering:

- Don't use static and multicast **`networkConnectors`** at the same time. If you enable static **`networkConnectors`**, then you should disable any multicast **`networkConnectors`**, and vice versa.

- When configuring a network of containers in **`installation_directory/conf/activemq.xml`**, ensure that the **`brokerName`** attribute is unique for each node in the cluster. This will enable the instances in the network to uniquely identify each other.

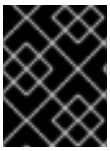
- When configuring a network of containers you must ensure that you have unique persistent stores for each **`ACTIVE`** instance. If you have a JDBC data source, you must use a separate database for each **`ACTIVE`** instance. For example:

```
<property name="url"
          value="jdbc:mysql://localhost/broker_activemq_host1?
relaxAutoCommit=true"/>
```

- You can setup a network of containers on the same host. To do this, you must change the JMS ports and **`transportConnector`** ports to avoid any port conflicts. Edit the **`installation_directory/conf/activemq.xml`** file, changing the **`rmi.port`** and **`activemq.port`** as appropriate. For example:

```
rmi.port = 1098
rmi.host       = localhost
jmx.url        =
service:jmx:rmi:///jndi/rmi://${rmi.host}:${rmi.port}/jmxrmi

activemq.port = 61616
activemq.host  = localhost
activemq.url   = tcp://${activemq.host}:${activemq.port}
```

CHAPTER 6. USING THE JBI ANT TASKS



IMPORTANT

The Java Business Integration components of Red Hat JBoss Fuse are considered deprecated. You should consider migrating any JBI applications to OSGi.

The JBI specification defines a number of Ant tasks that can be used to manage JBI components. These tasks allow you to install, start, stop, and uninstall components in the Red Hat JBoss Fuse container. You can use the JBI Ant tasks as either command line commands or as part of an Ant build file.

6.1. USING THE TASKS AS COMMANDS

Usage

This is the basic usage statement for the Red Hat JBoss Fuse Ant tasks when used from the command line:

ant -f *InstallDir*/ant/servicemix-ant-tasks.xml [-D*option*=*value* ...] *task*

The *task* argument is the name of the Ant task you are calling. Each task supports a number of options that are specified using the **-D*option*=*value*** flag.

Installing a component

The Ant task used to install a component to the Red Hat JBoss Fuse container is **install-component**. Its options are described in Table 6.1.

Table 6.1. Options for installing a JBI component with an Ant command

Option	Required	Description
sm.username	no	Specifies the username used to access the management features of the Red Hat JBoss Fuse container
sm.password	no	Specifies the password used to access the management features of the Red Hat JBoss Fuse container
sm.host	no	Specifies the host name where the container is running; the default value is **localhost**
sm.port	no	Specifies the port where the container's RMI registry is listening; the default value is **1099**

Option	Required	Description
sm.install.file	yes	Specifies the name of the installer file for the component

Example 6.1 shows an example of using **install-component** to install the Camel component to a container listening on port 1000.

Example 6.1. Installing a component using an Ant command

```
>ant -f ant/servicemix-ant-task.xml -Dsm.port=1000 -
Dsm.install.file=servicemix-camel-3.3.0.6-fuse-installer.zip install-
component
Buildfile: ant\servicemix-ant-task.xml install-component: [echo]
install-component [echo] Installing a service engine or binding
component. [echo] host=localhost [echo] port=1000 [echo]
file=hotdeploy\servicemix-camel-3.3.0.6-fuse-installer.zip BUILD
SUCCESSFUL Total time: 7 seconds
```

Removing a component

The Ant task used to remove a component from the Red Hat JBoss Fuse container is **uninstall-component**. Its options are described in Table 6.2.

Table 6.2. Options for removing a JBI component with an Ant command

Option	Required	Description
sm.username	no	Specifies the username used to access the management features of the Red Hat JBoss Fuse container
sm.password	no	Specifies the password used to access the management features of the Red Hat JBoss Fuse container
sm.host	no	Specifies the host name where the container is running; the default value is **localhost**
sm.port	no	Specifies the port where the container's RMI registry is listening; the default value is **1099**
sm.component.name	yes	Specifies the name of the JBI component

Example 6.2 shows an example of using **`uninstall-component`** to remove the drools component from a container listening on port 1000.

Example 6.2. Removing a component using an Ant command

```
>ant -f ant\servicemix-ant-task.xml -Dsm.port=1000 -
Dsm.component.name=servicemix-drools uninstall-component
Buildfile: ant\servicemix-ant-task.xml uninstall-component: [echo]
uninstall-component [echo] Uninstalling a Service Engine or Binding
Component. [echo] host=localhost [echo] port=1000 [echo]
name=servicemix-drools BUILD SUCCESSFUL Total time: 1 second
```

Starting a component

The Ant task used to start a component is **`start-component`**. Its options are described in Table 6.3.

Table 6.3. Options for starting a JBI component with an Ant command

Option	Required	Description
`sm.username`		Specifies the username used to access the management features of the Red Hat JBoss Fuse container
`sm.password`	no	Specifies the password used to access the management features of the Red Hat JBoss Fuse container.
`sm.host`	no	Specifies the host name where the container is running; the default value is **`localhost`**
`sm.port`	no	Specifies the port where the container's RMI registry is listening; the default value is **`1099`**
`sm.component.name`	yes	Specifies the name of the JBI component

Example 6.3 shows an example of using **`start-component`** to start the cxf-se component in a container listening on port 1000.

Example 6.3. Starting a component using an Ant command

```
>ant -f ant\servicemix-ant-task.xml -Dsm.port=1000 -
Dsm.component.name=servicemix-cxf-se start-component
Buildfile: ant\servicemix-ant-task.xml start-component: [echo] start-
```

```
component [echo] starts a particular component (service engine or
binding component) in Servicemix [echo] host=localhost [echo] port=1000
[echo] name=servicemix-cxf-se BUILD SUCCESSFUL Total time: 1 second
```

Stopping a component

The Ant task used to stop a component is **`stop-component`**. Its options are described in Table 6.4.

Table 6.4. Options for stopping a JBI component with an Ant command

Option	Required	Description
`sm.username`	no	Specifies the username used to access the management features of the Red Hat JBoss Fuse container
`sm.password`	no	Specifies the password used to access the management features of the Red Hat JBoss Fuse container
`sm.host`	no	Specifies the host name where the container is running; the default value is **`localhost`**
`sm.port`	no	Specifies the port where the container's RMI registry is listening; the default value is **`1099`**
`sm.component.name`	yes	Specifies the name of the JBI component

Example 6.4 shows an example of using **`stop-component`** to stop the cxf-se component in a container listening on port 1000.

Example 6.4. Stopping a component using an Ant command

```
>ant -f ant\servicemix-ant-task.xml -Dsm.port=1000 -
Dsm.component.name=servicemix-cxf-se stop-component
                Buildfile: ant\servicemix-ant-task.xml stop-component:
[echo] stop-component [echo] stops a particular component (service
engine or binding component) in Servicemix [echo] host=localhost [echo]
port=1000 [echo] name=servicemix-cxf-se BUILD SUCCESSFUL Total time: 1
second
```

Shutting down a component

The Ant task used to shutdown a component is **`shutdown-component`**. Its options are described in Table 6.5.

Table 6.5. Options for shutting down a JBI component with an Ant command

Option	Required	Description
`sm.username`	no	Specifies the username used to access the management features of the Red Hat JBoss Fuse container
`sm.password`	no	Specifies the password used to access the management features of the Red Hat JBoss Fuse container
`sm.host`	no	Specifies the host name where the container is running; the default value is **`localhost`**
`sm.port`	no	Specifies the port where the container's RMI registry is listening; the default value is **`1099`**
`sm.component.name`	yes	Specifies the name of the JBI component

Installing a shared library

The Ant task used to install a shared library to the Red Hat JBoss Fuse container is **`install-shared-library`**. Its options are described in Table 6.6.

Table 6.6. Options for installing a shared library with an Ant command

Option	Required	Description
`sm.username`	no	Specifies the username used to access the management features of the Red Hat JBoss Fuse container
`sm.password`	no	Specifies the password used to access the management features of the Red Hat JBoss Fuse container
`sm.host`	no	Specifies the host name where the container is running; the default value is **`localhost`**

Option	Required	Description
`sm.port`	no	Specifies the port where the container's RMI registry is listening; the default value is `1099`
`sm.install.file`	yes	Specifies the name of the library's installer file

Removing a shared library

The Ant task used to remove a shared library from the Red Hat JBoss Fuse container is `uninstall-shared-library`. Its options are described in Table 6.7.

Table 6.7. Options for removing a shared library with an Ant command

Option	Required	Description
`sm.username`	no	Specifies the username used to access the management features of the Red Hat JBoss Fuse container
`sm.password`	no	Specifies the password used to access the management features of the Red Hat JBoss Fuse container
`sm.host`	no	Specifies the host name where the container is running; the default value is `localhost`
`sm.port`	no	Specifies the port where the container's RMI registry is listening; the default value is `1099`
`sm.shared.library.name`	yes	Specifies the name of the shared library

6.2. USING THE TASKS IN BUILD FILES

Adding the JBI tasks to build an Ant file

Before you can use the JBI tasks in an Ant build file, you must add the tasks using a `taskdef` element, as shown in Example 6.5.

Example 6.5. Adding the JBI tasks to an Ant build file

```
...
1 <property name="fuseesb.install_dir" value="/home/fuse_esb"/>

<taskdef
2 file="${fuseesb.install_dir}/ant/servicemix_ant_taskdef.properties">
  3 <classpath id="fuseesb.classpath">
      <fileset dir="${fuseesb.install_dir}">
      <include name="*.jar"/>
    </fileset>
    <fileset dir="${fuseesb.install_dir}/lib">
      <include name="*.jar"/>
    </fileset>
  </classpath>
</taskdef>
...
```

The build file fragment in Example 6.5 does the following:

1. Sets a property, fuseesb.install_dir, the installation directory for Red Hat JBoss Fuse
2. Loads the tasks using the **`ant/servicemix_ant_taskdef.properties`**
3. Sets the classpath to make all of the required jars from the Red Hat JBoss Fuse installation available

Installing a component

The Ant task used to install a JBI component is **`jbi-install-component`**. Its attributes are listed in Table 6.8.

Table 6.8. Attributes for installing a JBI component using an Ant task

Attribute	Required	Description
`host`	no	Specifies the host name where the container is running; the default value is **`localhost`**
`port`	no	Specifies the port where the container's RMI registry is listening; the default value is **`1099`**
`username`	no	Specifies the username used to access the management features of the container
`password`	no	Specifies the password used to access the management features of the container

Attribute	Required	Description
failOnError	no	Specifies if an error will cause the entire build to fail
file	yes	Specifies the name of the installer file for the component

Example 6.6 shows an Ant target that installs the drools component.

Example 6.6. Ant target that installs a JBI component

```
...
<target name="installDrools" description="Installs the drools engine.">
  <jbi-install-component port="1099"
                         file="servicemix-drools-3.3.0.6-fuse-
installer.zip" />
</target>
...
```

Removing a component

The Ant task used to remove a JBI component is **jbi-uninstall-component**. Its attributes are listed in Table 6.9.

Table 6.9. Attributes for removing a JBI component using an Ant task

Attribute	Required	Description
host	no	Specifies the host name where the container is running; the default value is **localhost**
port	no	Specifies the port where the container's RMI registry is listening; the default value is **1099**
username	no	Specifies the username used to access the management features of the container
password	no	Specifies the password used to access the management features of the container
failOnError	no	Specifies if an error will cause the entire build to fail

Attribute	Required	Description
name	yes	Specifies the component's name

Example 6.7 shows an Ant target that removes the drools component.

Example 6.7. Ant target that removes a JBI component

```
...
<target name="removeDrools" description="Removes the drools engine.">
  <jbi-uninstall-component port="1099"
                           name="servicemix-drools" />
</target>
...
```

Starting a component

The Ant task used to start a JBI component is **jbi-start-component**. Its attributes are listed in Table 6.10.

Table 6.10. Attributes for starting a JBI component using an Ant task

Attribute	Required	Description
host	no	Specifies the host name where the container is running; the default value is **localhost**
port	no	Specifies the port where the container's RMI registry is listening; the default value is **1099**.
username	no	Specifies the username used to access the management features of the container
password	no	Specifies the password used to access the management features of the container
failOnError	no	Specifies if an error will cause the entire build to fail
name	yes	Specifies the component's name

Example 6.8 shows an Ant target that starts the drools component.

Example 6.8. Ant target that starts a JBI component

```
...
<target name="startDrools" description="Starts the drools engine.">
  <jbi-start-component port="1099" name="servicemix-drools" />
</target>
...
```

Stopping a component

The Ant task used to stop a JBI component is **`jbi-start-component`**. Its attributes are listed in Table 6.11.

Table 6.11. Attributes for stopping a JBI component using an Ant task

Attribute	Required	Description
`host`	no	Specifies the host name where the container is running; the default value is **`localhost`**
`port`	no	Specifies the port where the container's RMI registry is listening; the default value is **`1099`**
`username`	no	Specifies the username used to access the management features of the container
`password`	no	Specifies the password used to access the management features of the container
`failOnError`	no	Specifies if an error will cause the entire build to fail
`name`	yes	Specifies the component's name

Example 6.9 shows an Ant target that stops the drools component.

Example 6.9. Ant target that stops a JBI component

```
...
<target name="stopDrools" description="Stops the drools engine.">
  <jbi-stop-component port="1099" name="servicemix-drools" />
</target>
...
```

Shutting down a component

The Ant task used to shut down a JBI component is **`jbi-shut-down-component`**. Its attributes are listed in Table 6.12.

Table 6.12. Attributes for shutting down a JBI component using an Ant task

Attribute	Required	Description
`host`	no	Specifies the host name where the container is running; the default value is **`localhost`**
`port`	no	Specifies the port where the container's RMI registry is listening; the default value is **`1099`**
`username`	no	Specifies the username used to access the management features of the container
`password`	no	Specifies the password used to access the management features of the container
`failOnError`	no	Specifies if an error will cause the entire build to fail
`name`	yes	Specifies the component's name

Example 6.10 shows an Ant target that shuts down the drools component.

Example 6.10. Ant target that shuts down a JBI component

```
...
<target name="shutdownDrools" description="Stops the drools engine.">
  <jbi-shut-down-component port="1099" name="servicemix-drools" />
</target>
...
```

Installing a shared library

The Ant task used to install a shared library is **`jbi-install-shared-library`**. Its attributes are listed in Table 6.13.

Table 6.13. Attributes for installing a shared library using an Ant task

Attribute	Required	Description
host	no	Specifies the host name where the container is running; the default value is **localhost**
port	no	Specifies the port where the container's RMI registry is listening; the default value is **1099**
username	no	Specifies the username used to access the management features of the container
password	no	Specifies the password used to access the management features of the container
failOnError	no	Specifies if an error will cause the entire build to fail
file	yes	Specifies the name of the installer file for the library

Removing a shared library

The Ant task used to remove a shared library is **jbi-uninstall-shared-library**. Its attributes are listed in Table 6.14.

Table 6.14. Attributes for removing a shared library using an Ant task

Attribute	Required	Description
host	no	Specifies the host name where the container is running; the default value is **localhost**
port	no	Specifies the port where the container's RMI registry is listening; the default value is **1099**
username	no	Specifies the username used to access the management features of the container
password	no	Specifies the password used to access the management features of the container

Attribute	Required	Description
`failOnError`	no	Specifies if an error will cause the entire build to fail
`name`	yes	Specifies the name of the library

CHAPTER 7. BUILDING JBI COMPONENTS USING MAVEN

IMPORTANT

The Java Business Integration components of Red Hat JBoss Fuse are considered deprecated. You should consider migrating any JBI applications to OSGi.

OVERVIEW

Red Hat JBoss Fuse provides Maven tooling that simplifies the creation and deployment of JBI artifacts. Among the tools provided are:

- Plug-ins for packaging JBI components
- A plug-in for packaging shared libraries
- Archetypes that create starting point projects for JBI artifacts

The Red Hat JBoss Fuse Maven tools also include plug-ins for creating service units and service assemblies. However, those plug-ins are not described in this book.

SETTING UP THE MAVEN TOOLS

In order to use the Red Hat JBoss Fuse Maven tools, you add the elements shown in Example 7.1 to your POM file.

Example 7.1. POM elements for using Red Hat JBoss Fuse Maven tools

```
...
<pluginRepositories>
  <pluginRepository>
    <id>fusesource.m2</id>
    <name>JBoss Fuse 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>JBoss Fuse 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>JBoss Fuse 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 Red Hat JBoss Fuse Maven tools and to load the plug-in that implements the tools.

CREATING A JBI MAVEN PROJECT

The Red Hat JBoss Fuse Maven tools provide a number of archetypes that can be used to seed a JBI project. The archetype generates the proper file structure for the project along with a POM file that contains the metadata required for the specified project type.

Example 7.2 shows the command for using the JBI archetypes.

Example 7.2. Command for JBI maven archetypes

mvn archetype:create -DarchetypeGroupId=org.apache.servicemix.tooling -DarchetypeArtifactId=*servicemix-archetype-name* -DarchetypeVersion=fuse-4.0.0.0 [-DgroupId=*org.apache.servicemix.samples.embedded*] [-DartifactId=*servicemix-embedded-example*]

The value passed to the **-DarchetypeArtifactId** argument specifies the type of project you are creating.

JBI COMPONENTS

As shown in Example 7.3, you specify a value of **jbi-component** for the project's **packaging** element, which informs the Red Hat JBoss Fuse Maven tooling that the project is for a JBI component.

Example 7.3. Specifying that a maven project results in a JBI component

```
<project ...>
  ...
  <groupId>org.apache.servicemix</groupId>
  <artifactId>MyBindingComponent</artifactId>
  <packaging>jbi-component</packaging>
  ...
</project>
```

The **plugin** element responsible for packaging the JBI component is shown in Example 7.4. The **groupId** element, the **artifactId** element, the **version** element, and the **extensions** element are common to all instances of the Red Hat JBoss Fuse Maven plug-in. If you use the Maven archetypes to generate the project, you should not have to change them.

Example 7.4. Plug-in specification for packaging a JBI component

```
...
<plugin>
  <groupId>org.apache.servicemix.tooling</groupId>
  <artifactId>jbi-maven-plugin</artifactId>
  <version>${servicemix-version}</version>
  <extensions>true</extensions>
  <configuration>
    <type>service-engine</type>
    <bootstrap>org.apache.servicemix.samples.MyBootstrap</bootstrap>
    <component>org.apache.servicemix.samples.MyComponent</component>
  </configuration>
</plugin>
...
```

The **configuration** element, along with its children, provides the Red Hat JBoss Fuse tooling with the metadata necessary to construct the **jbi.xml** file required by the component.

type

Specifies the type of JBI component the project is building. Valid values are:

- **service-engine** for creating a service engine
- **binding-component** for creating a binding component

bootstrap

Specifies the name of the class that implements the JBI **Bootstrap** interface for the component.

TIP

You can omit this element if you intend to use the default **Bootstrap** implementation provided with Red Hat JBoss Fuse.

component

Specifies the name of the class that implements the JBI **Component** interface for that component.

Once the project is properly configured, you can build the JBI component by using the **mvn install** command. The Red Hat JBoss Fuse Maven tooling will generate a standard jar containing both the component and an installable JBI package for the component.

SHARED LIBRARIES

As shown in Example 7.5, to instruct the Red Hat JBoss Fuse Maven tooling that the project is for a shared library you specify a value of **jbi-shared-library** for the project's **packaging** element.

Example 7.5. Specifying that a maven project results in a JBI shared library

```
<project ...>
  ...
  <groupId>org.apache.servicemix</groupId>
  <artifactId>MyBindingComponent</artifactId>
  <packaging>jbi-shared-library</packaging>
  ...
</project>
```

You build the shared library using the **mvn install** command. The Red Hat JBoss Fuse Maven tooling generates a standard jar containing the shared library and an installable JBI package for the shared library.

CHAPTER 8. DEPLOYING JBI ENDPOINTS USING MAVEN

Abstract

Red Hat JBoss Fuse provides a Maven plug-in and a number of Maven archetypes that make developing, packaging, and deploying applications easier.

IMPORTANT

The Java Business Integration components of Red Hat JBoss Fuse are considered deprecated. You should consider migrating any JBI applications to OSGi.

The tooling provides you with a number of benefits, including:

- Automatic generation of JBI descriptors
- Dependency checking
- Service assembly deployment

Because Red Hat JBoss Fuse only allows you to deploy service assemblies, you must do the following when using Maven tooling:

1. Set up a top-level project to build all of the service units and the final service assembly (see Section 8.1, “Setting up a Red Hat JBoss Fuse JBI project”).
2. Create a project for each of your service units (see Section 8.2, “A service unit project”).
3. Create a project for the service assembly (see Section 8.3, “A service assembly project”).

8.1. SETTING UP A RED HAT JBOSS FUSE JBI PROJECT

Overview

When working with the Red Hat JBoss Fuse 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 JBoss Fuse JBI Maven tooling, add the elements shown in Example 8.1 to your top-level POM file.

Example 8.1. POM elements for using Red Hat JBoss Fuse Maven tooling

```
...
<pluginRepositories>
  <pluginRepository>
    <id>fusesource.m2</id>
    <name>FuseSource Open Source Community Release Repository</name>
    <url>http://repo.fusesource.com/nexus/content/groups/public/</url>
    <snapshots>
      <enabled>false</enabled>
    </snapshots>
    <releases>
      <enabled>true</enabled>
    </releases>
  </pluginRepository>
</pluginRepositories>
<repositories>
  <repository>
    <id>fusesource.m2</id>
    <name>FuseSource Open Source Community Release Repository</name>
    <url>http://repo.fusesource.com/nexus/content/groups/public/</url>
    <snapshots>
       <enabled>false</enabled>
    </snapshots>
    <releases>
      <enabled>true</enabled>
    </releases>
  </repository>
  <repository>
    <id>fusesource.m2-snapshot</id>
    <name>FuseSource Open Source Community Snapshot Repository</name>
    <url>http://repo.fusesource.com/nexus/content/groups/public-
snapshots/</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 JBoss Fuse 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 8.2 shows a top-level POM for a project that contains a single service unit.

Example 8.2. Top-level POM for a Red Hat JBoss Fuse 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>

(1) <pluginRepositories>
      <pluginRepository>
      <id>fusesource.m2</id>
      <name>FuseSource Open Source Community Release Repository</name>
```

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

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

  <build>
    <plugins>
3       <plugin>
          <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>
4           <plugin>
              <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 8.2, “Top-level POM for a Red Hat JBoss Fuse JBI project” does the following:

1. Configures Maven to use the FuseSource repositories for loading the JBoss Fuse plug-ins.
2. 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
3. Configures the Maven assembly plug-in.
4. Loads the JBoss Fuse JBI plug-in.

8.2. 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

Red Hat JBoss Fuse 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 8.3, 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 8.3. 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 8.1 lists the possible values and describes what type of project is seeded.

Table 8.1. 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
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 JBI Development Guide)
jms-provider	Creates a provider endpoint project targeted at the JMS binding component (see JBI Development Guide)
file-poller	Creates a polling (consumer) endpoint project targeted at the file binding component (see Chapter 10, *Using Poller Endpoints*)
file-sender	Creates a sender (provider) endpoint project targeted at the file binding component (see Chapter 11, *Using Sender Endpoints*)
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

Name	Description
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]
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 8.4.

Example 8.4. 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 8.5 shows the configuration for a service unit targeting the Apache CXF binding component.

Example 8.5. 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>
...
```

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, Red Hat JBoss Fuse 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 8.6.

Example 8.6. 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 8.7 shows the POM file for a project that is building a service unit targeted to the Apache CXF binding component.

Example 8.7. 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>

1     <parent>
          <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>
2  <packaging>jbi-service-unit</packaging>

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

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

The POM file in Example 8.7, “POM file for a service unit project” does the following:

1. Specifies that it is a part of the top-level project shown in Example 8.2, “Top-level POM for a Red Hat JBoss Fuse JBI project”
2. Specifies that this project builds a service unit
3. Specifies that the service unit targets the Apache CXF binding component
4. Specifies to use the Red Hat JBoss Fuse Maven plug-in

8.3. A SERVICE ASSEMBLY PROJECT

Overview

Red Hat JBoss Fuse requires that all service units are bundled into a service assembly before they can be deployed to a container. The JBoss Fuse 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

Red Hat JBoss Fuse provides a Maven artifact for seeding a service assembly project. You can seed a project with the **`smx-arch`** command. As shown in Example 8.8, 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 8.8. 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 8.9.

Example 8.9. 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 8.10 shows the configuration for a service assembly that bundles two service units.

Example 8.10. 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 8.11 shows a POM file for a project that is building a service assembly.

Example 8.11. 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>

(1)    <parent>
           <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>
(2)  <packaging>jbi-service-assembly</packaging>

(3)  <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>

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

The POM in Example 8.11, “POM for a service assembly project” does the following:

(1) Specifies that it is a part of the top-level project shown in Example 8.2, “Top-level POM for a Red Hat JBoss Fuse JBI project”

(2) Specifies that this project builds a service assembly

(3) Specifies the service units being bundled by the service assembly

(4) Specifies to use the JBoss Fuse Maven plug-in

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

APPENDIX A. USING THE JBI CONSOLE COMMANDS

ACCESSING THE JBI COMMANDS

The **jbi** commands allow you to manage JBI artifacts that are deployed in the Red Hat JBoss Fuse runtime.

Type **jbi:** then press **Tab** at the **JBossFuse:karaf@root>** prompt to view the available commands.

COMMANDS

Table A.1 describes the **jbi** commands available . For detailed information about the console commands in Red Hat JBoss Fuse, see the "Console Reference".

Table A.1. JBI Commands

Command	Description
jbi:list	Lists all of the JBI artifacts deployed into the Red Hat JBoss Fuse container. The list is separated into JBI components and JBI service assemblies. It displays the name of the artifact and its life-cycle state.
jbi:shutdown *artifact*	Moves the specified artifact from the stopped state to the shutdown state.
jbi:stop *artifact*	Moves the specified artifact into the stopped state.
jbi:start *artifact*	Moves the specified artifact into the started state.

PART II. FILE BINDING COMPONENT

Abstract

This guide provides an overview of the JBI file binding component; describes configuring and using poller and sender endpoints and file marshallers; describes the properties of poller and sender endpoints; and describes how to use the Maven tooling.

CHAPTER 9. INTRODUCTION TO THE FILE BINDING COMPONENT

Abstract

The file binding component allows you to create endpoints that read files from a file system and write files out to the file system.

IMPORTANT

The Java Business Integration components of Red Hat JBoss Fuse are considered deprecated. You should consider migrating any JBI applications to OSGi.

OVERVIEW

The file component provides integration to the file system. It can be used to read and write files via URI. It can also be configured to periodically poll directories for new files.

It allows for the creation of two types of endpoint:

poller endpoint

A poller endpoint polls a specified location on the file system for files. When it finds a file it reads the file and sends it to the NMR for delivery to the appropriate endpoint.

IMPORTANT

A poller endpoint can only create in-only message exchanges.

sender endpoint

A sender endpoint receives messages from the NMR. It then writes the contents of the message to a specified location on the file system.

KEY FEATURES

The file component has the following advanced features:

- custom filters for selecting files
- custom marshalers for converting the contents of a file to and from a normalized message
- custom locking mechanism for controlling file access during reads
- archiving of read files

CONTENTS OF A FILE COMPONENT SERVICE UNIT

A service unit that configures the file 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 9.1, "JBI descriptor for a file component service unit" shows a JBI descriptor for a file component service unit.

Example 9.1. JBI descriptor for a file component service unit

```
<jbi xmlns="http://java.sun.com/xml/ns/jbi" version="1.0">
  <services binding-component="false" />
</jbi>
```

TIP

The developer typically does not need to hand code this file. It is generated by the Red Hat JBoss Fuse Maven tooling.

OSGI PACKAGING

You can package file endpoints in an OSGi bundle. To do so, you need to make two minor changes:

- you will need to include an OSGi bundle manifest in the **META-INF** folder of the bundle.
- You need to add the following to your service unit's configuration file:

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

IMPORTANT

When you deploy file 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 H, *Using the Maven OSGi Tooling*.

NAMESPACE

The elements used to configure file endpoints are defined in the **http://servicemix.apache.org/file/1.0** namespace. You will need to add a namespace declaration similar to the one in Example 9.2, "Namespace declaration for using file endpoints" to your **xbean.xml** file's **beans** element.

Example 9.2. Namespace declaration for using file endpoints

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

In addition, you need to add the schema location to the Spring **beans** element's **xsi:schemaLocation** as shown in Example 9.3, “Schema location for using file endpoints”.

Example 9.3. Schema location for using file endpoints

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

CHAPTER 10. USING POLLER ENDPOINTS

Abstract

Poller endpoints poll the file system for files and passes the file to a target endpoint inside an in-only message exchange.

IMPORTANT

The Java Business Integration components of Red Hat JBoss Fuse are considered deprecated. You should consider migrating any JBI applications to OSGi.

10.1. INTRODUCTION TO POLLER ENDPOINTS

Overview

The function of a poller endpoint is to read data, in the form of files, from a location on a file system and pass that information to other endpoints in the ESB. Poller endpoints create an in-only message exchange containing the data read in from a file.

A poller endpoint, as its name implies, works by continually polling the file system to see if a file is present for consumption. The polling interval is completely customizable.

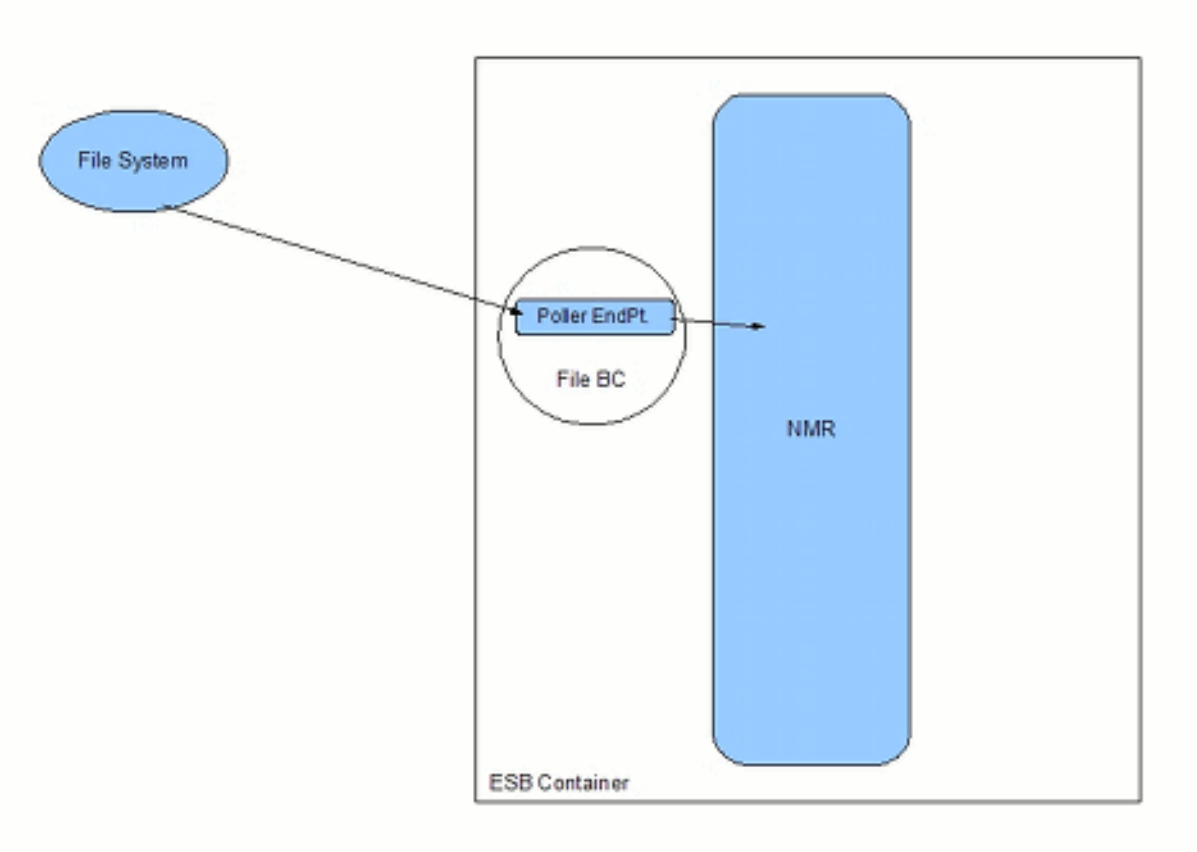
You can also control the files a poller endpoint consumes. Using the basic configuration attributes, you can configure the endpoint to poll for a specific file or you can poll it to monitor a specific directory on the file system. In addition, you can configure the endpoint to use a custom file filter.

By default, poller endpoints will only consume valid XML files. You can customize this behavior by configuring the endpoint to use a custom marshaler.

Where does a poller endpoint fit into a solution?

Poller endpoints play the role of consumer from the vantage point of the other endpoints in the ESB. As shown in Figure 10.1, “Poller endpoint”, a poller endpoint watches the file system for files to consume. When the endpoint consumes a file, it transfers its contents into a message and starts off an in-only message exchange. Poller endpoints cannot receive messages from the NMR.

Figure 10.1. Poller endpoint



Configuration element

Poller endpoints are configured using the **`poller`** element. All its configuration can be specified using attributes of this element.

The more complex features, such as custom marshalers, require the addition of other elements. These can either be separate **`bean`** elements or child elements of the **`poller`** element.

10.2. BASIC CONFIGURATION

Overview

The basic requirements for configuring a poller endpoint are straightforward. You need to supply the following information:

- the endpoint's name
- the endpoint's service name
- the file or directory to be monitored
- the endpoint to which the resulting messages will be sent

All of this information is provided using attributes of the **`poller`** element.

Identifying the endpoint

All endpoints need to have a unique identity. An endpoint's identity is made up of two pieces of information:

- a service name
- an endpoint name

Table 10.1, “Attributes for identifying a poller endpoint” describes the attributes used to identify a poller endpoint.

Table 10.1. Attributes for identifying a poller endpoint

Name	Description
`service`	Specifies the service name of the endpoint. This value must be a valid QName and does not need to be unique.
`endpoint`	Specifies the name of the endpoint. This value is a simple string. It must be unique among all of the endpoints associated with a given service name.

Specifying the message source

You specify the location in which the poller endpoint looks for new messages using the **`poller`** element's **`file`** attribute. This attribute takes a URI that identifies a location on the file system.

If you want the endpoint to poll a specific file, you use the standard file:*location* URI. If you do not use the **`file`** prefix, the endpoint will assume the URI specifies a directory on the file system and will consume all valid XML files placed in the specified directory.

For example, the URI file:inbox tells the endpoint to poll for a file called **`inbox`**. The URI inbox instructs the endpoint to poll the directory **`inbox`**.

IMPORTANT

Relative URIs are resolved from the directory in which the Red Hat JBoss Fuse container was started.

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:

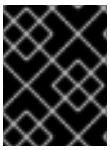
1. If you explicitly specify an endpoint using both the **`targetService`** attribute and the **`targetEndpoint`** attribute, the ESB will use that endpoint.

   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 NMR 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 the name of an interface that can accept the message using the **`targetInterface`** attribute. The NMR 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 poller endpoint will use the most specific information.

Example

Example 10.1, “Simple poller endpoint” shows the configuration for a simple poller endpoint.

Example 10.1. Simple poller endpoint

```
<beans xmlns:file="http://servicemix.apache.org/file/1.0"
       xmlns:foo="http://servicemix.org/demo/">

  <file:poller service="foo:filePoller"
               endpoint="filePoller"
               targetService="foo:fileSender"
               file="file:inbox/test.xml" />
  ...
</beans>
```

10.3. CONFIGURING POLLER ENDPOINTS INTERACTIONS WITH THE FILE SYSTEM

Overview

Poller endpoints interact with the file system in basic ways. You can configure a number of the aspects of this behavior including:

- if the endpoint creates the directory it is configured to poll
- if the endpoint polls the subdirectories of the configured directory
- if the endpoint deletes the files it consumes
- where the endpoint archives copies of the consumed files

Directory handling

The default behavior of a poller endpoint that is configured to poll a directory on the file system is to create the directory if it does not exist and to poll all of that directory's subdirectories. You can configure an endpoint to do only one, both, or none of these behaviors.

To configure an endpoint to not create the configured directory, you set its **autoCreateDirectory** attribute to **false**. If the directory does not exist, the endpoint will do nothing. You will then have to create the directory manually.

To configure the endpoint to only poll the configured directory and ignore its subdirectories, you set the endpoint's **recursive** attribute to **false**.

Example 10.2, “Poller endpoint that does not check subdirectories” shows the configuration for a poller endpoint that does not recurse into the subdirectories of the directory it polls.

Example 10.2. Poller endpoint that does not check subdirectories

```
<beans xmlns:file="http://servicemix.apache.org/file/1.0"
       xmlns:foo="http://servicemix.org/demo/">

  <file:poller service="foo:filePoller"
               endpoint="filePoller"
               targetService="foo:fileSender"
               file="inbox"
               recursive="false" />
  ...
</beans>
```

File retention

By default, poller endpoints delete a file once it is consumed. To configure the endpoint to leave the file in place after is consumed, set its **deleteFile** attribute to **false**.

Example 10.3, “Poller endpoint that leaves files behind” shows the configuration for a poller endpoint that does not delete files.

Example 10.3. Poller endpoint that leaves files behind

```
<beans xmlns:file="http://servicemix.apache.org/file/1.0"
       xmlns:foo="http://servicemix.org/demo/">

  <file:poller service="foo:filePoller"
               endpoint="filePoller"
               targetService="foo:fileSender"
               file="inbox"
               deleteFile="false" />
  ...
</beans>
```

IMPORTANT

When the poller endpoint does not automatically delete consumed files, the list of consumed files is stored in memory. If the Red Hat JBoss Fuse container is stopped and restarted, files that have been consumed, but not removed from the polling folder, will be reprocessed. One possible solution is to use a custom lock manager that stores a list of the consumed files to an external data store.

Archiving files

By default, poller endpoints do not archive files after they are consumed. If you want the files consumed by a poller endpoint to be archived you set the endpoint's **archive** attribute. The value of the **archive** attribute is a URI pointing to the directory into which the consumed files will archived.

IMPORTANT

Relative URIs are resolved from the directory in which the Red Hat JBoss Fuse container was started.

Example 10.4, “Poller endpoint that archives files” shows the configuration for a poller endpoint that files into a directory called **archives**.

Example 10.4. Poller endpoint that archives files

```
<beans xmlns:file="http://servicemix.apache.org/file/1.0"
       xmlns:foo="http://servicemix.org/demo/">

  <file:poller service="foo:filePoller"
               endpoint="filePoller"
               targetService="foo:fileSender"
               file="inbox"
               archive="archives" />
  ...
</beans>
```

10.4. CONFIGURING THE POLLING INTERVAL

Overview

A default poller endpoint provides limited scheduling facilities. You can configure when the endpoint starts polling and the interval between polling attempts.

Scheduling the first poll

By default, poller endpoints begin polling as soon as they are started. You can control when a poller endpoint first attempts to poll the file system using an attribute that controls the date of the first polling attempt.

You specify a date for the first poll using the endpoint's **firstTime** attribute. The **firstTime** attribute specifies a date using the standard xsd:date format of **YYYY-MM-DD**. For example, you would specify April 1, 2025 as **2025-04-01**. The first polling attempt will be made at 00:00:00 GMT on the specified date.

NOTE

If you schedule the first polling attempt in the past, the endpoint will begin polling immediately.

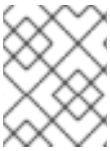
Example 10.5, “Poller endpoint with a scheduled start time” shows the configuration for a poller endpoint that starts polling at 1am GMT on April 1, 2010.

Example 10.5. Poller endpoint with a scheduled start time

```
<beans xmlns:file="http://servicemix.apache.org/file/1.0"
       xmlns:foo="http://servicemix.org/demo/">

  <file:poller service="foo:filePoller"
               endpoint="filePoller"
               targetService="foo:fileSender"
               file="inbox"
               firstTime="2010-04-01" />
  ...
</beans>
```

Delaying the first poll

In addition to controlling the specific date on which polling will start, you can also specify how long to delay the first polling attempt. The delay is specified using the endpoint's **delay** attribute which specifies the delay interval in milliseconds.



NOTE

If you have specified a date for the first polling attempt, the delay will be added to the date to determine when to make the first polling attempt.

Example 10.6, “Poller endpoint with a delayed start time” shows the configuration for a poller endpoint that begins polling five minutes after it is started.

Example 10.6. Poller endpoint with a delayed start time

```
<beans xmlns:file="http://servicemix.apache.org/file/1.0"
       xmlns:foo="http://servicemix.org/demo/">

  <file:poller service="foo:filePoller"
               endpoint="filePoller"
               targetService="foo:fileSender"
               file="inbox"
               delay="300000" />
  ...
</beans>
```

Configuring the polling interval

By default, poller endpoints poll the file system every five seconds. You can configure the polling interval by providing a value for the endpoint's **period** attribute. The **period** attribute specifies the number of milliseconds the endpoint waits between polling attempts.

Example 10.7, “Poller Endpoint with a thirty second polling interval” shows the configuration for a poller endpoint that uses a thirty second polling interval.

Example 10.7. Poller Endpoint with a thirty second polling interval

```
<beans xmlns:file="http://servicemix.apache.org/file/1.0"
       xmlns:foo="http://servicemix.org/demo/">

  <file:poller service="foo:filePoller"
               endpoint="filePoller"
               targetService="foo:fileSender"
               file="inbox"
               period="30000" />
  ...
</beans>
```

10.5. FILE LOCKING

Overview

It is possible to have multiple instances of a poller endpoint attempting to read a file on the system. To ensure that there are no conflicts in accessing the file, poller endpoints obtain an exclusive lock on a file while it is being processed.

The locking behavior is controlled by an implementation of the **`org.apache.servicemix.common.locks.LockManager`** interface. By default, poller endpoints use a provided implementation of this interface. If the default behavior is not appropriate for your application, you can implement the **`LockManager`** interface and configure your endpoints to use your implementation.

Implementing a lock manager

To implement a custom lock manager, you need to provide your own implementation of the **`org.apache.servicemix.common.locks.LockManager`** interface. The **`LockManager`** has single method, **`getLock()`** that needs to be implemented. Example 10.8, “The lock manager's get lock method” shows the signature for **`getLock()`**.

Example 10.8. The lock manager's get lock method

```
Lock getLock(String id);
```

The **`getLock()`** method takes a string that represents the URI of the file being processes and it returns a **`java.util.concurrent.locks.Lock`** object. The returned **`Lock`** object holds the lock for the specified file.

Example 10.9, “Simple lock manager implementation” shows a simple lock manager implementation.

Example 10.9. Simple lock manager implementation

```
package org.apache.servicemix.demo;
```

```
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.ConcurrentMap;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

import org.apache.servicemix.common.locks.LockManager;

public class myLockManager implements LockManager
{
  private ConcurrentMap<String, Lock> locks = new
ConcurrentHashMap<String, Lock>();

  public Lock getLock(String id)
  {
    Lock lock = locks.get(id);
    if (lock == null)
    {
      lock = new ReentrantLock();
      Lock oldLock = locks.putIfAbsent(id, lock);
      if (oldLock != null)
      {
        lock = oldLock;
      }
    }
    return lock;
  }

}
```

Configuring the endpoint to use a lock manager

You configure a poller endpoint to use a custom lock manager using its **`lockManager`** attribute. The **`lockManager`** attribute's value is a reference to a **`bean`** element specifying the class of your custom lock manager implementation.

Example 10.10, “Poller endpoint using a custom lock manager” shows configuration for a poller endpoint that uses a custom lock manager.

Example 10.10. Poller endpoint using a custom lock manager

```
<beans xmlns:file="http://servicemix.apache.org/file/1.0"
       xmlns:foo="http://servicemix.org/demo/">

  <file:poller service="foo:filePoller"
               endpoint="filePoller"
               targetService="foo:fileSender"
               file="inbox"
               lockManager="#myLockManager" />

  <bean id="myLockManager"
class="org.apache.servicemix.demo.myLockManager" />
  ...
</beans>
```

NOTE

You can also configure a poller endpoint to use a custom lock manager by adding a child **`lockManager`** element to the endpoint's configuration. The **`lockManager`** element simply wraps the **`bean`** element that configures the lock manager.

10.6. FILE FILTERING

Overview

When a poller endpoint is configured to poll a directory it will attempt to consume any file placed into that directory. If you want to limit the files a poller endpoint will attempt to consume, you can configure the endpoint to filter files based on their names. To do so, you must supply the endpoint with an implementation of the **`java.io.FileFilter`** interface.

There are several file filter implementation available in open source including the Apache Commons IO implementations and the Apache Jakarta-ORO implementations. You can also implement your own file filter if you need specific filtering capabilities.

Implementing a file filter

To implement a file filter, you need to provide an implementation of the **`java.io.FileFilter`** interface. The **`FileFilter`** interface has a single method, **`accept()`**, that needs to be implemented. Example 10.11, “File filter's accept method” shows the signature of the **`accept()`** method.

Example 10.11. File filter's accept method

```
public boolean accept()(java.io.File pathname);
```

The **`accept()`** method takes a **`File`** object that represents the file being checked against the filter. If the file passes the filter, the **`accept()`** method should return **`true`**. If the file does not pass, then the method should return **`false`**.

Example 10.12, “Simple file filter implementation” shows a file filter implementation that matches against a string passed into its constructor.

Example 10.12. Simple file filter implementation

```
package org.apache.servicemix.demo;

import java.io.File;
import java.io.FileFilter;

public class myFileFilter implements FileFilter
{
  String filtername = "joe.xml";

  public myFileFilter()
  {
  }
```

```
  public myFileFilter(String filtername)
  {
    this.filtername = filtername;
  }

  public boolean accept(File file)
  {
    String name = file.getName();
    return name.equals(this.filtername);
  }
}
```

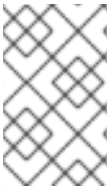
Configuring an endpoint to use a file filter

You configure a poller endpoint to use a file filter using its **`filter`** attribute. The **`filter`** attribute's value is a reference to a **bean** element specifying the class of the file filter implementation.

Example 10.13, “Poller endpoint using a file filter” shows configuration for a poller endpoint that uses the file filter implemented in Example 10.11, “File filter's accept method”. The **`constructor-arg`** element sets the filter's fitlername by passing a value into the constructor.

Example 10.13. Poller endpoint using a file filter

```
<beans xmlns:file="http://servicemix.apache.org/file/1.0"
       xmlns:foo="http://servicemix.org/demo/">

  <file:poller service="foo:filePoller"
               endpoint="filePoller"
               targetService="foo:fileSender"
               file="inbox"
               filter="#myFilter" />

  <bean id="myFilter" class="org.apache.servicemix.demo.myFileFilter">
    <constructor-arg value="joefred.xml" />
  </bean>
  ...
</beans>
```



NOTE

You can also configure a poller endpoint to use a file filter by adding a child **`filter`** element to the endpoint's configuration. The **`filter`** element simply wraps the **bean** element that configures the file filter.

CHAPTER 11. USING SENDER ENDPOINTS

Abstract

Sender endpoints write messages to the file system.

IMPORTANT

The Java Business Integration components of Red Hat JBoss Fuse are considered deprecated. You should consider migrating any JBI applications to OSGi.

11.1. INTRODUCTION TO SENDER ENDPOINTS

Overview

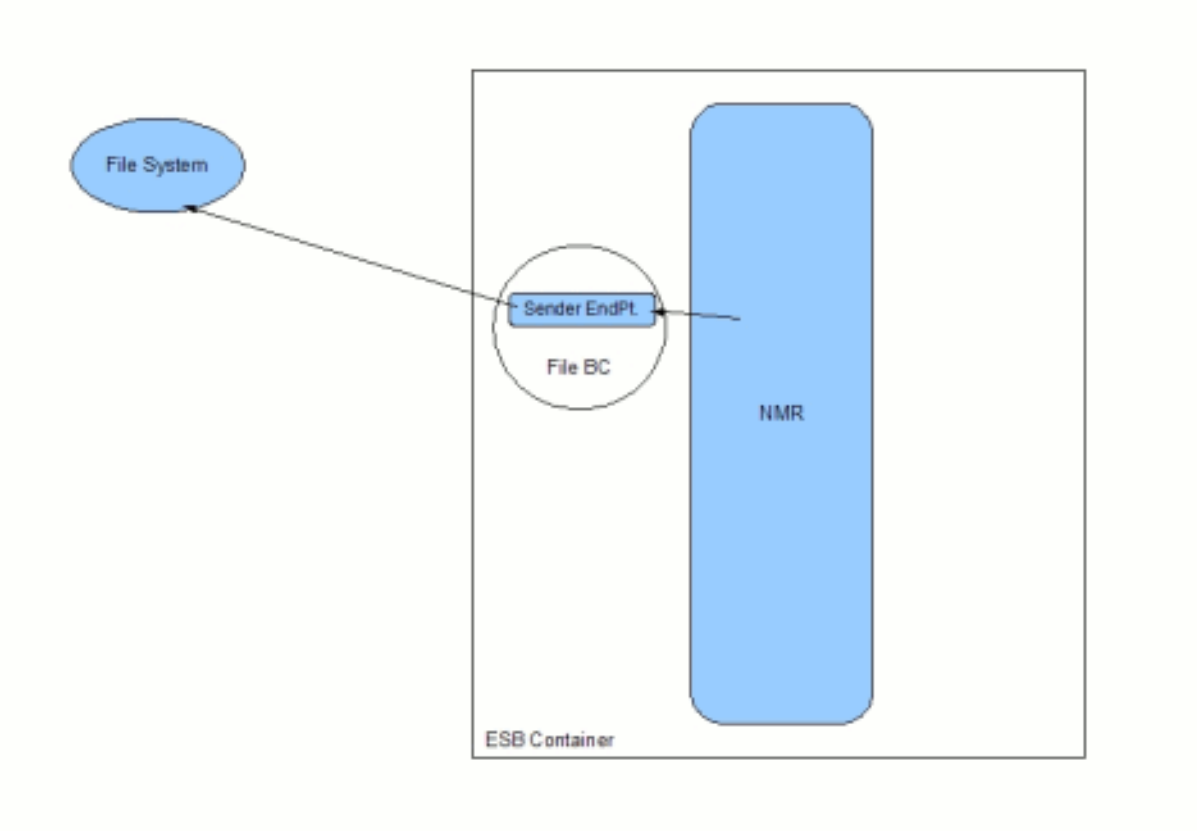
The function of a sender endpoint is to write data, in the form of files, to a location on a file system. You can control the location of the files written to the file system and have some control over the name of the generated files. You can also control if data is appended to existing files or if new copies of a file are created.

By default, sender endpoints write XML data to the file system. You can change this behavior by configuring the endpoint to use a custom marshaler.

Where does a sender endpoint fit into a solution?

Sender endpoints play the role of provider from the vantage point of the other endpoints in the ESB. As shown in Figure 11.1, “Sender endpoint”, a sender endpoint receives messages from the NMR and writes the message data to the file system.

Figure 11.1. Sender endpoint



Configuration element

Sender endpoints are configured using the **sender** element. All its configuration can be specified using attributes of this element.

Configuring a sender endpoint to use custom marshalers require the addition of other elements. These can either be separate **bean** elements or child elements of the **sender** element.

11.2. BASIC CONFIGURATION

Overview

The basic requirements for configuring a sender endpoint are straightforward. You need to supply the following information:

- the endpoint's name
- the endpoint's service name
- the file or directory to which files are written

All of this information is provided using attributes of the **sender** element.

Identifying the endpoint

All endpoints in the ESB need to have a unique identity. An endpoint's identity is made up of two pieces:

- a service name

- an endpoint name

Table 11.1, “Attributes for identifying a sender endpoint” describes the attributes used to identify a sender endpoint.

Table 11.1. Attributes for identifying a sender endpoint

Name	Description
`service`	Specifies the service name of the endpoint. This value must be a valid QName and does not need to be unique across the ESB.
`endpoint`	Specifies the name of the endpoint. This value is a simple string. It must be unique among all of the endpoints associated with a given service name.

Specifying the file destination

You specify the location the sender endpoint writes files using the `sender` element's `directory` attribute. This attribute takes a URI that identifies a location on the file system.

IMPORTANT

Relative URIs are resolved from the directory in which the Red Hat JBoss Fuse container was started.

Using the default marshaler, the name of the file is determined by the org.apache.servicemix.file.name property. This property is set on either the message exchange or the message by the endpoint originating the message exchange.

IMPORTANT

The marshaler is responsible for determining the name of the file being written. For more information on marshalers see Chapter 12, *File Marshalers*.

Example

Example 11.1, “Simple sender endpoint” shows the configuration for a simple sender endpoint.

Example 11.1. Simple sender endpoint

```
<beans xmlns:file="http://servicemix.apache.org/file/1.0"
       xmlns:foo="http://servicemix.org/demo/">

  <file:sender service="foo:fileSender"
               endpoint="sender"
               directory="outbox" />
  ...
</beans>
```

11.3. CONFIGURING A SENDER ENDPOINT'S INTERACTION WITH THE FILE SYSTEM

Overview

Sender endpoints interact with the file system in basic ways. You can configure a number of the aspects of this behavior including:

- if the endpoint creates the directory where it writes files
- how the endpoint names temporary files

Directory creation

The default behavior of a sender endpoint is to automatically create the target directory for its files if that directory does not already exist. To configure an endpoint to not create the target directory, you set its **`autoCreateDirectory`** attribute to **`false`**. If the directory does not exist, the endpoint will do nothing. You will then have to create the directory manually.

Example 11.2, “Sender endpoint that creates its target directory” shows the configuration for a sender endpoint that does not automatically create its target directory.

Example 11.2. Sender endpoint that creates its target directory

```
<beans xmlns:file="http://servicemix.apache.org/file/1.0"
       xmlns:foo="http://servicemix.org/demo/">

  <file:sender service="foo:fileSender"
               endpoint="fileSender"
               directory="outbox"
               autoCreateDirectory="false" />
  ...
</beans>
```

Appending data

By default, sender endpoints overwrite existing files. If a message wants to reuse the name of an existing file, the file on the file system is overwritten. You can configure a sender endpoint to append the message to the existing file by setting the endpoint's **`append`** attribute to **`true`**.

Example 11.3, “Sender endpoint that appends existing files” shows the configuration for an endpoint that appends messages to a file if it already exists.

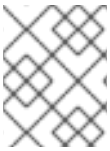
Example 11.3. Sender endpoint that appends existing files

```
<beans xmlns:file="http://servicemix.apache.org/file/1.0"
       xmlns:foo="http://servicemix.org/demo/">

  <file:sender service="foo:fileSender"
               endpoint="fileSender"
               directory="outbox"
```

```
                append="true" />
  ...
</beans>
```

Temporary file naming

By default, sender endpoints check the message exchange, or the message itself, for the name to use for the file being written. If the endpoint cannot determine a name for the target file, it will use a temporary file name. Table 11.2, “Attributes used to determine a temporary file name” describes the attributes used to generate the temporary file name.



NOTE

Checking for the name of the file to write is handled by the marshaler. For more information on marshalers see Chapter 12, *File Marshalers*.

Table 11.2. Attributes used to determine a temporary file name

Name	Description	Default
tempFilePrefix	Specifies the prefix used when creating output files.	**servicemix-**
tempFileSuffix	Specifies the file extension to use when creating output files.	**.xml**

The generated file names will have the form ***tempFilePrefixXXXXXtempFileSuffix***. The five Xs in the middle of the filename will be filled with randomly generated characters. So given the configuration shown in Example 11.4, “Configuring a sender endpoint's temporary file prefix”, a possible temporary filename would be **widgets-xy60s.xml**.

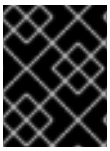
Example 11.4. Configuring a sender endpoint's temporary file prefix

```
<beans xmlns:file="http://servicemix.apache.org/file/1.0"
       xmlns:foo="http://servicemix.org/demo/">

  <file:sender service="foo:fileSender"
               endpoint="fileSender"
               directory="outbox"
               tempFilePrefix="widgets-" />
  ...
</beans>
```

CHAPTER 12. FILE MARSHALERS

Abstract

When using file component endpoints, you may want to customize how messages are processed as they pass in and out of the ESB. The Red Hat JBoss Fuse file binding component allows you to write custom marshalers for your file component endpoints.



IMPORTANT

The Java Business Integration components of Red Hat JBoss Fuse are considered deprecated. You should consider migrating any JBI applications to OSGi.

OVERVIEW

File component endpoints use a marshaler for processing messages. Poller endpoints rely on the marshaler for reading data off of the file system and normalizing it so it can be passed to the NMR. Sender endpoints rely on the marshaler for determining the name of the file to be written and for converting the normalized messages into the format to be written to the file system.

The default marshaler used by file component endpoints reads and writes valid XML files. It queries the message exchange, and the message, received from the NMR for the name of the outgoing message. The default marshaler expects the file name to be stored in a property called org.apache.servicemix.file.name.

If your application requires different functionality from the marshaler, you can provide a custom marshaler by implementing the **`org.apache.servicemix.components.util.FileMarshaler`** interface. You can easily configure your endpoints to use your custom marshaler instead of the default one.

PROVIDED FILE MARSHALERS

In addition to the default file marshaler, Red Hat JBoss Fuse provides two other file marshalers that file component endpoints can use:

Binary File Marshaler

The binary file marshaler is provided by the class **`org.apache.servicemix.components.util.BinaryFileMarshaler`**. It reads in binary data and adds the data to the normalized message as an attachment. You can set the name of the attachment and specify a content type for the attachment using the properties shown in Table 12.1, “Properties for configuring the binary file marshaler”.

Table 12.1. Properties for configuring the binary file marshaler

Name	Description	Default
attachment	Specifies the name of the attachment added to the normalized message.	**`content`**

Name	Description	Default
contentType	Specifies the content type of the binary data being used. Content types are specified using MIME types. MIME types are specified by RFC 2045.	

Flat File Marshaler

The flat file marshaler is provided by the class **`org.apache.servicemix.components.util.SimpleFlatFileMarshaler`**. It reads in flat text files and converts them into XML messages.

By default, the file is wrapped in a **`File`** element. Each line in the file is wrapped in a **`Line`** element with a **`number`** attribute that represents the position of the line in the original file.

You can control some aspects of the generated XML file using the properties described in Table 12.2, “Properties used to control the flat file marshaler”.

Table 12.2. Properties used to control the flat file marshaler

Name	Description	Default
docElementname	Specifies the name of the root element generated by a file.	**`File`**
lineElementname	Specifies the name of the element generated for each line of the file.	**`Line`**
insertLineNumbers	Specifies if the elements corresponding to a line will use the **`number`** attribute.	**`true`**

IMPLEMENTING A FILE MARSHALER

To develop a custom file marshaler, you need to implement the **`org.apache.servicemix.components.util.FileMarshaler`** interface. Example 12.1, “The file marshaler interface” shows the interface.

Example 12.1. The file marshaler interface

```
package org.apache.servicemix.components.util;

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

import javax.jbi.JBIException;
```

```
import javax.jbi.messaging.MessageExchange;
import javax.jbi.messaging.MessagingException;
import javax.jbi.messaging.NormalizedMessage;

public interface FileMarshaler
{
  void readMessage(MessageExchange exchange, NormalizedMessage message,
InputStream in, String path) throws IOException, JBIException;

  String getOutputName(MessageExchange exchange, NormalizedMessage
message) throws MessagingException;

  void writeMessage(MessageExchange exchange, NormalizedMessage message,
OutputStream out, String path) throws IOException, JBIException;
}
```

The **FileMarshaler** interface has three methods that need to be implemented:

readMessage()

The **readMessage()** method is responsible for reading a file from the file system and converting the data into a normalized message. Table 12.3, “Parameters for reading messages from the file system” describes the parameters used by the method.

Table 12.3. Parameters for reading messages from the file system

Name	Description
exchange	Contains the **MessageExchange** object that is going to be passed to the NMR.
message	Contains the **NormalizedMessage** object that is going to be passed to the NMR.
in	Contains the **BufferedInputStream** which points to the file in the file system.
path	Contains the full path to the file on the file system as determined by the Java **getCanonicalPath()** method.

getOutputName()

The **getOutputName()** method returns the name of the file to be written to the file system. The message exchange and the message received by the sender endpoint are passed to the method.

IMPORTANT

The returned file name does not contain a directory path. The sender endpoint uses the directory it was configured to use.

writeMessage()

The **writeMessage()** method is responsible for writing messages received from the NMR to the file system as files. Table 12.4, "Parameters for writing messages to the file system" describes the parameters used by the method.

Table 12.4. Parameters for writing messages to the file system

Name	Description
exchange	Contains the **MessageExchange** object received from the ESB.
message	Contains the **NormalizedMessage** object received from the ESB.
out	Contains the **BufferedOutputStream** which points to the file in the file system.
path	Contains the path to the file are returned from the **getOutputName()** method.

Example 12.2, "Simple file marshaler" shows a simple file mashaler.

Example 12.2. Simple file marshaler

```
package org.apache.servicemix.demos;

import java.io.File;
import java.io.IOException;
import java.io.InputStream;
import java.io.ObjectOutputStream;
import java.io.OutputStream;

import javax.jbi.JBIException;
import javax.jbi.messaging.MessageExchange;
import javax.jbi.messaging.MessagingException;
import javax.jbi.messaging.NormalizedMessage;

public class myFileMarshaler implements FileMarshaler
{

  public void readMessage(MessageExchange exchange, NormalizedMessage
message,
                                                    InputStream in,
String path)
  throws IOException, JBIException
  {
    message.setContent(new StreamSource(in, path));
  }

  public String getOutputName(MessageExchange exchange,
NormalizedMessage message)
  throws MessagingException
```

```
  {
    return "fred.xml";
  }

  public void writeMessage(MessageExchange exchange, NormalizedMessage
message,
                                                    OutputStream out,
String path)
  throws IOException, JBIException
  {
    Source src = message.getContent();
    if (src == null)
    {
      throw new NoMessageContentAvailableException(exchange);
    }
    try
    {
      ObjectOutputStream objectOut = new ObjectOutputStream(out);
      objectOut.writeObject(src);
    }
  }
}
```

CONFIGURING AN ENDPOINT TO USE A FILE MARSHALER

You configure a file component endpoint to use a file marshaler using its **marshaler** attribute. The **marshaler** attribute's value is a reference to a **bean** element specifying the class of the file filter implementation.

Example 12.3, “Poller endpoint using a file marshaler” shows configuration for a poller endpoint that uses the file marshaler implemented in Example 12.2, “Simple file marshaler”.

Example 12.3. Poller endpoint using a file marshaler

```
<beans xmlns:file="http://servicemix.apache.org/file/1.0"
       xmlns:foo="http://servicemix.org/demo/">

  <file:poller service="foo:filePoller"
               endpoint="filePoller"
               targetService="foo:fileSender"
               file="inbox"
               marshaler="#myMarshaler" />

  <bean id="myMarshaler"
class="org.apache.servicemix.demo.myFileMarshaler" />
```

NOTE

You can also configure a file component endpoint to use a file marshaler by adding a child **marshaler** element to the endpoint's configuration. The **marshaler** element simply wraps the **bean** element that configures the file marshaler.

APPENDIX B. POLLER ENDPOINT PROPERTIES

ATTRIBUTES

Table B.1, “Attributes for configuring a poller endpoint” describes the attributes used to configure a poller endpoint.

Table B.1. Attributes for configuring a poller endpoint

Name	Type	Description	Default
`service`	QName	Specifies the service name of the endpoint.	*required*
`endpoint`	String	Specifies the name of the endpoint.	*required*
`interfaceName`	QName	Specifies the interface name of the endpoint.	
`targetService`	QName	Specifies the service name of the target endpoint.	
`targetEndPoint`	String	Specifies the name of the target endpoint.	
`targetInterface`	QName	Specifies the interface name of the target endpoint.	
`targetUri`	string	Specifies the URI of the target endpoint.	
`autoCreateDirectory`	boolean	Specifies if the endpoint will create the target directory if it does not exist.	**`true`**
`firstTime`	date	Specifies the date and the time the first poll will take place.	null (The first poll will happen right after start up.)
`delay`	long	Specifies amount of time, in milliseconds, to wait before performing the first poll.	**`0`**

Name	Type	Description	Default
period	long	Specifies the amount of time, in milliseconds, between polls.	**5000**
file	String	Speficies the file or directory to poll.	*required*
deleteFile	boolean	Specifies if the file is deleted after it is processed.	**true**
recursive	boolean	Specifies if the endpoint processes sub directories when polling.	**true**
archive	string	Specifies the name of the directory to archive files into before deleting them.	null (no archiving)

BEANS

Table B.2, “Beans for configuring a poller endpoint” describes the beans which can be used to configure a poller endpoint.

Table B.2. Beans for configuring a poller endpoint

Name	Type	Description	Default
marshaler	**org.apache.servicemix.components.util.FileMarshaler**	Specifies the class used to marshal data from the file.	**DefaultFileMarshaler**
lockManager	**org.apache.servicemix.locks.LockManager**	Specifies the class implementing the file locking.	**SimpleLockManager**
filter	**java.io.FileFilter**	Specifies the class implementing the filtering logic to use for selecting files.	

APPENDIX C. SENDER ENDPOINT PROPERTIES

ATTRIBUTES

Table C.1, “Attributes for configuring a sender endpoint” describes the attributes used to configure a sender endpoint.

Table C.1. Attributes for configuring a sender endpoint

Name	Type	Description	Default
`service`	QName	Specifies the service name of the endpoint.	*required*
`endpoint`	String	Specifies the name of the endpoint.	*required*
`directory`	String	Specifies the name of the directory into which data is written.	*required*
`autoCreateDirectory`	boolean	Specifies if the endpoint creates the output directory if it does not exist.	**`true`**
`append`	boolean	Specifies if the data is appended to the end of an existing file or if the data is written to a new file.	**`false`**
`tempFilePrefix`	String	Specifies the prefix used when creating output files.	**`servicemix-`**
`tempFileSuffix`	String	Specifies the file extension to use when creating output files.	**`.xml`**

BEANS

Table C.2, “Attributes for configuring a sender endpoint” describes the beans used to configure a sender endpoint.

Table C.2. Attributes for configuring a sender endpoint

Name	Type	Description	Default

Name	Type	Description	Default
`marshaler`	`org.apache.servicemix.components.util.FileMarshaler`	Specifies the marshaler to use when writing data from the NMR to the file system.	`DefaultFileMarshaler`

PART III. JMS BINDING COMPONENT

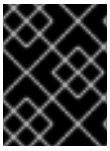
Abstract

This guide provides an overview of the JBI JMS binding component; describes how to configure the connection factory, how to create and configure various types of endpoints, and how to use the Maven tooling.

CHAPTER 13. INTRODUCTION TO THE RED HAT JBOSS FUSE JMS BINDING COMPONENT

Abstract

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



IMPORTANT

The Java Business Integration components of Red Hat JBoss Fuse are considered deprecated. You should consider migrating any JBI applications to OSGi.

OVERVIEW

The Red Hat JBoss Fuse 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 Red Hat JBoss Fuse 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 Red Hat JBoss Fuse 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

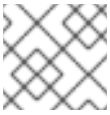
- SOAP support
- MIME support
- Customizable message marshaling

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 13.1, “JBI Descriptor for a JMS Service Unit” shows a JBI descriptor for a JMS service unit.

Example 13.1. JBI Descriptor for a JMS Service Unit

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

The elements shown in Example 13.1, “JBI Descriptor for a JMS Service Unit” do the following:

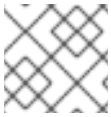
1. The **`service`** element is the root element of all service unit descriptors. The value of the **`binding-component`** attribute is always **`false`**.
2. The **`service`** element contains namespace references for all of the namespaces defined in the **`xbean.xml`** file's **`bean`** element.
3. 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.

4. 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.

5 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 Red Hat JBoss Fuse 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.

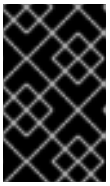
TIP

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 H, *Using the Maven OSGi Tooling*.

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 13.2, “Namespace Declaration for Using JMS Endpoints” to your **xbeans.xml** file's **beans** element.

Example 13.2. 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 13.3, “Schema Location for Using JMS Endpoints”.

Example 13.3. 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 14. CONFIGURING THE CONNECTION FACTORY

Abstract

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.

IMPORTANT

The Java Business Integration components of Red Hat JBoss Fuse are considered deprecated. You should consider migrating any JBI applications to OSGi.

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.

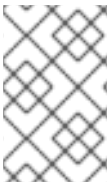
When using the Red Hat JBoss Fuse JMS binding component, you must configure each service unit with the information it needs to load a **`ConnectionFactory`** object. Often the **`ConnectionFactory`** object is looked up through JNDI. However, the information needed depends on the JMS provider you are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

Commonly used JMS providers include Red Hat JBoss A-MQ, Apache ActiveMQ, IBM's WebShere® MQ, BEA's WebLogic®, and Progress Software's SonicMQ®. JBoss A-MQ 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.

14.1. USING APACHE ACTIVEMQ CONNECTION FACTORIES

Overview

The recommended method for creating connections to 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 Red Hat JBoss Fuse 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 14.1, “Attributes for Configuring the Simple AMQPool Connection Factory”.

Table 14.1. 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 14.1, “Configuring a Simple AMQPool Connection Factory” shows a configuration snippet for configuring the simple AMQPool connection factory.

Example 14.1. 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 14.2, “Attributes for Configuring the XA AMQPool Connection Factory”.

Table 14.2. 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
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 14.2, “Configuring an XA AMQPool Connection Factory” shows a configuration snippet for configuring an XA AMQPool connection factory.

Example 14.2. 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 14.3, "Attributes for Configuring the JCA AMQPool Connection Factory".

Table 14.3. 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 14.3, "Configuring a JCA AMQPool Connection Factory" shows a configuration snippet for configuring the JCA AMQPool connection factory.

Example 14.3. 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>
```

14.2. USING JNDI

Overview

Many JMS providers store a reference to their connection factory in a JNDI service to ease retrieval. Red Hat JBoss Fuse 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 14.4, “Attributes for Using Spring's JEE JNDI Lookup”.

   Table 14.4. 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 14.4, “Getting the WebLogic Connection Factory Using Spring's JEE JNDI Look-up” shows a configuration snippet for using the JNDI look-up with WebLogic.

Example 14.4. 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:

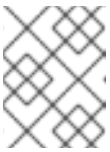
1. Add a **`bean`** element to your configuration for the JNDI template.
   a. Set the **`bean`** element's **`id`** attribute to a unique identifier.
   b. Set the **`bean`** element's **`class`** attribute to **`org.springframework.jndi.JndiTemplate`**.
   c. Add a **`property`** child element to the **`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 14.5, “Setting a Java Property” shows a **`prop`** element for setting the java.naming.factory.initial property.

      Example 14.5. 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 14.6, “Using a JNDI Template to Look Up a Connection Factory” shows a configuration fragment for retrieving the WebSphere MQ connection factory using Sun's reference JNDI implementation.

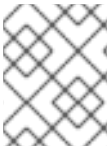
Example 14.6. Using a JNDI Template to Look Up a Connection Factory

```
<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>
```

14.3. 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 14.7, “Configuring a Connection Factory with a Spring Bean” shows an example of a WebSphere MQ connection factory configured as a Spring bean.

Example 14.7. 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 15. CREATING A CONSUMER ENDPOINT

Abstract

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.

IMPORTANT

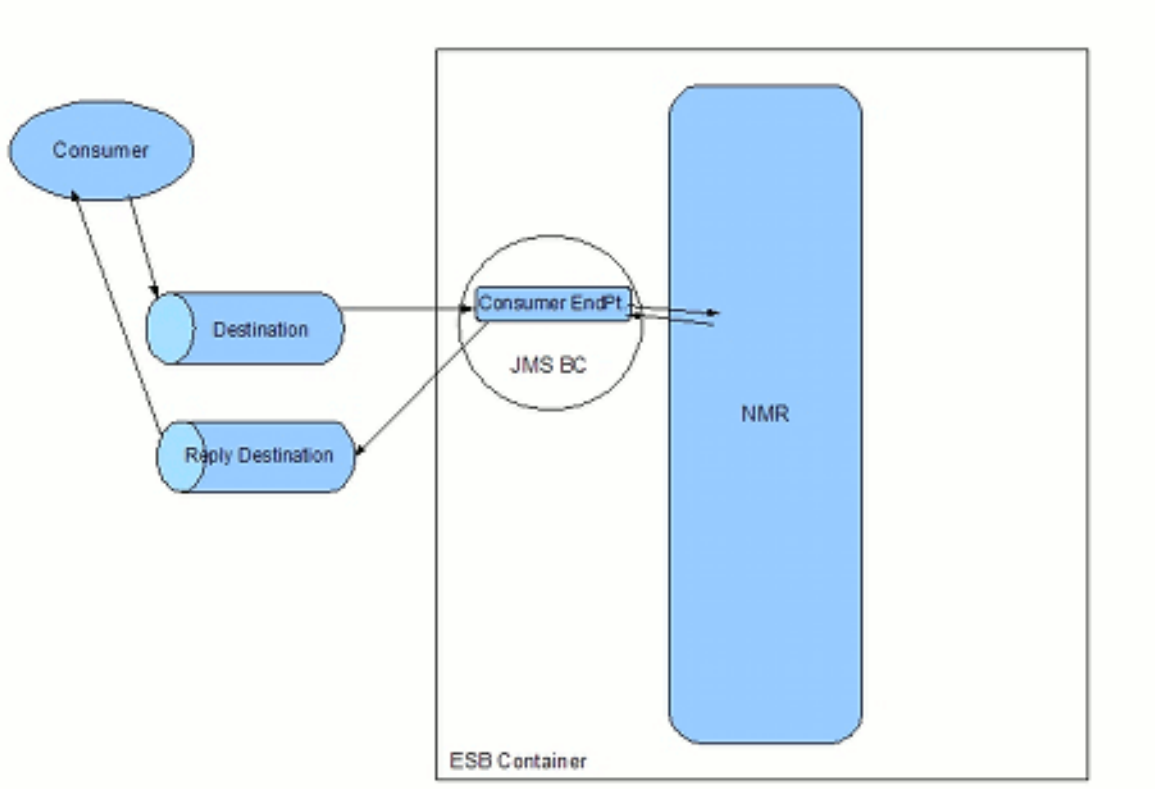
The Java Business Integration components of Red Hat JBoss Fuse are considered deprecated. You should consider migrating any JBI applications to OSGi.

15.1. 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 15.1, “Consumer Endpoint”,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 15.1. 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 Section 15.3, “Using the JCA Consumer Endpoint”.

15.2. USING THE GENERIC ENDPOINT OR THE SOAP ENDPOINT

15.2.1. 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 the section called “Types of consumer endpoints”.

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 Chapter 14, *Configuring the Connection Factory*.

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

   For more information see the section called “Configuring a destination”.

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

   For more information see the section called “Specifying the target endpoint”.

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.

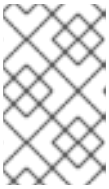
   You configure a destination using a Spring bean. You can add the bean directly to the endpoint by wrapping it in a **jms:destination** child element. You can also configure the bean separately and refer the bean using the endpoint's **destination** attribute as shown in Example 15.1, “Configuring a Consumer's Destination”.

Example 15.1. 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 Section 19.2, “Using a Custom Destination Resolver”.

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 15.2, “Basic Configuration for a Generic Consumer Endpoint” shows the basic configuration for a plain JMS provider endpoint.

Example 15.2. 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 15.3, “Basic Configuration for a SOAP Consumer Endpoint” shows the basic configuration for a SOAP JMS provider endpoint.

Example 15.3. 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>
```

15.2.2. 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

Red Hat JBoss Fuse'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 Red Hat JBoss Fuse 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 15.1, “Values for Configuring a Consumer's Listener Container” lists the values for the **`listenerType`** attribute.

Table 15.1. 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 15.4, “Configuring a SOAP Consumer to Use the Simple Listener Container” shows configuration for SOAP consumer that uses the simple listener container.

Example 15.4. 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 15.2, “Attributes Used to Performance Tune Standard JMS Consumers and SOAP JMS Consumers” describes the attributes used to tune endpoint performance.

Table 15.2. 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), **2**(CACHE_SESSION), and **3**(CACHE_CONSUMER).	**0**
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 15.5, “Tuning a Generic Consumer Endpoint” shows an example of a generic consumer that allows consumer level message caching and only tries once to receive a message.

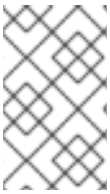
Example 15.5. Tuning a Generic Consumer Endpoint

```
<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 Red Hat JBoss Fuse 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 15.6, “Configuring a Consumer to Use a Pooled Session Factory” shows an example of configuring an endpoint to use the Spring framework's **`CommonsPoolServerSessionFactory`** object as a session factory.

Example 15.6. 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.CommonsPoolServerS
essionFactory" />
  ...
</beans>
```

15.2.3. 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 15.3, "Consumer Transaction Support" describes the possible values for the **`transacted`** attribute.

Table 15.3. 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 15.7, "Consumer using a Durable Subscription" shows a configuration snipit for a consumer that registers for a durable subscription.

Example 15.7. 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>
```

15.2.4. 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 15.8, “Configuring a SOAP Consumer to Use the JBI Wrapper” shows a configuration fragment for configuring a SOAP consumer to use the JBI wrapper.

Example 15.8. 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`**.

15.3. 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 Chapter 14, *Configuring the Connection Factory*.

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

   For more information see the section called “Configuring a destination”.

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 the section called “Specifying the target endpoint”.

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 15.9, “Configuring a JCA Consumer's Destination”.

Example 15.9. 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 Section 19.2, “Using a Custom Destination Resolver”.

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:

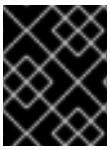
1. If you explicitly specify an endpoint using both the **targetService** attribute and the **targetEndpoint** attribute, the ESB will use that endpoint.

   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 15.10, “Basic Configuration for a JCA Consumer Endpoint” shows the configuration for a JCA consumer endpoint.

Example 15.10. 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>
```

15.4. 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.

15.4.1. Configuring the Reply Destination

Overview

Red Hat JBoss Fuse 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 Section 19.1, “Using a Custom Destination Chooser”.
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 15.11, “Configuring a Consumer's Reply Destination”. 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 Section 19.2, “Using a Custom Destination Resolver”.

Example

Example 15.11, “Configuring a Consumer's Reply Destination” shows an example of configuring a consumer endpoint to use a dedicated JMS destination.

Example 15.11. 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>
```

15.4.2. 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 15.12, “Consumer with Reply QoS Properties” shows the configuration for a consumer whose reply messages are set to have low priority and to be non-persistent.

Example 15.12. 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>
```

-

15.4.3. 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 Red Hat JBoss Fuse 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 15.13, “Adding Custom Properties to a Reply Message” shows an example of a SOAP consumer whose reply messages have a set of custom properties added to their header.

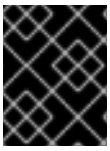
Example 15.13. Adding Custom Properties to a Reply Message

```
<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 16. CREATING A PROVIDER ENDPOINT

Abstract

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.



IMPORTANT

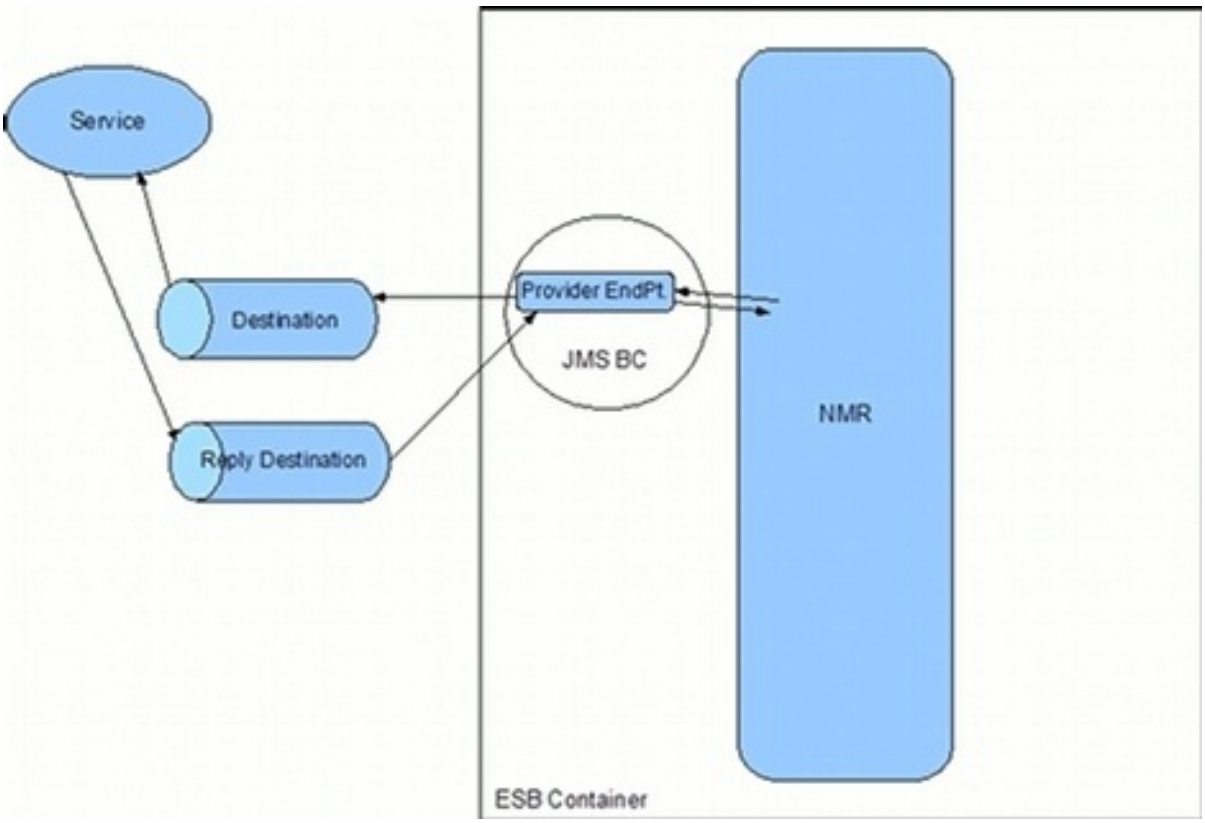
The Java Business Integration components of Red Hat JBoss Fuse are considered deprecated. You should consider migrating any JBI applications to OSGi.

16.1. 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 16.1, “Provider Endpoint”, 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 16.1. 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.

16.2. BASIC CONFIGURATION

Procedure

To configure a provider endpoint do the following:

1. Decide what type of provider endpoint to use.

   See the section called “Types of providers”.

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 Chapter 14, *Configuring the Connection Factory*.

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

   For more information see the section called “Configuring a destination”.

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 Section 19.1, “Using a Custom Destination Chooser”.

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 16.1, “Configuring a Provider's Destination”. You can also add the bean directly to the endpoint by wrapping it in a **jms:destination** child element.

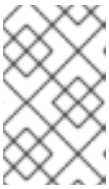
Example 16.1. Configuring a Provider's Destination

```
<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 Section 19.2, “Using a Custom Destination Resolver”.

Examples

Example 16.2, “Basic Configuration for a Generic Provider Endpoint” shows the basic configuration for a plain JMS provider endpoint.

Example 16.2. 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 16.3, “Basic Configuration for a SOAP Provider Endpoint” shows the basic configuration for a SOAP JMS provider endpoint.

Example 16.3. 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>
```

16.3. 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 Section 19.1, “Using a Custom Destination Chooser”.

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

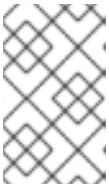
   The default destination chooser checks the message exchange received from the NMR for a DESTINATION_KEY property. If the message exchange has that property set, it returns that destination.

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 16.1, “Configuring a Provider's Destination”. 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 Section 19.2, “Using a Custom Destination Resolver”.

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 16.4, “JMS Provider Endpoint with a Response Destination” shows a JMS provider endpoint that will wait for a response for one minute.

Example 16.4. 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>
```

16.4. ADVANCED PROVIDER CONFIGURATION

16.4.1. 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 16.5, “Setting JMS Provider Endpoint Message Properties” shows configuration for a JMS SOAP provider whose messages have a priority of 1.

Example 16.5. 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>
```

16.4.2. 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, Red Hat JBoss Fuse 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.

16.4.3. 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 16.6, “Configuring a SOAP Provider to Use the JBI Wrapper” shows a configuration fragment for configuring a SOAP provider to use the JBI wrapper.

Example 16.6. 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 17. MAKING ENDPOINTS STATEFUL

Abstract

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.

IMPORTANT

The Java Business Integration components of Red Hat JBoss Fuse are considered deprecated. You should consider migrating any JBI applications to OSGi.

OVERVIEW

Red Hat JBoss Fuse 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 JBoss Fuse 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 17.1, “Properties Used to Configure a JDBC Store Factory” list the properties you can set for the JDBC store factory.

Table 17.1. 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.

Name	Description
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 17.1, “Configuring a Statefull JMS Provider Endpoint” shows the configuration needed for a stateful JMS provider endpoint using MySQL as a JDBC accessible datastore.

Example 17.1. Configuring a Statefull JMS Provider Endpoint

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

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

4<bean id="mysql-ds"
        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 17.1, “Configuring a Statefull JMS Provider Endpoint” does the following:

1. Configures the endpoint's store factory by providing a reference to the bean configuring the factory.
2. Configures the endpoint to store a copy of the current message in the datastore.
3. Configures the JDBC factory store to create a datastore that can be accessed by a cluster of endpoints.
4. Configures the MySQL JDBC driver.

CHAPTER 18. WORKING WITH MESSAGE MARSHALERS

Abstract

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

IMPORTANT

The Java Business Integration components of Red Hat JBoss Fuse are considered deprecated. You should consider migrating any JBI applications to OSGi.

18.1. CONSUMER MARSHALERS

Overview

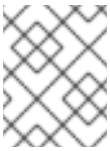
Consumer endpoints use an implementation of the **`org.apache.servicemix.jms.endpoints.JmsConsumerMarshaler`** interface to process the incoming JMS messages and convert them into normalized messages. Consumer marshalers also convert fault messages and response messages into JMS messages that can be 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 18.1, “The Consumer Marshaler Interface”, has five methods that need implementing:

Example 18.1. 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 18.2, “Consumer Marshaler Implementation” shows a simple consumer marshaler implementation.

Example 18.2. 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().createExchange(M
essageExchangeSupport.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, JmsContext 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 18.3, “Configuring a Consumer to Use a Customer Marshaler” shows configuration for a consumer that uses a custom marshaler.

Example 18.3. 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.

18.2. 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 18.4, “The Provider Marshaler Interface”, has two methods you need to implement:

Example 18.4. The Provider Marshaler Interface

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

```
  void populateMessage(Message message, MessageExchange exchange,
NormalizedMessage normalizedMessage) throws Exception;
}
```

`createMessage()`

The **`createMessage()`** method uses information from the Red Hat JBoss Fuse 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 Red Hat JBoss Fuse core.

Example 18.5, “Provider Marshaler Implementation” shows a simple provider marshaler implementation.

Example 18.5. 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 18.6, “Configuring a Provider to Use a Customer Marshaler” shows configuration for a provider that uses a custom marshaler.

Example 18.6. 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 19. IMPLEMENTING DESTINATION RESOLVING LOGIC

Abstract

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.

IMPORTANT

The Java Business Integration components of Red Hat JBoss Fuse are considered deprecated. You should consider migrating any JBI applications to OSGi.

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 Red Hat JBoss Fuse JMS binding component provides two mechanisms for endpoints to dynamically discover destinations:

destination choosers

Destination choosers are specific to the Red Hat JBoss Fuse 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.

19.1. USING A CUSTOM DESTINATION CHOOSER

Overview

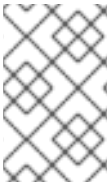
Provider endpoints use a destination chooser to determine the JMS destination on which to send requests and receive replies. They have a default destination chooser that queries the message exchange for a property that specifies the destination to use. Consumer endpoints use destination choosers to determine 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 19.1, "Destination Chooser Method", 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 Section 19.2, "Using a Custom Destination Resolver".

Example 19.1. 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 19.2, "Simple Destination Chooser" 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 19.2. 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 19.3, “Configuring a Destination Chooser with a Bean Reference”, 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 19.3. 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 19.4, “Explicitly Configuring a Destination Chooser” 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 19.4. 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>
```

19.2. USING A CUSTOM DESTINATION RESOLVER

Overview

Destination resolvers are a part of the JMS technology Red Hat JBoss Fuse 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.

Red Hat JBoss Fuse 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.

Red Hat JBoss Fuse 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 the section called “Configuring an endpoint to use a destination resolver”.

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 19.5, “Destination Resolver Method”, takes three parameters: a JMS session, a destination name, and a boolean specifying if the destination is a JMS topic.[2] It returns a JMS destination that correlates to the provided destination name.

Example 19.5. Destination Resolver Method

```
Destination resolveDestinationName(Session session,
                                   String destinationName,
                                   boolean pubSubDomain)
    throws JMSException;
```

Example 19.6, “Simple Destination Resolver” shows a simple destination resolver implementation.

Example 19.6. 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 19.7, “Configuring a Destination Resolver with a Bean Reference”, 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 19.7. 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 19.8, “Explicitly Configuring a Destination Resolver” 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 19.8. 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>
```

[2] If the value is **false**, a JMS queue will be returned.

APPENDIX D. CONSUMER ENDPOINT PROPERTIES

D.1. COMMON PROPERTIES

Attributes

The attributes described in Table D.1, “Common Consumer Endpoint Property Attributes” can be used on all elements used to configure a consumer endpoint.

Table D.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)

Name	Type	Description	Required
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)
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 D.2, “Common Consumer Endpoint Property Beans” can be used on all elements used to configure a consumer endpoint.

Table D.2. Common Consumer Endpoint Property Beans

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

D.2. PROPERTIES SPECIFIC TO GENERIC CONSUMERS AND SOAP CONSUMERS

Common Attributes

The attributes described in Table D.3, “Attributes Uses to Configure Standard JMS Consumers and SOAP JMS Consumers” are specific to the `jms:consumer` element and the `jms:soap-consumer` elements.

Table D.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 when creating a **`Session`** to send a message.	no (defaults to **`Session.AUTO_ACKNOWLEDGE`**)

Attribute	Type	Listener(s)	Description	Required
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 D.4, “Elements Uses to Configure Standard JMS Consumers and SOAP JMS Consumers” are specific to the **jms:consumer** element and the **jms:soap-consumer** elements.

Table D.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 D.5, “Attributes for the JMS SOAP Consumer” are specific to the `jms:soap-consumer` element.

Table D.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

D.3. PROPERTIES SPECIFIC TO A JCA CONSUMER

The elements described in Table D.6, “Elements Used to Configure a JCA Consumer” are specific to the **`jms:jca-consumer`** element.

Table D.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 E. PROVIDER ENDPOINT PROPERTIES

E.1. COMMON PROPERTIES

Attributes

The attributes described in Table E.1, “Common Provider Endpoint Property Attributes” can be used on all elements used to configure a provider endpoint.

Table E.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`**)

Attribute	Type	Description	Required
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**
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 E.2, “Common Provider Endpoint Property Beans” can be used on all elements used to configure a JMS provider endpoint.

Table E.2. Common Provider Endpoint Property Beans

Element	Type	Description	Required
destination	**Destination**	Specifies the JMS destination used to send messages.	no

Element	Type	Description	Required
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**)
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

E.2. PROPERTIES SPECIFIC TO SOAP PROVIDERS

Attributes

The attributes described in Table E.3, “Attributes Used to Configure SOAP JMS Providers” are specific to **jms:soap-provider** elements.

Table E.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**)

Attribute	Type	Description	Required
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 E.4, “Elements Used to Configure SOAP JMS Providers” are specific to **jms:soap-provider** elements.

Table E.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

PART IV. CXF BINDING COMPONENT

Abstract

This guide provides an overview of the JBI CXF binding component; describes how to define endpoints in WSDL, how to configure and package endpoints, and how to configure the CXF runtime; describes the properties of consumer and provider endpoints; and describes how to use the Maven tooling.

CHAPTER 20. INTRODUCTION TO THE APACHE CXF BINDING COMPONENT

Abstract

The Apache CXF binding component allows you to create SOAP/HTTP and SOAP/JMS endpoints.

IMPORTANT

The Java Business Integration components of Red Hat JBoss Fuse are considered deprecated. You should consider migrating any JBI applications to OSGi.

OVERVIEW

The Apache CXF binding component provides connectivity to external endpoints using either SOAP/HTTP or SOAP/JMS. The endpoints are defined using WSDl files that contain Apache CXF specific extensions for defining the transport. In addition, you can add Apache CXF-based Spring configuration to use the advanced features.

It allows for the creation of two types of endpoint:

consumer endpoint

A consumer endpoint listens for messages on a specified address. When it receives a message it sends it to the NMR for delivery to the appropriate endpoint. If the message is part of a two-way exchange, then the consumer endpoint is also responsible for returning the response to the external endpoint.

For information about configuring consumer endpoints see Chapter 28, *Consumer Endpoints*.

provider endpoint

A provider endpoint receives messages from the NMR. It then packages the message as a SOAP message and sends it to the specified external address. If the message is part of a two-way message exchange, the provider endpoint waits for the response from the external endpoint. The provider endpoint will then direct the response back to the NMR.

For information about configuring provider endpoints see Chapter 29, *Provider Endpoints*.

KEY FEATURES

The Apache CXF binding component has the following features:

- HTTP support
- JMS 1.1 support
- SOAP 1.1 support
- SOAP 1.2 support
- MTOM support

- Support for all MEPs as consumers or providers
- SSL support
- WS-Security support
- WS-Policy support
- WS-RM support
- WS-Addressing support

STEPS FOR WORKING WITH THE APACHE CXF BINDING COMPONENT

Using the Apache CXF binding component to expose SOAP endpoints usually involves the following steps:

1. Defining the contract for your endpoint in WSDL.

   See ???.

2. Configuring the endpoint and packaging it into a service unit.

   See ???.

3. Bundling the service unit into a service assembly for deployment into the Red Hat JBoss Fuse container.

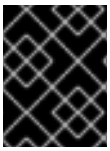
MORE INFORMATION

For more information about using Apache CXF to create SOAP endpoints see the Apache CXF documentation.

CHAPTER 21. INTRODUCING WSDL CONTRACTS

Abstract

WSDL documents define services using Web Service Description Language and a number of possible extensions. The documents have a logical part and a concrete part. The abstract part of the contract defines the service in terms of implementation neutral data types and messages. The concrete part of the document defines how an endpoint implementing a service will interact with the outside world.



IMPORTANT

The Java Business Integration components of Red Hat JBoss Fuse are considered deprecated. You should consider migrating any JBI applications to OSGi.

The recommended approach to design services is to define your services in WSDL and XML Schema before writing any code. When hand-editing WSDL documents you must make sure that the document is valid, as well as correct. To do this you must have some familiarity with WSDL. You can find the standard on the W3C web site, www.w3.org.

21.1. STRUCTURE OF A WSDL DOCUMENT

A WSDL document is, at its simplest, a collection of elements contained within a root **`definition`** element. These elements describe a service and how an endpoint implementing that service is accessed.

A WSDL document has two distinct parts:

- A logical part that defines the service in implementation neutral terms
- A concrete part that defines how an endpoint implementing the service is exposed on a network

The logical part

The logical part of a WSDL document contains the **`types`**, the **`message`**, and the **`portType`** elements. It describes the service's interface and the messages exchanged by the service. Within the **`types`** element, XML Schema is used to define the structure of the data that makes up the messages. A number of **`message`** elements are used to define the structure of the messages used by the service. The **`portType`** element contains one or more **`operation`** elements that define the messages sent by the operations exposed by the service.

The concrete part

The concrete part of a WSDL document contains the **`binding`** and the **`service`** elements. It describes how an endpoint that implements the service connects to the outside world. The **`binding`** elements describe how the data units described by the **`message`** elements are mapped into a concrete, on-the-wire data format, such as SOAP. The **`service`** elements contain one or more **`port`** elements which define the endpoints implementing the service.

21.2. WSDL ELEMENTS

A WSDL document is made up of the following elements:

- **`definitions`** — The root element of a WSDL document. The attributes of this element specify the name of the WSDL document, the document's target namespace, and the shorthand definitions for the namespaces referenced in the WSDL document.
- **`types`** — The XML Schema definitions for the data units that form the building blocks of the messages used by a service. For information about defining data types see Chapter 22, *Defining Logical Data Units*.
- **`message`** — The description of the messages exchanged during invocation of a services operations. These elements define the arguments of the operations making up your service. For information on defining messages see Chapter 23, *Defining Logical Messages Used by a Service*.
- **`portType`** — A collection of **`operation`** elements describing the logical interface of a service. For information about defining port types see Chapter 24, *Defining Your Logical Interfaces*.
- **`operation`** — The description of an action performed by a service. Operations are defined by the messages passed between two endpoints when the operation is invoked. For information on defining operations see the section called "Operations".
- **`binding`** — The concrete data format specification for an endpoint. A **`binding`** element defines how the abstract messages are mapped into the concrete data format used by an endpoint. This element is where specifics such as parameter order and return values are specified.
- **`service`** — A collection of related **`port`** elements. These elements are repositories for organizing endpoint definitions.
- **`port`** — The endpoint defined by a binding and a physical address. These elements bring all of the abstract definitions together, combined with the definition of transport details, and they define the physical endpoint on which a service is exposed.

21.3. DESIGNING A CONTRACT

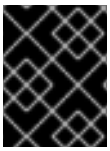
To design a WSDL contract for your services you must perform the following steps:

1. Define the data types used by your services.
2. Define the messages used by your services.
3. Define the interfaces for your services.
4. Define the bindings between the messages used by each interface and the concrete representation of the data on the wire.
5. Define the transport details for each of the services.

CHAPTER 22. DEFINING LOGICAL DATA UNITS

Abstract

When describing a service in a WSDL contract complex data types are defined as logical units using XML Schema.



IMPORTANT

The Java Business Integration components of Red Hat JBoss Fuse are considered deprecated. You should consider migrating any JBI applications to OSGi.

When defining a service, the first thing you must consider is how the data used as parameters for the exposed operations is going to be represented. Unlike applications that are written in a programming language that uses fixed data structures, services must define their data in logical units that can be consumed by any number of applications. This involves two steps:

1. Breaking the data into logical units that can be mapped into the data types used by the physical implementations of the service
2. Combining the logical units into messages that are passed between endpoints to carry out the operations

This chapter discusses the first step. Chapter 23, *Defining Logical Messages Used by a Service* discusses the second step.

22.1. MAPPING DATA INTO LOGICAL DATA UNITS

The interfaces used to implement a service define the data representing operation parameters as XML documents. If you are defining an interface for a service that is already implemented, you must translate the data types of the implemented operations into discreet XML elements that can be assembled into messages. If you are starting from scratch, you must determine the building blocks from which your messages are built, so that they make sense from an implementation standpoint.

Available type systems for defining service data units

According to the WSDL specification, you can use any type system you choose to define data types in a WSDL contract. However, the W3C specification states that XML Schema is the preferred canonical type system for a WSDL document. Therefore, XML Schema is the intrinsic type system in Apache CXF.

XML Schema as a type system

XML Schema is used to define how an XML document is structured. This is done by defining the elements that make up the document. These elements can use native XML Schema types, like xsd:int, or they can use types that are defined by the user. User defined types are either built up using combinations of XML elements or they are defined by restricting existing types. By combining type definitions and element definitions you can create intricate XML documents that can contain complex data.

When used in WSDL XML Schema defines the structure of the XML document that holds the data used to interact with a service. When defining the data units used by your service, you can define them as types that specify the structure of the message parts. You can also define your data units as elements that make up the message parts.

Considerations for creating your data units

You might consider simply creating logical data units that map directly to the types you envision using when implementing the service. While this approach works, and closely follows the model of building RPC-style applications, it is not necessarily ideal for building a piece of a service-oriented architecture.

The Web Services Interoperability Organization's WS-I basic profile provides a number of guidelines for defining data units and can be accessed at http://www.ws-i.org/Profiles/BasicProfile-1.1-2004-08-24.html#WSDLTYPES. In addition, the W3C also provides the following guidelines for using XML Schema to represent data types in WSDL documents:

- Use elements, not attributes.
- Do not use protocol-specific types as base types.

22.2. ADDING DATA UNITS TO A CONTRACT

Depending on how you choose to create your WSDL contract, creating new data definitions requires varying amounts of knowledge. The Apache CXF GUI tools provide a number of aids for describing data types using XML Schema. Other XML editors offer different levels of assistance. Regardless of the editor you choose, it is a good idea to have some knowledge about what the resulting contract should look like.

Procedure

Defining the data used in a WSDL contract involves the following steps:

1. Determine all the data units used in the interface described by the contract.
2. Create a **types** element in your contract.
3. Create a **schema** element, shown in Example 22.1, "Schema entry for a WSDL contract", as a child of the **type** element.

   The **targetNamespace** attribute specifies the namespace under which new data types are defined. The remaining entries should not be changed.

   Example 22.1. Schema entry for a WSDL contract

   ```
   <schema targetNamespace="http://schemas.iona.com/bank.idl"
           xmlns="http://www.w3.org/2001/XMLSchema"
           xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
   ```

4. For each complex type that is a collection of elements, define the data type using a **complexType** element. See Section 22.4.1, "Defining data structures".
5. For each array, define the data type using a **complexType** element. See Section 22.4.2, "Defining arrays".
6. For each complex type that is derived from a simple type, define the data type using a **simpleType** element. See Section 22.4.4, "Defining types by restriction".
7. For each enumerated type, define the data type using a **simpleType** element. See Section 22.4.5, "Defining enumerated types".

8. For each element, define it using an **`element`** element. See Section 22.5, “Defining elements”.

22.3. XML SCHEMA SIMPLE TYPES

If a message part is going to be of a simple type it is not necessary to create a type definition for it. However, the complex types used by the interfaces defined in the contract are defined using simple types.

Entering simple types

XML Schema simple types are mainly placed in the **`element`** elements used in the types section of your contract. They are also used in the **`base`** attribute of **`restriction`** elements and **`extension`** elements.

Simple types are always entered using the **`xsd`** prefix. For example, to specify that an element is of type int, you would enter xsd:int in its **`type`** attribute as shown in Example 22.2, “Defining an element with a simple type”.

Example 22.2. Defining an element with a simple type

```
<element name="simpleInt" type="xsd:int" />
```

Supported XSD simple types

Apache CXF supports the following XML Schema simple types:

- xsd:string
- xsd:normalizedString
- xsd:int
- xsd:unsignedInt
- xsd:long
- xsd:unsignedLong
- xsd:short
- xsd:unsignedShort
- xsd:float
- xsd:double
- xsd:boolean
- xsd:byte
- xsd:unsignedByte
- xsd:integer

- xsd:positiveInteger
- xsd:negativeInteger
- xsd:nonPositiveInteger
- xsd:nonNegativeInteger
- xsd:decimal
- xsd:dateTime
- xsd:time
- xsd:date
- xsd:QName
- xsd:base64Binary
- xsd:hexBinary
- xsd:ID
- xsd:token
- xsd:language
- xsd:Name
- xsd:NCName
- xsd:NMTOKEN
- xsd:anySimpleType
- xsd:anyURI
- xsd:gYear
- xsd:gMonth
- xsd:gDay
- xsd:gYearMonth
- xsd:gMonthDay

22.4. DEFINING COMPLEX DATA TYPES

XML Schema provides a flexible and powerful mechanism for building complex data structures from its simple data types. You can create data structures by creating a sequence of elements and attributes. You can also extend your defined types to create even more complex types.

In addition to building complex data structures, you can also describe specialized types such as enumerated types, data types that have a specific range of values, or data types that need to follow certain patterns by either extending or restricting the primitive types.

22.4.1. Defining data structures

In XML Schema, data units that are a collection of data fields are defined using **complexType** elements. Specifying a complex type requires three pieces of information:

1. The name of the defined type is specified in the **name** attribute of the **complexType** element.
2. The first child element of the **complexType** describes the behavior of the structure's fields when it is put on the wire. See the section called "Complex type varieties".
3. Each of the fields of the defined structure are defined in **element** elements that are grandchildren of the **complexType** element. See the section called "Defining the parts of a structure".

For example, the structure shown in Example 22.3, "Simple Structure" is be defined in XML Schema as a complex type with two elements.

Example 22.3. Simple Structure

```
struct personalInfo
{
  string name;
  int age;
};
```

Example 22.4, "A complex type" shows one possible XML Schema mapping for the structure shown in Example 22.3, "Simple Structure".

Example 22.4. A complex type

```
<complexType name="personalInfo">
  <sequence>
    <element name="name" type="xsd:string" />
    <element name="age" type="xsd:int" />
  </sequence>
</complexType>
```

Complex type varieties

XML Schema has three ways of describing how the fields of a complex type are organized when represented as an XML document and passed on the wire. The first child element of the **complexType** element determines which variety of complex type is being used. Table 22.1, "Complex type descriptor elements" shows the elements used to define complex type behavior.

Table 22.1. Complex type descriptor elements

Element	Complex Type Behavior
sequence	All the complex type's fields must be present and they must be in the exact order they are specified in the type definition.

Element	Complex Type Behavior
all	All of the complex type's fields must be present but they can be in any order.
choice	Only one of the elements in the structure can be placed in the message.

If a **sequence** element, an **all** element, or a **choice** is not specified, then a **sequence** is assumed. For example, the structure defined in Example 22.4, "A complex type" generates a message containing two elements: **name** and **age**.

If the structure is defined using a **choice** element, as shown in Example 22.5, "Simple complex choice type", it generates a message with either a **name** element or an **age** element.

Example 22.5. Simple complex choice type

```
<complexType name="personalInfo">
  <choice>
    <element name="name" type="xsd:string"/>
    <element name="age" type="xsd:int"/>
  </choice>
</complexType>
```

Defining the parts of a structure

You define the data fields that make up a structure using **element** elements. Every **complexType** element should contain at least one **element** element. Each **element** element in the **complexType** element represents a field in the defined data structure.

To fully describe a field in a data structure, **element** elements have two required attributes:

- The **name** attribute specifies the name of the data field and it must be unique within the defined complex type.
- The **type** attribute specifies the type of the data stored in the field. The type can be either one of the XML Schema simple types, or any named complex type that is defined in the contract.

In addition to **name** and **type**, **element** elements have two other commonly used optional attributes: **minOcurrs** and **maxOccurs**. These attributes place bounds on the number of times the field occurs in the structure. By default, each field occurs only once in a complex type. Using these attributes, you can change how many times a field must or can appear in a structure. For example, you can define a field, **previousJobs**, that must occur at least three times, and no more than seven times, as shown in Example 22.6, "Simple complex type with occurrence constraints".

Example 22.6. Simple complex type with occurrence constraints

```
<complexType name="personalInfo>
  <all>
    <element name="name" type="xsd:string"/>
    <element name="age" type="xsd:int"/>
```

```
    <element name="previousJobs" type="xsd:string:
             minOccurs="3" maxOccurs="7"/>
  </all>
</complexType>
```

You can also use the **minOccurs** to make the **age** field optional by setting the **minOccurs** to zero as shown in Example 22.7, “Simple complex type with minOccurs set to zero”. In this case **age** can be omitted and the data will still be valid.

Example 22.7. Simple complex type with minOccurs set to zero

```
<complexType name="personalInfo>
  <choice>
    <element name="name" type="xsd:string"/>
    <element name="age" type="xsd:int" minOccurs="0"/>
  </choice>
</complexType>
```

Defining attributes

In XML documents attributes are contained in the element’s tag. For example, in the **complexType** element **name** is an attribute. They are specified using the **attribute** element. It comes after the **all**, **sequence**, or **choice** element and are a direct child of the **complexType** element. Example 22.8, “Complex type with an attribute” shows a complex type with an attribute.

Example 22.8. Complex type with an attribute

```
<complexType name="personalInfo>
  <all>
    <element name="name" type="xsd:string"/>
    <element name="previousJobs" type="xsd:string"
             minOccurs="3" maxOccurs="7"/>
  </all>
  <attribute name="age" type="xsd:int" use="optional" />
</complexType>
```

The **attribute** element has three attributes:

- **name** — A required attribute that specifies the string identifying the attribute.
- **type** — Specifies the type of the data stored in the field. The type can be one of the XML Schema simple types.
- **use** — Specifies if the attribute is required or optional. Valid values are **required** or **optional**.

If you specify that the attribute is optional you can add the optional attribute **default**. The **default** attribute allows you to specify a default value for the attribute.

22.4.2. Defining arrays

Apache CXF supports two methods for defining arrays in a contract. The first is define a complex type with a single element whose **maxOccurs** attribute has a value greater than one. The second is to use SOAP arrays. SOAP arrays provide added functionality such as the ability to easily define multi-dimensional arrays and to transmit sparsely populated arrays.

Complex type arrays

Complex type arrays are a special case of a sequence complex type. You simply define a complex type with a single element and specify a value for the **maxOccurs** attribute. For example, to define an array of twenty floating point numbers you use a complex type similar to the one shown in Example 22.9, "Complex type array".

Example 22.9. Complex type array

```
<complexType name="personalInfo">
  <element name="averages" type="xsd:float" maxOccurs="20"/>
</complexType>
```

You can also specify a value for the **minOccurs** attribute.

SOAP arrays

SOAP arrays are defined by deriving from the SOAP-ENC:Array base type using the **wsdl:arrayType** element. The syntax for this is shown in Example 22.10, "Syntax for a SOAP array derived using wsdl:arrayType".

Example 22.10. Syntax for a SOAP array derived using wsdl:arrayType

```
<complexType name="TypeName">
  <complexContent>
    <restriction base="SOAP-ENC:Array">
      <attribute ref="SOAP-ENC:arrayType"
                 wsdl:arrayType="ElementType<ArrayBounds>"/>
    </restriction>
  </complexContent>
</complexType>
```

Using this syntax, *TypeName* specifies the name of the newly-defined array type. *ElementType* specifies the type of the elements in the array. *ArrayBounds* specifies the number of dimensions in the array. To specify a single dimension array use **[]**; to specify a two-dimensional array use either **[][]** or **[,]**.

For example, the SOAP Array, SOAPStrings, shown in Example 22.11, "Definition of a SOAP array", defines a one-dimensional array of strings. The **wsdl:arrayType** attribute specifies the type of the array elements, xsd:string, and the number of dimensions, with **[]** implying one dimension.

Example 22.11. Definition of a SOAP array

```
<complexType name="SOAPStrings">
  <complexContent>
```

```
    <restriction base="SOAP-ENC:Array">
      <attribute ref="SOAP-ENC:arrayType"
                 wsdl:arrayType="xsd:string[]"/>
    </restriction>
  </complexContent>
</complexType>
```

You can also describe a SOAP Array using a simple element as described in the SOAP 1.1 specification. The syntax for this is shown in Example 22.12, “Syntax for a SOAP array derived using an element”.

Example 22.12. Syntax for a SOAP array derived using an element

```
<complexType name="TypeName">
  <complexContent>
    <restriction base="SOAP-ENC:Array">
      <sequence>
        <element name="ElementName" type="ElementType"
                 maxOccurs="unbounded"/>
      </sequence>
    </restriction>
  </complexContent>
</complexType>
```

When using this syntax, the element's **maxOccurs** attribute must always be set to **unbounded**.

22.4.3. Defining types by extension

Like most major coding languages, XML Schema allows you to create data types that inherit some of their elements from other data types. This is called defining a type by extension. For example, you could create a new type called **alienInfo**, that extends the **personalInfo** structure defined in Example 22.4, “A complex type” by adding a new element called **planet**.

Types defined by extension have four parts:

1. The name of the type is defined by the **name** attribute of the **complexType** element.
2. The **complexContent** element specifies that the new type will have more than one element.

   NOTE

   If you are only adding new attributes to the complex type, you can use a **simpleContent** element.

3. The type from which the new type is derived, called the *base* type, is specified in the **base** attribute of the **extension** element.
4. The new type’s elements and attributes are defined in the **extension** element, the same as they are for a regular complex type.

For example, **alienInfo** is defined as shown in Example 22.13, “Type defined by extension”.

Example 22.13. Type defined by extension

```
<complexType name="alienInfo">
  <complexContent>
    <extension base="personalInfo">
      <sequence>
        <element name="planet" type="xsd:string"/>
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

22.4.4. Defining types by restriction

XML Schema allows you to create new types by restricting the possible values of an XML Schema simple type. For example, you can define a simple type, **SSN**, which is a string of exactly nine characters. New types defined by restricting simple types are defined using a **simpleType** element.

The definition of a type by restriction requires three things:

1. The name of the new type is specified by the **name** attribute of the **simpleType** element.
2. The simple type from which the new type is derived, called the *base type*, is specified in the **restriction** element. See the section called “Specifying the base type”.
3. The rules, called *facets*, defining the restrictions placed on the base type are defined as children of the **restriction** element. See the section called “Defining the restrictions”.

Specifying the base type

The base type is the type that is being restricted to define the new type. It is specified using a **restriction** element. The **restriction** element is the only child of a **simpleType** element and has one attribute, **base**, that specifies the base type. The base type can be any of the XML Schema simple types.

For example, to define a new type by restricting the values of an xsd:int you use a definition like the one shown in Example 22.14, “Using int as the base type”.

Example 22.14. Using int as the base type

```
<simpleType name="restrictedInt">
  <restriction base="xsd:int">
    ...
  </restriction>
</simpleType>
```

Defining the restrictions

The rules defining the restrictions placed on the base type are called *facets*. Facets are elements with one attribute, **value**, that defines how the facet is enforced. The available facets and their valid **value** settings depend on the base type. For example, xsd:string supports six facets, including:

- **length**
- **minLength**
- **maxLength**
- **pattern**
- **whitespace**
- **enumeration**

Each facet element is a child of the **restriction** element.

Example

Example 22.15, “SSN simple type description” shows an example of a simple type, **SSN**, which represents a social security number. The resulting type is a string of the form **xxx-xx-xxxx**. <SSN>032-43-9876<SSN> is a valid value for an element of this type, but <SSN>032439876</SSN> is not.

Example 22.15. SSN simple type description

```
<simpleType name="SSN">
  <restriction base="xsd:string">
    <pattern value="\d{3}-\d{2}-\d{4}"/>
  </restriction>
</simpleType>
```

22.4.5. Defining enumerated types

Enumerated types in XML Schema are a special case of definition by restriction. They are described by using the **enumeration** facet which is supported by all XML Schema primitive types. As with enumerated types in most modern programming languages, a variable of this type can only have one of the specified values.

Defining an enumeration in XML Schema

The syntax for defining an enumeration is shown in Example 22.16, “Syntax for an enumeration”.

Example 22.16. Syntax for an enumeration

```
<simpleType name="EnumName">
  <restriction base="EnumType">
    <enumeration value="Case1Value"/>
    <enumeration value="Case2Value"/>
    ...
    <enumeration value="CaseNValue"/>
  </restriction>
</simpleType>
```

EnumName specifies the name of the enumeration type. *EnumType* specifies the type of the case values. *CaseNValue*, where *N* is any number one or greater, specifies the value for each specific case of the enumeration. An enumerated type can have any number of case values, but because it is derived from a simple type, only one of the case values is valid at a time.

Example

For example, an XML document with an element defined by the enumeration **`widgetSize`**, shown in Example 22.17, "widgetSize enumeration", would be valid if it contained <widgetSize>big</widgetSize>, but it would not be valid if it contained <widgetSize>big,mungo</widgetSize>.

Example 22.17. widgetSize enumeration

```
<simpleType name="widgetSize">
  <restriction base="xsd:string">
    <enumeration value="big"/>
    <enumeration value="large"/>
    <enumeration value="mungo"/>
  </restriction>
</simpleType>
```

22.5. DEFINING ELEMENTS

Elements in XML Schema represent an instance of an element in an XML document generated from the schema. The most basic element consists of a single **`element`** element. Like the **`element`** element used to define the members of a complex type, they have three attributes:

- **`name`** — A required attribute that specifies the name of the element as it appears in an XML document.
- **`type`** — Specifies the type of the element. The type can be any XML Schema primitive type or any named complex type defined in the contract. This attribute can be omitted if the type has an in-line definition.
- **`nillable`** — Specifies whether an element can be omitted from a document entirely. If **`nillable`** is set to **`true`**, the element can be omitted from any document generated using the schema.

An element can also have an *in-line* type definition. In-line types are specified using either a **`complexType`** element or a **`simpleType`** element. Once you specify if the type of data is complex or simple, you can define any type of data needed using the tools available for each type of data. In-line type definitions are discouraged because they are not reusable.

CHAPTER 23. DEFINING LOGICAL MESSAGES USED BY A SERVICE

Abstract

A service is defined by the messages exchanged when its operations are invoked. In a WSDL contract these messages are defined using **`message`** element. The messages are made up of one or more parts that are defined using **`part`** elements.

IMPORTANT

The Java Business Integration components of Red Hat JBoss Fuse are considered deprecated. You should consider migrating any JBI applications to OSGi.

A service's operations are defined by specifying the logical messages that are exchanged when an operation is invoked. These logical messages define the data that is passed over a network as an XML document. They contain all of the parameters that are a part of a method invocation.

Logical messages are defined using the **`message`** element in your contracts. Each logical message consists of one or more parts, defined in **`part`** elements.

TIP

While your messages can list each parameter as a separate part, the recommended practice is to use only a single part that encapsulates the data needed for the operation.

MESSAGES AND PARAMETER LISTS

Each operation exposed by a service can have only one input message and one output message. The input message defines all of the information the service receives when the operation is invoked. The output message defines all of the data that the service returns when the operation is completed. Fault messages define the data that the service returns when an error occurs.

In addition, each operation can have any number of fault messages. The fault messages define the data that is returned when the service encounters an error. These messages usually have only one part that provides enough information for the consumer to understand the error.

MESSAGE DESIGN FOR INTEGRATING WITH LEGACY SYSTEMS

If you are defining an existing application as a service, you must ensure that each parameter used by the method implementing the operation is represented in a message. You must also ensure that the return value is included in the operation's output message.

One approach to defining your messages is RPC style. When using RPC style, you define the messages using one part for each parameter in the method's parameter list. Each message part is based on a type defined in the **`types`** element of the contract. Your input message contains one part for each input parameter in the method. Your output message contains one part for each output parameter, plus a part to represent the return value, if needed. If a parameter is both an input and an output parameter, it is listed as a part for both the input message and the output message.

RPC style message definition is useful when service enabling legacy systems that use transports such as Tibco or CORBA. These systems are designed around procedures and methods. As such, they are

easiest to model using messages that resemble the parameter lists for the operation being invoked. RPC style also makes a cleaner mapping between the service and the application it is exposing.

MESSAGE DESIGN FOR SOAP SERVICES

While RPC style is useful for modeling existing systems, the service's community strongly favors the wrapped document style. In wrapped document style, each message has a single part. The message's part references a wrapper element defined in the **types** element of the contract. The wrapper element has the following characteristics:

- It is a complex type containing a sequence of elements. For more information see Section 22.4, "Defining complex data types".
- If it is a wrapper for an input message:
  - It has one element for each of the method's input parameters.
  - Its name is the same as the name of the operation with which it is associated.
- If it is a wrapper for an output message:
  - It has one element for each of the method's output parameters and one element for each of the method's inout parameters.
  - Its first element represents the method's return parameter.
  - Its name would be generated by appending **Response** to the name of the operation with which the wrapper is associated.

MESSAGE NAMING

Each message in a contract must have a unique name within its namespace. It is recommended that you use the following naming conventions:

- Messages should only be used by a single operation.
- Input message names are formed by appending **Request** to the name of the operation.
- Output message names are formed by appending **Response** to the name of the operation.
- Fault message names should represent the reason for the fault.

MESSAGE PARTS

Message parts are the formal data units of the logical message. Each part is defined using a **part** element, and is identified by a **name** attribute and either a **type** attribute or an **element** attribute that specifies its data type. The data type attributes are listed in Table 23.1, "Part data type attributes".

Table 23.1. Part data type attributes

Attribute	Description
element="*elem_name*"	The data type of the part is defined by an element called *elem_name*.

Attribute	Description
type="*type_name*"	The data type of the part is defined by a type called *type_name*.

Messages are allowed to reuse part names. For instance, if a method has a parameter, ***foo***, that is passed by reference or is an in/out, it can be a part in both the request message and the response message, as shown in Example 23.1, “Reused part”.

Example 23.1. Reused part

```
<message name="fooRequest">
  <part name="foo" type="xsd:int"/>
<message>
<message name="fooReply">
  <part name="foo" type="xsd:int"/>
<message>
```

EXAMPLE

For example, imagine you had a server that stored personal information and provided a method that returned an employee’s data based on the employee's ID number. The method signature for looking up the data is similar to Example 23.2, “personalInfo lookup method”.

Example 23.2. personalInfo lookup method

```
personalInfo lookup(long empId)
```

This method signature can be mapped to the RPC style WSDL fragment shown in Example 23.3, “RPC WSDL message definitions”.

Example 23.3. RPC WSDL message definitions

```
<message name="personalLookupRequest">
  <part name="empId" type="xsd:int"/>
<message/>
<message name="personalLookupResponse>
  <part name="return" element="xsd1:personalInfo"/>
<message/>
```

It can also be mapped to the wrapped document style WSDL fragment shown in Example 23.4, “Wrapped document WSDL message definitions”.

Example 23.4. Wrapped document WSDL message definitions

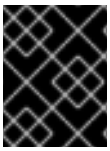
```
<types>
  <schema ... >
```

```
  ...
  <element name="personalLookup">
    <complexType>
      <sequence>
        <element name="empID" type="xsd:int" />
      </sequence>
    </complexType>
  </element>
  <element name="personalLookupResponse">
    <complexType>
      <sequence>
        <element name="return" type="personalInfo" />
      </sequence>
    </complexType>
  </element>
  </schema>
</types>
<message name="personalLookupRequest">
  <part name="empId" element="xsd1:personalLookup"/>
<message/>
<message name="personalLookupResponse">
  <part name="return" element="xsd1:personalLookupResponse"/>
<message/>
```

CHAPTER 24. DEFINING YOUR LOGICAL INTERFACES

Abstract

Logical service interfaces are defined using the **portType** element.



IMPORTANT

The Java Business Integration components of Red Hat JBoss Fuse are considered deprecated. You should consider migrating any JBI applications to OSGi.

Logical service interfaces are defined using the WSDL **portType** element. The **portType** element is a collection of abstract operation definitions. Each operation is defined by the input, output, and fault messages used to complete the transaction the operation represents. When code is generated to implement the service interface defined by a **portType** element, each operation is converted into a method containing the parameters defined by the input, output, and fault messages specified in the contract.

PROCESS

To define a logical interface in a WSDL contract you must do the following:

1. Create a **portType** element to contain the interface definition and give it a unique name. See the section called “Port types”.
2. Create an **operation** element for each operation defined in the interface. See the section called “Operations”.
3. For each operation, specify the messages used to represent the operation’s parameter list, return type, and exceptions. See the section called “Operation messages”.

PORT TYPES

A WSDL **portType** element is the root element in a logical interface definition. While many Web service implementations map **portType** elements directly to generated implementation objects, a logical interface definition does not specify the exact functionality provided by the the implemented service. For example, a logical interface named **ticketSystem** can result in an implementation that either sells concert tickets or issues parking tickets.

The **portType** element is the unit of a WSDL document that is mapped into a binding to define the physical data used by an endpoint exposing the defined service.

Each **portType** element in a WSDL document must have a unique name, which is specified using the **name** attribute, and is made up of a collection of operations, which are described in **operation** elements. A WSDL document can describe any number of port types.

OPERATIONS

Logical operations, defined using WSDL **operation** elements, define the interaction between two endpoints. For example, a request for a checking account balance and an order for a gross of widgets can both be defined as operations.

Each operation defined within a **portType** element must have a unique name, specified using the **name** attribute. The **name** attribute is required to define an operation.

OPERATION MESSAGES

Logical operations are made up of a set of elements representing the logical messages communicated between the endpoints to execute the operation. The elements that can describe an operation are listed in Table 24.1, “Operation message elements”.

Table 24.1. Operation message elements

Element	Description
input	Specifies the message the client endpoint sends to the service provider when a request is made. The parts of this message correspond to the input parameters of the operation.
output	Specifies the message that the service provider sends to the client endpoint in response to a request. The parts of this message correspond to any operation parameters that can be changed by the service provider, such as values passed by reference. This includes the return value of the operation.
fault	Specifies a message used to communicate an error condition between the endpoints.

An operation is required to have at least one **input** or one **output** element. An operation can have both **input** and **output** elements, but it can only have one of each. Operations are not required to have any **fault** elements, but can, if required, have any number of **fault** elements.

The elements have the two attributes listed in Table 24.2, “Attributes of the input and output elements”.

Table 24.2. Attributes of the input and output elements

Attribute	Description
name	Identifies the message so it can be referenced when mapping the operation to a concrete data format. The name must be unique within the enclosing port type.
message	Specifies the abstract message that describes the data being sent or received. The value of the **message** attribute must correspond to the **name** attribute of one of the abstract messages defined in the WSDL document.

It is not necessary to specify the **name** attribute for all **input** and **output** elements; WSDL provides a default naming scheme based on the enclosing operation’s name. If only one element is used in the

operation, the element name defaults to the name of the operation. If both an **input** and an **output** element are used, the element name defaults to the name of the operation with either **Request** or **Response** respectively appended to the name.

RETURN VALUES

Because the **operation** element is an abstract definition of the data passed during an operation, WSDL does not provide for return values to be specified for an operation. If a method returns a value it will be mapped into the **output** element as the last part of that message.

EXAMPLE

For example, you might have an interface similar to the one shown in Example 24.1, “personalInfo lookup interface”.

Example 24.1. personalInfo lookup interface

```
interface personalInfoLookup
{
  personalInfo lookup(in int empID)
  raises(idNotFound);
}
```

This interface can be mapped to the port type in Example 24.2, “personalInfo lookup port type”.

Example 24.2. personalInfo lookup port type

```
<message name="personalLookupRequest">
  <part name="empId" element="xsd1:personalLookup"/>
<message/>
<message name="personalLookupResponse">
  <part name="return" element="xsd1:personalLookupResponse"/>
<message/>
<message name="idNotFoundException">
  <part name="exception" element="xsd1:idNotFound"/>
<message/>
<portType name="personalInfoLookup">
  <operation name="lookup">
    <input name="empID" message="personalLookupRequest"/>
    <output name="return" message="personalLookupResponse"/>
    <fault name="exception" message="idNotFoundException"/>
  </operation>
</portType>
```

CHAPTER 25. USING HTTP

Abstract

HTTP is the underlying transport for the Web. It provides a standardized, robust, and flexible platform for communicating between endpoints. Because of these factors it is the assumed transport for most WS-* specifications and is integral to RESTful architectures.

IMPORTANT

The Java Business Integration components of Red Hat JBoss Fuse are considered deprecated. You should consider migrating any JBI applications to OSGi.

25.1. ADDING A BASIC HTTP ENDPOINT

Overview

There are three ways of specifying an HTTP endpoint's address depending on the payload format you are using.

- SOAP 1.1 uses the standardized **`soap:address`** element.
- SOAP 1.2 uses the **`soap12:address`** element.
- All other payload formats use the **`http:address element.`**

SOAP 1.1

When you are sending SOAP 1.1 messages over HTTP you must use the SOAP 1.1 **`address`** element to specify the endpoint's address. It has one attribute, **`location`**, that specifies the endpoint's address as a URL. The SOAP 1.1 **`address`** element is defined in the namespace http://schemas.xmlsoap.org/wsdl/soap/.

Example 25.1, "SOAP 1.1 Port Element" shows a **`port`** element used to send SOAP 1.1 messages over HTTP.

Example 25.1. SOAP 1.1 Port Element

```
<definitions ...
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" ...>
  ...
  <service name="SOAP11Service">
    <port binding="SOAP11Binding" name="SOAP11Port">
      <soap:address location="http://artie.com/index.xml">
    </port>
  </service>
  ...
<definitions>
```

SOAP 1.2

When you are sending SOAP 1.2 messages over HTTP you must use the SOAP 1.2 **address** element to specify the endpoint's address. It has one attribute, **location**, that specifies the endpoint's address as a URL. The SOAP 1.2 **address** element is defined in the namespace http://schemas.xmlsoap.org/wsdl/soap12/.

Example 25.2, "SOAP 1.2 Port Element" shows a **port** element used to send SOAP 1.2 messages over HTTP.

Example 25.2. SOAP 1.2 Port Element

```
<definitions ...
             xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/" ...
>
  <service name="SOAP12Service">
    <port binding="SOAP12Binding" name="SOAP12Port">
      <soap12:address location="http://artie.com/index.xml">
    </port>
  </service>
  ...
</definitions>
```

Other messages types

When your messages are mapped to any payload format other than SOAP you must use the HTTP **address** element to specify the endpoint's address. It has one attribute, **location**, that specifies the endpoint's address as a URL. The HTTP **address** element is defined in the namespace http://schemas.xmlsoap.org/wsdl/http/.

Example 25.3, "HTTP Port Element" shows a **port** element used to send an XML message.

Example 25.3. HTTP Port Element

```
<definitions ...
             xmlns:http="http://schemas.xmlsoap.org/wsdl/http/" ... >
  <service name="HTTPService">
    <port binding="HTTPBinding" name="HTTPPort">
      <http:address location="http://artie.com/index.xml">
    </port>
  </service>
  ...
</definitions>
```

25.2. CONSUMER CONFIGURATION

Namespace

The WSDL extension elements used to configure an HTTP consumer endpoint are defined in the namespace http://cxf.apache.org/transports/http/configuration. It is commonly referred to using the prefix **http-conf**. In order to use the HTTP configuration elements you must add the line shown in

Example 25.4, “HTTP Consumer WSDL Element's Namespace” to the **definitions** element of your endpoint's WSDL document.

Example 25.4. HTTP Consumer WSDL Element's Namespace

```
<definitions ...
       xmlns:http-
conf="http://cxf.apache.org/transports/http/configuration"
```

Configuring the endpoint

The **http-conf:client** element is used to specify the connection properties of an HTTP consumer in a WSDL document. The **http-conf:client** element is a child of the WSDL **port** element. The attributes are described in Table 25.1, “HTTP Consumer Configuration Attributes”.

Table 25.1. HTTP Consumer Configuration Attributes

Attribute	Description
ConnectionTimeout	Specifies the amount of time, in milliseconds, that the consumer attempts to establish a connection before it times out. The default is **30000**. **0** specifies that the consumer will continue to send the request indefinitely.
ReceiveTimeout	Specifies the amount of time, in milliseconds, that the consumer will wait for a response before it times out. The default is **30000**. **0** specifies that the consumer will wait indefinitely.
AutoRedirect	Specifies if the consumer will automatically follow a server issued redirection. The default is **false**.
MaxRetransmits	Specifies the maximum number of times a consumer will retransmit a request to satisfy a redirect. The default is **-1** which specifies that unlimited retransmissions are allowed.

Attribute	Description
AllowChunking	Specifies whether the consumer will send requests using chunking. The default is **true** which specifies that the consumer will use chunking when sending requests. Chunking cannot be used if either of the following are true: • **http-conf:basicAuthSupplier** is configured to provide credentials preemptively. • **AutoRedirect** is set to **true**. In both cases the value of **AllowChunking** is ignored and chunking is disallowed.
Accept	Specifies what media types the consumer is prepared to handle. The value is used as the value of the HTTP Accept property. The value of the attribute is specified using multipurpose internet mail extensions (MIME) types.
AcceptLanguage	Specifies what language (for example, American English) the consumer prefers for the purpose of receiving a response. The value is used as the value of the HTTP AcceptLanguage property. Language tags are regulated by the International Organization for Standards (ISO) and are typically formed by combining a language code, determined by the ISO-639 standard, and country code, determined by the ISO-3166 standard, separated by a hyphen. For example, en-US represents American English.
AcceptEncoding	Specifies what content encodings the consumer is prepared to handle. Content encoding labels are regulated by the Internet Assigned Numbers Authority (IANA). The value is used as the value of the HTTP AcceptEncoding property.
ContentType	Specifies the media type of the data being sent in the body of a message. Media types are specified using multipurpose internet mail extensions (MIME) types. The value is used as the value of the HTTP ContentType property. The default is **text/xml**. For web services, this should be set to **text/xml**. If the client is sending HTML form data to a CGI script, this should be set to **application/x-www-form-urlencoded**. If the HTTP POST request is bound to a fixed payload format (as opposed to SOAP), the content type is typically set to **application/octet-stream**.

Attribute	Description
`Host`	Specifies the Internet host and port number of the resource on which the request is being invoked. The value is used as the value of the HTTP Host property. This attribute is typically not required. It is only required by certain DNS scenarios or application designs. For example, it indicates what host the client prefers for clusters (that is, for virtual servers mapping to the same Internet protocol (IP) address).
`Connection`	Specifies whether a particular connection is to be kept open or closed after each request/response dialog. There are two valid values: • **`Keep-Alive`** — Specifies that the consumer wants the connection kept open after the initial request/response sequence. If the server honors it, the connection is kept open until the consumer closes it. • **`close`**(default) — Specifies that the connection to the server is closed after each request/response sequence.
`CacheControl`	Specifies directives about the behavior that must be adhered to by caches involved in the chain comprising a request from a consumer to a service provider. See the section called “Consumer Cache Control Directives”.
`Cookie`	Specifies a static cookie to be sent with all requests.
`BrowserType`	Specifies information about the browser from which the request originates. In the HTTP specification from the World Wide Web consortium (W3C) this is also known as the *user-agent*. Some servers optimize based on the client that is sending the request.

Attribute	Description
Referer	Specifies the URL of the resource that directed the consumer to make requests on a particular service. The value is used as the value of the HTTP Referer property. This HTTP property is used when a request is the result of a browser user clicking on a hyperlink rather than typing a URL. This can allow the server to optimize processing based upon previous task flow, and to generate lists of back-links to resources for the purposes of logging, optimized caching, tracing of obsolete or mistyped links, and so on. However, it is typically not used in web services applications. If the **AutoRedirect** attribute is set to **true** and the request is redirected, any value specified in the **Referer** attribute is overridden. The value of the HTTP Referer property is set to the URL of the service that redirected the consumer's original request.
DecoupledEndpoint	Specifies the URL of a decoupled endpoint for the receipt of responses over a separate provider->consumer connection. For more information on using decoupled endpoints see, Section 25.4, "Using the HTTP Transport in Decoupled Mode". You must configure both the consumer endpoint and the service provider endpoint to use WS-Addressing for the decoupled endpoint to work.
ProxyServer	Specifies the URL of the proxy server through which requests are routed.
ProxyServerPort	Specifies the port number of the proxy server through which requests are routed.
ProxyServerType	Specifies the type of proxy server used to route requests. Valid values are: • **HTTP**(default) • **SOCKS**

Consumer Cache Control Directives

Table 25.2, "**http-conf:client** Cache Control Directives" lists the cache control directives supported by an HTTP consumer.

Table 25.2. http-conf:client Cache Control Directives

Directive	Behavior

Directive	Behavior
no-cache	Caches cannot use a particular response to satisfy subsequent requests without first revalidating that response with the server. If specific response header fields are specified with this value, the restriction applies only to those header fields within the response. If no response header fields are specified, the restriction applies to the entire response.
no-store	Caches must not store either any part of a response or any part of the request that invoked it.
max-age	The consumer can accept a response whose age is no greater than the specified time in seconds.
max-stale	The consumer can accept a response that has exceeded its expiration time. If a value is assigned to max-stale, it represents the number of seconds beyond the expiration time of a response up to which the consumer can still accept that response. If no value is assigned, the consumer can accept a stale response of any age.
min-fresh	The consumer wants a response that is still fresh for at least the specified number of seconds indicated.
no-transform	Caches must not modify media type or location of the content in a response between a provider and a consumer.
only-if-cached	Caches should return only responses that are currently stored in the cache, and not responses that need to be reloaded or revalidated.
cache-extension	Specifies additional extensions to the other cache directives. Extensions can be informational or behavioral. An extended directive is specified in the context of a standard directive, so that applications not understanding the extended directive can adhere to the behavior mandated by the standard directive.

Example

Example 25.5, “WSDL to Configure an HTTP Consumer Endpoint” shows a WSDL fragment that configures an HTTP consumer endpoint to specify that it does not interact with caches.

Example 25.5. WSDL to Configure an HTTP Consumer Endpoint

```
<service ... >
  <port ... >
    <soap:address ... />
    <http-conf:client CacheControl="no-cache" />
  </port>
</service>
```

25.3. PROVIDER CONFIGURATION

Namespace

The WSDL extension elements used to configure an HTTP provider endpoint are defined in the namespace http://cxf.apache.org/transports/http/configuration. It is commonly referred to using the prefix **`http-conf`**. To use the HTTP configuration elements you must add the line shown in Example 25.6, "HTTP Provider WSDL Element's Namespace" to the **`definitions`** element of your endpoint's WSDL document.

Example 25.6. HTTP Provider WSDL Element's Namespace

```
<definitions ...
       xmlns:http-
conf="http://cxf.apache.org/transports/http/configuration"
```

Configuring the endpoint

The **`http-conf:server`** element is used to specify the connection properties of an HTTP service provider in a WSDL document. The **`http-conf:server`** element is a child of the WSDL **`port`** element. The attributes are described in Table 25.3, "HTTP Service Provider Configuration Attributes".

Table 25.3. HTTP Service Provider Configuration Attributes

Attribute	Description
`ReceiveTimeout`	Sets the length of time, in milliseconds, the service provider attempts to receive a request before the connection times out. The default is **`30000`**. **`0`** specifies that the provider will not timeout.
`SuppressClientSendErrors`	Specifies whether exceptions are to be thrown when an error is encountered on receiving a request. The default is **`false`**; exceptions are thrown on encountering errors.
`SuppressClientReceiveErrors`	Specifies whether exceptions are to be thrown when an error is encountered on sending a response to a consumer. The default is **`false`**; exceptions are thrown on encountering errors.
`HonorKeepAlive`	Specifies whether the service provider honors requests for a connection to remain open after a response has been sent. The default is **`false`**; keep-alive requests are ignored.

Attribute	Description
`RedirectURL`	Specifies the URL to which the client request should be redirected if the URL specified in the client request is no longer appropriate for the requested resource. In this case, if a status code is not automatically set in the first line of the server response, the status code is set to **`302`** and the status description is set to **`Object Moved`**. The value is used as the value of the HTTP RedirectURL property.
`CacheControl`	Specifies directives about the behavior that must be adhered to by caches involved in the chain comprising a response from a service provider to a consumer. See the section called "Service Provider Cache Control Directives".
`ContentLocation`	Sets the URL where the resource being sent in a response is located.
`ContentType`	Specifies the media type of the information being sent in a response. Media types are specified using multipurpose internet mail extensions (MIME) types. The value is used as the value of the HTTP ContentType location.
`ContentEncoding`	Specifies any additional content encodings that have been applied to the information being sent by the service provider. Content encoding labels are regulated by the Internet Assigned Numbers Authority (IANA). Possible content encoding values include **`zip`**, **`gzip`**, **`compress`**, **`deflate`**, and **`identity`**. This value is used as the value of the HTTP ContentEncoding property. The primary use of content encodings is to allow documents to be compressed using some encoding mechanism, such as zip or gzip. Apache CXF performs no validation on content codings. It is the user's responsibility to ensure that a specified content coding is supported at application level.
`ServerType`	Specifies what type of server is sending the response. Values take the form ***`program-name/version`***; for example, **`Apache/1.2.5`**.

Service Provider Cache Control Directives

Table 25.4, "**`http-conf:server`** Cache Control Directives" lists the cache control directives supported by an HTTP service provider.

Table 25.4. `http-conf:server` Cache Control Directives

Directive	Behavior
no-cache	Caches cannot use a particular response to satisfy subsequent requests without first revalidating that response with the server. If specific response header fields are specified with this value, the restriction applies only to those header fields within the response. If no response header fields are specified, the restriction applies to the entire response.
public	Any cache can store the response.
private	Public (*shared*) caches cannot store the response because the response is intended for a single user. If specific response header fields are specified with this value, the restriction applies only to those header fields within the response. If no response header fields are specified, the restriction applies to the entire response.
no-store	Caches must not store any part of the response or any part of the request that invoked it.
no-transform	Caches must not modify the media type or location of the content in a response between a server and a client.
must-revalidate	Caches must revalidate expired entries that relate to a response before that entry can be used in a subsequent response.
proxy-revalidate	Does the same as must-revalidate, except that it can only be enforced on shared caches and is ignored by private unshared caches. When using this directive, the public cache directive must also be used.
max-age	Clients can accept a response whose age is no greater that the specified number of seconds.
s-max-age	Does the same as max-age, except that it can only be enforced on shared caches and is ignored by private unshared caches. The age specified by s-max-age overrides the age specified by max-age. When using this directive, the proxy-revalidate directive must also be used.
cache-extension	Specifies additional extensions to the other cache directives. Extensions can be informational or behavioral. An extended directive is specified in the context of a standard directive, so that applications not understanding the extended directive can adhere to the behavior mandated by the standard directive.

Example

Example 25.7, “WSDL to Configure an HTTP Service Provider Endpoint” shows a WSDL fragment that configures an HTTP service provider endpoint specifying that it will not interact with caches.

Example 25.7. WSDL to Configure an HTTP Service Provider Endpoint

```
<service ... >
  <port ... >
    <soap:address ... />
    <http-conf:server CacheControl="no-cache" />
  </port>
</service>
```

25.4. USING THE HTTP TRANSPORT IN DECOUPLED MODE

Overview

In normal HTTP request/response scenarios, the request and the response are sent using the same HTTP connection. The service provider processes the request and responds with a response containing the appropriate HTTP status code and the contents of the response. In the case of a successful request, the HTTP status code is set to **200**.

In some instances, such as when using WS-RM or when requests take an extended period of time to execute, it makes sense to decouple the request and response message. In this case the service providers sends the consumer a **202 Accepted** response to the consumer over the back-channel of the HTTP connection on which the request was received. It then processes the request and sends the response back to the consumer using a new decoupled server->client HTTP connection. The consumer runtime receives the incoming response and correlates it with the appropriate request before returning to the application code.

Configuring decoupled interactions

Using the HTTP transport in decoupled mode requires that you do the following:

1. Configure the consumer to use WS-Addressing.

   See the section called “Configuring an endpoint to use WS-Addressing”.

2. Configure the consumer to use a decoupled endpoint.

   See the section called “Configuring the consumer”.

3. Configure any service providers that the consumer interacts with to use WS-Addressing.

   See the section called “Configuring an endpoint to use WS-Addressing”.

Configuring an endpoint to use WS-Addressing

Specify that the consumer and any service provider with which the consumer interacts use WS-Addressing.

You can specify that an endpoint uses WS-Addressing in one of two ways:

- Adding the **wswa:UsingAddressing** element to the endpoint's WSDL **port** element as shown in Example 25.8, “Activating WS-Addressing using WSDL”.

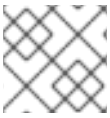
Example 25.8. Activating WS-Addressing using WSDL

```
...
<service name="WidgetSOAPService">
  <port name="WidgetSOAPPort" binding="tns:WidgetSOAPBinding">
    <soap:address="http://widgetvendor.net/widgetSeller" />
    <wswa:UsingAddressing
xmlns:wswa="http://www.w3.org/2005/02/addressing/wsdl"/>
  </port>
</service>
...
```

- Adding the WS-Addressing policy to the endpoint's WSDL **port** element as shown in Example 25.9, “Activating WS-Addressing using a Policy”.

Example 25.9. Activating WS-Addressing using a Policy

```
...
<service name="WidgetSOAPService">
  <port name="WidgetSOAPPort" binding="tns:WidgetSOAPBinding">
    <soap:address="http://widgetvendor.net/widgetSeller" />
    <wsp:Policy xmlns:wsp="http://www.w3.org/2006/07/ws-policy">
      <wsam:Addressing
xmlns:wsam="http://www.w3.org/2007/02/addressing/metadata">
        <wsp:Policy/>
      </wsam:Addressing>
    </wsp:Policy>
  </port>
</service>
...
```



NOTE

The WS-Addressing policy supersedes the **wswa:UsingAddressing** WSDL element.

Configuring the consumer

Configure the consumer endpoint to use a decoupled endpoint using the **DecoupledEndpoint** attribute of the **http-conf:conduit** element.

Example 25.10, “Configuring a Consumer to Use a Decoupled HTTP Endpoint” shows the configuration for setting up the endpoint defined in Example 25.8, “Activating WS-Addressing using WSDL” to use use a decoupled endpoint. The consumer now receives all responses at http://widgetvendor.net/widgetSellerInbox.

Example 25.10. Configuring a Consumer to Use a Decoupled HTTP Endpoint

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

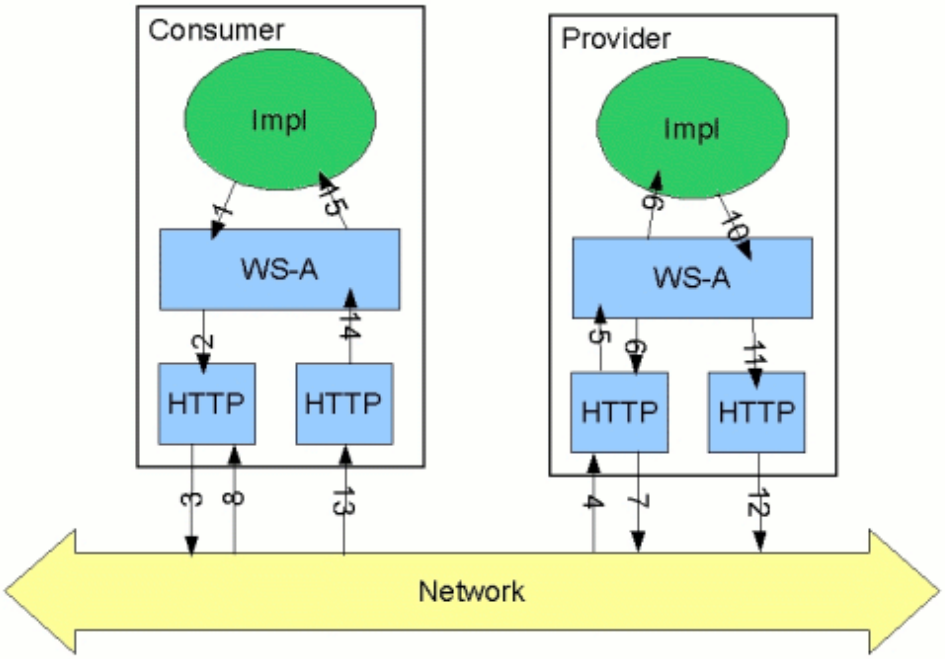
```
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:http="http://cxf.apache.org/transports/http/configuration"

xsi:schemaLocation="http://cxf.apache.org/transports/http/configuration

http://cxf.apache.org/schemas/configuration/http-conf.xsd
                            http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd">

  <http:conduit name="
{http://widgetvendor.net/services}WidgetSOAPPort.http-conduit">
    <http:client
DecoupledEndpoint="http://widgetvendor.net:9999/decoupled_endpoint" />
  </http:conduit>
</beans>
```

How messages are processed

Using the HTTP transport in decoupled mode adds extra layers of complexity to the processing of HTTP messages. While the added complexity is transparent to the implementation level code in an application, it might be important to understand what happens for debugging reasons.

Figure 25.1, “Message Flow in for a Decoupled HTTP Transport” shows the flow of messages when using HTTP in decoupled mode.

Figure 25.1. Message Flow in for a Decoupled HTTP Transport



A request starts the following process:

1. The consumer implementation invokes an operation and a request message is generated.
2. The WS-Addressing layer adds the WS-A headers to the message.

   When a decoupled endpoint is specified in the consumer's configuration, the address of the decoupled endpoint is placed in the WS-A ReplyTo header.
3. The message is sent to the service provider.
4. The service provider receives the message.
5. The request message from the consumer is dispatched to the provider's WS-A layer.
6. Because the WS-A ReplyTo header is not set to anonymous, the provider sends back a message with the HTTP status code set to **`202`**, acknowledging that the request has been received.
7. The HTTP layer sends a **`202 Accepted`** message back to the consumer using the original connection's back-channel.
8. The consumer receives the **`202 Accepted`** reply on the back-channel of the HTTP connection used to send the original message.

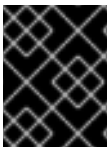
   When the consumer receives the **`202 Accepted`** reply, the HTTP connection closes.

9. The request is passed to the service provider's implementation where the request is processed.
10. When the response is ready, it is dispatched to the WS-A layer.
11. The WS-A layer adds the WS-Addressing headers to the response message.
12. The HTTP transport sends the response to the consumer's decoupled endpoint.
13. The consumer's decoupled endpoint receives the response from the service provider.
14. The response is dispatched to the consumer's WS-A layer where it is correlated to the proper request using the WS-A RelatesTo header.
15. The correlated response is returned to the client implementation and the invoking call is unblocked.

CHAPTER 26. USING JMS

Abstract

HTTP is the underlying transport for the Web. It provides a standardized, robust, and flexible platform for communicating between endpoints. Because of these factors it is the assumed transport for most WS-* specifications and is integral to RESTful architectures.



IMPORTANT

The Java Business Integration components of Red Hat JBoss Fuse are considered deprecated. You should consider migrating any JBI applications to OSGi.

26.1. USING SOAP/JMS

Apache CXF implements the W3C standard SOAP/JMS transport. This standard is intended to provide a more robust alternative to SOAP/HTTP services. Apache CXF applications using this transport should be able to interoperate with applications that also implement the SOAP/JMS standard. The transport is configured directly in an endpoint's WSDL.

26.1.1. Basic configuration

Overview

The SOAP over JMS protocol is defined by the World Wide Web Consortium(W3C) as a way of providing a more reliable transport layer to the customary SOAP/HTTP protocol used by most services. The Apache CXF implementation is fully compliant with the specification and should be compatible with any framework that is also compliant.

This transport uses JNDI to find the JMS destinations. When an operation is invoked, the request is packaged as a SOAP message and sent in the body of a JMS message to the specified destination.

To use the SOAP/JMS transport:

1. Specify that the transport type is SOAP/JMS.
2. Specify the target destination using a JMS URI.
3. Optionally, configure the JNDI connection.
4. Optionally, add additional JMS configuration.

Specifying the JMS transport type

You configure a SOAP binding to use the JMS transport when specifying the WSDL binding. You set the **`soap:binding`** element's **`transport`** attribute to **`http://www.w3.org/2010/soapjms/`**. Example 26.1, “SOAP over JMS binding specification” shows a WSDL binding that uses SOAP/JMS.

Example 26.1. SOAP over JMS binding specification

```
<wsdl:binding ... >
  <soap:binding style="document"
```

```
                    transport="http://www.w3.org/2010/soapjms/" />
  ...
</wsdl:binding>
```

Specifying the target destination

You specify the address of the JMS target destination when specifying the WSDL port for the endpoint. The address specification for a SOAP/JMS endpoint uses the same **`soap:address`** element and attribute as a SOAP/HTTP endpoint. The difference is the address specification. JMS endpoints use a JMS URI as defined in the URI Scheme for JMS 1.0. Example 26.2, “JMS URI syntax” shows the syntax for a JMS URI.

Example 26.2. JMS URI syntax

```
jms:variant:destination?options
```

Table 26.1, “JMS URI variants” describes the available variants for the JMS URI.

Table 26.1. JMS URI variants

Variant	Description
jndi	Specifies that the destination is a JNDI name for the target destination. When using this variant, you must provide the configuration for accessing the JNDI provider.
topic	Specifies that the destination is the name of the topic to be used as the target destination. The string provided is passed into **`Session.createTopic()`** to create a representation of the destination.
queue	Specifies that the destination is the name of the queue to be used as the target destination. The string provided is passed into **`Session.createQueue()`** to create a representation of the destination.

The *options* portion of a JMS URI are used to configure the transport and are discussed in Section 26.1.2, “JMS URIs”.

Example 26.3, “SOAP/JMS endpoint address” shows the WSDL port entry for a SOAP/JMS endpoint whose target destination is looked up using JNDI.

Example 26.3. SOAP/JMS endpoint address

```
<wsdl:port ... >
  ...
  <soap:address
```

```
location="jms:jndi:dynamicQueues/test.cxf.jmstransport.queue" />
</wsdl:port>
```

For working with SOAP/JMS services in Java see ???.

Configuring JNDI and the JMS transport

The SOAP/JMS provides several ways to configure the JNDI connection and the JMS transport:

- Using the JMS URI
- Using WSDL extensions

26.1.2. JMS URIs

Overview

When using SOAP/JMS, a JMS URI is used to specify the endpoint's target destination. The JMS URI can also be used to configure JMS connection by appending one or more options to the URI. These options are detailed in the IETF standard, URI Scheme for Java Message Service 1.0. They can be used to configure the JNDI system, the reply destination, the delivery mode to use, and other JMS properties.

Syntax

As shown in Example 26.2, “JMS URI syntax”, you can append one or more options to the end of a JMS URI by separating them from the destination's address with a question mark(**?**). Multiple options are separated by an ampersand(**&**). Example 26.4, “Syntax for JMS URI options” shows the syntax for using multiple options in a JMS URI.

Example 26.4. Syntax for JMS URI options

```
jmsAddress?option1=value1&option2=value2&...optionN=valueN
```

JMS properties

Table 26.2, “JMS properties settable as URI options” shows the URI options that affect the JMS transport layer.

Table 26.2. JMS properties settable as URI options

Property	Default	Description

Property	Default	Description
`deliveryMode`	`PERSISTENT`	Specifies whether to use JMS `PERSISTENT` or `NON_PERSISTENT` message semantics. In the case of `PERSISTENT` delivery mode, the JMS broker stores messages in persistent storage before acknowledging them; whereas `NON_PERSISTENT` messages are kept in memory only.
`replyToName`		Explicitly specifies the reply destination to appear in the `JMSReplyTo` header. Setting this property is recommended for applications that have request-reply semantics because the JMS provider will assign a temporary reply queue if one is not explicitly set. The value of this property has an interpretation that depends on the variant specified in the JMS URI: • `jndi` variant—the JNDI name of the destination • `queue` or `topic` variants—the actual name of the destination
`priority`	`4`	Specifies the JMS message priority, which ranges from 0 (lowest) to 9 (highest).
`timeToLive`	`0`	Time (in milliseconds) after which the message will be discarded by the JMS provider. A value of `0` represents an infinite lifetime (the default).

JNDI properties

Table 26.3, “JNDI properties settable as URI options” shows the URI options that can be used to configure JNDI for this endpoint.

Table 26.3. JNDI properties settable as URI options

Property	Description

Property	Description
jndiConnectionFactoryName	Specifies the JNDI name of the JMS connection factory.
jndiInitialContextFactory	Specifies the fully qualified Java class name of the JNDI provider (which must be of **javax.jms.InitialContextFactory** type). Equivalent to setting the **java.naming.factory.initial** Java system property.
jndiURL	Specifies the URL that initializes the JNDI provider. Equivalent to setting the **java.naming.provider.url** Java system property.

Additional JNDI properties

The properties, **java.naming.factory.initial** and **java.naming.provider.url**, are standard properties, which are required to initialize any JNDI provider. Sometimes, however, a JNDI provider might support custom properties in addition to the standard ones. In this case, you can set an arbitrary JNDI property by setting a URI option of the form **jndi-*PropertyName***.

For example, if you were using SUN's LDAP implementation of JNDI, you could set the JNDI property, **java.naming.factory.control**, in a JMS URI as shown in Example 26.5, “Setting a JNDI property in a JMS URI”.

Example 26.5. Setting a JNDI property in a JMS URI

```
jms:queue:FOO.BAR?jndi-
java.naming.factory.control=com.sun.jndi.ldap.ResponseControlFactory
```

Example

If the JMS provider is *not* already configured, it is possible to provide the requisite JNDI configuration details in the URI using options (see Table 26.3, “JNDI properties settable as URI options”). For example, to configure an endpoint to use the Apache ActiveMQ JMS provider and connect to the queue called **test.cxf.jmstransport.queue**, use the URI shown in Example 26.6, “JMS URI that configures a JNDI connection”.

Example 26.6. JMS URI that configures a JNDI connection

```
jms:jndi:dynamicQueues/test.cxf.jmstransport.queue
?
jndiInitialContextFactory=org.apache.activemq.jndi.ActiveMQInitialContex
tFactory
&jndiConnectionFactoryName=ConnectionFactory
&jndiURL=tcp://localhost:61616
```

26.1.3. WSDL extensions

Overview

You can specify the basic configuration of the JMS transport by inserting WSDL extension elements into the contract, either at binding scope, service scope, or port scope. The WSDL extensions enable you to specify the properties for bootstrapping a JNDI **`InitialContext`**, which can then be used to look up JMS destinations. You can also set some properties that affect the behavior of the JMS transport layer.

SOAP/JMS namespace

the SOAP/JMS WSDL extensions are defined in the **`http://www.w3.org/2010/soapjms/`** namespace. To use them in your WSDL contracts add the following setting to the **`wsdl:definitions`** element:

```
<wsdl:definitions ...
    xmlns:soapjms="http://www.w3.org/2010/soapjms/"
  ... >
```

WSDL extension elements

Table 26.4, “SOAP/JMS WSDL extension elements” shows all of the WSDL extension elements you can use to configure the JMS transport.

Table 26.4. SOAP/JMS WSDL extension elements

Element	Default	Description
`soapjms:jndiInitialContextFactory`		Specifies the fully qualified Java class name of the JNDI provider. Equivalent to setting the **`java.naming.factory.initial`** Java system property.
`soapjms:jndiURL`		Specifies the URL that initializes the JNDI provider. Equivalent to setting the **`java.naming.provider.url`** Java system property.
`soapjms:jndiContextParameter`		Enables you to specify an additional property for creating the JNDI **`InitialContext`**. Use the **`name`** and **`value`** attributes to specify the property.
`soapjms:jndiConnectionFactoryName`		Specifies the JNDI name of the JMS connection factory.

Element	Default	Description
soapjms:deliveryMode	**PERSISTENT**	Specifies whether to use JMS **PERSISTENT** or **NON_PERSISTENT** message semantics. In the case of **PERSISTENT** delivery mode, the JMS broker stores messages in persistent storage before acknowledging them; whereas **NON_PERSISTENT** messages are kept in memory only.
soapjms:replyToName		Explicitly specifies the reply destination to appear in the **JMSReplyTo** header. Setting this property is recommended for SOAP invocations that have request-reply semantics. If this property is not set the JMS provider allocates a temporary queue with an automatically generated name. The value of this property has an interpretation that depends on the variant specified in the JMS URI, as follows: • **jndi** variant—the JNDI name of the destination. • **queue** or **topic** variants—the actual name of the destination.
soapjms:priority	**4**	Specifies the JMS message priority, which ranges from 0 (lowest) to 9 (highest).
soapjms:timeToLive	**0**	Time, in milliseconds, after which the message will be discarded by the JMS provider. A value of **0** represents an infinite lifetime.

Configuration scopes

The WSDL elements placement in the WSDL contract effect the scope of the configuration changes on the endpoints defined in the contract. The SOAP/JMS WSDL elements can be placed as children of either the **wsdl:binding** element, the **wsdl:service** element, or the **wsdl:port** element. The parent of the SOAP/JMS elements determine which of the following scopes the configuration is placed into.

Binding scope

You can configure the JMS transport at the *binding scope* by placing extension elements inside the

wsdl:binding element. Elements in this scope define the default configuration for all endpoints that use this binding. Any settings in the binding scope can be overridden at the service scope or the port scope.

Service scope

You can configure the JMS transport at the *service scope* by placing extension elements inside a **wsdl:service** element. Elements in this scope define the default configuration for all endpoints in this service. Any settings in the service scope can be overridden at the port scope.

Port scope

You can configure the JMS transport at the *port scope* by placing extension elements inside a **wsdl:port** element. Elements in the port scope define the configuration for this port. They override any defaults defined at the service scope or at the binding scope.

Example

Example 26.7, "WSDL contract with SOAP/JMS configuration" shows a WSDL contract for a SOAP/JMS service. It configures the JNDI layer in the binding scope, the message delivery details in the service scope, and the reply destination in the port scope.

Example 26.7. WSDL contract with SOAP/JMS configuration

```
<wsd;definitions ...
1     xmlns:soapjms="http://www.w3.org/2010/soapjms/"
    ... >
  ...
  <wsdl:binding name="JMSGreeterPortBinding"
type="tns:JMSGreeterPortType">
    ...
2     <soapjms:jndiInitialContextFactory>
        org.apache.activemq.jndi.ActiveMQInitialContextFactory
    </soapjms:jndiInitialContextFactory>
    <soapjms:jndiURL>tcp://localhost:61616</soapjms:jndiURL>
    <soapjms:jndiConnectionFactoryName>
      ConnectionFactory
    </soapjms:jndiConnectionFactoryName>
    ...
  </wsdl:binding>
  ...
  <wsdl:service name="JMSGreeterService">
    ...
3     <soapjms:deliveryMode>NON_PERSISTENT</soapjms:deliveryMode>
      <soapjms:timeToLive>60000</soapjms:timeToLive>
    ...
    <wsdl:port binding="tns:JMSGreeterPortBinding" name="GreeterPort">
4       <soap:address
 location="jms:jndi:dynamicQueues/test.cxf.jmstransport.queue" />
5       <soapjms:replyToName>
          dynamicQueues/greeterReply.queue
      </soapjms:replyToName>
      ...
    </wsdl:port>
    ...
```

```
    </wsdl:service>
    ...
</wsdl:definitions>
```

The WSDL in Example 26.7, “WSDL contract with SOAP/JMS configuration” does the following:

1. Declare the namespace for the SOAP/JMS extensions.
2. Configure the JNDI connections in the binding scope.
3. Configure the JMS delivery style to non-persistent and each message to live for one minute.
4. Specify the target destination.
5. Configure the JMS transport so that reply messages are delivered on the **greeterReply.queue** queue.

26.2. USING WSDL TO CONFIGURE JMS

The WSDL extensions for defining a JMS endpoint are defined in the namespace http://cxf.apache.org/transports/jms. In order to use the JMS extensions you will need to add the line shown in Example 26.8, “JMS WSDL extension namespace” to the definitions element of your contract.

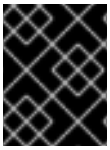
Example 26.8. JMS WSDL extension namespace

```
xmlns:jms="http://cxf.apache.org/transports/jms"
```

26.2.1. Basic JMS configuration

Overview

The JMS address information is provided using the **jms:address** element and its child, the **jms:JMSNamingProperties** element. The **jms:address** element’s attributes specify the information needed to identify the JMS broker and the destination. The **jms:JMSNamingProperties** element specifies the Java properties used to connect to the JNDI service.



IMPORTANT

Information specified using the JMS feature will override the information in the endpoint's WSDL file.

Specifying the JMS address

The basic configuration for a JMS endpoint is done by using a **jms:address** element as the child of your service’s **port** element. The **jms:address** element used in WSDL is identical to the one used in the configuration file. Its attributes are listed in Table 26.5, “JMS endpoint attributes”.

Table 26.5. JMS endpoint attributes

Attribute	Description
destinationStyle	Specifies if the JMS destination is a JMS queue or a JMS topic.
jndiConnectionFactoryName	Specifies the JNDI name bound to the JMS connection factory to use when connecting to the JMS destination.
jmsDestinationName	Specifies the JMS name of the JMS destination to which requests are sent.
jmsReplyDestinationName	Specifies the JMS name of the JMS destinations where replies are sent. This attribute allows you to use a user defined destination for replies. For more details see Section 26.3, “Using a Named Reply Destination”.
jndiDestinationName	Specifies the JNDI name bound to the JMS destination to which requests are sent.
jndiReplyDestinationName	Specifies the JNDI name bound to the JMS destinations where replies are sent. This attribute allows you to use a user defined destination for replies. For more details see Section 26.3, “Using a Named Reply Destination”.
connectionUserName	Specifies the user name to use when connecting to a JMS broker.
connectionPassword	Specifies the password to use when connecting to a JMS broker.

The **jms:address** WSDL element uses a **jms:JMSNamingProperties** child element to specify additional information needed to connect to a JNDI provider.

Specifying JNDI properties

To increase interoperability with JMS and JNDI providers, the **jms:address** element has a child element, **jms:JMSNamingProperties**, that allows you to specify the values used to populate the properties used when connecting to the JNDI provider. The **jms:JMSNamingProperties** element has two attributes: **name** and **value**. **name** specifies the name of the property to set. **value** attribute specifies the value for the specified property. **jms:JMSNamingProperties** element can also be used for specification of provider specific properties.

The following is a list of common JNDI properties that can be set:

1. **java.naming.factory.initial**
2. **java.naming.provider.url**

3. **java.naming.factory.object**
4. **java.naming.factory.state**
5. **java.naming.factory.url.pkgs**
6. **java.naming.dns.url**
7. **java.naming.authoritative**
8. **java.naming.batchsize**
9. **java.naming.referral**
10. **java.naming.security.protocol**
11. **java.naming.security.authentication**
12. **java.naming.security.principal**
13. **java.naming.security.credentials**
14. **java.naming.language**
15. **java.naming.applet**

For more details on what information to use in these attributes, check your JNDI provider's documentation and consult the Java API reference material.

Example

Example 26.9, "JMS WSDL port specification" shows an example of a JMS WSDL **port** specification.

Example 26.9. JMS WSDL port specification

```
<service name="JMSService">
  <port binding="tns:Greeter_SOAPBinding" name="SoapPort">
    <jms:address jndiConnectionFactoryName="ConnectionFactory"

jndiDestinationName="dynamicQueues/test.Celtix.jmstransport" >
      <jms:JMSNamingProperty name="java.naming.factory.initial"

value="org.activemq.jndi.ActiveMQInitialContextFactory" />
      <jms:JMSNamingProperty name="java.naming.provider.url"
                             value="tcp://localhost:61616" />
    </jms:address>
  </port>
</service>
```

26.2.2. JMS client configuration

Overview

JMS consumer endpoints specify the type of messages they use. JMS consumer endpoint can use either a JMS **ByteMessage** or a JMS **TextMessage**.

When using an **ByteMessage** the consumer endpoint uses a byte[] as the method for storing data into and retrieving data from the JMS message body. When messages are sent, the message data, including any formating information, is packaged into a byte[] and placed into the message body before it is placed on the wire. When messages are received, the consumer endpoint will attempt to unmarshall the data stored in the message body as if it were packed in a byte[].

When using a **TextMessage**, the consumer endpoint uses a string as the method for storing and retrieving data from the message body. When messages are sent, the message information, including any format-specific information, is converted into a string and placed into the JMS message body. When messages are received the consumer endpoint will attempt to unmarshall the data stored in the JMS message body as if it were packed into a string.

When native JMS applications interact with Apache CXF consumers, the JMS application is responsible for interpreting the message and the formatting information. For example, if the Apache CXF contract specifies that the binding used for a JMS endpoint is SOAP, and the messages are packaged as **TextMessage**, the receiving JMS application will get a text message containing all of the SOAP envelope information.

Specifying the message type

The type of messages accepted by a JMS consumer endpoint is configured using the optional **jms:client** element. The **jms:client** element is a child of the WSDL **port** element and has one attribute:

Table 26.6. JMS Client WSDL Extensions

messageType	Specifies how the message data will be packaged as a JMS message. **text** specifies that the data will be packaged as a **TextMessage**. **binary** specifies that the data will be packaged as an **ByteMessage**.

Example

Example 26.10, “WSDL for a JMS consumer endpoint” shows the WSDL for configuring a JMS consumer endpoint.

Example 26.10. WSDL for a JMS consumer endpoint

```
<service name="JMSService">
  <port binding="tns:Greeter_SOAPBinding" name="SoapPort">
    <jms:address jndiConnectionFactoryName="ConnectionFactory"

jndiDestinationName="dynamicQueues/test.Celtix.jmstransport" >
      <jms:JMSNamingProperty name="java.naming.factory.initial"

value="org.activemq.jndi.ActiveMQInitialContextFactory" />
      <jms:JMSNamingProperty name="java.naming.provider.url"
                             value="tcp://localhost:61616" />
    </jms:address>
```

```
      <jms:client messageType="binary" />
    </port>
  </service>
```

26.2.3. JMS provider configuration

Overview

JMS provider endpoints have a number of behaviors that are configurable. These include:

- how messages are correlated
- the use of durable subscriptions
- if the service uses local JMS transactions
- the message selectors used by the endpoint

Specifying the configuration

Provider endpoint behaviors are configured using the optional **`jms:server`** element. The **`jms:server`** element is a child of the WSDL **`wsdl:port`** element and has the following attributes:

Table 26.7. JMS provider endpoint WSDL extensions

Attribute	Description
`useMessageIDAsCorrealationID`	Specifies whether JMS will use the message ID to correlate messages. The default is **`false`**.
`durableSubscriberName`	Specifies the name used to register a durable subscription.
`messageSelector`	Specifies the string value of a message selector to use. For more information on the syntax used to specify message selectors, see the JMS 1.1 specification.
`transactional`	Specifies whether the local JMS broker will create transactions around message processing. The default is **`false`**. [a]

[a] Currently, setting the **`transactional`** attribute to **`true`** is not supported by the runtime.

Example

Example 26.11, “WSDL for a JMS provider endpoint” shows the WSDL for configuring a JMS provider endpoint.

Example 26.11. WSDL for a JMS provider endpoint

```
<service name="JMSService">
  <port binding="tns:Greeter_SOAPBinding" name="SoapPort">
    <jms:address jndiConnectionFactoryName="ConnectionFactory"

jndiDestinationName="dynamicQueues/test.Celtix.jmstransport" >
      <jms:JMSNamingProperty name="java.naming.factory.initial"

value="org.activemq.jndi.ActiveMQInitialContextFactory" />
      <jms:JMSNamingProperty name="java.naming.provider.url"
                             value="tcp://localhost:61616" />
    </jms:address>
    <jms:server messageSelector="cxf_message_selector"
                useMessageIDAsCorrelationID="true"
                transactional="true"
                durableSubscriberName="cxf_subscriber" />
  </port>
</service>
```

26.3. USING A NAMED REPLY DESTINATION

Overview

By default, Apache CXF endpoints using JMS create a temporary queue for sending replies back and forth. If you prefer to use named queues, you can configure the queue used to send replies as part of an endpoint's JMS configuration.

Setting the reply destination name

You specify the reply destination using either the **`jmsReplyDestinationName`** attribute or the **`jndiReplyDestinationName`** attribute in the endpoint's JMS configuration. A client endpoint will listen for replies on the specified destination and it will specify the value of the attribute in the **`ReplyTo`** field of all outgoing requests. A service endpoint will use the value of the **`jndiReplyDestinationName`** attribute as the location for placing replies if there is no destination specified in the request's **`ReplyTo`** field.

Example

Example 26.12, "JMS Consumer Specification Using a Named Reply Queue" shows the configuration for a JMS client endpoint.

Example 26.12. JMS Consumer Specification Using a Named Reply Queue

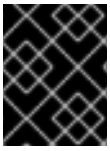
```
<jms:conduit name="
{http://cxf.apache.org/jms_endpt}HelloWorldJMSPort.jms-conduit">
    <jms:address destinationStyle="queue"
                 jndiConnectionFactoryName="myConnectionFactory"
                 jndiDestinationName="myDestination"
                 jndiReplyDestinationName="myReplyDestination" >
      <jms:JMSNamingProperty name="java.naming.factory.initial"
```

```
value="org.apache.cxf.transport.jms.MyInitialContextFactory" />
      <jms:JMSNamingProperty name="java.naming.provider.url"
                             value="tcp://localhost:61616" />
    </jms:address>
  </jms:conduit>
```

CHAPTER 27. INTRODUCTION TO THE APACHE CXF BINDING COMPONENT

Abstract

Endpoints being deployed using the Apache CXF binding component are packaged into a service unit. The service unit will container the WSDL document defining the endpoint's interface and a configuration file that sets-up the endpoint's runtime behavior.



IMPORTANT

The Java Business Integration components of Red Hat JBoss Fuse are considered deprecated. You should consider migrating any JBI applications to OSGi.

CONTENTS OF A FILE COMPONENT SERVICE UNIT

A service unit that configures the Apache CXF binding component will contain the following 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.

WSDL file

The WSDL file defines the endpoint the interface exposes.

Spring configuration file

The Spring configuration file contains configuration for the Apache CXF runtime.

`meta-inf/jbi.xml`

The **`jbi.xml`** file is the JBI descriptor for the service unit. Example 27.1, “JBI Descriptor for a Apache CXF Binding Component Service Unit” shows a JBI descriptor for a Apache CXF binding component service unit.

Example 27.1. JBI Descriptor for a Apache CXF Binding Component Service Unit

```
<jbi xmlns="http://java.sun.com/xml/ns/jbi" version="1.0">
  <services binding-component="false" />
</jbi>
```

For information on using the Maven tooling to package endpoints into a JBI service unit see ???.

OSGI PACKAGING

You can package Apache CXF binding component endpoints in an OSGi bundle. To do so you need to make two minor changes:

- you will need to include an OSGi bundle manifest in the **META-INF** folder of the bundle.
- You need to add the following to your service unit's configuration file:

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

IMPORTANT

When you deploy Apache CXF binding component 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 H, *Using the Maven OSGi Tooling*.

NAMESPACE

The elements used to configure Apache CXF binding component endpoints are defined in the **http://servicemix.apache.org/cxfbc/1.0** namespace. You will need to add a namespace declaration similar to the one in Example 27.2, "Namespace Declaration for Using Apache CXF Binding Component Endpoints" to your **xbeans.xml** file's **beans** element.

Example 27.2. Namespace Declaration for Using Apache CXF Binding Component Endpoints

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

In addition, you need to add the schema location to the Spring **beans** element's **xsi:schemaLocation** as shown in Example 27.3, "Schema Location for Using Apache CXF Binding Component Endpoints".

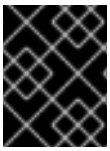
Example 27.3. Schema Location for Using Apache CXF Binding Component Endpoints

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

CHAPTER 28. CONSUMER ENDPOINTS

Abstract

A consumer endpoint listens for requests from external endpoints and delivers responses back to the requesting endpoint. It is configured using a single XML element that specifies the WSDL document defining the endpoint.



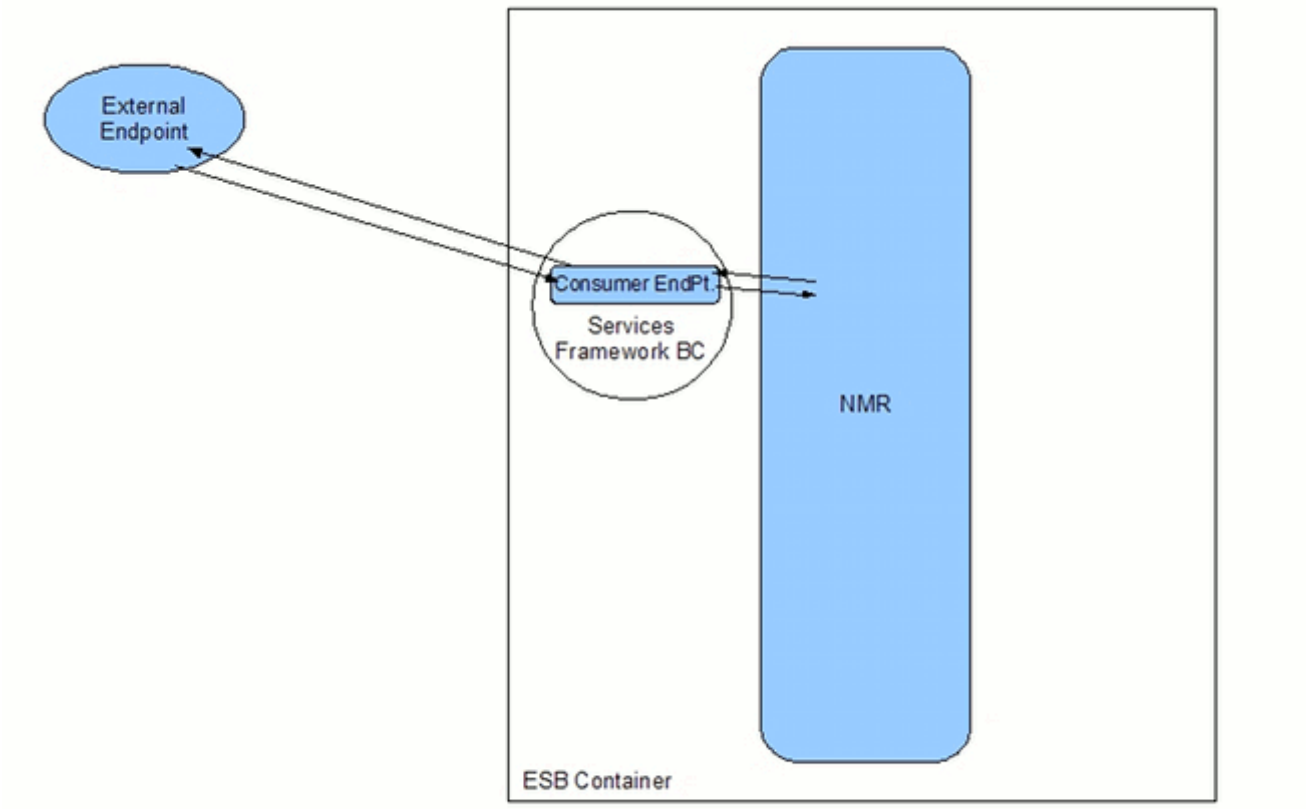
IMPORTANT

The Java Business Integration components of Red Hat JBoss Fuse are considered deprecated. You should consider migrating any JBI applications to OSGi.

OVERVIEW

Consumer endpoints play the role of consumer from the vantage point of other endpoints running inside of the ESB. However, from outside of the ESB a consumer endpoint plays the role of a service provider. As shown in Figure 28.1, “Consumer Endpoint”, consumer endpoints listen from incoming requests from external endpoints. When it receives a request, the consumer passes it off to the NMR fro delivery to endpoint that will process the request. If a response is generated, the consumer endpoint delivers the response back to the external endpoint.

Figure 28.1. Consumer Endpoint



IMPORTANT

Because consumer endpoint's behave like service providers to external endpoints, you configure the runtime behavior of the transport using the provider-specific WSDL entries.

PROCEDURE

To configure a consumer endpoint do the following:

1. Add a **`consumer`** element to your **`xbean.xml`** file.
2. Add a **`wsdl`** attribute to the **`consumer`** element.

   See the section called “Specifying the WSDL”.
3. If your WSDL defines more than one service, you will need to specify a value for the **`service`** attribute.

   See the section called “Specifying the endpoint details”.
4. If the service you choose defines more than one endpoint, you will need to specify a value for the **`endpoint`** attribute.

   See the section called “Specifying the endpoint details”.
5. Specify the details for the target of the requests received by the endpoint.

   See the section called “Specifying the target endpoint”.
6. If your endpoint is going to be receiving binary attachments set its **`mtomEnabled`** attribute to **`true`**.

   See Chapter 30, *Using MTOM to Process Binary Content*.
7. If your endpoint does not need to process the JBI wrapper set its **`useJbiWrapper`** attribute to **`false`**.

   See Chapter 31, *Working with the JBI Wrapper*.
8. If you are using any of the advanced features, such as WS-Addressing or WS-Policy, specify a value for the **`busCfg`** attribute.

   See ???.

SPECIFYING THE WSDL

The **`wsdl`** attribute is the only required attribute to configure a consumer endpoint. It specifies the location of the WSDL document that defines the endpoint being exposed. The path used is relative to the top-level of the exploded service unit.

TIP

If the WSDL document defines a single service with a single endpoint, then you do not require any additional information to expose a consumer endpoint.

Example 28.1, “Minimal Consumer Endpoint Configuration” shows the minimal configuration for a consumer endpoint.

Example 28.1. Minimal Consumer Endpoint Configuration

```
<beans xmlns:cxfbc="http://servicemix.apache.org/cxfbc/1.0"
       ... >
  ...
  <cxfbc:consumer wsdl="/wsdl/widget.wsdl" />

  ...
</beans>
```

For information on creating a WSDL document see ???.

SPECIFYING THE ENDPOINT DETAILS

If the endpoint's WSDL document defines a single service with a single endpoint, the ESB can easily determine which endpoint to use. It will use the values from the WSDL document to specify the service name, endpoint name and interface name for the instantiated endpoint.

However, if the endpoint's WSDL document defines multiple services or if it defines multiple endpoints for a service, you will need to provide the consumer endpoint with additional information so that it can determine the proper definition to use. What information you need to provide depends on the complexity of the WSDL document. You may need to supply values for both the service name and the endpoint name, or you may only have to supply one of these values.

If the WSDL document contains more than one **`service`** element you will need to specify a value for the consumer's **`service`** attribute. The value of the consumer's **`service`** attribute is the QName of the WSDL **`service`** element that defines the desired service in the WSDL document. For example, if you wanted your endpoint to use the WidgetSalesService in the WSDL shown in Example 28.2, “WSDL with Two Services” you would use the configuration shown in Example 28.3, “Consumer Endpoint with a Defined Service Name”.

Example 28.2. WSDL with Two Services

```
<definitions ...
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             targetNamespace="http://demos.widgetVendor.com" ...>
  ...
  <service name="WidgetSalesService">
    <port binding="WidgetSalesBinding" name="WidgetSalesPort">
      <soap:address location="http://widget.sales.com/index.xml">
    </port>
  </service>

  <service name="WidgetInventoryService">
    <port binding="WidgetInventoryBinding" name="WidgetInventoryPort">
      <soap:address location="http://widget.inventory.com/index.xml">
    </port>
  </service>
  ...
<definitions>
```

Example 28.3. Consumer Endpoint with a Defined Service Name

```
<beans xmlns:cxfbc="http://servicemix.apache.org/cxfbc/1.0"
       xmlns:widgets="http://demos.widgetVendor.com"
       ... >
  ...
  <cxfbc:consumer wsdl="/wsdl/widget.wsdl"
                  service="widgets:WidgetSalesService" />

  ...
</beans>
```

If the WSDL document's service definition contains more than one endpoint, then you will need to provide a value for the consumer's **endpoint** attribute. The value of the **endpoint** attribute corresponds to the value of the WSDL **port** element's **name** attribute. For example, if you wanted your endpoint to use the WidgetEasternSalesPort in the WSDL shown in Example 28.4, “Service with Two Endpoints” you would use the configuration shown in Example 28.5, “Consumer Endpoint with a Defined Endpoint Name”.

Example 28.4. Service with Two Endpoints

```
<definitions ...
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             targetNamespace="http://demos.widgetVendor.com" ...>
  ...
  <service name="WidgetSalesService">
    <port binding="WidgetSalesBinding" name="WidgetWesternSalesPort">
      <soap:address location="http://widget.sales.com/index.xml">
    </port>
    <port binding="WidgetSalesBinding" name="WidgetEasternSalesPort">
      <jms:address jndiConnectionFactoryName="ConnectionFactory"

jndiDestinationName="dynamicQueues/test.Celtix.jmstransport" >
        <jms:JMSNamingProperty name="java.naming.factory.initial"

value="org.activemq.jndi.ActiveMQInitialContextFactory" />
        <jms:JMSNamingProperty name="java.naming.provider.url"
                               value="tcp://localhost:61616" />
      </jms:address>
    </port>
  </service>
  ...
<definitions>
```

Example 28.5. Consumer Endpoint with a Defined Endpoint Name

```
<beans xmlns:cxfbc="http://servicemix.apache.org/cxfbc/1.0"
       xmlns:widgets="http://demos.widgetVendor.com"
       ... >
  ...
  <cxfbc:consumer wsdl="/wsdl/widget.wsdl"
```

```
                    endpoint="WidgetEasternSalesService" />
  ...
</beans>
```

SPECIFYING THE TARGET ENDPOINT

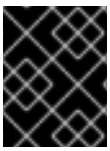
The consumer endpoint will determine the target endpoint in the following manner:

1. If you explicitly specify an endpoint using both the **`targetService`** attribute and the **`targetEndpoint`** attribute, the ESB will use that endpoint.
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 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.
4. If you do not use any of the target attributes, the ESB will use the values used in configuring the endpoint's service name and endpoint name to determine the target endpoint.

Example 28.6, “Consumer Endpoint Configuration Specifying a Target Endpoint” shows the configuration for a consumer endpoint that specifies the target endpoint to use.

Example 28.6. Consumer Endpoint Configuration Specifying a Target Endpoint

```
<beans xmlns:cxfbc="http://servicemix.apache.org/cxfbc/1.0"
       xmlns:widgets="http://demos.widgetVendor.com"
       ... >
  ...
  <cxfbc:consumer wsdl="/wsdl/widget.wsdl"
                  targetEndpoint="WidgetSalesTargetPort"
                  targetService="widgets:WidgetSalesTargetService" />

  ...
</beans>
```



IMPORTANT

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

CHAPTER 29. PROVIDER ENDPOINTS

Abstract

A provider endpoint sends requests to external endpoints and waits for the response. It is configured using a single XML element that specifies the WSDL document defining the endpoint.

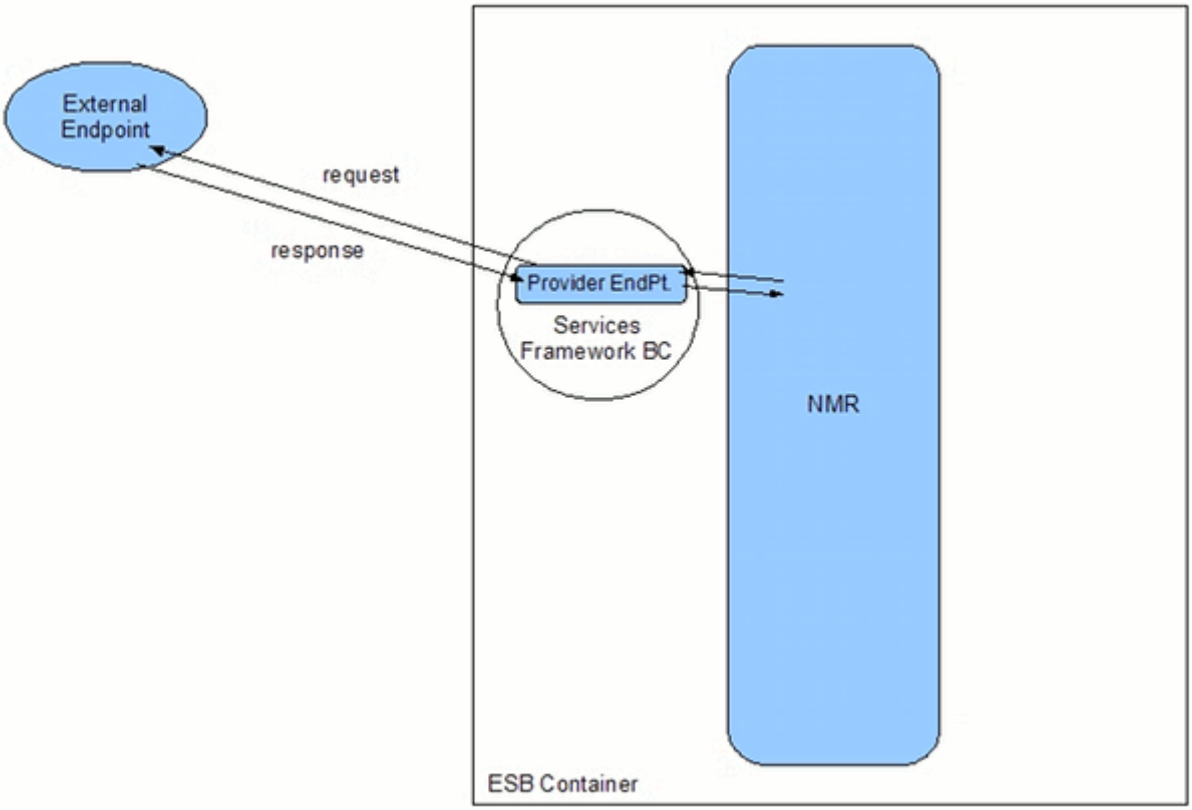
IMPORTANT

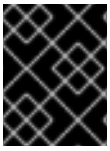
The Java Business Integration components of Red Hat JBoss Fuse are considered deprecated. You should consider migrating any JBI applications to OSGi.

OVERVIEW

Provider endpoints play the role of service provider from the vantage point of other endpoints running inside of the ESB. However, from outside of the ESB a provider endpoint plays the role of a consumer. As shown in Figure 29.1, “Provider Endpoint”, provider endpoints make requests on external endpoints. When it receives the response, the provider endpoint returns it back to the NMR.

Figure 29.1. Provider Endpoint





IMPORTANT

Because provider endpoint's behave like consumers to external endpoints, you configure the runtime behavior of the transport using the consumer-specific WSDL entries.

PROCEDURE

To configure a provider endpoint do the following:

1. Add a **`provider`** element to your **`xbean.xml`** file.
2. Add a **`wsdl`** attribute to the **`provider`** element.

   See the section called “Specifying the WSDL”.
3. If your WSDL defines more than one service, you will need to specify a value for the **`service`** attribute.

   See the section called “Specifying the endpoint details”.
4. If the service you choose defines more than one endpoint, you will need to specify a value for the **`endpoint`** attribute.

   See the section called “Specifying the endpoint details”.
5. If your endpoint is going to be receiving binary attachments set its **`mtomEnabled`** attribute to **`true`**.

   See Chapter 30, *Using MTOM to Process Binary Content*.
6. If your endpoint does not need to process the JBI wrapper set its **`useJbiWrapper`** attribute to **`false`**.

   See Chapter 31, *Working with the JBI Wrapper*.
7. If you are using any of the advanced features, such as WS-Addressing or WS-Policy, specify a value for the **`busCfg`** attribute.

   See ???.

SPECIFYING THE WSDL

The **`wsdl`** attribute is the only required attribute to configure a provider endpoint. It specifies the location of the WSDL document that defines the endpoint being exposed. The path used is relative to the top-level of the exploded service unit.

TIP

If the WSDL document defines a single service with a single endpoint, then you do not require any additional information to expose a provider endpoint.

Example 29.1, “Minimal Provider Endpoint Configuration” shows the minimal configuration for a provider endpoint.

Example 29.1. Minimal Provider Endpoint Configuration

```
<beans xmlns:cxfbc="http://servicemix.apache.org/cxfbc/1.0"
       ... >
  ...
  <cxfbc:provider wsdl="/wsdl/widget.wsdl" />
```

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

For information on creating a WSDL document see ???.

SPECIFYING THE ENDPOINT DETAILS

If the endpoint's WSDL document defines a single service with a single endpoint, the ESB can easily determine which endpoint to use. It will use the values from the WSDL document to specify the service name, endpoint name and interface name for the instantiated endpoint.

However, if the endpoint's WSDL document defines multiple services or if it defines multiple endpoints for a service, you will need to provide the provider endpoint with additional information so that it can determine the proper definition to use. What information you need to provide depends on the complexity of the WSDL document. You may need to supply values for both the service name and the endpoint name, or you may only have to supply one of these values.

If the WSDL document contains more than one **`service`** element you will need to specify a value for the provider's **`service`** attribute. The value of the provider's **`service`** attribute is the QName of the WSDL **`service`** element that defines the desired service in the WSDL document. For example, if you wanted your endpoint to use the WidgetInventoryService in the WSDL shown in Example 29.2, “WSDL with Two Services” you would use the configuration shown in Example 29.3, “Provider Endpoint with a Defined Service Name”.

Example 29.2. WSDL with Two Services

```
<definitions ...
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             targetNamespace="http://demos.widgetVendor.com" ...>
  ...
  <service name="WidgetSalesService">
    <port binding="WidgetSalesBinding" name="WidgetSalesPort">
      <soap:address location="http://widget.sales.com/index.xml">
    </port>
  </service>

  <service name="WidgetInventoryService">
    <port binding="WidgetInventoryBinding" name="WidgetInventoryPort">
      <soap:address location="http://widget.inventory.com/index.xml">
    </port>
  </service>
  ...
<definitions>
```

Example 29.3. Provider Endpoint with a Defined Service Name

```
<beans xmlns:cxfbc="http://servicemix.apache.org/cxfbc/1.0"
       xmlns:widgets="http://demos.widgetVendor.com"
       ... >
  ...
  <cxfbc:provider wsdl="/wsdl/widget.wsdl"
```

```
                    service="widgets:WidgetInventoryService" />

  ...
</beans>
```

If the WSDL document's service definition contains more than one endpoint, then you will need to provide a value for the provider's **endpoint** attribute. The value of the **endpoint** attribute corresponds to the value of the WSDL **port** element's **name** attribute. For example, if you wanted your endpoint to use the WidgetWesternSalesPort in the WSDL shown in Example 29.4, “Service with Two Endpoints” you would use the configuration shown in Example 29.5, “Provider Endpoint with a Defined Endpoint Name”.

Example 29.4. Service with Two Endpoints

```
<definitions ...
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             targetNamespace="http://demos.widgetVendor.com" ...>
  ...
  <service name="WidgetSalesService">
    <port binding="WidgetSalesBinding" name="WidgetWesternSalesPort">
      <soap:address location="http://widget.sales.com/index.xml">
    </port>
    <port binding="WidgetSalesBinding" name="WidgetEasternSalesPort">
      <jms:address jndiConnectionFactoryName="ConnectionFactory"

jndiDestinationName="dynamicQueues/test.Celtix.jmstransport" >
       <jms:JMSNamingProperty name="java.naming.factory.initial"

value="org.activemq.jndi.ActiveMQInitialContextFactory" />
       <jms:JMSNamingProperty name="java.naming.provider.url"
                              value="tcp://localhost:61616" />
      </jms:address>
    </port>
  </service>
  ...
<definitions>
```

Example 29.5. Provider Endpoint with a Defined Endpoint Name

```
<beans xmlns:cxfbc="http://servicemix.apache.org/cxfbc/1.0"
       xmlns:widgets="http://demos.widgetVendor.com"
       ... >
  ...
  <cxfbc:provider wsdl="/wsdl/widget.wsdl"
                  endpoint="WidgetWesternSalesService" />
  ...
</beans>
```

CHAPTER 30. USING MTOM TO PROCESS BINARY CONTENT

Abstract

Enabling MTOM support allows your endpoints to consume and produce messages that contain binary data.

IMPORTANT

The Java Business Integration components of Red Hat JBoss Fuse are considered deprecated. You should consider migrating any JBI applications to OSGi.

OVERVIEW

SOAP Message Transmission Optimization Mechanism (MTOM) specifies an optimized method for sending binary data as part of a SOAP message using the XML-binary Optimized Packaging (XOP) packages for transmitting binary data. The Apache CXF binding supports the use of MTOM to send and receive binary data. MTOM support is enabled on an endpoint by endpoint basis.

CONFIGURING AN ENDPOINT TO SUPPORT MTOM

As shown in Example 30.1, “Configuring an Endpoint to Use MTOM”, you configure an endpoint to support MTOM by setting its **`mtomEnabled`** attribute to true.

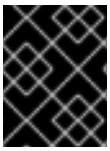
Example 30.1. Configuring an Endpoint to Use MTOM

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

  <cxfbc:consumer wsdl="/wsdl/widget.wsdl"
                  mtomEnabled="true" />

  ...
</beans>
```

CHAPTER 31. WORKING WITH THE JBI WRAPPER

Abstract

By default, all Apache CXF binding component endpoints expect SOAP messages to be inside of the JBI wrapper. You can turn off the extra processing if it is not required.



IMPORTANT

The Java Business Integration components of Red Hat JBoss Fuse are considered deprecated. You should consider migrating any JBI applications to OSGi.

OVERVIEW

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.

For the sake of compatibility, all endpoints exposed by the Apache CXF binding component will check for the JBI wrapper. If it is present the endpoint will unwrap the messages. If you are positive that your endpoints will never receive messages that use the JBI wrapper, you can turn off the extra processing.

TURNING OF JBI WRAPPER PROCESSING

If you are sure your endpoint will not receive messages using the JBI wrapper you can set its **`useJbiWrapper`** attribute to **`false`**. This instructs the endpoint to disable the processing of the JBI wrapper. If the endpoint does receive a message that uses the JBI wrapper, it will fail to process the message and generate an error.

EXAMPLE

Example 31.1, “Configuring a Consumer to Not Use the JBI Wrapper” shows a configuration fragment for configuring a consumer that does not process the JBI wrapper.

Example 31.1. Configuring a Consumer to Not Use the JBI Wrapper

```
<beans xmlns:cxfbc="http://servicemix.apache.org/cxfbc/1.0"
       ... >
  ...
  <cxfbc:consumer wsdl="/wsdl/widget.wsdl"
                  useJbiWrapper="false" />
  ...
</beans>
```

CHAPTER 32. USING MESSAGE INTERCEPTORS

Abstract

You can use low-level message interceptors to process messages before they are delivered to your endpoint's service implementation.

IMPORTANT

The Java Business Integration components of Red Hat JBoss Fuse are considered deprecated. You should consider migrating any JBI applications to OSGi.

OVERVIEW

Interceptors are a low-level pieces of code that process messages as they are passed between the message channel and service's implementation. They have access to the raw message data and can be used to process SOAP action entries, process security tokens, or correlate messages. Interceptors are called in a chain and you can configure what interceptors are used at a number of points along the chain.

CONFIGURING AN ENDPOINT'S INTERCEPTOR CHAIN

A Apache CXF binding component endpoint's interceptor chain has four points at which you can insert an interceptor:

in interceptors

On consumer endpoints the *in interceptors* process messages when they are received from the external endpoint.

On provider endpoints the *in interceptors* process messages when they are received from the NMR.

in fault interceptors

The *in fault interceptors* process fault messages that are generated before the service implementation gets called.

out interceptors

On consumer endpoints the *out interceptors* process messages as they pass from the service implementation to the external endpoint.

On provider endpoints the *out interceptors* process messages as they pass from the service implementation to the NMR.

out fault interceptors

The *out fault interceptors* process fault messages that are generated by the service implementation or by an out interceptor.

An endpoint's interceptor chain is configured using children of its **`consumer`** element or **`provider`** element. Table 32.1, “Elements Used to Configure an Endpoint's Interceptor Chain” lists the elements used to configure an endpoint's interceptor chain.

Table 32.1. Elements Used to Configure an Endpoint's Interceptor Chain

Name	Description
inInterceptors	Specifies a list of interceptors that process incoming messages.
inFaultInterceptors	Specifies a list of interceptors that process incoming fault messages.
outInterceptors	Specifies a list of interceptors that process outgoing messages.
outFaultInterceptors	Specifies a list of interceptors that process outgoing fault messages.

Example 32.1, “Configuring an Interceptor Chain” shows a consumer endpoint configured to use the Apache CXF logging interceptors.

Example 32.1. Configuring an Interceptor Chain

```
<cxfbc:consumer ...>
  ...
  <cxfbc:inInterceptors>
    <bean class="org.apache.cxf.interceptor.LoggingInInterceptor" />
  </cxfbc:inInterceptors>
  <cxfbc:outInterceptors>
    <bean class="org.apache.cxf.interceptor.LoggingOutInterceptor" />
  </cxfbc:outInterceptors>
  <cxfbc:inFaultInterceptors>
    <bean class="org.apache.cxf.interceptor.LoggingInInterceptor" />
  </cxfbc:inFaultInterceptors>
  <cxfbc:outFaultInterceptors>
    <bean class="org.apache.cxf.interceptor.LoggingOutInterceptor" />
  </cxfbc:outFaultInterceptors>
</cxfbc:consumer>
```

IMPLEMENTING AN INTERCEPTOR

You can implement a custom interceptor by extending the **org.apache.cxf.phase.AbstractPhaseInterceptor** class or one of its sub-classes. Extending **AbstractPhaseInterceptor** provides you with access to the generic message handling APIs used by Apache CXF. Extending one of the sub-classes provides you with more specific APIs. For example, extending the **AbstractSoapInterceptor** class allows your interceptor to work directly with the SOAP APIs.

MORE INFORMATION

For more information about writing Apache CXF interceptors see the Apache CXF documentation.

CHAPTER 33. CONFIGURING THE ENDPOINTS TO LOAD APACHE CXF RUNTIME CONFIGURATION

Abstract

Both consumers and providers use the **busCfg** attribute to configure the endpoint to load Apache CXF runtime configuration. Its value points to a Apache CXF configuration file.

IMPORTANT

The Java Business Integration components of Red Hat JBoss Fuse are considered deprecated. You should consider migrating any JBI applications to OSGi.

SPECIFYING THE CONFIGURATION TO LOAD

You instruct an endpoint to load Apache CXF runtime configuration using the **busCfg** attribute. Both the **provider** element and the **consumer** element accept this attribute. The attribute's value is the path to a file containing configuration information used by the Apache CXF runtime. This path is relative to the location of the endpoint's **xbean.xml** file.

TIP

The Apache CXF configuration file should be stored in the endpoint's service unit.

Each endpoint uses a separate Apache CXF runtime. If your service unit creates multiple endpoints, each endpoint can load its own Apache CXF runtime configuration.

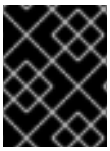
EXAMPLE

Example 33.1, “Provider Endpoint that Loads Apache CXF Runtime Configuration” shows the configuraiton for a provider endpoint that loads a Apache CXF configuration file called **jms-config.xml**.

Example 33.1. Provider Endpoint that Loads Apache CXF Runtime Configuration

```
<beans xmlns:cxfbc="http://servicemix.apache.org/cxfbc/1.0"
       xmlns:greeter="http://cxf.apache.org/jms_greeter"
       xmlns:test="http://test">

  <cxfbc:provider wsdl="classpath:jms_greeter.wsdl"
                  service="greeter:JMSGreeterService"
                  endpoint="GreeterPort"
                  interfaceName="greeter:JMSGreeterPortType"
                  useJBIWrapper="false"
                  busCfg="./jms-config.xml" />

</beans>
```

CHAPTER 34. TRANSPORT CONFIGURATION



IMPORTANT

The Java Business Integration components of Red Hat JBoss Fuse are considered deprecated. You should consider migrating any JBI applications to OSGi.

34.1. USING THE JMS CONFIGURATION BEAN

Overview

To simplify JMS configuration and make it more powerful, Apache CXF uses a single JMS configuration bean to configure JMS endpoints. The bean is implemented by the **`org.apache.cxf.transport.jms.JMSConfiguration`** class. It can be used to either configure endpoint's directly or to configure the JMS conduits and destinations.

Configuration namespace

The JMS configuration bean uses the Spring p-namespace to make the configuration as simple as possible. To use this namespace you need to declare it in the configuration's root element as shown in Example 34.1, “Declaring the Spring p-namespace”.

Example 34.1. Declaring the Spring p-namespace

```
<beans ...
  xmlns:p="http://www.springframework.org/schema/p"
  ... >
  ...
</beans>
```

Specifying the configuration

You specify the JMS configuration by defining a bean of class **`org.apache.cxf.transport.jms.JMSConfiguration`**. The properties of the bean provide the configuration settings for the transport.

Table 34.1, “General JMS Configuration Properties” lists properties that are common to both providers and consumers.

Table 34.1. General JMS Configuration Properties

Property	Default	Description
`connectionFactory-ref`		Specifies a reference to a bean that defines a JMS **`ConnectionFactory`**.

Property	Default	Description
wrapInSingleConnectionFactory	**true**	Specifies whether to wrap the **ConnectionFactory** with a Spring **SingleConnectionFactory**. Doing so can improve the performance of the JMS transport when the specified connection factory does not pool connections.
reconnectOnException	**false**	Specifies whether to create a new connection in the case of an exception. This property is only used when wrapping the connection factory with a Spring **SingleConnectionFactory**.
targetDestination		Specifies the JNDI name or provider specific name of a destination.
replyDestination		Specifies the JMS name of the JMS destinations where replies are sent. This attribute allows you to use a user defined destination for replies. For more details see Section 26.3, “Using a Named Reply Destination”.
destinationResolver		Specifies a reference to a Spring **DestinationResolver**. This allows you to define how destination names are resolved. By default a **DynamicDestinationResolver** is used. It resolves destinations using the JMS providers features. If you reference a **JndiDestinationResolver** you can resolve the destination names using JNDI.
transactionManager		Specifies a reference to a Spring transaction manager. This allows the service to participate in JTA Transactions.

Property	Default	Description
taskExecutor		Specifies a reference to a Spring **TaskExecutor**. This is used in listeners to decide how to handle incoming messages. By default the transport uses the Spring **SimpleAsyncTaskExecutor**.
useJms11	**false**	Specifies whether JMS 1.1 features are available.
messageIdEnabled	**true**	Specifies whether the JMS transport wants the JMS broker to provide message IDs. Setting this 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.
messageTimestampEnabled	**true**	Specifies whether the JMS transport wants the JMS broker to provide message time stamps. Setting this 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 time stamps or add them to the messages from the endpoint. The JMS broker can choose to accept the hint or ignore it.
cacheLevel	**3**	Specifies the level of caching allowed by the listener. Valid values are **0**(CACHE_NONE), **1**(CACHE_CONNECTION), **2**(CACHE_SESSION), **3**(CACHE_CONSUMER), **4**(CACHE_AUTO).

Property	Default	Description
pubSubNoLocal	**false**	Specifies whether to receive messages produced from the same connection.
receiveTimeout	**0**	Specifies, in milliseconds, the amount of time to wait for response messages. **0** means wait indefinitely.
explicitQosEnabled	**false**	Specifies whether the QoS settings like priority, persistence, and time to live are explicitly set for each message or if they are allowed to use default values.
deliveryMode	**1**	Specifies if a message is persistent. The two values are: • **1**(NON_PERSISTENT)—messages will be kept memory • **2**(PERSISTENT)—messages will be persisted to disk
priority	**4**	Specifies the message's priority for the messages. JMS priority values can range from 0 to 9. The lowest priority is 0 and the highest priority is 9.
timeToLive	**0**	Specifies, in milliseconds, the message will be available after it is sent. 0 specifies an infinite time to live.
sessionTransacted	**false**	Specifies if JMS transactions are used.
concurrentConsumers	**1**	Specifies the minimum number of concurrent consumers created by the listener.
maxConcurrentConsumers	**1**	Specifies the maximum number of concurrent consumers by listener.

Property	Default	Description
messageSelector		Specifies the string value of the selector. For more information on the syntax used to specify message selectors, see the JMS 1.1 specification.
subscriptionDurable	**false**	Specifies whether the server uses durrable subscriptions.
durableSubscriptionName		Specifies the string used to register the durable subscription.
messageType	**text**	Specifies how the message data will be packaged as a JMS message. **text** specifies that the data will be packaged as a **TextMessage**. **binary** specifies that the data will be packaged as an **ByteMessage**.
pubSubDomain	**false**	Specifies whether the target destination is a topic.
jmsProviderTibcoEms	**false**	Specifies if your JMS provider is Tibco EMS. This causes the principal in the security context to be populated from the **JMS_TIBCO_SENDER** header.
useMessageIDAsCorrelationID	**false**	Specifies whether JMS will use the message ID to correlate messages. If not, the client will set a generated correlation ID.

As shown in Example 34.2, “JMS configuration bean”, the bean's properties are specified as attributes to the **bean** element. They are all declared in the Spring **p** namespace.

Example 34.2. JMS configuration bean

```
<bean id="jmsConfig"
      class="org.apache.cxf.transport.jms.JMSConfiguration"
      p:connectionFactory-ref="connectionFactory"
      p:targetDestination="dynamicQueues/greeter.request.queue"
      p:pubSubDomain="false" />
```

Applying the configuration to an endpoint

The **JMSConfiguration** bean can be applied directly to both server and client endpoints using the

Apache CXF features mechanism. To do so:

1. Set the endpoint's **address** attribute to **jms://**.
2. Add a **jaxws:feature** element to the endpoint's configuration.
3. Add a bean of type **org.apache.cxf.transport.jms.JMSConfigFeature** to the feature.
4. Set the **bean** element's **p:jmsConfig-ref** attribute to the ID of the **JMSConfiguration** bean.

Example 34.3, “Adding JMS configuration to a JAX-WS client” shows a JAX-WS client that uses the JMS configuration from Example 34.2, “JMS configuration bean”.

Example 34.3. Adding JMS configuration to a JAX-WS client

```
<jaxws:client id="CustomerService"
              xmlns:customer="http://customerservice.example.com/"
              serviceName="customer:CustomerServiceService"
              endpointName="customer:CustomerServiceEndpoint"
              address="jms://"

serviceClass="com.example.customerservice.CustomerService">
  <jaxws:features>
    <bean class="org.apache.cxf.transport.jms.JMSConfigFeature"
          p:jmsConfig-ref="jmsConfig"/>
  </jaxws:features>
</jaxws:client>
```

Applying the configuration to the transport

The **JMSConfiguration** bean can be applied to JMS conduits and JMS destinations using the **jms:jmsConfig-ref** element. The **jms:jmsConfig-ref** element's value is the ID of the **JMSConfiguration** bean.

Example 34.4, “Adding JMS configuration to a JMS conduit” shows a JMS conduit that uses the JMS configuration from Example 34.2, “JMS configuration bean”.

Example 34.4. Adding JMS configuration to a JMS conduit

```
<jms:conduit name="
{http://cxf.apache.org/jms_conf_test}HelloWorldQueueBinMsgPort.jms-
conduit">
  ...
  <jms:jmsConfig-ref>jmsConf</jms:jmsConfig-ref>
</jms:conduit>
```

34.2. CONFIGURING THE JETTY RUNTIME

Overview

The Jetty runtime is used by HTTP service providers and HTTP consumers using a decoupled endpoint. The runtime's thread pool can be configured, and you can also set a number of the security settings for an HTTP service provider through the Jetty runtime.

Maven dependency

If you use Apache Maven as your build system, you can add the Jetty runtime to your project by including the following dependency in your project's **pom.xml** file:

```
<dependency>
    <groupId>org.apache.cxf</groupId>
    <artifactId>cxf-rt-transports-http-jetty</artifactId>
    <version>${cxf-version}</version>
</dependency>
```

Namespace

The elements used to configure the Jetty runtime are defined in the namespace http://cxf.apache.org/transports/http-jetty/configuration. It is commonly referred to using the prefix **httpj**. In order to use the Jetty configuration elements you must add the lines shown in Example 34.5, “Jetty Runtime Configuration Namespace” to the **beans** element of your endpoint's configuration file. In addition, you must add the configuration elements' namespace to the **xsi:schemaLocation** attribute.

Example 34.5. Jetty Runtime Configuration Namespace

```
<beans ...
       xmlns:httpj="http://cxf.apache.org/transports/http-
jetty/configuration"
       ...
       xsi:schemaLocation="...
                           http://cxf.apache.org/transports/http-
jetty/configuration

http://cxf.apache.org/schemas/configuration/http-jetty.xsd
                          ...">
```

The `engine-factory` element

The **httpj:engine-factory** element is the root element used to configure the Jetty runtime used by an application. It has a single required attribute, **bus**, whose value is the name of the **Bus** that manages the Jetty instances being configured.

TIP

The value is typically **cxf** which is the name of the default **Bus** instance.

The **httpj:engine-factory** element has three children that contain the information used to configure the HTTP ports instantiated by the Jetty runtime factory. The children are described in Table 34.2, “Elements for Configuring a Jetty Runtime Factory”.

Table 34.2. Elements for Configuring a Jetty Runtime Factory

Element	Description
`httpj:engine`	Specifies the configuration for a particular Jetty runtime instance. See the section called "The **`engine`** element".
`httpj:identifiedTLSServerParameters`	Specifies a reusable set of properties for securing an HTTP service provider. It has a single attribute, **`id`**, that specifies a unique identifier by which the property set can be referred.
`httpj:identifiedThreadingParameters`	Specifies a reusable set of properties for controlling a Jetty instance's thread pool. It has a single attribute, **`id`**, that specifies a unique identifier by which the property set can be referred. See the section called "Configuring the thread pool".

The `engine` element

The **`httpj:engine`** element is used to configure specific instances of the Jetty runtime. It has a single attribute, **`port`**, that specifies the number of the port being managed by the Jetty instance.

TIP

You can specify a value of **`0`** for the **`port`** attribute. Any threading properties specified in an **`httpj:engine`** element with its **`port`** attribute set to **`0`** are used as the configuration for all Jetty listeners that are not explicitly configured.

Each **`httpj:engine`** element can have two children: one for configuring security properties and one for configuring the Jetty instance's thread pool. For each type of configuration you can either directly provide the configuration information or you can provide a reference to a set of configuration properties defined in the parent **`httpj:engine-factory`** element.

The child elements used to provide the configuration properties are described in Table 34.3, "Elements for Configuring a Jetty Runtime Instance".

Table 34.3. Elements for Configuring a Jetty Runtime Instance

Element	Description
`httpj:tlsServerParameters`	Specifies a set of properties for configuring the security used for the specific Jetty instance.
`httpj:tlsServerParametersRef`	Refers to a set of security properties defined by a **`identifiedTLSServerParameters`** element. The **`id`** attribute provides the id of the referred **`identifiedTLSServerParameters`** element.
`httpj:threadingParameters`	Specifies the size of the thread pool used by the specific Jetty instance. See the section called "Configuring the thread pool".

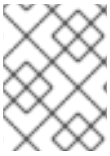
Element	Description
`httpj:threadingParametersRef`	Refers to a set of properties defined by a **`identifiedThreadingParameters`** element. The **`id`** attribute provides the id of the referred **`identifiedThreadingParameters`** element.

Configuring the thread pool

You can configure the size of a Jetty instance's thread pool by either:

- Specifying the size of the thread pool using a **`identifiedThreadingParameters`** element in the **`engine-factory`** element. You then refer to the element using a **`threadingParametersRef`** element.
- Specifying the size of the of the thread pool directly using a **`threadingParameters`** element.

The **`threadingParameters`** has two attributes to specify the size of a thread pool. The attributes are described in Table 34.4, “Attributes for Configuring a Jetty Thread Pool”.



NOTE

The **`httpj:identifiedThreadingParameters`** element has a single child **`threadingParameters`** element.

Table 34.4. Attributes for Configuring a Jetty Thread Pool

Attribute	Description
`minThreads`	Specifies the minimum number of threads available to the Jetty instance for processing requests.
`maxThreads`	Specifies the maximum number of threads available to the Jetty instance for processing requests.

Example

Example 34.6, “Configuring a Jetty Instance” shows a configuration fragment that configures a Jetty instance on port number 9001.

Example 34.6. Configuring a Jetty Instance

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:sec="http://cxf.apache.org/configuration/security"
  xmlns:http="http://cxf.apache.org/transports/http/configuration"
  xmlns:httpj="http://cxf.apache.org/transports/http-
jetty/configuration"
  xmlns:jaxws="http://java.sun.com/xml/ns/jaxws"
  xsi:schemaLocation="http://cxf.apache.org/configuration/security
          http://cxf.apache.org/schemas/configuration/security.xsd
```

```
            http://cxf.apache.org/transports/http/configuration
            http://cxf.apache.org/schemas/configuration/http-conf.xsd
            http://cxf.apache.org/transports/http-jetty/configuration
            http://cxf.apache.org/schemas/configuration/http-jetty.xsd
            http://www.springframework.org/schema/beans
            http://www.springframework.org/schema/beans/spring-beans-
2.0.xsd">
  ...

  <httpj:engine-factory bus="cxf">
    <httpj:identifiedTLSServerParameters id="secure">
      <sec:keyManagers keyPassword="password">
        <sec:keyStore type="JKS" password="password"
                      file="certs/cherry.jks"/>
      </sec:keyManagers>
    </httpj:identifiedTLSServerParameters>

    <httpj:engine port="9001">
      <httpj:tlsServerParametersRef id="secure" />
      <httpj:threadingParameters minThreads="5"
                                 maxThreads="15" />
    </httpj:engine>
  </httpj:engine-factory>
 </beans>
```

CHAPTER 35. DEPLOYING WS-ADDRESSING

Abstract

Apache CXF supports WS-Addressing for JAX-WS applications. This chapter explains how to deploy WS-Addressing in the Apache CXF runtime environment.

35.1. INTRODUCTION TO WS-ADDRESSING

Overview

WS-Addressing is a specification that allows services to communicate addressing information in a transport neutral way. It consists of two parts:

- A structure for communicating a reference to a Web service endpoint
- A set of Message Addressing Properties (MAP) that associate addressing information with a particular message

Supported specifications

Apache CXF supports both the WS-Addressing 2004/08 specification and the WS-Addressing 2005/03 specification.

Further information

For detailed information on WS-Addressing, see the 2004/08 submission at http://www.w3.org/Submission/ws-addressing/.

35.2. WS-ADDRESSING INTERCEPTORS

Overview

In Apache CXF, WS-Addressing functionality is implemented as interceptors. The Apache CXF runtime uses interceptors to intercept and work with the raw messages that are being sent and received. When a transport receives a message, it creates a message object and sends that message through an interceptor chain. If the WS-Addressing interceptors are added to the application's interceptor chain, any WS-Addressing information included with a message is processed.

WS-Addressing Interceptors

The WS-Addressing implementation consists of two interceptors, as described in Table 35.1, “WS-Addressing Interceptors”.

Table 35.1. WS-Addressing Interceptors

Interceptor	Description

Interceptor	Description
`org.apache.cxf.ws.addressing.MAPAggregator`	A logical interceptor responsible for aggregating the Message Addressing Properties (MAPs) for outgoing messages.
`org.apache.cxf.ws.addressing.soap.MAPCodec`	A protocol-specific interceptor responsible for encoding and decoding the Message Addressing Properties (MAPs) as SOAP headers.

35.3. ENABLING WS-ADDRESSING

Overview

To enable WS-Addressing the WS-Addressing interceptors must be added to the inbound and outbound interceptor chains. This is done in one of the following ways:

- Apache CXF Features
- RMAssertion and WS-Policy Framework
- Using Policy Assertion in a WS-Addressing Feature

Adding WS-Addressing as a Feature

WS-Addressing can be enabled by adding the WS-Addressing feature to the client and the server configuration as shown in Example 35.1, “client.xml—Adding WS-Addressing Feature to Client Configuration” and Example 35.2, “server.xml—Adding WS-Addressing Feature to Server Configuration” respectively.

Example 35.1. client.xml—Adding WS-Addressing Feature to Client Configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jaxws="http://cxf.apache.org/jaxws"
       xmlns:wsa="http://cxf.apache.org/ws/addressing"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

    <jaxws:client ...>
        <jaxws:features>
            <wsa:addressing/>
        </jaxws:features>
    </jaxws:client>
</beans>
```

Example 35.2. server.xml—Adding WS-Addressing Feature to Server Configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jaxws="http://cxf.apache.org/jaxws"
       xmlns:wsa="http://cxf.apache.org/ws/addressing"
       xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

    <jaxws:endpoint ...>
        <jaxws:features>
            <wsa:addressing/>
        </jaxws:features>
    </jaxws:endpoint>
</beans>
```

35.4. CONFIGURING WS-ADDRESSING ATTRIBUTES

Overview

The Apache CXF WS-Addressing feature element is defined in the namespace **`http://cxf.apache.org/ws/addressing`**. It supports the two attributes described in Table 35.2, "WS-Addressing Attributes".

Table 35.2. WS-Addressing Attributes

Attribute Name	Value
`allowDuplicates`	A boolean that determines if duplicate MessageIDs are tolerated. The default setting is **`true`**.
`usingAddressingAdvisory`	A boolean that indicates if the presence of the **`UsingAddressing`** element in the WSDL is advisory only; that is, its absence does not prevent the encoding of WS-Addressing headers.

Configuring WS-Addressing attributes

Configure WS-Addressing attributes by adding the attribute and the value you want to set it to the WS-Addressing feature in your server or client configuration file. For example, the following configuration extract sets the **`allowDublicates`** attribute to **`false`** on the server endpoint:

```
<beans ... xmlns:wsa="http://cxf.apache.org/ws/addressing" ...>
    <jaxws:endpoint ...>
        <jaxws:features>
            <wsa:addressing allowDuplicates="false"/>
        </jaxws:features>
    </jaxws:endpoint>
</beans>
```

Using a WS-Policy assertion embedded in a feature

In Example 35.3, “Using the Policies to Configure WS-Addressing” an addressing policy assertion to enable non-anonymous responses is embedded in the **policies** element.

Example 35.3. Using the Policies to Configure WS-Addressing

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:wsa="http://cxf.apache.org/ws/addressing"
       xmlns:wsp="http://www.w3.org/2006/07/ws-policy"
       xmlns:policy="http://cxf.apache.org/policy-config"
       xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
wss-wssecurity-utility-1.0.xsd"
       xmlns:jaxws="http://cxf.apache.org/jaxws"
       xsi:schemaLocation="
http://www.w3.org/2006/07/ws-policy http://www.w3.org/2006/07/ws-
policy.xsd
http://cxf.apache.org/ws/addressing
http://cxf.apache.org/schema/ws/addressing.xsd
http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

    <jaxws:endpoint name="
{http://cxf.apache.org/greeter_control}GreeterPort"
                    createdFromAPI="true">
        <jaxws:features>
            <policy:policies>
                <wsp:Policy
xmlns:wsam="http://www.w3.org/2007/02/addressing/metadata">
                    <wsam:Addressing>
                        <wsp:Policy>
                            <wsam:NonAnonymousResponses/>
                        </wsp:Policy>
                    </wsam:Addressing>
                </wsp:Policy>
            <policy:policies>
        </jaxws:features>
    </jaxws:endpoint>
</beans>
```

CHAPTER 36. ENABLING RELIABLE MESSAGING

Abstract

Apache CXF supports WS-Reliable Messaging(WS-RM). This chapter explains how to enable and configure WS-RM in Apache CXF.

36.1. INTRODUCTION TO WS-RM

Overview

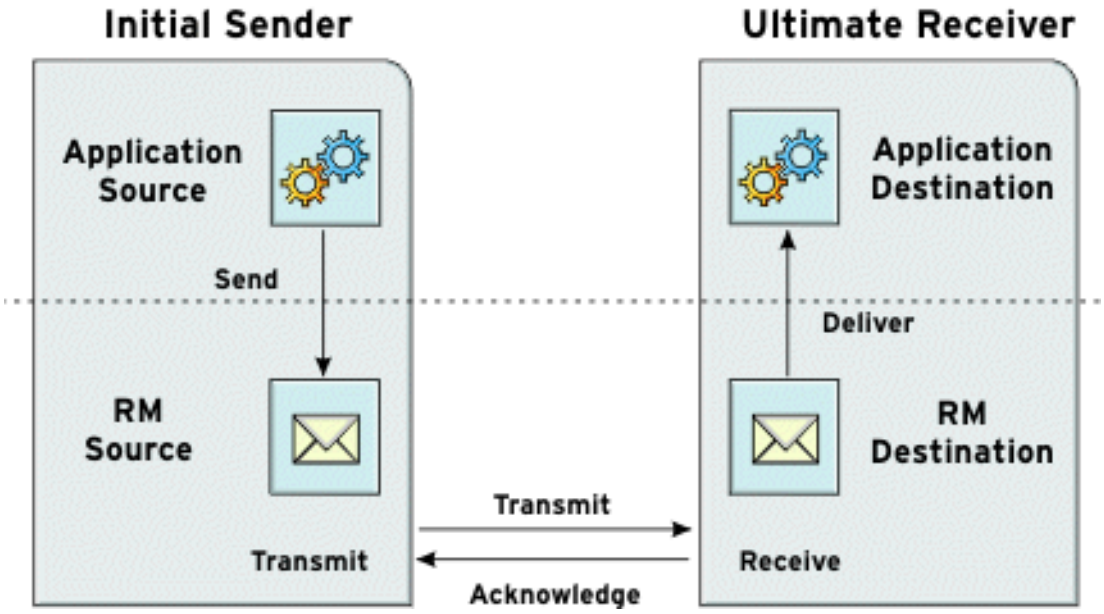
WS-ReliableMessaging (WS-RM) is a protocol that ensures the reliable delivery of messages in a distributed environment. It enables messages to be delivered reliably between distributed applications in the presence of software, system, or network failures.

For example, WS-RM can be used to ensure that the correct messages have been delivered across a network exactly once, and in the correct order.

How WS-RM works

WS-RM ensures the reliable delivery of messages between a source and a destination endpoint. The source is the initial sender of the message and the destination is the ultimate receiver, as shown in Figure 36.1, “Web Services Reliable Messaging”.

Figure 36.1. Web Services Reliable Messaging



The flow of WS-RM messages can be described as follows:

1. The RM source sends a **`CreateSequence`** protocol message to the RM destination. This contains a reference for the endpoint that receives acknowledgements (the **`wsrm:AcksTo`** endpoint).

2. The RM destination sends a **`CreateSequenceResponse`** protocol message back to the RM source. This message contains the sequence ID for the RM sequence session.

3. The RM source adds an RM **Sequence** header to each message sent by the application source. This header contains the sequence ID and a unique message ID.

4. The RM source transmits each message to the RM destination.

5. The RM destination acknowledges the receipt of the message from the RM source by sending messages that contain the RM **`SequenceAcknowledgement`** header.

6. The RM destination delivers the message to the application destination in an exactly-once-in-order fashion.

7. The RM source retransmits a message that it has not yet received an acknowledgement.

   The first retransmission attempt is made after a base retransmission interval. Successive retransmission attempts are made, by default, at exponential back-off intervals or, alternatively, at fixed intervals. For more details, see Section 36.4, “Configuring WS-RM”.

This entire process occurs symmetrically for both the request and the response message; that is, in the case of the response message, the server acts as the RM source and the client acts as the RM destination.

WS-RM delivery assurances

WS-RM guarantees reliable message delivery in a distributed environment, regardless of the transport protocol used. Either the source or the destination endpoint logs an error if reliable delivery can not be assured.

Supported specifications

Apache CXF supports the 2005/02 version of the WS-RM specification, which is based on the WS-Addressing 2004/08 specification.

Further information

For detailed information on WS-RM, see the specification at http://specs.xmlsoap.org/ws/2005/02/rm/ws-reliablemessaging.pdf.

36.2. WS-RM INTERCEPTORS

Overview

In Apache CXF, WS-RM functionality is implemented as interceptors. The Apache CXF runtime uses interceptors to intercept and work with the raw messages that are being sent and received. When a transport receives a message, it creates a message object and sends that message through an interceptor chain. If the application's interceptor chain includes the WS-RM interceptors, the application can participate in reliable messaging sessions. The WS-RM interceptors handle the collection and aggregation of the message chunks. They also handle all of the acknowledgement and retransmission logic.

Apache CXF WS-RM Interceptors

The Apache CXF WS-RM implementation consists of four interceptors, which are described in Table 36.1, “Apache CXF WS-ReliableMessaging Interceptors”.

Table 36.1. Apache CXF WS-ReliableMessaging Interceptors

Interceptor	Description
`org.apache.cxf.ws.rm.RMOutInterceptor`	Deals with the logical aspects of providing reliability guarantees for outgoing messages. Responsible for sending the `CreateSequence` requests and waiting for their `CreateSequenceResponse` responses. Also responsible for aggregating the sequence properties—ID and message number—for an application message.
`org.apache.cxf.ws.rm.RMInInterceptor`	Responsible for intercepting and processing RM protocol messages and `SequenceAcknowledgement` messages that are piggybacked on application messages.
`org.apache.cxf.ws.rm.soap.RMSoapInterceptor`	Responsible for encoding and decoding the reliability properties as SOAP headers.
`org.apache.cxf.ws.rm.RetransmissionInterceptor`	Responsible for creating copies of application messages for future resending.

Enabling WS-RM

The presence of the WS-RM interceptors on the interceptor chains ensures that WS-RM protocol messages are exchanged when necessary. For example, when intercepting the first application message on the outbound interceptor chain, the `RMOutInterceptor` sends a `CreateSequence` request and waits to process the original application message until it receives the `CreateSequenceResponse` response. In addition, the WS-RM interceptors add the sequence headers to the application messages and, on the destination side, extract them from the messages. It is not necessary to make any changes to your application code to make the exchange of messages reliable.

For more information on how to enable WS-RM, see Section 36.3, “Enabling WS-RM”.

Configuring WS-RM Attributes

You control sequence demarcation and other aspects of the reliable exchange through configuration. For example, by default Apache CXF attempts to maximize the lifetime of a sequence, thus reducing the overhead incurred by the out-of-band WS-RM protocol messages. To enforce the use of a separate sequence per application message configure the WS-RM source’s sequence termination policy (setting the maximum sequence length to `1`).

For more information on configuring WS-RM behavior, see Section 36.4, “Configuring WS-RM”.

36.3. ENABLING WS-RM

Overview

To enable reliable messaging, the WS-RM interceptors must be added to the interceptor chains for both inbound and outbound messages and faults. Because the WS-RM interceptors use WS-Addressing, the WS-Addressing interceptors must also be present on the interceptor chains.

You can ensure the presence of these interceptors in one of two ways:

- Explicitly, by adding them to the dispatch chains using Spring beans
- Implicitly, using WS-Policy assertions, which cause the Apache CXF runtime to transparently add the interceptors on your behalf.

Spring beans—explicitly adding interceptors

To enable WS-RM add the WS-RM and WS-Addressing interceptors to the Apache CXF bus, or to a consumer or service endpoint using Spring bean configuration. This is the approach taken in the WS-RM sample that is found in the ***InstallDir*/samples/ws_rm** directory. The configuration file, **ws-rm.cxf**, shows the WS-RM and WS-Addressing interceptors being added one-by-one as Spring beans (see Example 36.1, “Enabling WS-RM Using Spring Beans”).

Example 36.1. Enabling WS-RM Using Spring Beans

```
<?xml version="1.0" encoding="UTF-8"?>
1 <beans xmlns="http://www.springframework.org/schema/beans"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/
   beans http://www.springframework.org/schema/beans/spring-beans.xsd">
2    <bean id="mapAggregator"
  class="org.apache.cxf.ws.addressing.MAPAggregator"/>
   <bean id="mapCodec"
class="org.apache.cxf.ws.addressing.soap.MAPCodec"/>
3    <bean id="rmLogicalOut"
  class="org.apache.cxf.ws.rm.RMOutInterceptor">
       <property name="bus" ref="cxf"/>
   </bean>
   <bean id="rmLogicalIn" class="org.apache.cxf.ws.rm.RMInInterceptor">
       <property name="bus" ref="cxf"/>
   </bean>
   <bean id="rmCodec"
class="org.apache.cxf.ws.rm.soap.RMSoapInterceptor"/>
   <bean id="cxf" class="org.apache.cxf.bus.CXFBusImpl">
4        <property name="inInterceptors">
             <list>
               <ref bean="mapAggregator"/>
               <ref bean="mapCodec"/>
               <ref bean="rmLogicalIn"/>
               <ref bean="rmCodec"/>
           </list>
       </property>
5        <property name="inFaultInterceptors">
             <list>
               <ref bean="mapAggregator"/>
               <ref bean="mapCodec"/>
               <ref bean="rmLogicalIn"/>
               <ref bean="rmCodec"/>
           </list>
       </property>
```

```
6         <property name="outInterceptors">
              <list>
                <ref bean="mapAggregator"/>
                <ref bean="mapCodec"/>
                <ref bean="rmLogicalOut"/>
                <ref bean="rmCodec"/>
            </list>
        </property>
7         <property name="outFaultInterceptors">
              <list>
                <ref bean="mapAggregator">
                <ref bean="mapCodec"/>
                <ref bean="rmLogicalOut"/>
                <ref bean="rmCodec"/>
            </list>
        </property>
    </bean>
</beans>
```

The code shown in Example 36.1, “Enabling WS-RM Using Spring Beans” can be explained as follows:

1. A Apache CXF configuration file is a Spring XML file. You must include an opening Spring **beans** element that declares the namespaces and schema files for the child elements that are encapsulated by the **beans** element.
2. Configures each of the WS-Addressing interceptors—**`MAPAggregator`** and **`MAPCodec`**. For more information on WS-Addressing, see Chapter 35, *Deploying WS-Addressing*.
3. Configures each of the WS-RM interceptors—**`RMOutInterceptor`**, **`RMInInterceptor`**, and **`RMSoapInterceptor`**.
4. Adds the WS-Addressing and WS-RM interceptors to the interceptor chain for inbound messages.
5. Adds the WS-Addressing and WS-RM interceptors to the interceptor chain for inbound faults.
6. Adds the WS-Addressing and WS-RM interceptors to the interceptor chain for outbound messages.
7. Adds the WS-Addressing and WS-RM interceptors to the interceptor chain for outbound faults.

WS-Policy framework—implicitly adding interceptors

The WS-Policy framework provides the infrastructure and APIs that allow you to use WS-Policy. It is compliant with the November 2006 draft publications of the Web Services Policy 1.5—Framework and Web Services Policy 1.5—Attachment specifications.

To enable WS-RM using the Apache CXF WS-Policy framework, do the following:

1. Add the policy feature to your client and server endpoint. Example 36.2, “Configuring WS-RM using WS-Policy” shows a reference bean nested within a **`jaxws:feature`** element. The reference bean specifies the **`AddressingPolicy`**, which is defined as a separate element within the same configuration file.

Example 36.2. Configuring WS-RM using WS-Policy

```
<jaxws:client>
    <jaxws:features>
      <ref bean="AddressingPolicy"/>
    </jaxws:features>
</jaxws:client>
<wsp:Policy wsu:Id="AddressingPolicy"
xmlns:wsam="http://www.w3.org/2007/02/addressing/metadata">
    <wsam:Addressing>
      <wsp:Policy>
        <wsam:NonAnonymousResponses/>
      </wsp:Policy>
    </wsam:Addressing>
</wsp:Policy>
```

2. Add a reliable messaging policy to the **`wsdl:service`** element—or any other WSDL element that can be used as an attachment point for policy or policy reference elements—to your WSDL file, as shown in Example 36.3, “Adding an RM Policy to Your WSDL File”.

Example 36.3. Adding an RM Policy to Your WSDL File

```
<wsp:Policy wsu:Id="RM"
   xmlns:wsp="http://www.w3.org/2006/07/ws-policy"
   xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
wss-wssecurity-utility-1.0.xsd">
    <wsam:Addressing
xmlns:wsam="http://www.w3.org/2007/02/addressing/metadata">
        <wsp:Policy/>
    </wsam:Addressing>
    <wsrmp:RMAssertion
xmlns:wsrmp="http://schemas.xmlsoap.org/ws/2005/02/rm/policy">
        <wsrmp:BaseRetransmissionInterval Milliseconds="10000"/>
    </wsrmp:RMAssertion>
</wsp:Policy>
...
<wsdl:service name="ReliableGreeterService">
    <wsdl:port binding="tns:GreeterSOAPBinding"
name="GreeterPort">
        <soap:address
location="http://localhost:9020/SoapContext/GreeterPort"/>
        <wsp:PolicyReference URI="#RM"
xmlns:wsp="http://www.w3.org/2006/07/ws-policy"/>
    </wsdl:port>
</wsdl:service>
```

36.4. CONFIGURING WS-RM

You can configure WS-RM by:

- Setting Apache CXF-specific attributes that are defined in the Apache CXF WS-RM manager namespace, **`http://cxf.apache.org/ws/rm/manager`**.

- Setting standard WS-RM policy attributes that are defined in the http://schemas.xmlsoap.org/ws/2005/02/rm/policy namespace.

36.4.1. Configuring Apache CXF-Specific WS-RM Attributes

Overview

To configure the Apache CXF-specific attributes, use the **rmManager** Spring bean. Add the following to your configuration file:

- The **http://cxf.apache.org/ws/rm/manager** namespace to your list of namespaces.
- An **rmManager** Spring bean for the specific attribute that your want to configure.

Example 36.4, “Configuring Apache CXF-Specific WS-RM Attributes” shows a simple example.

Example 36.4. Configuring Apache CXF-Specific WS-RM Attributes

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:wsrm-mgr="http://cxf.apache.org/ws/rm/manager"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
 http://cxf.apache.org/ws/rm/manager
http://cxf.apache.org/schemas/configuration/wsrm-manager.xsd">
...
<wsrm-mgr:rmManager>
<!--
  ...Your configuration goes here
-->
</wsrm-mgr:rmManager>
```

Children of the rmManager Spring bean

Table 36.2, “Children of the rmManager Spring Bean” shows the child elements of the **rmManager** Spring bean, defined in the **http://cxf.apache.org/ws/rm/manager** namespace.

Table 36.2. Children of the rmManager Spring Bean

Element	Description
RMAssertion	An element of type RMAssertion
deliveryAssurance	An element of type DeliveryAssuranceType that describes the delivery assurance that should apply
sourcePolicy	An element of type SourcePolicyType that allows you to configure details of the RM source
destinationPolicy	An element of type DestinationPolicyType that allows you to configure details of the RM destination

Example

For an example, see the section called “Maximum unacknowledged messages threshold”.

36.4.2. Configuring Standard WS-RM Policy Attributes

Overview

You can configure standard WS-RM policy attributes in one of the following ways:

- RMAssertion in rmManager Spring bean
- Policy within a feature
- WSDL file
- External attachment

WS-Policy RMAssertion Children

Table 36.3, “Children of the WS-Policy RMAssertion Element” shows the elements defined in the **`http://schemas.xmlsoap.org/ws/2005/02/rm/policy`** namespace:

Table 36.3. Children of the WS-Policy RMAssertion Element

Name	Description
`InactivityTimeout`	Specifies the amount of time that must pass without receiving a message before an endpoint can consider an RM sequence to have been terminated due to inactivity.
`BaseRetransmissionInterval`	Sets the interval within which an acknowledgement must be received by the RM Source for a given message. If an acknowledgement is not received within the time set by the **`BaseRetransmissionInterval`**, the RM Source will retransmit the message.
`ExponentialBackoff`	Indicates the retransmission interval will be adjusted using the commonly known exponential backoff algorithm (Tanenbaum). For more information, see *Computer Networks*, Andrew S. Tanenbaum, Prentice Hall PTR, 2003.
`AcknowledgementInterval`	In WS-RM, acknowledgements are sent on return messages or sent stand-alone. If a return message is not available to send an acknowledgement, an RM Destination can wait for up to the acknowledgement interval before sending a stand-alone acknowledgement. If there are no unacknowledged messages, the RM Destination can choose not to send an acknowledgement.

More detailed reference information

For more detailed reference information, including descriptions of each element's sub-elements and attributes, please refer to http://schemas.xmlsoap.org/ws/2005/02/rm/wsrm-policy.xsd.

RMAssertion in rmManager Spring bean

You can configure standard WS-RM policy attributes by adding an **`RMAssertion`** within a Apache CXF **`rmManager`** Spring bean. This is the best approach if you want to keep all of your WS-RM configuration in the same configuration file; that is, if you want to configure Apache CXF-specific attributes and standard WS-RM policy attributes in the same file.

For example, the configuration in Example 36.5, "Configuring WS-RM Attributes Using an RMAssertion in an rmManager Spring Bean" shows:

- A standard WS-RM policy attribute, **`BaseRetransmissionInterval`**, configured using an **`RMAssertion`** within an **`rmManager`** Spring bean.
- An Apache CXF-specific RM attribute, **`intraMessageThreshold`**, configured in the same configuration file.

Example 36.5. Configuring WS-RM Attributes Using an RMAssertion in an rmManager Spring Bean

```
<beans xmlns:wsrm-
policy="http://schemas.xmlsoap.org/ws/2005/02/rm/policy"
       xmlns:wsrm-mgr="http://cxf.apache.org/ws/rm/manager"
...>
<wsrm-mgr:rmManager id="org.apache.cxf.ws.rm.RMManager">
    <wsrm-policy:RMAssertion>
        <wsrm-policy:BaseRetransmissionInterval Milliseconds="4000"/>
    </wsrm-policy:RMAssertion>
    <wsrm-mgr:destinationPolicy>
        <wsrm-mgr:acksPolicy intraMessageThreshold="0" />
    </wsrm-mgr:destinationPolicy>
</wsrm-mgr:rmManager>
</beans>
```

Policy within a feature

You can configure standard WS-RM policy attributes within features, as shown in Example 36.6, "Configuring WS-RM Attributes as a Policy within a Feature".

Example 36.6. Configuring WS-RM Attributes as a Policy within a Feature

```
<xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:wsa="http://cxf.apache.org/ws/addressing"
       xmlns:wsp="http://www.w3.org/2006/07/ws-policy"
       xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-
wss-wssecurity-utility-1.0.xsd"
       xmlns:jaxws="http://cxf.apache.org/jaxws"
```

```
        xsi:schemaLocation="
http://www.w3.org/2006/07/ws-policy http://www.w3.org/2006/07/ws-
policy.xsd
http://cxf.apache.org/ws/addressing
http://cxf.apache.org/schema/ws/addressing.xsd
http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
    <jaxws:endpoint name="
{http://cxf.apache.org/greeter_control}GreeterPort"
createdFromAPI="true">
        <jaxws:features>
               <wsp:Policy>
                   <wsrm:RMAssertion
xmlns:wsrm="http://schemas.xmlsoap.org/ws/2005/02/rm/policy">
                     <wsrm:AcknowledgementInterval Milliseconds="200"
/>
                   </wsrm:RMAssertion>
                   <wsam:Addressing
xmlns:wsam="http://www.w3.org/2007/02/addressing/metadata">
                       <wsp:Policy>
                            <wsam:NonAnonymousResponses/>
                       </wsp:Policy>
                   </wsam:Addressing>
              </wsp:Policy>
        </jaxws:features>
    </jaxws:endpoint>
</beans>
```

WSDL file

If you use the WS-Policy framework to enable WS-RM, you can configure standard WS-RM policy attributes in a WSDL file. This is a good approach if you want your service to interoperate and use WS-RM seamlessly with consumers deployed to other policy-aware Web services stacks.

For an example, see the section called “WS-Policy framework—implicitly adding interceptors” where the base retransmission interval is configured in the WSDL file.

External attachment

You can configure standard WS-RM policy attributes in an external attachment file. This is a good approach if you cannot, or do not want to, change your WSDL file.

Example 36.7, “Configuring WS-RM in an External Attachment” shows an external attachment that enables both WS-A and WS-RM (base retransmission interval of 30 seconds) for a specific EPR.

Example 36.7. Configuring WS-RM in an External Attachment

```
<attachments xmlns:wsp="http://www.w3.org/2006/07/ws-policy"
xmlns:wsa="http://www.w3.org/2005/08/addressing">
    <wsp:PolicyAttachment>
        <wsp:AppliesTo>
           <wsa:EndpointReference>
```

```
<wsa:Address>http://localhost:9020/SoapContext/GreeterPort</wsa:Address>
            </wsa:EndpointReference>
        </wsp:AppliesTo>
        <wsp:Policy>
            <wsam:Addressing
xmlns:wsam="http://www.w3.org/2007/02/addressing/metadata">
                <wsp:Policy/>
            </wsam:Addressing>
            <wsrmp:RMAssertion
xmlns:wsrmp="http://schemas.xmlsoap.org/ws/2005/02/rm/policy">
                <wsrmp:BaseRetransmissionInterval
Milliseconds="30000"/>
            </wsrmp:RMAssertion>
        </wsp:Policy>
    </wsp:PolicyAttachment>
</attachments>/
```

36.4.3. WS-RM Configuration Use Cases

Overview

This subsection focuses on configuring WS-RM attributes from a use case point of view. Where an attribute is a standard WS-RM policy attribute, defined in the http://schemas.xmlsoap.org/ws/2005/02/rm/policy namespace, only the example of setting it in an **`RMAssertion`** within an **`rmManager`** Spring bean is shown. For details of how to set such attributes as a policy within a feature; in a WSDL file, or in an external attachment, see Section 36.4.2, “Configuring Standard WS-RM Policy Attributes”.

The following use cases are covered:

- Base retransmission interval
- Exponential backoff for retransmission
- Acknowledgement interval
- Maximum unacknowledged messages threshold
- Maximum length of an RM sequence
- Message delivery assurance policies

Base retransmission interval

The **`BaseRetransmissionInterval`** element specifies the interval at which an RM source retransmits a message that has not yet been acknowledged. It is defined in the http://schemas.xmlsoap.org/ws/2005/02/rm/wsrm-policy.xsd schema file. The default value is 3000 milliseconds.

Example 36.8, “Setting the WS-RM Base Retransmission Interval” shows how to set the WS-RM base retransmission interval.

Example 36.8. Setting the WS-RM Base Retransmission Interval

```
<beans xmlns:wsrm-
policy="http://schemas.xmlsoap.org/ws/2005/02/rm/policy
...>
<wsrm-mgr:rmManager id="org.apache.cxf.ws.rm.RMManager">
    <wsrm-policy:RMAssertion>
        <wsrm-policy:BaseRetransmissionInterval Milliseconds="4000"/>
    </wsrm-policy:RMAssertion>
</wsrm-mgr:rmManager>
</beans>
```

Exponential backoff for retransmission

The **`ExponentialBackoff`** element determines if successive retransmission attempts for an unacknowledged message are performed at exponential intervals.

The presence of the **`ExponentialBackoff`** element enables this feature. An exponential backoff ratio of **2** is used by default.

Example 36.9, “Setting the WS-RM Exponential Backoff Property” shows how to set the WS-RM exponential backoff for retransmission.

Example 36.9. Setting the WS-RM Exponential Backoff Property

```
<beans xmlns:wsrm-
policy="http://schemas.xmlsoap.org/ws/2005/02/rm/policy
...>
<wsrm-mgr:rmManager id="org.apache.cxf.ws.rm.RMManager">
    <wsrm-policy:RMAssertion>
        <wsrm-policy:ExponentialBackoff="4"/>
    </wsrm-policy:RMAssertion>
</wsrm-mgr:rmManager>
</beans>
```

Acknowledgement interval

The **`AcknowledgementInterval`** element specifies the interval at which the WS-RM destination sends asynchronous acknowledgements. These are in addition to the synchronous acknowledgements that it sends on receipt of an incoming message. The default asynchronous acknowledgement interval is **`0`** milliseconds. This means that if the **`AcknowledgementInterval`** is not configured to a specific value, acknowledgements are sent immediately (that is, at the first available opportunity).

Asynchronous acknowledgements are sent by the RM destination only if both of the following conditions are met:

- The RM destination is using a non-anonymous **`wsrm:acksTo`** endpoint.
- The opportunity to piggyback an acknowledgement on a response message does not occur before the expiry of the acknowledgement interval.

Example 36.10, “Setting the WS-RM Acknowledgement Interval” shows how to set the WS-RM acknowledgement interval.

Example 36.10. Setting the WS-RM Acknowledgement Interval

```
<beans xmlns:wsrm-
policy="http://schemas.xmlsoap.org/ws/2005/02/rm/policy
...>
<wsrm-mgr:rmManager id="org.apache.cxf.ws.rm.RMManager">
    <wsrm-policy:RMAssertion>
        <wsrm-policy:AcknowledgementInterval Milliseconds="2000"/>
    </wsrm-policy:RMAssertion>
</wsrm-mgr:rmManager>
</beans>
```

Maximum unacknowledged messages threshold

The **maxUnacknowledged** attribute sets the maximum number of unacknowledged messages that can accrue per sequence before the sequence is terminated.

Example 36.11, “Setting the WS-RM Maximum Unacknowledged Message Threshold” shows how to set the WS-RM maximum unacknowledged messages threshold.

Example 36.11. Setting the WS-RM Maximum Unacknowledged Message Threshold

```
<beans xmlns:wsrm-mgr="http://cxf.apache.org/ws/rm/manager
...>
<wsrm-mgr:reliableMessaging>
    <wsrm-mgr:sourcePolicy>
        <wsrm-mgr:sequenceTerminationPolicy maxUnacknowledged="20" />
    </wsrm-mgr:sourcePolicy>
</wsrm-mgr:reliableMessaging>
</beans>
```

Maximum length of an RM sequence

The **maxLength** attribute sets the maximum length of a WS-RM sequence. The default value is **0**, which means that the length of a WS-RM sequence is unbound.

When this attribute is set, the RM endpoint creates a new RM sequence when the limit is reached, and after receiving all of the acknowledgements for the previously sent messages. The new message is sent using a newsequence.

Example 36.12, “Setting the Maximum Length of a WS-RM Message Sequence” shows how to set the maximum length of an RM sequence.

Example 36.12. Setting the Maximum Length of a WS-RM Message Sequence

```
<beans xmlns:wsrm-mgr="http://cxf.apache.org/ws/rm/manager
...>
<wsrm-mgr:reliableMessaging>
    <wsrm-mgr:sourcePolicy>
        <wsrm-mgr:sequenceTerminationPolicy maxLength="100" />
```

```
    </wsrm-mgr:sourcePolicy>
</wsrm-mgr:reliableMessaging>
</beans>
```

Message delivery assurance policies

You can configure the RM destination to use the following delivery assurance policies:

- **`AtMostOnce`** — The RM destination delivers the messages to the application destination only once. If a message is delivered more than once an error is raised. It is possible that some messages in a sequence may not be delivered.
- **`AtLeastOnce`** — The RM destination delivers the messages to the application destination at least once. Every message sent will be delivered or an error will be raised. Some messages might be delivered more than once.
- **`InOrder`** — The RM destination delivers the messages to the application destination in the order that they are sent. This delivery assurance can be combined with the **`AtMostOnce`** or **`AtLeastOnce`** assurances.

Example 36.13, “Setting the WS-RM Message Delivery Assurance Policy” shows how to set the WS-RM message delivery assurance.

Example 36.13. Setting the WS-RM Message Delivery Assurance Policy

```
<beans xmlns:wsrm-mgr="http://cxf.apache.org/ws/rm/manager
...>
<wsrm-mgr:reliableMessaging>
    <wsrm-mgr:deliveryAssurance>
        <wsrm-mgr:AtLeastOnce />
    </wsrm-mgr:deliveryAssurance>
</wsrm-mgr:reliableMessaging>
</beans>
```

36.5. CONFIGURING WS-RM PERSISTENCE

Overview

The Apache CXF WS-RM features already described in this chapter provide reliability for cases such as network failures. WS-RM persistence provides reliability across other types of failure such as an RM source or an RM destination crash.

WS-RM persistence involves storing the state of the various RM endpoints in persistent storage. This enables the endpoints to continue sending and receiving messages when they are reincarnated.

Apache CXF enables WS-RM persistence in a configuration file. The default WS-RM persistence store is JDBC-based. For convenience, Apache CXF includes Derby for out-of-the-box deployment. In addition, the persistent store is also exposed using a Java API.

IMPORTANT

WS-RM persistence is supported for oneway calls only, and it is disabled by default.

How it works

Apache CXF WS-RM persistence works as follows:

- At the RM source endpoint, an outgoing message is persisted before transmission. It is evicted from the persistent store after the acknowledgement is received.
- After a recovery from crash, it recovers the persisted messages and retransmits until all the messages have been acknowledged. At that point, the RM sequence is closed.
- At the RM destination endpoint, an incoming message is persisted, and upon a successful store, the acknowledgement is sent. When a message is successfully dispatched, it is evicted from the persistent store.
- After a recovery from a crash, it recovers the persisted messages and dispatches them. It also brings the RM sequence to a state where new messages are accepted, acknowledged, and delivered.

Enabling WS-persistence

To enable WS-RM persistence, you must specify the object implementing the persistent store for WS-RM. You can develop your own or you can use the JDBC based store that comes with Apache CXF.

The configuration shown in Example 36.14, “Configuration for the Default WS-RM Persistence Store” enables the JDBC-based store that comes with Apache CXF.

Example 36.14. Configuration for the Default WS-RM Persistence Store

```
<bean id="RMTxStore"
class="org.apache.cxf.ws.rm.persistence.jdbc.RMTxStore"/>
<wsrm-mgr:rmManager id="org.apache.cxf.ws.rm.RMManager">
    <property name="store" ref="RMTxStore"/>
</wsrm-mgr:rmManager>
```

Configuring WS-persistence

The JDBC-based store that comes with Apache CXF supports the properties shown in Table 36.4, “JDBC Store Properties”.

Table 36.4. JDBC Store Properties

Attribute Name	Type	Default Setting
driverClassName	String	`org.apache.derby.jdbc.EmbeddedDriver`
userName	String	null

Attribute Name	Type	Default Setting
passWord	String	null
url	String	jdbc:derby:rmdb;create=true

The configuration shown in Example 36.15, “Configuring the JDBC Store for WS-RM Persistence” enables the JDBC-based store that comes with Apache CXF, while setting the driverClassName and url to non-default values.

Example 36.15. Configuring the JDBC Store for WS-RM Persistence

```
<bean id="RMTxStore"
class="org.apache.cxf.ws.rm.persistence.jdbc.RMTxStore">
    <property name="driverClassName" value="com.acme.jdbc.Driver"/>
    <property name="url" value="jdbc:acme:rmdb;create=true"/>
</bean>
```

APPENDIX F. CONSUMER ENDPOINT PROPERTIES

The attributes described in Table F.1, “Consumer Endpoint Attributes” are used to configure a consumer endpoint.

Table F.1. Consumer Endpoint Attributes

Name	Type	Description	Required
wsdl	String	Specifies the location of the WSDL defining the endpoint.	yes
service	QName	Specifies the service name of the proxied endpoint. This corresponds to WSDL **service** element's **name** attribute.	no[a]
endpoint	String	Specifies the endpoint name of the proxied endpoint. This corresponds to WSDL **port** element's **name** attribute.	no[b]
interfaceName	QName	Specifies the interface name of the proxied endpoint. This corresponds to WSDL **portType** element's **name** attribute.	no
targetService	QName	Specifies the service name of the target endpoint.	no (defaults to the value of the **service** attribute)
targetEndpoint	String	Specifies the endpoint name of the target endpoint.	no (defaults to the value of the **endpoint** attribute)
targetInterfaceName	QName	Specifies the interface name of the target endpoint.	no
busCfg	String	Specifies the location of a spring configuration file used for Apache CXF bus initialization.	no

Name	Type	Description	Required
`mtomEnabled`	boolean	Specifies if MTOM / attachment support is enabled.	no (defaults to **`false`**)
`useJbiWrapper`	boolean	Specifies if the JBI wrapper is sent in the body of the message.	no (defaults to **`true`**)
`timeout`	int	Specifies the number of seconds to wait for a response.	no (defaults to **`10`**

[a] If the WSDL defining the service has more than one **`service`** element, this attribute is required.

[b] If the service being used defines more than one endpoint, this attribute is required.

APPENDIX G. PROVIDER ENDPOINT PROPERTIES

The attributes described in Table G.1, “Provider Endpoint Attributes” are used to configure a provider endpoint.

Table G.1. Provider Endpoint Attributes

Attribute	Type	Description	Required
`wsdl`	String	Specifies the location of the WSDL defining the endpoint.	yes
`service`	QName	Specifies the service name of the exposed endpoint.	no[a]
`endpoint`	String	Specifies the endpoint name of the exposed endpoint.	no[b]
`locationURI`	URI	Specifies the URL of the target service.	no[c][d]
`interfaceName`	QName	Specifies the interface name of the exposed jbi endpoint.	no
`busCfg`	String	Specifies the location of the spring configuration file used for Apache CXF bus initialization.	no
`mtomEnabled`	boolean	Specifies if MTOM / attachment support is enabled.	no (defaults to **`false`**)
`useJbiWrapper`	boolean	Specifies if the JBI wrapper is sent in the body of the message.	no (defaults to **`true`**)

[a] If the WSDL defining the service has more than one **`service`** element, this attribute is required.

[b] If the service being used defines more than one endpoint, this attribute is required.

[c] If specified, the value of this attribute overrides the HTTP address specified in the WSDL contract.

[d] This attribute is ignored if the endpoint uses a JMS address in the WSDL.

APPENDIX H. USING THE MAVEN OSGI TOOLING

Abstract

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 Red Hat JBoss Fuse OSGi tooling uses the Maven bundle plug-in from Apache Felix. The bundle plug-in is based on the **bnd** 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:

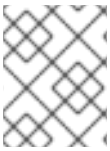
1. Add the bundle plug-in to your project's POM file.
2. Configure the plug-in to correctly populate your bundle's manifest.

H.1. SETTING UP A RED HAT JBOSS FUSE 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.



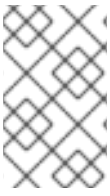
NOTE

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

Red Hat JBoss Fuse 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 H.1, “Adding an OSGi bundle plug-in to a POM” shows the POM entries required to add the bundle plug-in to your project.

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

```
...
<dependencies>
1   <dependency>
      <groupId>org.apache.felix</groupId>
    <artifactId>org.osgi.core</artifactId>
    <version>1.0.0</version>
  </dependency>
...
</dependencies>
...
<build>
  <plugins>
2    <plugin>
       <groupId>org.apache.felix</groupId>
     <artifactId>maven-bundle-plugin</artifactId>
     <configuration>
       <instructions>
         <Bundle-SymbolicName>${pom.artifactId}</Bundle-SymbolicName>
3          <Import-Package>*,org.apache.camel.osgi</Import-Package>
4            <Private-
5 Package>org.apache.servicemix.examples.camel</Private-Package>
             </instructions>
     </configuration>
    </plugin>
  </plugins>
</build>
...
```

The entries in Example H.1, “Adding an OSGi bundle plug-in to a POM” do the following:

1. Adds the dependency on Apache Felix
2. Adds the bundle plug-in to your project
3. Configures the plug-in to use the project's artifact ID as the bundle's symbolic name
4. Configures the plug-in to include all Java packages imported by the bundled classes; also imports the org.apache.camel.osgi package
5. 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 Section H.2, “Configuring the Bundle Plug-In”.

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:

- the section called “Spring OSGi archetype”
- the section called “Apache CXF code-first archetype”
- the section called “Apache CXF wsdl-first archetype”
- the section called “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 JBoss Fuse, 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
```

H.2. 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 H.2.

Example H.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 H.3.

Example H.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 H.4.

Example H.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.

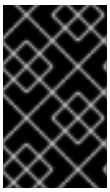
For example, to include all packages starting with com.fuse.demo except the package com.fuse.demo.private, list the packages using:

```
!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 H.5 shows the configuration for including a private package in a bundle

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

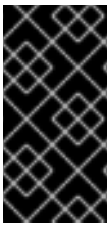
```
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <configuration>
   <instructions>
     <Private-Package>org.apache.cxf.wsdlFirst.impl</Private-Package>
     ...
    </instructions>
  </configuration>
</plugin>
```

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

Example H.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
- Peter Kriens' aQute Software Consultancy web site

INDEX

A

B

F

G

H

O

P

R

S

T

U

W

X