Progress FUSE

FUSE™ ESB

Using JBI to Develop Applications

Version 4.1
April 2009

PROGRESS SOFTWARE

Using JBI to Develop Applications

Version 4.1

Publication date 22 Jul 2009

Legal Notices

Table of Contents

List of Figures

List of Tables

List of Examples

Part I. Overview of Java Business Integration(JBI)

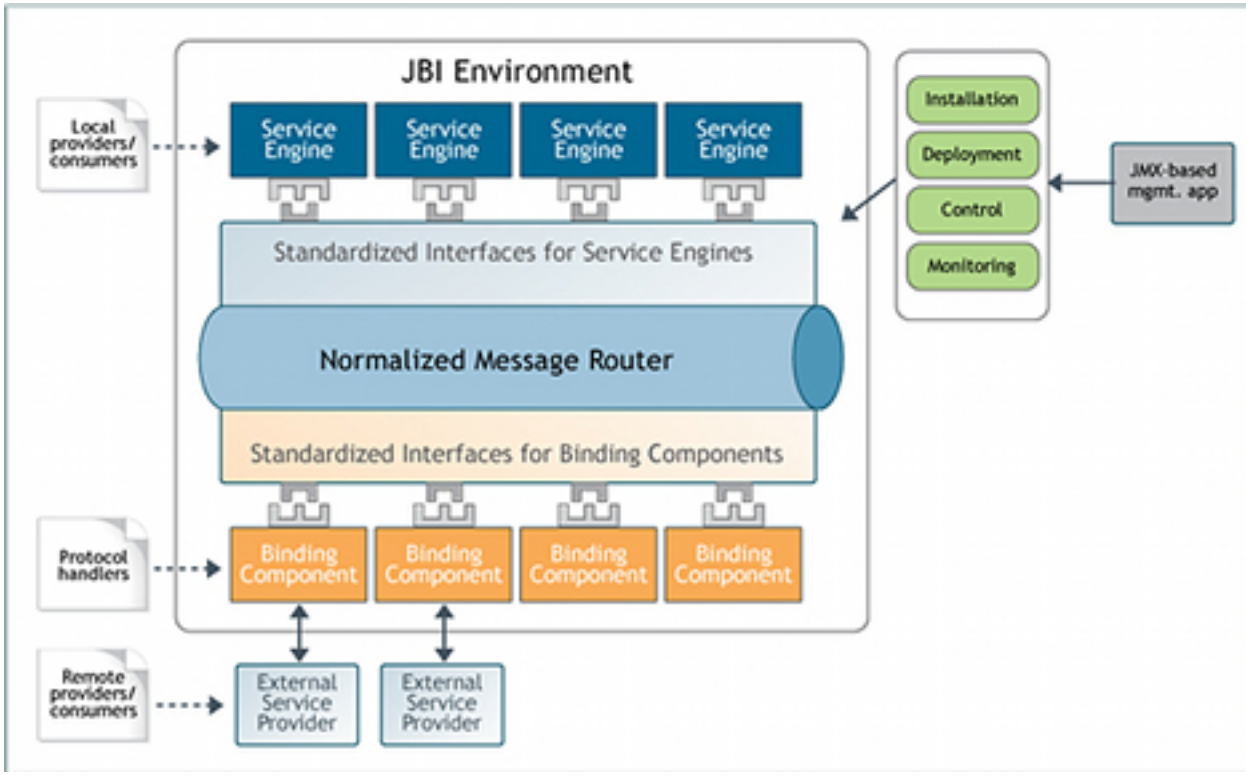
Chapter 1. Introduction to JBI

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

The Java Business Integration(JBI) specification defines an integration architecture based on service oriented concepts. Applications are broken up 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 between the JBI components.

As shown in Figure 1.1 on page 14,the JBI environment is made up of the following parts:

Figure 1.1. The JBI Architecture



- The JBI component framework hosts and manages the JBI components. For more information see on page 15.
- The normalized message router provides message mediation between the JBI components. For more information see on page 19.
- 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 deployed into the JBI environment. For more information see on page 23.

Chapter 2. The Component Framework

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

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 Engines

> Service engines are components that provide some of the logic needed to provide services, like message transformation, orchestration, or advanced message routing, inside of the JBI environment. They can only communicate with other components inside of the JBI environment. Service engines act as containers for the functional units deployed into the FUSE ESB.

Binding Components

> Binding components provide access to services outside the JBI environment via a particular protocol. They implement the logic needed 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 component is purely a matter of convention. The 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 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 of packages encountered by an application developer are:

service assembly

Service assemblies are 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

Service units are packages that contain functionality to be deployed into a JBI component. For example a service unit intended for a routing service engine would contain the definition for one or more routes. 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. In doing so, the JBI component takes on one of the following roles:

service provider

Service providers receive request messages and return response messages if required.

service consumer

Service consumers initiate message exchanges by sending requests to a service provider.

Depending on the number and 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 FUSE ESB. 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, 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

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

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 between each other. Instead, they pass messages to the NMR. The NMR 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 worrying 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 needed 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 one of four message exchange patterns:

in-out

> In this pattern a consumer sends a request message to a provider. The provider responds to the request with a response message. The provider may also respond with a fault message indicating that an error occured during processing.

in-optional-out

> In this pattern a consumer sends a request message to a provider. The provider may send a response message back to the consumer, but the consumer does not require a response. The provider may also respond with a fault message. In addition, the consumer can send a fault message to the provider.

in-only

In this pattern a consumer sends a message to a provider, but the provider does not send a response. The provider does not even send fault messages back to the consumer.

robust-in-only

In this pattern a consumer sends a message to a provider. The provider may send a fault message back to the consumer to signal an error condition, but otherwise does not respond to the consumer.

Normalized messages

In order to fully decouple the entities involved in message exchanges JBI uses *normalized messages*. A normalized message is a genericized format used to represent all of message data passed through the NMR. It consists of three parts:

meta-data, properties
: The meta-data 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
: The payload of a message is an XML document that conforms to the XML Schema definition in the WSDL document defining the message exchange. The XML document holds the substance of the message.

attachments
: Attachments hold any binary data associated with the message. For example, an attachment could be an image file sent as an attachment to SOAP message.

security `Subject`
: The security `Subject` holds security information, such as authentication credentials, associated with the message. For more information of the security `Sublect`, see Sun's API documentation[1].

JBI binding components are responsible for normalizing all of the messages placed onto the NMR. Binding components normalize messages received from external sources before passing them to the NMR, The binding component

[1] http://java.sun.com/j2se/1.5.0/docs/api/javax/security/auth/Subject.html

will also denormalize the message so that it is in the appropriate format for the external source.

Chapter 4. Management Structure

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

Overview

The JBI environment is managed using JMX. The internal components of the JBI environment provides a set of MBeans to facilitate the management of the JBI environment and the deployed components. In addition, the JBI environment also supplies a number of Apache Ant tasks managing the JBI environment.

The management of the JBI environment largely consists of the following:

- 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. The Ant tasks 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. It 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, and free, tool for managing a JBI environment. FUSE HQ(http://fusesource.com/products/fuse-hq/) is 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**

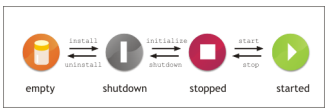
When a service assembly is installed into a JBI environment all of the service units contained within the assembly are deployed to their respective JBI components. Service assemblies and service units are installed using the `DeploymentService` MBean that is exposed through the JMX console. In addition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

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

Managing JBI components

Figure 4.1 on page 24 shows the life-cycle of a JBI component.

Figure 4.1. JBI Component Life-cycle



Components start life in a *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 needed 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.

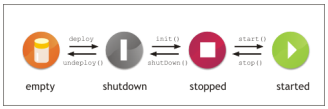
Components can be moved back and forth through the shutdown, stopped, and started states without being uninstalled. You can manage the lifecycle of an installed JBI component using the `InstallationService` MBean and the component's `ComponentLifeCycle` MBean. In addition, you can manage a component's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

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

Managing service units

Figure 4.2 on page 25 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 needed 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 unti is in the *started* state, the service unit 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 without being undeployed. 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 FUSE ESB Runtime

The FUSE ESB runtime is a container into which you deploy services. You also need to deploy components to the container to support your services. FUSE ESB supports the JBI packaging and deployment model for deploying functionality into the runtime.

Chapter 5. Using the JBI Ant Tasks

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 into the FUSE ESB container. You can use the JBI Ant tasks as either command line commands or as part of an Ant build file.

Using the Tasks as Commands

Usage

Example 5.1 on page 30 shows the basic usage statement for the FUSE ESB Ant tasks when used from the command line.

Example 5.1. JBI Ant Task Command Line Usage

`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 `-Doption=value` flag.

Installing a component

The Ant task used to install a component to the FUSE ESB container is **install-component**. Its options are described in Table 5.1 on page 30.

Table 5.1. Options for Installing a JBI Component with an Ant Command

Option	Required	Description
`sm.username`	no	Specifies the username used to access the FUSE ESB container's management features.
`sm.password`	no	Specifies the password used to access the FUSE ESB container's management features.
`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.install.file`	yes	Specifies the name of the component's installer file.

Example 5.2 on page 30 shows an example of using **install-component** to install the Camel component to a container listening on port 1000.

Example 5.2. Installing a Component Using an Ant Command

```
>ant -f ant/servicemix-ant-task.xml -Dsm.port=1000 -Dsm.in
stall.file=servicemix-camel-3.3.0.6-fuse-installer.zip install-com
ponent
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 FUSE ESB container is **uninstall-component**. Its options are described in Table 5.2 on page 31.

Table 5.2. Options for Removing a JBI Component with an Ant Command

Option	Required	Description
`sm.username`	no	Specifies the username used to access the FUSE ESB container's management features.
`sm.password`	no	Specifies the password used to access the FUSE ESB container's management features.
`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 5.3 on page 31 shows an example of using **uninstall-component** to remove the drools component from a container listening on port 1000.

Example 5.3. Removing a Component Using an Ant Command

```
>ant -f ant\servicemix-ant-task.xml -Dsm.port=1000 -Dsm.compon
ent.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 5.3 on page 32.

Table 5.3. Options for Starting a JBI Component with an Ant Command

Option	Required	Description
`sm.username`		Specifies the username used to access the FUSE ESB container's management features.
`sm.password`	no	Specifies the password used to access the FUSE ESB container's management features.
`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 5.4 on page 32 shows an example of using **start-component** to start the cxf-se component in a container listening on port 1000.

Example 5.4. Starting a Component Using an Ant Command

```
>ant -f ant\servicemix-ant-task.xml -Dsm.port=1000 -Dsm.compon
ent.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 5.4 on page 33.

Table 5.4. Options for Stopping a JBI Component with an Ant Command

Option	Required	Description
`sm.username`	no	Specifies the username used to access the FUSE ESB container's management features.
`sm.password`	no	Specifies the password used to access the FUSE ESB container's management features.
`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 5.5 on page 33 shows an example of using **stop-component** to stop the cxf-se component in a container listening on port 1000.

Example 5.5. Stopping a Component Using an Ant Command

```
>ant -f ant\servicemix-ant-task.xml -Dsm.port=1000 -Dsm.compon
ent.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 5.5 on page 34.

Table 5.5. Options for Shutting down a JBI Component with an Ant Command

Option	Required	Description
`sm.username`	no	Specifies the username used to access the FUSE ESB container's management features.
`sm.password`	no	Specifies the password used to access the FUSE ESB container's management features.
`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 FUSE ESB container is **install-shared-library**. Its options are described in Table 5.6 on page 34.

Table 5.6. Options for Installing a Shared Library with an Ant Command

Option	Required	Description
`sm.username`	no	Specifies the username used to access the FUSE ESB container's management features.
`sm.password`	no	Specifies the password used to access the FUSE ESB container's management features.
`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 library's installer file.

Removing a shared library

The Ant task used to remove a shared library from the FUSE ESB container is **uninstall-shared-library**. Its options are described in Table 5.7 on page 35.

Table 5.7. Options for Removing a Shared Library with an Ant Command

Option	Required	Description
`sm.username`	no	Specifies the username used to access the FUSE ESB container's management features.
`sm.password`	no	Specifies the password used to access the FUSE ESB container's management features.
`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.

Using the Tasks in Build Files

Adding the JBI tasks to a build 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 5.6 on page 36.

Example 5.6. Adding the JBI Tasks to an Ant Build File

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

<taskdef file="${fuseesb.install_dir}/ant/servicemix_ant_taskdef.properties"> ❷
  <classpath id="fuseesb.classpath"> ❸
    <fileset dir="${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 5.6 on page 36 does the following:

❶ Sets a property, fuseesb.install_dir, the FUSE ESB's installation directory.

❷ Loads the tasks using the `ant/servicemix_ant_taskdef.properties`.

❸ Sets the classpath so that all of the required jars from the FUSE ESB installation are available.

Installing a component

The Ant task used to install a JBI component is `jbi-install-component`. Its attributes are listed in Table 5.8 on page 36.

Table 5.8. 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 container's management features.

Attribute	Required	Description
password	no	Specifies the password used to access the container's management features.
failOnError	no	Specifies if an error will cause the entire build to fail.
file	yes	Specifies the name of the component's installer file.

Example 5.7 on page 37 shows an Ant target that installs the drools component.

Example 5.7. 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 5.9 on page 37.

Table 5.9. 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 container's management features.
password	no	Specifies the password used to access the container's management features.
failOnError	no	Specifies if an error will cause the entire build to fail.
name	yes	Specifies the component's name.

Example 5.8 on page 38 shows an Ant target that removes the drools component.

Example 5.8. 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 5.10 on page 38.

Table 5.10. 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 container's management features.
`password`	no	Specifies the password used to access the container's management features.
`failOnError`	no	Specifies if an error will cause the entire build to fail.
`name`	yes	Specifies the component's name.

Example 5.9 on page 38 shows an Ant target that starts the drools component.

Example 5.9. 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 5.11 on page 39.

Table 5.11. 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 container's management features.
`password`	no	Specifies the password used to access the container's management features.
`failOnError`	no	Specifies if an error will cause the entire build to fail.
`name`	yes	Specifies the component's name.

Example 5.10 on page 39 shows an Ant target that stops the drools component.

Example 5.10. 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 5.12 on page 39.

Table 5.12. 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 container's management features.
`password`	no	Specifies the password used to access the container's management features.

Attribute	Required	Description
failOnError	no	Specifies if an error will cause the entire build to fail.
name	yes	Specifies the component's name.

Example 5.11 on page 40 shows an Ant target that shuts down the drools component.

Example 5.11. 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 5.13 on page 40.

Table 5.13. 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 container's management features.
password	no	Specifies the password used to access the container's management features.
failOnError	no	Specifies if an error will cause the entire build to fail.
file	yes	Specifies the name of the library's installer file.

Removing a shared library

The Ant task used to remove a shared library is `jbi-uninstall-shared-library`. Its attributes are listed in Table 5.14 on page 41.

Table 5.14. 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 container's management features.
`password`	no	Specifies the password used to access the container's management features.
`failOnError`	no	Specifies if an error will cause the entire build to fail.
`name`	yes	Specifies the library's name.

Chapter 6. Building JBI Components using Maven

Overview

FUSE ESB 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 FUSE ESB Maven tooling also includes plug-ins for creating service units and service assemblies. However, those plug-ins are outside of the scope of this book.

Setting up the Maven tools

In order to use the FUSE ESB Maven tooling, you add the elements shown in Example 6.1 on page 43 to your POM file.

Example 6.1. POM Elements for Using FUSE ESB Tooling

```
...
<pluginRepositories>
  <pluginRepository>
    <id>fusesource.m2</id>
    <name>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>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>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 FUSE ESB Maven tooling and load the plug-in that implements the tooling.

Creating a JBI Maven project

The FUSE ESB Maven tooling provides a number of archetypes that can be used to seed a JBI project. The archetype will generate the proper file structure for the project and a POM file containing the essential metadata required for the specified project type.

Example 6.2 on page 44 shows the command for using the JBI archetypes.

Example 6.2. 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 wist to create.

JBI components

As shown in Example 6.3 on page 45, you instruct the FUSE ESB Maven tooling that the project is for a JBI component by specifying a value of `jbi-component` for the project's `packaging` element.

Example 6.3. 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 6.4 on page 45. The `groupId` element, the `artifactId` element, the `version` element, and the `extensions` element are common to all instances of the FUSE ESB Maven plugin. If you used the Maven archetypes to generate the project, you should not have to change them.

Example 6.4. Plugin 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 and its children provides the FUSE ESB tooling with the metadata needed to construct the `jbi.xml` file required by the component.

`type`

The `type` element 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`

The `bootstrap` element specifies the name of the class that implements the JBI `Bootstrap` interface for the component.

Tip

If you intend to use the default `Bootstrap` implementation provided with FUSE ESB you can omit this element.

`component`

The `component` element specifies the name of the class that implement the JBI `Component` interface for the component.

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

Shared libraries

As shown in Example 6.5 on page 46, you instruct the FUSE ESB Maven tooling that the project is for a shared library by specifying a value of `jbi-shared-library` for the project's `packaging` element.

Example 6.5. Specifying that a Maven Project Results in a JBI Component

```
<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 FUSE ESB Maven tooling will generate a standard jar containing the shared library and an installable JBI package for the shared library.

Chapter 7. Deploying JBI Endpoints Using Maven

FUSE ESB provides a Maven plug-in and a number of Maven archetypes that make developing, packaging, and deploying applications easier.

The tooling provides you with a number of benefits. These benefits include:

- automatic generation of JBI descriptors
- dependency checking
- service assembly deployment

Because FUSE ESB only allows you to deploy service assemblies, you will need to do at least the following when using the Maven tooling:

1. Set up a top-level project on page 50 to build all of the service units and the final service assembly.
2. Create a project for each of your service units. on page 55.
3. Create a project for the service assembly on page 61.

Setting Up a FUSE ESB JBI Project

Overview

When working with the FUSE ESB JBI Maven tooling, you will want to create a top-level project that can build all of the service units and 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 also gives you a central location from which to build and deploy the application.

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

Directory structure

Your top-level project will contain the following directories:

- a source directory containing the information needed by the Maven assembly plug-in
- a directory to hold 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

In order to use the FUSE ESB JBI Maven tooling, you add the elements shown in Example 7.1 on page 50 to your top-level POM file.

Example 7.1. POM Elements for Using FUSE ESB Tooling

```
...
<pluginRepositories>
  <pluginRepository>
    <id>fusesource.m2</id>
    <name>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>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>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 FUSE ESB Maven tooling and load the plug-in that implements the tooling.

Listing the subprojects

Your top-level POM lists all of the service units and the service assembly that will be 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 will also need a `module` element for the service assembly.

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

Example JBI Project POM

Example 7.2 on page 52 shows a top-level pom for a project that contains a single service unit.

Example 7.2. Top-Level POM for a FUSE ESB JBI Project

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

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

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

  <pluginRepositories> ❶
    <pluginRepository>
      <id>fusesource.m2</id>
      <name>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>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>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>

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

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

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

The POM shown in Example 7.2 on page 52 does the following:

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

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

❸ Configures the Maven assembly plug-in.

❹ Loads the FUSE ESB JBI plug-in.

A Service Unit Project

Overview

Each service unit in the service assembly needs to 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 a minimum, a service unit project will contain a POM and an XML configuration file.

Seeding a project using a Maven artifact

FUSE ESB provides Maven artifacts for a number of service unit types. You can use them to seed a project with the **smx-arch** command. As shown in Example 7.3 on page 55, 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 7.3. 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 7.1 on page 55 lists the possible values and describes what type of project will be seeded.

Table 7.1. Service Unit Archetypes

Name	Description
camel	Creates a project for using the FUSE Mediation Router service engine.
cxf-se	Creates a project for developing a Java-first service using the FUSE Services Framework service engine.
cxf-se-wsdl-first	Creates a project for developing a WSDL-first service using the FUSE Services Framework service engine.
cxf-bc	Creates an endpoint project targeted at the FUSE Services Framework binding component.

Name	Description
http-consumer	Creates a consumer endpoint project targeted at the HTTP binding component.
http-provider	Creates a provider endpoint project targeted at the HTTP binding component.
jms-consumer	Creates a consumer endpoint project targeted at the JMS binding component. See *Using the JMS Binding Component*.
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 "Using Poller Endpoints" in *Using the File Binding Component*.
file-sender	Creates a sender (provider) endpoint project targeted at the file binding component. See "Using Sender Endpoints" in *Using the File Binding Component*.
ftp-poller	Creates a polling (consumer) endpoint project targeted at the FTP binding component.
ftp-sender	Creates a sender (provider) endpoint project targeted at the FTP binding component.
jsr181-annotated	Creates a project for developing an annotated Java service to be run by the JSR181 service engine. [a]
jsr181-wsdl-first	Creates a project for developing a WSDL generated Java service to be run by the JSR181 service engine.[a]
saxon-xquery	Create a project for executing xquery statements using the Saxon service engine.
saxon-xslt	Create a project for executing XSLT scripts using the Saxon service engine.
eip	Creates a project for using the EIP service engine. [b]
lwcontainer	Create 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.

Name	Description
ode	Create a project for deploying a BPEL process into the ODE service engine.

[a]The JSR181 has been deprecated. The FUSE Services Framework service engine has superseded it.

[b]The EIP service engine has been deprecated. The FUSE Mediation Router 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 FUSE Services Framework 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 FUSE Mediation Router 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 7.4 on page 57.

Example 7.4. 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

In order to properly fill in the metadata required for packaging a service unit, the Maven plug-in needs to 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 need to configure the project as follows:

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

Example 7.5 on page 58 shows configuration for a service unit targeting the FUSE Services Framework binding component.

Example 7.5. 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 check if the targeted component is deployed in the container. If one of the components is not deployed, FUSE ESB will not hold off deploying the service unit until all of the required components are deployed.

[1] You replace this with the version of FUSE Services Framework you are using.

Tip

A message identifying the missing component(s) is typically written to the log.

If your service unit targets 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 specifies the name of a targeted component. Example 7.6 on page 59 shows how to use the `componentName` element to specify the target component.

Example 7.6. Specifying the Target Components 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. Maven also cannot download the required component.

Example

Example 7.7 on page 59 shows the POM file for a project building a service unit targeted to the FUSE Services Framework binding component.

Example 7.7. POM for a Service Unit Project

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

    <modelVersion>4.0.0</modelVersion>

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

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

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

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

The POM in Example 7.7 on page 59 does the following:

❶ Specifies that it is a part of the top-level project described in Example 7.2 on page 52.

❷ Specifies that this project builds a service unit.

❸ Specifies that the service unit targets the FUSE Services Framework binding component.

❹ Specifies that the FUSE ESB Maven plug-in is to be used.

A Service Assembly Project

Overview

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

Seeding a project using a Maven artifact

FUSE ESB provides a Maven artifact for seeding a service assembly project. You can seed a project with the **smx-arch** command. As shown in Example 7.8 on page 61, the **smx-arch** command takes two arguments. The `groupId` value and the `artifactId` values correspond to the project's group ID and artifact ID.

Example 7.8. 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

You configure the Maven plug-in to package the results of the project build as a service assembly by changing the value of the project's `packaging` element to `jbi-service-assembly` as shown in Example 7.9 on page 61.

Example 7.9. 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 needs to be told what service units are being bundled into the service assembly. You do this by specifying the service units as a dependencies using the standard Maven `dependencies` element. You add a `dependency` child element for each service unit. Example 7.10 on page 62 shows configuration for a service assembly that bundles two service units.

Example 7.10. 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 7.11 on page 62 shows the POM file for a project building a service assembly.

Example 7.11. POM for a Service Assembly Project

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

    <modelVersion>4.0.0</modelVersion>

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

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

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

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

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

The POM in Example 7.11 on page 62 does the following:

❶ Specifies that it is a part of the top-level project described in Example 7.2 on page 52.

❷ Specifies that this project builds a service assembly.

❸ Specifies the service units the service assembly bundles.

❹ Specifies that the FUSE ESB Maven plug-in is to be used.

Appendix A. Using the JBI Console Commands

Accessing the JBI shell

The FUSE ESB console provides a shell to manage JBI artifacts that are deployed in the container. You access the JBI shell by typing **jbi** at the `servicemix>` prompt.

You can also access the JBI shell commands by prefixing the command with **jbi**.

Commands

Table A.1 on page 65 describes the commands available from the JBI shell.

Table A.1. JBI Shell Commands

Command	Description
list	Lists all of the JBI artifacts deployed into the FUSE ESB container. The list is separated into JBI components and JBI service assemblies. It displays the name of the artifact and its life-cycle state.
ls	Performs the same action as **list**.
shutdown *`artifact`*	Moves the specified artifact from the stopped state to the shutdown state.
stop *`artifact`*	Moves the specified artifact into the stopped state.
start *`artifact`*	Moves the specified artifact into the started state.

Index

M

P

S

U