



Red Hat JBoss Fuse 6.0

Using Java Business Integration

Using an alternative packaging standard

Red Hat JBoss Fuse 6.0 Using Java Business Integration

Using an alternative packaging standard

JBoss A-MQ Docs Team

Content Services

fuse-docs-support@redhat.com

Legal Notice

Copyright © 2013 Red Hat.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution-Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at

<http://creativecommons.org/licenses/by-sa/3.0/>

. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat Software Collections is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

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.

Table of Contents

PART I. OVERVIEW OF JAVA BUSINESS INTEGRATION	4
CHAPTER 1. INTRODUCTION TO JBI	5
CHAPTER 2. THE COMPONENT FRAMEWORK	6
OVERVIEW	6
COMPONENT TYPES	6
PACKAGING	6
COMPONENT ROLES	7
CHAPTER 3. THE NORMALIZED MESSAGE ROUTER	8
OVERVIEW	8
MESSAGE EXCHANGE PATTERNS	8
NORMALIZED MESSAGES	8
CHAPTER 4. MANAGEMENT STRUCTURE	10
OVERVIEW	10
JMX	10
INSTALLING AND UNINSTALLING ARTIFACTS INTO THE JBI ENVIRONMENT	10
MANAGING JBI COMPONENTS	11
MANAGING SERVICE UNITS	12
PART II. DEPLOYING JBI ARTIFACTS INTO THE RED HAT JBOSS FUSE RUNTIME	13
CHAPTER 5. CLUSTERING JBI ENDPOINTS	14
OVERVIEW	14
FEATURES	14
STEPS TO SET UP CLUSTERING	14
INSTALLING THE CLUSTERING FEATURE	15
DEFAULT CLUSTERING ENGINE CONFIGURATION	15
CHANGING THE DEFAULT CONFIGURATION	16
CHANGING THE JMS BROKER	16
USING CLUSTERING IN AN APPLICATION	16
ESTABLISHING NETWORK CONNECTIONS BETWEEN CONTAINERS	17
HIGH AVAILABILITY	18
CLUSTER CONFIGURATION CONVENTIONS	18
CHAPTER 6. USING THE JBI ANT TASKS	20
6.1. USING THE TASKS AS COMMANDS	20
6.2. USING THE TASKS IN BUILD FILES	25
CHAPTER 7. BUILDING JBI COMPONENTS USING MAVEN	33
OVERVIEW	33
SETTING UP THE MAVEN TOOLS	33
CREATING A JBI MAVEN PROJECT	34
JBI COMPONENTS	34
SHARED LIBRARIES	36
CHAPTER 8. DEPLOYING JBI ENDPOINTS USING MAVEN	37
8.1. SETTING UP A RED HAT JBOSS FUSE JBI PROJECT	37
8.2. A SERVICE UNIT PROJECT	41
8.3. A SERVICE ASSEMBLY PROJECT	46
APPENDIX A. USING THE JBI CONSOLE COMMANDS	50

ACCESSING THE JBI COMMANDS	50
COMMANDS	50
INDEX	50

PART I. OVERVIEW OF JAVA BUSINESS INTEGRATION



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.



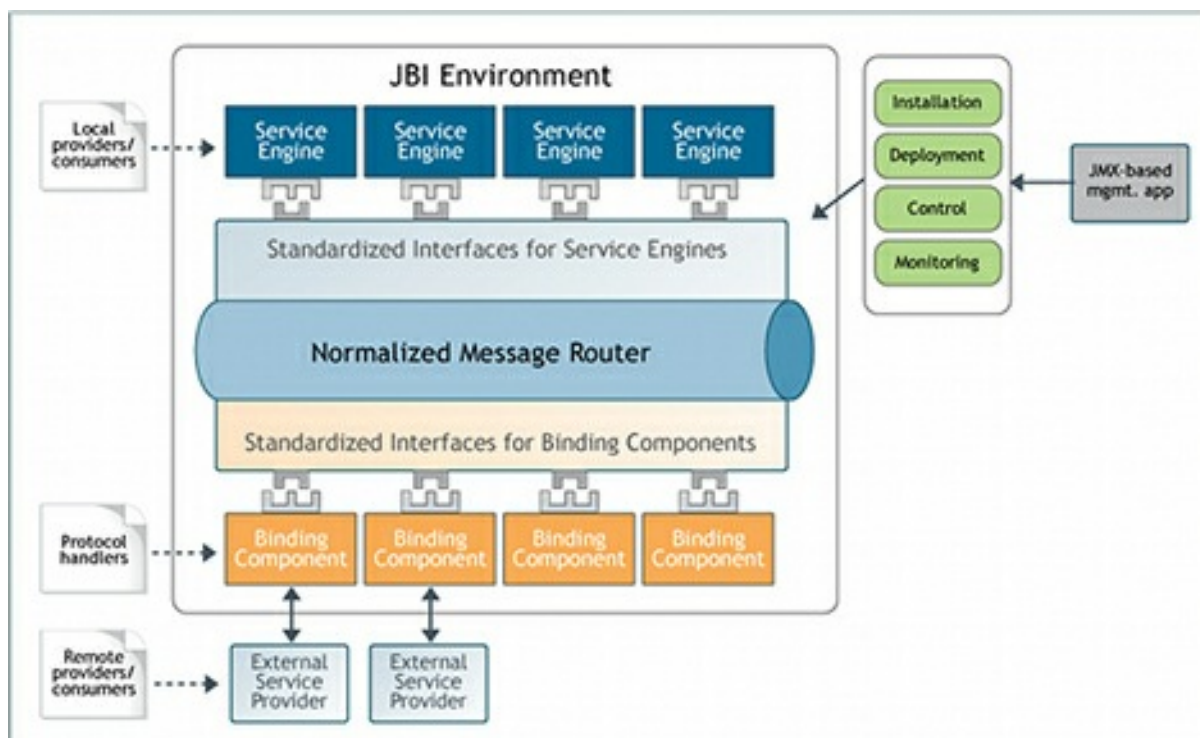
IMPORTANT

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.

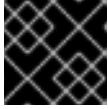
PACKAGING

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 occurred 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 **Subject**, 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`

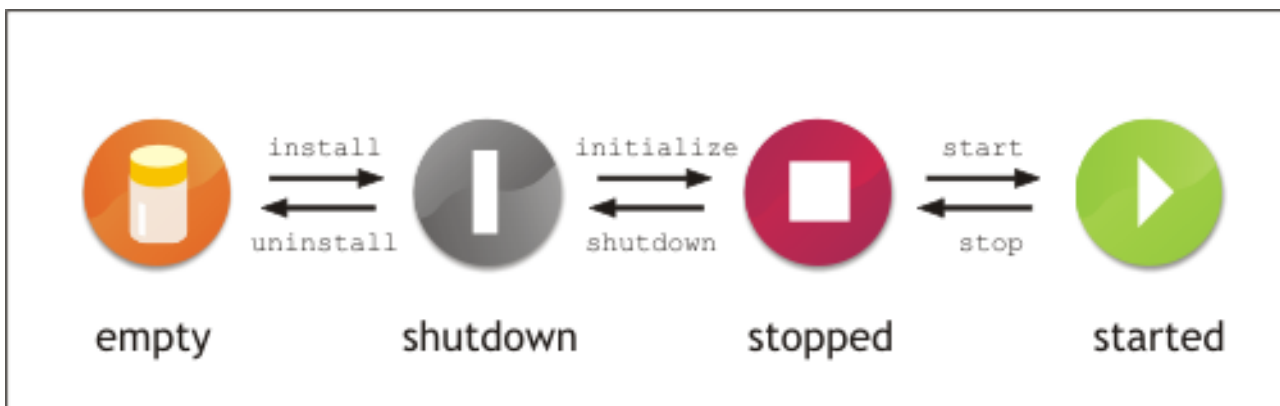
When a service assembly is installed into a JBI environment, all service units contained within the assembly are deployed to their respective JBI components. Service assemblies and service units are installed using the `DeploymentService` MBean that is exposed through the JMX console. In addition to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

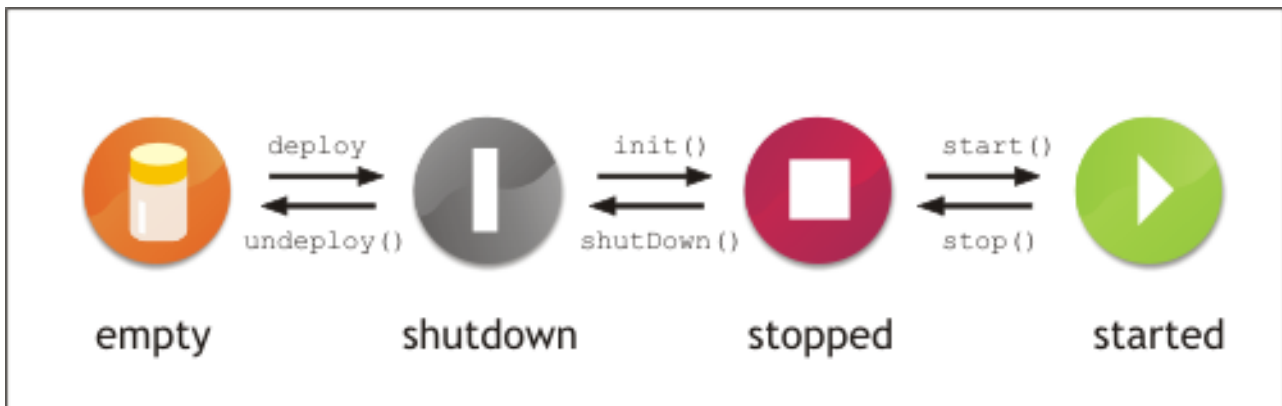
Components can be moved back and forth through the shutdown, stopped, and started states without being uninstalled. You can manage the lifecycle of an installed JBI component using the `InstallationService` MBean and the component's `ComponentLifeCycle` MBean. In addition, you can manage a component's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

PART II. DEPLOYING JBI ARTIFACTS INTO THE RED HAT JBOSS FUSE RUNTIME

Abstract

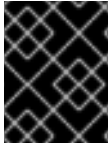
The Red Hat JBoss Fuse runtime is a container into which you deploy services. You must also deploy components to the container to support those services. Red Hat JBoss Fuse supports the JBI packaging and deployment model for deploying functionality into the runtime.



IMPORTANT

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

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 ActiveMQ. 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:

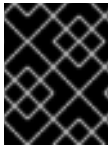
```
<property name="url"
          value="jdbc:mysql://localhost/broker_activemq_host1?
          relaxAutoCommit=true"/>
```

- 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 [-Doption=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 *-Doption=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
<code>sm.username</code>	no	Specifies the username used to access the management features of the Red Hat JBoss Fuse container
<code>sm.password</code>	no	Specifies the password used to access the management features of the Red Hat JBoss Fuse container
<code>sm.host</code>	no	Specifies the host name where the container is running; the default value is <code>localhost</code>
<code>sm.port</code>	no	Specifies the port where the container's RMI registry is listening; the default value is <code>1099</code>

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

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

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

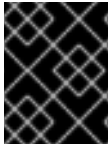
```
<project ...>
  ...
  <groupId>org.apache.servicemix</groupId>
  <artifactId>MyBindingComponent</artifactId>
  <packaging>jbi-shared-library</packaging>
  ...
</project>
```

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>cxfr-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

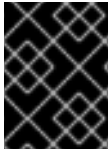
Each service unit in the service assembly must be its own project. These projects are placed at the same level as the service assembly project. The contents of a service unit's project depends on the component at which the service unit is targeted. At the minimum, a service unit project contains a POM 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 "Using the JMS Binding Component")
jms-provider	Creates a provider endpoint project targeted at the JMS binding component (see "Using the JMS Binding Component")
file-poller	Creates a polling (consumer) endpoint project targeted at the file binding component (see chapter "Using Poller Endpoints" in "Using the File Binding Component")
file-sender	Creates a sender (provider) endpoint project targeted at the file binding component (see chapter "Using Sender Endpoints" in "Using the File Binding Component")
ftp-poller	Creates a polling (consumer) endpoint project targeted at the FTP binding component

Name	Description
ftp-sender	Creates a sender (provider) endpoint project targeted at the FTP binding component
jsr181-annotated	Creates a project for developing an annotated Java service to be run by the JSR181 service engine [a]
jsr181-wsdl-first	Creates a project for developing a WSDL generated Java service to be run by the JSR181 service engine [a]
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
<p>[a] The JSR181 has been deprecated. The Apache CXF service engine has superseded it.</p> <p>[b] The EIP service engine has been deprecated. The Apache Camel service engine has superseded it.</p> <p>[c] The lightweight container has been deprecated.</p>	

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 Firrt 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>

  ❶ <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>
  ❷ <packaging>jbi-service-unit</packaging>

  ❸ <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

To 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>

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

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

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

  <build>
    <plugins>
      ❹ <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:

- ❶ 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”](#)
- ❷ Specifies that this project builds a service assembly
- ❸ Specifies the service units being bundled by the service assembly
- ❹ Specifies to use the 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
<code>jbi:list</code>	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.
<code>jbi:shutdown artifact</code>	Moves the specified artifact from the stopped state to the shutdown state.
<code>jbi:stop artifact</code>	Moves the specified artifact into the stopped state.
<code>jbi:start artifact</code>	Moves the specified artifact into the started state.

INDEX

A

Ant task

`install-component`, [Installing a component](#)

`install-shared-library`, [Installing a shared library](#)

installing components, [Installing a component](#), [Installing a component](#)

installing shared libraries, [Installing a shared library](#), [Installing a shared library](#)

`jbi-install-component`, [Installing a component](#)

`jbi-install-shared-library`, [Installing a shared library](#)

`jbi-shut-down-component`, [Shutting down a component](#)

`jbi-start-component`, [Starting a component](#)

`jbi-stop-component`, [Stopping a component](#)

jbi-uninstall-component, [Removing a component](#)

jbi-uninstall-shared-library, [Removing a shared library](#)

removing components, [Removing a component](#), [Removing a shared library](#), [Removing a component](#)

removing shared libraries, [Removing a shared library](#)

shutdown-component, [Shutting down a component](#)

shutting down components, [Shutting down a component](#), [Shutting down a component](#)

start-component, [Starting a component](#)

starting components, [Starting a component](#), [Starting a component](#)

stop-component, [Stopping a component](#)

stopping components, [Stopping a component](#), [Stopping a component](#)

uninstall-component, [Removing a component](#)

uninstall-shared-library, [Removing a shared library](#)

uninstalling components, [Removing a component](#), [Removing a shared library](#), [Removing a component](#)

B

binding component, [Component types](#)

C

clustering JBI endpoints, [Overview](#)

component life-cycle, [Managing JBI components](#)

componentName, [Specifying the target components](#)

consumer, [Component roles](#)

I

install-component, [Installing a component](#)

sm.host, [Installing a component](#)

sm.install.file, [Installing a component](#)

sm.password, [Installing a component](#)

sm.port, [Installing a component](#)

sm.username, [Installing a component](#)

install-shared-library, [Installing a shared library](#)

sm.host, [Installing a shared library](#)

sm.install.file, [Installing a shared library](#)

sm.password, [Installing a shared library](#)

sm.port, [Installing a shared library](#)

sm.username, [Installing a shared library](#)

installing components, [Installing a component](#), [Installing a component](#)

J

Java Management Extensions, [JMX](#)

JBI clustering, [Overview](#)

jbi-install-component, [Installing a component](#)

failOnError, [Installing a component](#)

file, [Installing a component](#)

host, [Installing a component](#)

password, [Installing a component](#)

port, [Installing a component](#)

username, [Installing a component](#)

jbi-install-shared-library, [Installing a shared library](#)

failOnError, [Installing a shared library](#)

file, [Installing a shared library](#)

host, [Installing a shared library](#)

password, [Installing a shared library](#)

port, [Installing a shared library](#)

username, [Installing a shared library](#)

jbi-shut-down-component, [Shutting down a component](#)

failOnError, [Shutting down a component](#)

host, [Shutting down a component](#)

name, [Shutting down a component](#)

password, [Shutting down a component](#)

port, [Shutting down a component](#)

username, [Shutting down a component](#)

jbi-start-component, [Starting a component](#)

failOnError, [Starting a component](#)

host, [Starting a component](#)

name, [Starting a component](#)

password, [Starting a component](#)

port, [Starting a component](#)

username, [Starting a component](#)

jbi-stop-component, [Stopping a component](#)

failOnError, [Stopping a component](#)

host, [Stopping a component](#)

name, [Stopping a component](#)

password, [Stopping a component](#)

port, [Stopping a component](#)

username, [Stopping a component](#)

jbi-uninstall-component, [Removing a component](#)

failOnError, [Removing a component](#)

host, [Removing a component](#)

name, [Removing a component](#)

password, [Removing a component](#)

port, [Removing a component](#)

username, [Removing a component](#)

jbi-uninstall-shared-library, [Removing a shared library](#)

failOnError, [Removing a shared library](#)

host, [Removing a shared library](#)

name, [Removing a shared library](#)

password, [Removing a shared library](#)

port, [Removing a shared library](#)

username, [Removing a shared library](#)

JMX, [JMX](#)

M

Maven tooling

binding component, [JBI components](#)

component bootstrap class, [JBI components](#)

component implementation class, [JBI components](#)

component type, [JBI components](#)

JBI component, [JBI components](#)

project creation, [Creating a JBI Maven project](#)
service engine, [JBI components](#)
set up, [Setting up the Maven tools](#), [Setting up the Maven tools](#)
shared libraries, [Shared libraries](#)

message exchange patterns, [Message exchange patterns](#)
in-only, [Message exchange patterns](#)
in-optional-out, [Message exchange patterns](#)
in-out, [Message exchange patterns](#)
robust-in-only, [Message exchange patterns](#)

P

provider, [Component roles](#)

S

service assembly, [Packaging](#)
seeding, [Seeding a project using a Maven artifact](#)
specifying the service units, [Specifying the target components](#)

service consumer, [Component roles](#)

service engine, [Component types](#)

service provider, [Component roles](#)

service unit, [Packaging](#)
seeding, [Seeding a project using a Maven artifact](#)
specifying the target component, [Specifying the target components](#)

service unit life-cycle, [Managing service units](#)

shutdown-component, [Shutting down a component](#)
sm.component.name, [Shutting down a component](#)
sm.host, [Shutting down a component](#)
sm.password, [Shutting down a component](#)
sm.port, [Shutting down a component](#)
sm.username, [Shutting down a component](#)

sm.component.name, [Removing a component](#), [Starting a component](#), [Stopping a component](#),
[Shutting down a component](#)

sm.host, [Installing a component](#), [Removing a component](#), [Starting a component](#), [Stopping a component](#), [Shutting down a component](#), [Installing a shared library](#), [Removing a shared library](#)

sm.install.file, [Installing a component](#), [Installing a shared library](#)

sm.password, [Installing a component](#), [Removing a component](#), [Starting a component](#), [Stopping a component](#), [Shutting down a component](#), [Installing a shared library](#), [Removing a shared library](#)

sm.port, [Installing a component](#), [Removing a component](#), [Starting a component](#), [Stopping a component](#), [Shutting down a component](#), [Installing a shared library](#), [Removing a shared library](#)

sm.shared.library.name, [Removing a shared library](#)

sm.username, [Installing a component](#), [Removing a component](#), [Starting a component](#), [Stopping a component](#), [Shutting down a component](#), [Installing a shared library](#), [Removing a shared library](#)

smx-arch, [Seeding a project using a Maven artifact](#) , [Seeding a project using a Maven artifact](#)

start-component, [Starting a component](#)

sm.component.name, [Starting a component](#)

sm.host, [Starting a component](#)

sm.password, [Starting a component](#)

sm.port, [Starting a component](#)

sm.username, [Starting a component](#)

stop-component, [Stopping a component](#)

sm.component.name, [Stopping a component](#)

sm.host, [Stopping a component](#)

sm.password, [Stopping a component](#)

sm.port, [Stopping a component](#)

sm.username, [Stopping a component](#)

U

uninstall-component, [Removing a component](#)

sm.component.name, [Removing a component](#)

sm.host, [Removing a component](#)

sm.password, [Removing a component](#)

sm.port, [Removing a component](#)

sm.username, [Removing a component](#)

uninstall-shared-library, [Removing a shared library](#)

sm.host, [Removing a shared library](#)

sm.password, [Removing a shared library](#)

sm.port, [Removing a shared library](#)

`sm.shared.library.name`, [Removing a shared library](#)

`sm.username`, [Removing a shared library](#)