Red Hat JBoss Fuse 6.0

Deploying into the Container

Getting application packages into the container

Last Updated: 2017-10-13

Red Hat JBoss Fuse 6.0 Deploying into the Container

Getting application packages into the container

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

Legal Notice

Abstract

The guide describes the options for deploying applications into a Red Hat JBoss Fuse container.

Table of Contents

PART I. THE RED HAT JBOSS FUSE CONTAINER

Abstract

The Red Hat JBoss Fuse container is a flexible container that supports a variety of different deployment models: FAB deployment, WAR deployment, and OSGi bundle deployment. The container is also integrated with Apache Maven, so that required artifacts can be downloaded and installed dynamically at deploy time.

CHAPTER 1. RED HAT JBOSS FUSE OVERVIEW

08/15/12

Added section to introduce Fuse Fabric

Abstract

Red Hat JBoss Fuse is a flexible container that allows you to deploy applications in a range of different package types (FAB, WAR, OSGi bundle, or JBI SA) and has support for both synchronous and asynchronous communication.

1.1. RED HAT JBOSS FUSE CONTAINER ARCHITECTURE

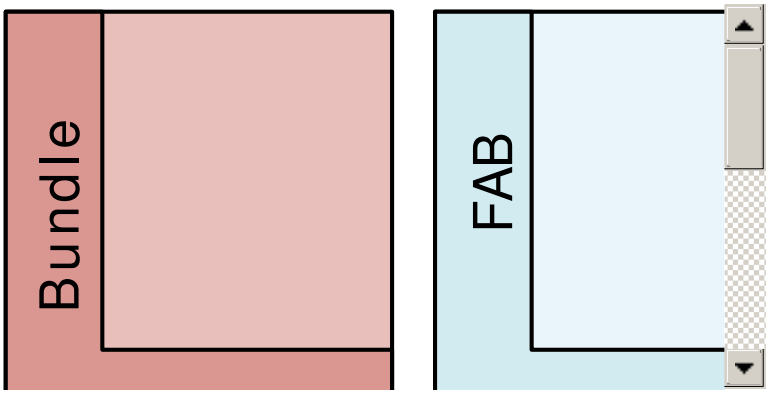
09/27/12

Reordered the deployment models to make OSGi more prominent

Overview

Figure 1.1, “Red Hat JBoss Fuse Container Architecture” shows a high-level overview of the Red Hat JBoss Fuse container architecture, showing the variety of deployment models that are supported.

Figure 1.1. Red Hat JBoss Fuse Container Architecture



Deployment models

Red Hat JBoss Fuse is a multi-faceted container that supports a variety of deployment models. You can deploy any of the following kinds of deployment unit:

OSGi bundle

An *OSGi bundle* is a JAR file augmented with metadata in the JAR's **`META-INF/MANIFEST.MF`** file. Because the Red Hat JBoss Fuse container is fundamentally an OSGi container, the OSGi bundle is also the native format for the container. Ultimately, after deployment, all of the other deployment unit types are converted into OSGi bundles.

Fuse Application Bundle, FAB

A *Fuse Application Bundle* (FAB) is a Fuse-specific deployment unit optimised for the JBoss Fuse container. FABs are a plain JAR built using the Apache Maven. FABs address some of the issues when using OSGi such as dependency resolution and class-loader conflicts. The FAB deployer in JBoss Fuse scans the metadata in the POM and automatically downloads any dependencies needed by the bundle.

WAR

A *Web application ARchive* (WAR) is the standard archive format for applications that run inside a Web server. As originally conceived by the Java servlet specification, a WAR packages Web pages, JSP pages, Java classes, servlet code, and so on, as required for a typical Web application. More generally, however, a WAR can be any deployment unit that obeys the basic WAR packaging rules (which, in particular, require the presence of a Web application deployment descriptor, **`web.xml`**).

JBI service assembly

A Java Business Integration (JBI) service assembly is the basic unit of deployment for JBI applications. A discussion of the JBI container lies outside the scope of this document. For details, see "Using Java Business Integration".

Spring framework

The Spring framework is a popular dependency injection framework, which is fully integrated into the JBoss Fuse container. In other words, Spring enables you to create instances of Java objects and wire them together by defining a file in XML format. In addition, you can also access a wide variety of utilities and services (such as security, persistence, and transactions) through the Spring framework.

Blueprint framework

The blueprint framework is a dependency injection framework defined by the OSGi Alliance. It is similar to Spring (in fact, it was originally sponsored by SpringSource), but is a more lightweight framework that is optimized for the OSGi environment.

OSGi core framework

At its heart, Red Hat JBoss Fuse is an OSGi container, based on Apache Karaf, whose architecture is defined by the *OSGi Service Platform Core Specification* (available from http://www.osgi.org/Release4/Download). OSGi is a flexible and dynamic container, whose particular strengths include: sophisticated management of version dependencies; sharing libraries between applications; and support for dynamically updating libraries at run time (hot fixes).

For more details about the OSGi framework, see Chapter 10, *Introduction to OSGi*.

Red Hat JBoss Fuse kernel

The JBoss Fuse kernel extends the core framework of OSGi, adding features such as the runtime console, administration, logging, deployment, provisioning, management, and so on. For more details, see Section 10.1, “Red Hat JBoss Fuse”.

1.2. DEPLOYMENT MODELS

09/27/12

Reordered to make OSGi more prominent

Overview

Although Red Hat JBoss Fuse is an OSGi container at heart, it supports a variety of different deployment models. You can think of these as virtual containers, which hide the details of the OSGi framework. In this section we compare the deployment models to give you some idea of the weaknesses and strengths of each model.

Table 1.1, “Alternative Deployment Packages” shows an overview of the package types associated with each deployment model.

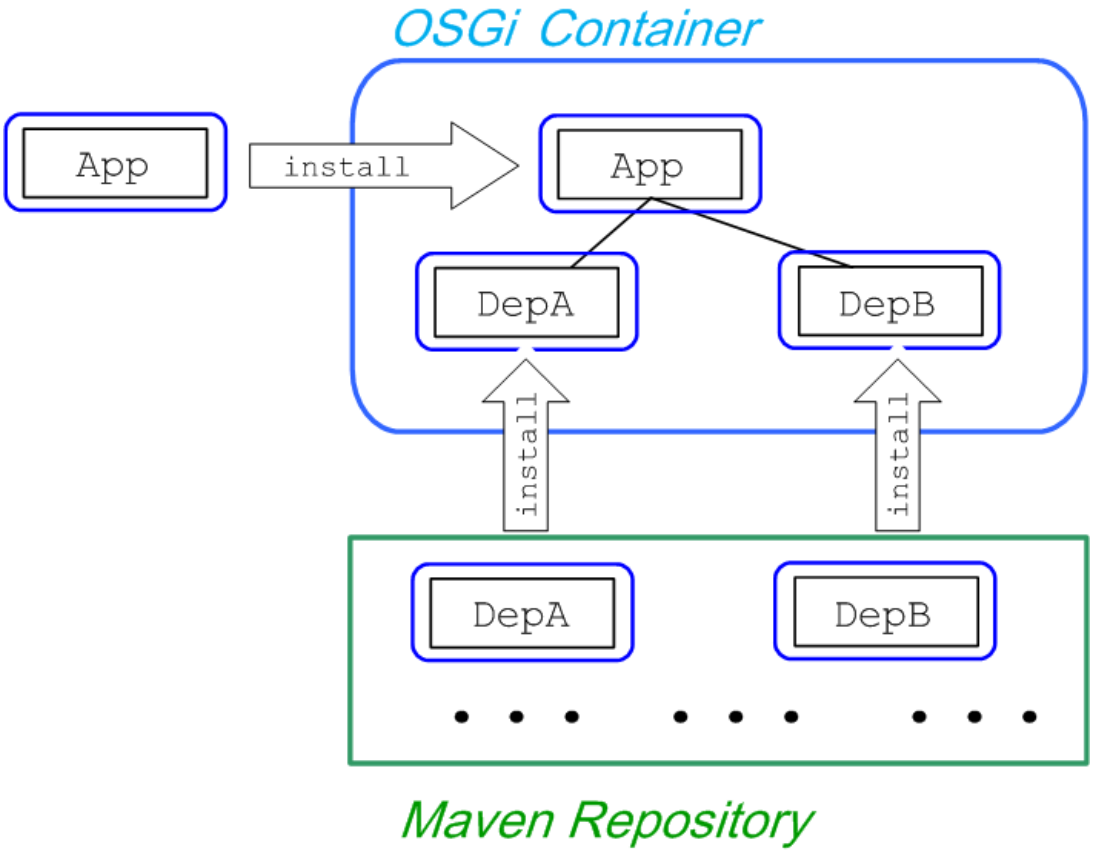
Table 1.1. Alternative Deployment Packages

Package	Metadata	Maven Plug-in	URI Scheme	File Suffix
Bundle	`MANIFEST.MF`	`maven-bundle-plugin`	*None*	`.jar`
FAB	`pom.xml`	`maven-jar-plugin`	`fab:`	`.jar` or `.fab`
WAR	`web.xml`	`maven-war-plugin`	`war:`	`.war`

OSGi bundle deployment model

Figure 1.2, “Installing an OSGi Bundle” gives an overview of what happens when you install an OSGi bundle into the Red Hat JBoss Fuse container, where the bundle depends on several other bundles.

Figure 1.2. Installing an OSGi Bundle

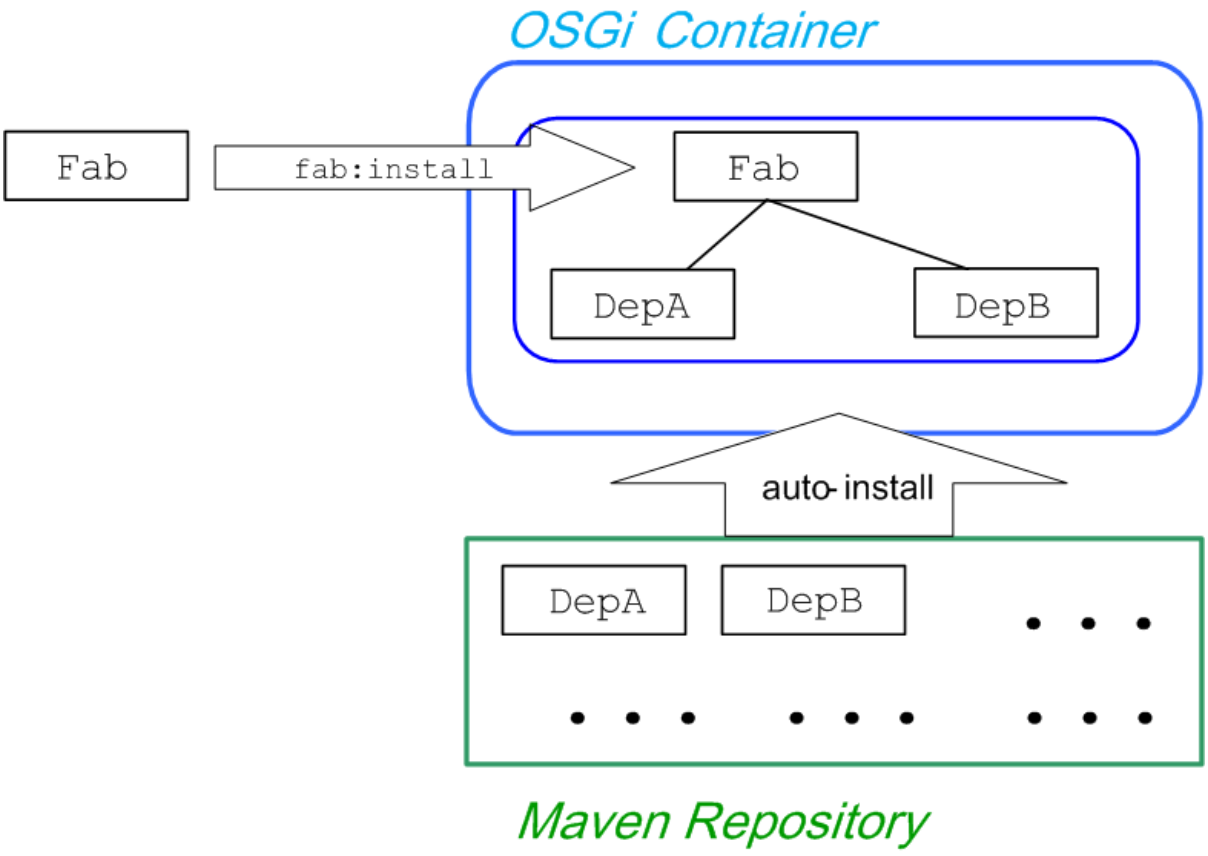


Implicitly, a bundle shares all of its dependencies. This is a flexible approach to deployment, which minimizes resource consumption. But it also introduces a degree of complexity when working with large applications. A bundle does *not* automatically load all of its requisite dependencies, so a bundle might fail to resolve, due to missing dependencies. The recommended way to remedy this is to use *features* to deploy the bundle together with its dependencies (see Chapter 13, *Deploying Features*).

FAB deployment model

Figure 1.3, “Installing a FAB” gives an overview of what happens when you install a typical FAB into the JBoss Fuse container (where it is assumed that all of the FAB's dependencies are either FABs or plain JARs).

Figure 1.3. Installing a FAB

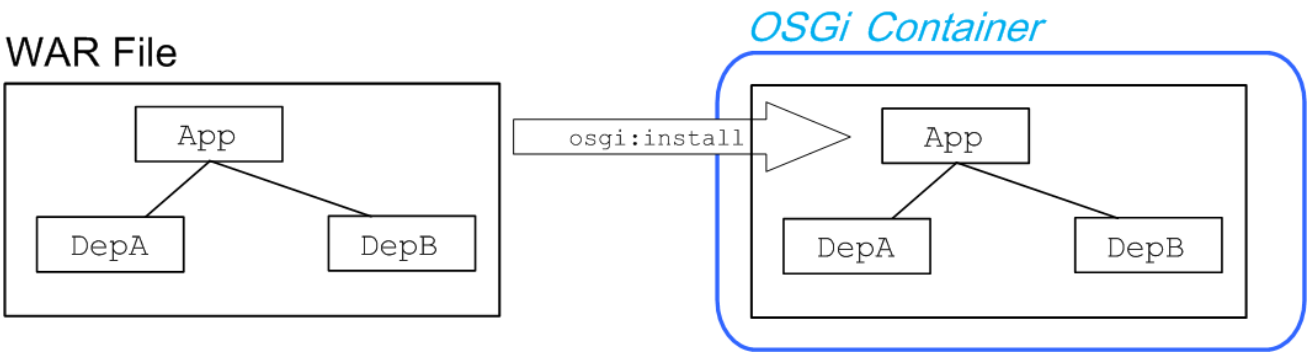


After you install the FAB into the Red Hat JBoss Fuse container, the runtime analyzes the metadata embedded in the FAB and automatically installs the requisite dependencies by pulling them from your local Maven repository (or, if necessary, by downloading them from a remote repository).

WAR deployment model

Figure 1.4, “Installing a WAR” gives an overview of what happens when you install a WAR into the JBoss Fuse container.

Figure 1.4. Installing a WAR



The WAR has a relatively simple deployment model, because the WAR is typically packaged together with all of its dependencies. Hence, the container usually does not have to do any work to resolve the

WAR's dependencies. The drawback of this approach, however, is that the WAR is typically large and it duplicates libraries already available in the container (thus consuming more resources).

1.3. DEPENDENCY INJECTION FRAMEWORKS

09/27/12

Reordered to make blueprint more prominent

Dependency injection

Dependency injection or inversion of control (IOC) is a design paradigm for initializing and configuring applications. Instead of writing Java code that explicitly finds and sets the properties and attributes required by an object, you declare setter methods for all of the properties and objects that this object depends on. The framework takes responsibility for injecting dependencies and properties into the object, using the declared setter methods. This approach reduces dependencies between components and reduces the amount of code that must be devoted to retrieving configuration properties.

There are many popular dependency injection frameworks in current use. In particular, the Spring framework and the blueprint framework are fully integrated with Red Hat JBoss Fuse.

OSGi framework extensions

One of the important characteristics of the OSGi framework is that it is extensible. OSGi provides a framework extension API, which makes it possible to implement OSGi plug-ins that are tightly integrated with the OSGi core. An OSGi extension can be deployed into the OSGi container as an *extension bundle*, which is a special kind of bundle that enjoys privileged access to the OSGi core framework.

Red Hat JBoss Fuse defines extension bundles to integrate the following dependency injection frameworks with OSGi:

- *Blueprint*—the blueprint extensor is based on the blueprint implementation from Apache Karaf.
- *Spring*—the Spring extensor is based on Spring Dynamic Modules (Spring-DM), which is the OSGi integration component from SpringSource.

Activating a framework

The framework extension mechanism enables both the Spring extensor and the blueprint extensor to be integrated with the bundle lifecycle. In particular, the extenders receive notifications whenever a bundle is activated (using the command, **`osgi:start`**) or de-activated (using the command, **`osgi:stop`**). This gives the extenders a chance to scan the bundle, look for configuration files of the appropriate type and, if necessary, activate the dependency injection framework for that bundle.

For example, when you activate a bundle that is configured using Spring, the blueprint extensor scans the bundle package, looking for any blueprint XML files in the standard location and, if it finds one or more such files, activates the blueprint framework for this bundle.

Blueprint XML file location

The blueprint extensor searches a bundle for blueprint XML files whose location matches the following pattern:

```
OSGI-INF/blueprint/*.xml
```

NOTE

A blueprint XML file can also be placed in a non-standard location, by specifying the location in a bundle header (see the section called “Custom Blueprint file locations”).

Spring XML file location

The Spring extensor searches a bundle for Spring XML files whose location matches the following pattern:

```
META-INF/spring/*.xml
```

NOTE

A WAR package uses a different mechanism to specify the location of Spring XML files (see Section 8.3, “Bootstrapping a Spring Context in a WAR”).

1.4. SYNCHRONOUS COMMUNICATION

Overview

Synchronous communication between bundles in the OSGi container is realized by publishing a Java object as an OSGi service. Clients of the OSGi service can then invoke the methods of the published object.

OSGi services

An OSGi service is a plain Java object, which is *published* to make it accessible to other bundles deployed in the OSGi container. Other bundles then *bind* to the Java object and invoke its methods *synchronously*, using the normal Java syntax. OSGi services thus support a model of *synchronous communication* between bundles.

One of the strengths of this model is that the OSGi service is a *perfectly ordinary Java object*. The object is not required to inherit from specific interfaces nor is it required to have any annotations. In other words, your application code is not polluted by the OSGi deployment model.

OSGi registry

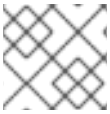
To publish an OSGi service, you must register it in the *OSGi registry*. The OSGi specification defines a Java API for registering services, but the simplest way to publish an OSGi service is to exploit the special syntax provided by the blueprint framework. Use the blueprint **`service`** element to register a Java object in the OSGi registry. For example, to create a **`SavingsAccountImpl`** object and export it as an OSGi service (exposing it through the **`org.fusesource.example.Account`** Java interface)

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">

  <bean id="savings" class="org.fusesource.example.SavingsAccountImpl"/>

 <service ref="savings" interface="org.fusesource.example.Account"/>

</blueprint>
```

Another bundle in the container can then bind to the published OSGi service, by defining a blueprint **`reference`** element that searches the OSGi registry for a service that supports the **`org.fusesource.example.Account`** interface.

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">

  <reference id="savingsRef"
             interface="org.fusesource.example.Account"/>

  <bean id="client" class="org.fusesource.example.client.Client">
    <property name="savingsAccount" ref="savingsRef"/>
  </bean>

</blueprint>
```

For more details, see Section 16.1.2, “Defining a Service Bean”.



NOTE

Spring XML also supports the publication and binding of OSGi services.

Dynamic interaction between bundles

Another important feature of OSGi services is that they are tightly integrated with the bundle lifecycle. When a bundle is activated (for example, using the console command, **`osgi:start`**), its OSGi services are published to the OSGi registry. And when a bundle is de-activated (for example, using the console command, **`osgi:stop`**), its OSGi services are removed from the OSGi registry. Clients of a service can elect to receive notifications whenever a service is published to or removed from the OSGi registry.

Consequently, the presence or absence of an OSGi service in the registry can serve as a flag that signals to other bundles whether or not a particular bundle is available (active). With appropriate programming, clients can thus be implemented to respond flexibly as other bundles are uninstalled and re-installed in the container. For example, a client could be programmed to suspend certain kinds of processing until a bundle is re-activated, thus facilitating dynamic reconfiguration or updating of that dependent bundle.

1.5. ASYNCHRONOUS COMMUNICATION

Overview

Red Hat JBoss Fuse supports two alternative mechanisms of asynchronous communication within the container: the JMS broker and the NMR bus.

JMS broker

You can optionally install a broker instance, typically Apache ActiveMQ, into the Red Hat JBoss Fuse container, to provide support for asynchronous communication between bundles in the container. Apache ActiveMQ is a sophisticated implementation of a JMS broker, which supports asynchronous communication using either queues or topics (publish-subscribe model). Some of the basic features of this JMS broker are as follows:

- *VM protocol*—the Virtual Machine (VM) transport protocol is ideal for communicating within the container. VM is optimized for sending messages within the same JVM instance.

- *Persistent or non-persistent messaging*—you can choose whether the broker should persist messages or not, depending on the requirements of your application.
- *Ease of use*—there is no need to create and configure destinations (that is, queues or topics) before you can use them. After connecting to a broker, you can immediately start sending and receiving messages. If you start sending messages to a queue that does not exist yet, Apache ActiveMQ creates it dynamically.
- *External communication*—you can also configure a TCP port on the broker, opening it up to external JMS clients or other brokers.

For details of how to set up a JMS broker in Red Hat JBoss Fuse, see Chapter 17, *JMS Broker*.

NMR bus

The Normalized Message Router (NMR) bus is an asynchronous messaging system that was originally developed in the context of the JBI standard. In the original specification of the NMR, the message was required to be in XML format and the destination addresses were required to conform to specific URI formats.

The NMR in Red Hat JBoss Fuse, however, has been modified from the original JBI specification. Although you can still use the NMR in the standard way in the context of the JBI container, there are some important differences when you use the NMR in the context of the OSGi container. In the context of OSGi, the NMR bus is much less restrictive, in particular:

- A message body is *not* required to be in XML format. You can use *any* data format in the NMR message body.
- A destination address can be an arbitrary string.
- Messages can be sent within the OSGi container; within the JBI container; and between the OSGi container and the JBI container. Therefore, the NMR bus integrates the two containers.

1.6. FUSE FABRIC

Overview

Fuse Fabric is a technology layer that allows a group of containers to form a cluster that shares a common set of configuration information and a common set of repositories from which to access runtime artifacts. Fabric containers are managed by a Fabric Agent that installs a set of bundles that are specified in the profiles assigned to the container. The agent requests artifacts from the Fabric Ensemble. The ensemble has a list of repositories that it can access. These repositories are managed using a Maven proxy and include a repository that is local to the ensemble.

The added layer imposed on fabric containers does not change the basic deployment models, but it does impact how you specify what needs to be deployed. It also impacts how dependencies are located.

Bundle deployment

In a fabric container, you cannot directly deploy bundles to a container. A container's configuration is managed by a Fabric Agent that updates its contents and configuration based on one or more profiles. So to add a bundle to a container, you must either add the bundle to an existing profile or create a new profile containing the bundle. When the profile is applied to a container the Fuse Agent will install the bundle.

The installation process will download the bundle from a Maven repository and use the appropriate install command to load it into the container. Once the bundle is installed, the dependency resolution process proceeds as it would in a standalone container.

Things to consider

While installing bundles to a fabric container is not radically different from installing bundles in a standalone container, there are a number of things to consider when thinking about creating profiles to deploy your applications:

- Bundles must be accessible through the fabric's Maven proxy

  When a Fabric Agent installs a bundle, it must first copy the bundle to the container's host computer. To do so, the agent uses the fabric's Maven Proxy to locate the bundle in one of the accessible Maven repositories and downloads it. This mechanism ensures that all of the containers in the fabric have access to the same set of bundles.

  If the installed bundle is a FAB, the dependencies are also resolved using the fabric's Maven proxy. This means that all of the dependencies must be in one of the repositories the proxy is configured to access. If any dependency is not accessible through the proxy, it will not be resolved.

  To address this issue, you need to ensure that the fabric's Maven proxy is configured to have access to all of the repositories from which your applications will need to download bundles. For more information see chapter "Configuring a Fabric's Maven Proxy" in "Configuring and Running Red Hat JBoss Fuse".

- Fabric Agents only load the bundles specified in a profile

  A fabric container's contents is completely controlled by the profiles associated with it. The fabric agent managing the container inspects each of the profiles associated with the container, downloads the listed bundles, and features, and installs them. If one of the bundles in a profile depends on a bundle that is not specified in the profile, or one of the other profiles associated with the container, the bundle will not be able to resolve that dependency.

  To address this issue you can do one of the following:

  - construct your profiles to ensure that it contains all of the required bundles and their dependencies
  - deploy the application as a feature that contains all of the required bundles and their dependencies
  - package the application as a FAB and allow the container to resolve the dependencies and download the required bundles

CHAPTER 2. DEPENDENCY INJECTION FRAMEWORKS

Abstract

Red Hat JBoss Fuse supports two alternative dependency injection frameworks: Spring and OSGi blueprint. These frameworks are fully integrated with the Red Hat JBoss Fuse container, so that Spring XML files and blueprint XML files are automatically activated at the same time the corresponding bundle is activated.

2.1. SPRING AND BLUEPRINT FRAMEWORKS

Overview

The OSGi framework allows third-party frameworks to be piggybacked on top of it. In particular, Red Hat JBoss Fuse enables the Spring framework and the blueprint framework, by default. In the case of the *Spring framework*, OSGi automatically activates any Spring XML files under the **`META-INF/spring/`** directory in a JAR, and Spring XML files can also be hot-deployed to the ***`ESBInstallDir`*`/deploy`** directory. In the case of the *blueprint framework*, OSGi automatically activates any blueprint XML files under the **`OSGI-INF/blueprint/`** directory in a JAR, and blueprint XML files can also be hot-deployed to the ***`ESBInstallDir`*`/deploy`** directory.

Configuration files

There are two kinds of file that you can use to configure your project:

- *Spring configuration*—in the standard Maven directory layout, Spring XML configuration files are located under ***`ProjectDir`*`/src/main/resources/META-INF/spring`**.
- *Blueprint configuration*—in the standard Maven directory layout, blueprint XML configuration files are located under ***`ProjectDir`*`/src/main/resources/OSGI-INF/blueprint`**.

If you decide to use the blueprint configuration, you can embed **`camelContext`** elements in the blueprint file, as described in the section called “Blueprint configuration file”.

Prerequisites for blueprint configuration

If you decide to configure your Apache Camel application using blueprint, you must ensure that the **`camel-blueprint`** feature is installed. If necessary, install it by entering the following console command:

```
JBossFuse:karaf@root> features:install camel-blueprint
```

Spring configuration file

You can deploy a **`camelContext`** using a Spring configuration file, where the root element is a Spring **`beans`** element and the **`camelContext`** element is a child of the **`beans`** element. In this case, the **`camelContext`** namespace must be **`http://camel.apache.org/schema/spring`**.

For example, the following Spring configuration defines a route that generates timer messages every two seconds, sending the messages to the **`ExampleRouter`** log (which get incorporated into the console log file, ***`InstallDir`*`/data/log/servicemix.log`**):

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       >

  <camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
      <from uri="timer://myTimer?fixedRate=true&amp;period=2000"/>
      <to uri="log:ExampleRouter"/>
    </route>
  </camelContext>

</beans>
```

It is *not* necessary to specify schema locations in the configuration. But if you are editing the configuration file with an XML editor, you might want to add the schema locations in order to support schema validation and content completion in the editor. For the preceding example, you could specify the schema locations as follows:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
       http://camel.apache.org/schema/spring
http://camel.apache.org/schema/spring/camel-spring.xsd">
    ...
```

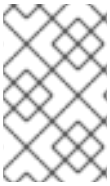
Blueprint configuration file

Before deploying routes in a blueprint configuration file, check that the camel-blueprint feature is already installed.

You can deploy a **camelContext** using a blueprint configuration file, where the root element is **blueprint** and the **camelContext** element is a child of the **blueprint** element. In this case, the **camelContext** namespace must be **http://camel.apache.org/schema/blueprint**.

For example, the following blueprint configuration defines a route that generates timer messages every two seconds, sending the messages to the **ExampleRouter** log (which get incorporated into the console log file, ***InstallDir*/data/log/servicemix.log**):

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    >

  <camelContext xmlns="http://camel.apache.org/schema/blueprint">
    <route>
      <from uri="timer://myTimer?fixedRate=true&amp;period=2000"/>
      <to uri="log:ExampleRouter"/>
    </route>
  </camelContext>

</blueprint>
```



NOTE

Blueprint is a dependency injection framework, defined by the OSGi standard, which is similar to Spring in many respects. For more details about blueprint, see Section 16.1, “The Blueprint Container”.

2.2. HOT DEPLOYMENT

Types of configuration file

You can hot deploy the following types of configuration file:

- Spring XML file, deployable with the suffix, **`.xml`**.
- Blueprint XML file, deployable with the suffix, **`.xml`**.

Hot deploy directory

If you have an existing Spring XML or blueprint XML configuration file, you can deploy the configuration file directly by copying it into the following hot deploy directory:

```
InstallDir/deploy
```

After deploying, the configuration file is activated immediately.

Prerequisites

If you want to deploy Apache Camel routes in a blueprint configuration file, the **`camel-blueprint`** feature must be installed (which it is by default). If the **`camel-blueprint`** feature has been disabled, however, you can re-install it by entering the following console command:

```
JBossFuse:karaf@root> features:install camel-blueprint
```

Default bundle version

When a Spring XML file or a Blueprint XML file is hot deployed, the XML file is automatically wrapped in an OSGi bundle and deployed as a bundle in the OSGi container. By default, the generated bundle has the version, **`0.0.0`**.

Customizing the bundle version

If you prefer to customize the bundle version, use the **`manifest`** element in the XML file. The **`manifest`** element enables you to *override* any of the headers in the generated bundle's **`META-INF/MANIFEST.MF`** file. In particular, you can use it to specify the bundle version.

Specifying the bundle version in a Spring XML file

To specify the bundle version in a hot-deployed Spring XML file, define a **`manifest`** element as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
       >

  <manifest xmlns="http://karaf.apache.org/xmlns/deployer/spring/v1.0.0">
      Bundle-Version = 1.2.3.4
  </manifest>

  <camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
      <from uri="timer://myTimer?fixedRate=true&amp;period=2000"/>
      <to uri="log:ExampleRouter"/>
    </route>
  </camelContext>

</beans>
```

The **manifest** element for Spring XML files belongs to the following schema namespace:

```
http://karaf.apache.org/xmlns/deployer/spring/v1.0.0
```

The contents of the **manifest** element are specified using the syntax of a Java properties file.

Specifying the bundle version in a Blueprint XML file

To specify the bundle version in a hot-deployed Blueprint XML file, define a **manifest** element as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    >

  <manifest
xmlns="http://karaf.apache.org/xmlns/deployer/blueprint/v1.0.0">
      Bundle-Version = 1.2.3.4
  </manifest>

  <camelContext xmlns="http://camel.apache.org/schema/blueprint">
    <route>
      <from uri="timer://myTimer?fixedRate=true&amp;period=2000"/>
      <to uri="log:ExampleRouter"/>
    </route>
  </camelContext>

</blueprint>
```

The **manifest** element for Blueprint XML files belongs to the following schema namespace:

```
http://karaf.apache.org/xmlns/deployer/blueprint/v1.0.0
```

The contents of the **manifest** element are specified using the syntax of a Java properties file.

2.3. USING OSGI CONFIGURATION PROPERTIES

Overview

The OSGi Configuration Admin service defines a mechanism for passing configuration settings to an OSGi bundle. You do not have to use this service for configuration, but it is typically the most convenient way of configuring applications deployed in Red Hat JBoss Fuse (including FABs and OSGi bundles).

Persistent ID

In the OSGi Configuration Admin service, a *persistent ID* is a name that identifies a group of related configuration properties. In JBoss Fuse, every persistent ID, *PersistentID*, is implicitly associated with a file named ***`PersistentID`*`.cfg`** in the ***`ESBInstallDir`*`/etc/`** directory. If the corresponding file exists, it can be used to initialize the values of properties belonging to the *PersistentID* property group.

For example, the **`etc/org.ops4j.pax.url.mvn.cfg`** file is used to set the properties associated with the **`org.ops4j.pax.url.mvn`** persistent ID (for the PAX Mvn URL handler).

Using OSGi configuration properties in Spring

Spring DM provides support for OSGi configuration, enabling you to substitute variables in a Spring XML file using values obtained from the OSGi Configuration Admin service.

Spring example

Example 2.1, “Using OSGi Configuration Properties in Spring XML” shows how to pass the value of the **`prefix`** variable to the constructor of the **`myTransform`** bean in Spring XML, where the value of **`prefix`** is set by the OSGi Configuration Admin service.

Example 2.1. Using OSGi Configuration Properties in Spring XML

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:ctx="http://www.springframework.org/schema/context"
       xmlns:osgi="http://camel.apache.org/schema/osgi"
 xmlns:osgix="http://www.springframework.org/schema/osgi-compendium"
       ... >
  ...
  <bean id="myTransform" class="org.fusesource.example.MyTransform">
    <property name="prefix" value="${prefix}"/>
  </bean>

  <osgix:cm-properties id="preProps" persistent-
id="org.fusesource.example">
    <prop key="prefix">MyTransform</prop>
  </osgix:cm-properties>

  <ctx:property-placeholder properties-ref="preProps" />

</beans>
```

The syntax, **`${prefix}`**, substitutes the value of the **`prefix`** variable into the Spring XML file. The OSGi properties are set up using the following XML elements:

`osgix:cm-properties`

To integrate Spring properties with the properties from the OSGi Configuration Admin service, insert an **osgix:cm-properties** element into the Spring XML file. This element creates a bean that gets injected with all of the properties from the OSGi **ManagedService** instance that is identified by the **persistent-id** attribute. The minimal configuration consists of an empty **osgix:cm-properties** element that sets the **persistent-id** attribute and the **id** attribute—for example:

```
<osgix:cm-properties id="preProps" persistent-
id="org.fusesource.example"/>
```

For an example of how the persistent ID relates to OSGi configuration settings, see the example in the section called "Add OSGi configurations to the feature".

If you want to define defaults for some of the properties in the Spring XML file, add **prop** elements as children of the **osgix:cm-properties** element, as shown in Example 2.1, "Using OSGi Configuration Properties in Spring XML".

ctx:property-placeholder

Property placeholder is a Spring mechanism that enables you to use the syntax, **${*PropName*}**, to substitute variables in a Spring XML file. By defining a **ctx:property-placeholder** element with a reference to the **preProps** bean (as in Example 2.1, "Using OSGi Configuration Properties in Spring XML"), you enable the property placeholder mechanism to substitute any of the variables from the **preProps** bean (which encapsulates the OSGi configuration properties) into the Spring XML file.

Blueprint example

Example 2.2, "Using OSGi Configuration Properties in Blueprint" shows how to pass the value of the **prefix** variable to the constructor of the **myTransform** bean in blueprint XML, where the value of **prefix** is set by the OSGi Configuration Admin service.

Example 2.2. Using OSGi Configuration Properties in Blueprint

```
<blueprint
    xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
 xmlns:cm="http://aries.apache.org/blueprint/xmlns/blueprint-cm/v1.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    ... >
    ...
    <cm:property-placeholder persistent-id="org.fusesource.example">
        <cm:default-properties>
            <cm:property name="prefix" value="Blueprint-Example"/>
        </cm:default-properties>
    </cm:property-placeholder>

    <bean id="myTransform"
class="org.apache.servicemix.examples.camel.MyTransform">
        <property name="prefix" value="{{prefix}}" />
    </bean>

</blueprint>
```

The syntax, **{{prefix}}**, substitutes the value of the **prefix** variable into the blueprint XML file. The OSGi properties are set up using the following XML elements:

cm:property-placeholder

This element gives you access to the properties associated with the specified persistent ID. After defining this element, you can use the syntax, ***{{PropName}}***, to substitute variables belonging to the specified persistent ID.

cm:property-placeholder/cm:default-properties

You can optionally specify default values for properties by defining **cm:property** elements inside the **cm:default-properties** element. If the corresponding ***etc/PersistentID*.cfg** file defines property values, however, these will be used instead.

CHAPTER 3. BUILDING WITH MAVEN

Abstract

Maven is an open source build system which is available from the Apache Maven project. This chapter explains some of the basic Maven concepts and describes how to set up Maven to work with Red Hat JBoss Fuse. In principle, you could use any build system to build an OSGi bundle. But Maven is strongly recommended, because it is well supported by Red Hat JBoss Fuse. Moreover, Maven is a requirement for building FABs.

3.1. MAVEN DIRECTORY STRUCTURE

Overview

One of the most important principles of the Maven build system is that there are *standard locations* for all of the files in the Maven project. There are several advantages to this principle. One advantage is that Maven projects normally have an identical directory layout, making it easy to find files in a project. Another advantage is that the various tools integrated with Maven need almost *no* initial configuration. For example, the Java compiler knows that it should compile all of the source files under **`src/main/java`** and put the results into **`target/classes`**.

Standard directory layout

Example 3.1, “Standard Maven Directory Layout” shows the elements of the standard Maven directory layout that are relevant to building OSGi bundle projects. In addition, the standard locations for Spring-DM and Blueprint configuration files (which are *not* defined by Maven) are also shown.

Example 3.1. Standard Maven Directory Layout

```
ProjectDir/
    pom.xml
    src/
        main/
            java/
                ...
            resources/
                META-INF/
                    spring/
                        *.xml
                OSGI-INF/
                    blueprint/
                        *.xml
        test/
            java/
            resources/
    target/
        ...
```

NOTE

It is possible to override the standard directory layout, but this is *not* a recommended practice in Maven.

pom.xml file

The **pom.xml** file is the Project Object Model (POM) for the current project, which contains a complete description of how to build the current project. A **pom.xml** file can be completely self-contained, but frequently (particular for more complex Maven projects) it can import settings from a *parent POM* file.

After building the project, a copy of the **pom.xml** file is automatically embedded at the following location in the generated JAR file:

```
META-INF/maven/groupId/artifactId/pom.xml
```

src and target directories

The **src/** directory contains all of the code and resource files that you will work on while developing the project.

The **target/** directory contains the result of the build (typically a JAR file), as well as all all of the intermediate files generated during the build. For example, after performing a build, the **target/classes/** directory will contain a copy of the resource files and the compiled Java classes.

main and test directories

The **src/main/** directory contains all of the code and resources needed for building the artifact.

The **src/test/** directory contains all of the code and resources for running unit tests against the compiled artifact.

java directory

Each **java/** sub-directory contains Java source code (***.java** files) with the standard Java directory layout (that is, where the directory pathnames mirror the Java package names, with **/** in place of the **.** character). The **src/main/java/** directory contains the bundle source code and the **src/test/java/** directory contains the unit test source code.

resources directory

If you have any configuration files, data files, or Java properties to include in the bundle, these should be placed under the **src/main/resources/** directory. The files and directories under **src/main/resources/** will be copied into the root of the JAR file that is generated by the Maven build process.

The files under **src/test/resources/** are used only during the testing phase and will *not* be copied into the generated JAR file.

Spring integration

By default, Red Hat JBoss Fuse installs and activates support for Spring Dynamic Modules (Spring DM), which integrates Spring with the OSGi container. This means that it is possible for you to include Spring

configuration files, **META-INF/spring/*.xml**, in your bundle. One of the key consequences of having Spring DM enabled in the OSGi container is that the lifecycle of the Spring application context is automatically synchronized with the OSGi bundle lifecycle:

- *Activation*—when a bundle is activated, Spring DM automatically scans the bundle to look for Spring configuration files in the standard location (any **.xml** files found under the **META-INF/spring/** directory). If any Spring files are found, Spring DM creates an application context for the bundle and creates the beans defined in the Spring configuration files.
- *Stopping*—when a bundle is stopped, Spring DM automatically shuts down the bundle's Spring application context, causing any Spring beans to be deleted.

In practice, this means that you can treat your Spring-enabled bundle as if it is being deployed in a Spring container. Using Spring DM, the features of the OSGi container and a Spring container are effectively merged. In addition, Spring DM provides additional features to support the OSGi container environment—some of these features are discussed in Chapter 16, *OSGi Services*.

Blueprint container

OSGi R4.2 defines a blueprint container, which is effectively a standardized version of Spring DM. Red Hat JBoss Fuse has built-in support for the blueprint container, which you can enable simply by including blueprint configuration files, **OSGI-INF/blueprint/*.xml**, in your project. For more details about the blueprint container, see Chapter 16, *OSGi Services*.

3.2. PREPARING TO USE MAVEN

Overview

This section gives a brief overview of how to prepare Maven for building Red Hat JBoss Fuse projects and introduces the concept of Maven coordinates, which are used to locate Maven artifacts.

Prerequisites

In order to build a project using Maven, you must have the following prerequisites:

- *Maven installation*—Maven is a free, open source build tool from Apache. You can download the latest version from the Maven download page. The minimum supported version is 2.2.1.
- *Network connection*—whilst performing a build, Maven dynamically searches external repositories and downloads the required artifacts on the fly. By default, Maven looks for repositories that are accessed over the Internet. You can change this behavior so that Maven will prefer searching repositories that are on a local network.

NOTE

Maven can run in an offline mode. In offline mode Maven will only look for artifacts in its local repository.

Adding the Red Hat JBoss Fuse repository

In order to access artifacts from the Red Hat JBoss Fuse Maven repository, you need to add it to Maven's **settings.xml** file. Maven looks for your **settings.xml** file in the **.m2** directory of the user's home directory. If there is not a user specified **settings.xml** file, Maven will use the system-level **settings.xml** file at **M2_HOME/conf/settings.xml**.

To add the JBoss Fuse repository to Maven's list of repositories, you can either create a new **.m2/settings.xml** file or modify the system-level settings. In the **settings.xml** file, add the **repository** element for the JBoss Fuse repository as shown in bold text in Example 3.2, “Adding the Red Hat JBoss Fuse Repositories to Maven”.

Example 3.2. Adding the Red Hat JBoss Fuse Repositories to Maven

```
<settings>
    <profiles>
        <profile>
            <id>my-profile</id>
            <activation>
                <activeByDefault>true</activeByDefault>
            </activation>
            <repositories>
                <repository>
                    <id>fusesource</id>

<url>http://repo.fusesource.com/nexus/content/groups/public/</url>
                    <snapshots>
                        <enabled>false</enabled>
                    </snapshots>
                    <releases>
                        <enabled>true</enabled>
                    </releases>
                </repository>                <repository>
                    <id>fusesource.snapshot</id>

<url>http://repo.fusesource.com/nexus/content/groups/public-
snapshots/</url>
                    <snapshots>
                        <enabled>true</enabled>
                    </snapshots>
                    <releases>
                        <enabled>false</enabled>
                    </releases>
                </repository>
                <repository>
                    <id>apache-public</id>

<url>https://repository.apache.org/content/groups/public/</url>
                    <snapshots>
                        <enabled>true</enabled>
                    </snapshots>
                    <releases>
                        <enabled>true</enabled>
                    </releases>
                </repository>
                ...
            </repositories>
        </profile>
    </profiles>
    ...
</settings>
```

The preceding example also shows repository element for the following repositories:

- **`fusesource-snapshot`** repository—if you want to experiment with building your application using an Red Hat JBoss Fuse snapshot kit, you can include this repository.
- **`apache-public`** repository—you might not always need this repository, but it is often useful to include it, because JBoss Fuse depends on many of the artifacts from Apache.

Artifacts

The basic building block in the Maven build system is an *artifact*. The output of an artifact, after performing a Maven build, is typically an archive, such as a JAR or a WAR.

Maven coordinates

A key aspect of Maven functionality is the ability to locate artifacts and manage the dependencies between them. Maven defines the location of an artifact using the system of *Maven coordinates*, which uniquely define the location of a particular artifact. A basic coordinate tuple has the form, **`{groupId, artifactId, version}`**. Sometimes Maven augments the basic set of coordinates with the additional coordinates, *packaging* and *classifier*. A tuple can be written with the basic coordinates, or with the additional *packaging* coordinate, or with the addition of both the *packaging* and *classifier* coordinates, as follows:

```
groupdId:artifactId:version
groupdId:artifactId:packaging:version
groupdId:artifactId:packaging:classifier:version
```

Each coordinate can be explained as follows:

groupdId

Defines a scope for the name of the artifact. You would typically use all or part of a package name as a group ID—for example, **`org.fusesource.example`**.

artifactId

Defines the artifact name (relative to the group ID).

version

Specifies the artifact's version. A version number can have up to four parts: **`n.n.n.n`**, where the last part of the version number can contain non-numeric characters (for example, the last part of **`1.0-SNAPSHOT`** is the alphanumeric substring, **`0-SNAPSHOT`**).

packaging

Defines the packaged entity that is produced when you build the project. For OSGi projects, the packaging is **`bundle`**. The default value is **`jar`**.

classifier

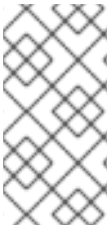
Enables you to distinguish between artifacts that were built from the same POM, but have different content.

The group ID, artifact ID, packaging, and version are defined by the corresponding elements in an artifact's POM file. For example:

```
<project ... >
  ...
  <groupId>org.fusesource.example</groupId>
  <artifactId>bundle-demo</artifactId>
  <packaging>bundle</packaging>
  <version>1.0-SNAPSHOT</version>
  ...
</project>
```

For example, to define a dependency on the preceding artifact, you could add the following **dependency** element to a POM:

```
<project ... >
  ...
  <dependencies>
    <dependency>
      <groupId>org.fusesource.example</groupId>
      <artifactId>bundle-demo</artifactId>
      <version>1.0-SNAPSHOT</version>
    </dependency>
  </dependencies>
  ...
</project>
```



NOTE

It is *not* necessary to specify the **bundle** package type in the preceding dependency, because a bundle is just a particular kind of JAR file and **jar** is the default Maven package type. If you do need to specify the packaging type explicitly in a dependency, however, you can use the **type** element.

CHAPTER 4. LOCATING DEPENDENCIES

08/20/12

Added section to explaining how artifacts are located with Fuse Fabric

Abstract

In Red Hat JBoss Fuse, Maven is the primary mechanism for locating artifacts and dependencies, both at build time and at run time. Normally, Maven requires Internet connectivity, so that dependencies can be downloaded from remote repositories on demand. But, as explained here, it is also possible to provide dependencies locally, so that the need for Internet connectivity is reduced.

4.1. UNDERSTANDING WHERE RED HAT JBOSS FUSE BUNDLES ARE STORED

09/19/2012

edited to clarify between online and no Web access

Overview

Red Hat JBoss Fuse uses Maven as the primary mechanism for locating features, bundles, and their dependencies. Maven is an inherently online tool and will automatically search remote repositories if it cannot locate a dependency in a local repository. Most of the repositories Maven uses by default are accessed through the Internet. A few of them are also public repositories.

It is important to understand where the bundles for the JBoss Fuse features are stored to make sure you understand the connectivity requirements for using JBoss Fuse. It is also useful to know this information so that you understand the potential risks involved. If your systems are not able to connect to the public Internet, you can create either request a copy of the JBoss Fuse off-line repository or build a repository to be hosted on your local network.

Core Red Hat JBoss Fuse features

If you start up a JBoss Fuse console and enter the **`features:list`** command you will see a complete list of the available features. The first column of the listing indicates whether each feature is installed or uninstalled. If you run this command immediately after installing JBoss Fuse, the installed features are the *core JBoss Fuse features*. These core features and all of their dependencies are provided in the JBoss Fuse installation under the ***`EsbInstallDir`*`/system`** directory.

All of the core features are contained locally to the installation. Maven will not need to access a network to search for anything.

Optional Red Hat JBoss Fuse features

If you run **`features:list`** immediately after installing JBoss Fuse, the features listed as uninstalled are the *optional JBoss Fuse features*. The optional features are *not* provided in the system repository and must be downloaded over a network connection.

The default configuration for a standalone container will look for these in the FuseSource repositories first. If it cannot find some artifacts, it will then begin looking in other repositories such as Maven central and SpringSource's repositories.

Custom offline repository

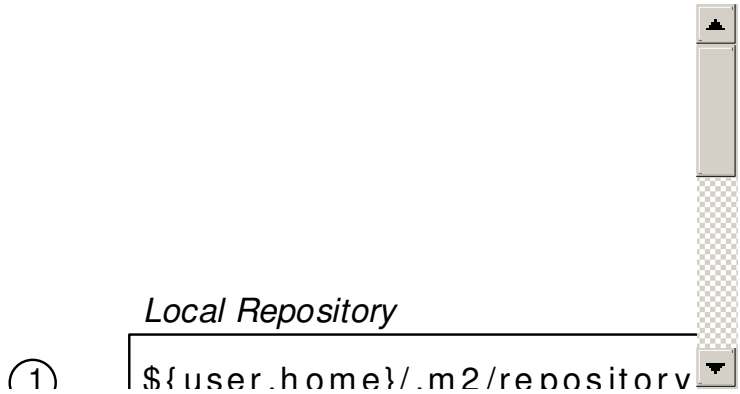
If you are working in an environment that does not allow access to the Internet, you need to make sure that all of the JBoss Fuse features you require are available from internal repositories. One way to achieve this is to create a smaller custom offline repository, which contains just the features and artifacts you need to run your application. For more details, see Section 4.5, “Generating a Custom Offline Repository”.

4.2. LOCATING MAVEN ARTIFACTS AT BUILD TIME

Overview

This section explains how Maven locates artifacts at build time. Essentially, Maven implements a simple caching scheme: artifacts are downloaded from remote repositories on the Internet and then cached in the local repository. Figure 4.1, “How Maven Locates Artifacts at Build Time” shows an overview of the procedure that Maven follows when locating artifacts at build time.

Figure 4.1. How Maven Locates Artifacts at Build Time



Procedure for locating artifacts

While building a project, Maven locates required artifacts (dependencies, required plug-ins, and so on) as follows:

1. The first place that Maven looks for artifacts is in the *local repository*, which is the local cache where Maven stores all of the artifacts it has downloaded or found elsewhere. The default location of the local repository is the `.m2/repository/` directory under the user's home directory.

2. If an artifact is not available in the local repository, Maven has an ordered list repositories, from which it can try to download the artifact. This list of repositories can include both internal and remote repositories. Normally, any internal repositories (that is, repositories maintained in the local network) should appear at the head of the repository list, so that they are consulted first.

3. If the artifact is not available from local or internal repositories, the next repositories to try are the remote repositories (which are accessible, for example, through the HTTP or the HTTPS protocols).

4. When a Maven project is built using the `mvn install` command, the project itself is installed into the local repository.

Configuration

You can configure the following kinds of repository for locating Maven artifacts at build time:

- the section called “Local repository”.
- the section called “Internal repositories”.
- the section called “Remote repositories”.

Local repository

Maven resolves the location of the local repository, by checking the following settings:

1. The location specified by the **`localRepository`** element in the **`~/.m2/settings.xml`** file (UNIX and Linux) or **`C:\Documents and Settings\`*`UserName`*`\.m2\settings.xml`** (Windows).
2. Otherwise, the location specified by the **`localRepository`** element in the ***`M2_HOME`*`/conf/settings.xml`** file.
3. Otherwise, the default location is in the user's home directory, **`~/.m2/repository/`** (UNIX and Linux) or **`C:\Documents and Settings\`*`UserName`*`\.m2\repository`** (Windows).

Internal repositories

Maven enables you to specify the location of internal repositories either in your **`settings.xml`** file (which applies to all projects) or in a **`pom.xml`** (which applies to that project only). Typically, the location of an internal repository is specified using either a **`file://`** URL or a **`http://`** URL (assuming you have set up a local Web server to serve up the artifacts) and you should generally ensure that internal repositories are listed *before* remote repositories. Otherwise, there is nothing special about an internal repository: it is just a repository that happens to be located in your internal network.

For an example of how to specify a repository in your **`settings.xml`** file, see the section called “Adding the Red Hat JBoss Fuse repository”.

Remote repositories

Remote repositories are configured in the same way as internal repositories, except that they should be listed *after* any internal repositories.

4.3. LOCATING MAVEN ARTIFACTS AT RUN TIME

Overview

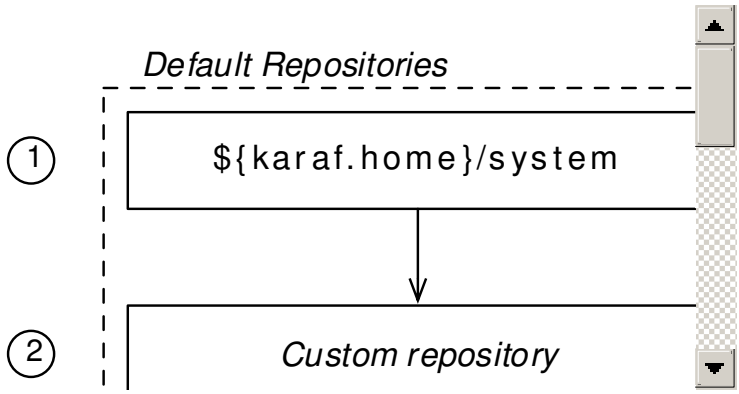
At run time the container strikes a balance between accessing artifacts locally and downloading artifacts from remote repositories. The container will first search all systems local to the container. If it cannot locate the artifacts in a local repository, it will then search remote repositories.

For default features, the artifacts are always stored in the container's system repository. For non-default features, third party bundles, or customer developed bundles, it is likely that Maven will need to search remote repositories to locate the artifacts.

Procedure for locating artifacts

Figure 4.2, “How the Container Locates Artifacts at Run Time” shows an overview of the procedure that Red Hat JBoss Fuse follows when a feature or bundle is installed at run time.

Figure 4.2. How the Container Locates Artifacts at Run Time



The steps followed to locate the required Maven artifacts are:

1. The container searches for artifacts in the system repository.

   This repository contains all of the artifacts provided with the JBoss Fuse installation. The system repository is located at ***EsbInstallDir*`/system`**.

2. If an artifact is not available in the system repository, the container searches any other configured default repositories.

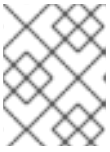
   JBoss Fuse allows you to specify one or more repositories into which you can place artifacts. For example, you could use a shared folder as a default repository that provides an easy way to distribute bundles to remote machines. See the section called “Default repositories” for details on configuring the default repositories.

3. If the artifact is not available in the default repositories, the container searches the Maven local repository.

   The default location of the local repository is the **`.m2/repository/`** directory under the user's home directory. See the section called “Local repository” for details on configuring the local repository.

4. If the artifact is not available in any of the local repositories, the container searches the remote repositories specified in the JBoss Fuse configuration.

   The remote repositories are specified by the **`org.ops4j.pax.url.mvn.repositories`** property in the **`org.ops4j.pax.url.mvn.`** PID. See the section called “Remote repositories” for details on configuring the remote repositories that the container will check.



NOTE

If an artifact is found in a remote repository, it is automatically downloaded and installed into the local repository.

Default repositories

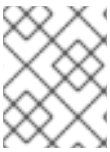
The default repositories are a list of repositories that the container always checks first. The list is specified by the **`org.ops4j.pax.url.mvn.defaultRepositories`** in the **`org.ops4j.pax.url.mvn`** PID. The property's initial setting is a single entry for the container's system repository as shown in Example 4.1, “Initial Setting for a Container's Default Repositories”.

Example 4.1. Initial Setting for a Container's Default Repositories

```
org.ops4j.pax.url.mvn.defaultRepositories=file:${karaf.home}/${karaf.def
ault.repository}@snapshots
```

The **org.ops4j.pax.url.mvn.defaultRepositories** property is a comma-separated list, so you can specify multiple default repositories. You can specify the repository location using a URL with a **file:**, **http:**, or **https:** scheme. You can optionally add the following suffixes to the URL:

- **@snapshots**—allow snapshot versions to be read from the repository
- **@noreleases**—do not allow release versions to be read from the repository



NOTE

It is recommended that you leave the container's system repository as the first entry in the list.

Local repository

The container resolves the location of the local repository in the following manner:

1. Use the location specified by the **org.ops4j.pax.url.mvn.localRepository.localRepository** property in the **org.ops4j.pax.url.mvn** PID.
2. Otherwise, use the location specified by the **localRepository** element in the **settings.xml** file specified by the **org.ops4j.pax.url.mvn.localRepository.settings** property in the **org.ops4j.pax.url.mvn** PID.
3. Otherwise, use the location specified by the **localRepository** element in the **.m2/settings.xml** file located under the user's home directory.
4. Otherwise, use the location specified by the **localRepository** element in the ***M2_HOME*/conf/settings.xml** file.
5. Otherwise, the default location is **.m2/repository/** under the user's home directory.

Remote repositories

The remote repositories checked by the container are specified by the **org.ops4j.pax.url.mvn.repositories** property in the **org.ops4j.pax.url.mvn** PID. The repositories are specified as a comma-separated list as shown in Example 4.2, “Setting a Container's Remote Repositories”.

Example 4.2. Setting a Container's Remote Repositories

```
org.ops4j.pax.url.mvn.repositories= \
    http://repo1.maven.org/maven2, \
    http://repo.fusesource.com/maven2, \
    http://repo.fusesource.com/maven2-snapshot@snapshots@noreleases, \
    http://repo.fusesource.com/nexus/content/repositories/releases, \

http://repo.fusesource.com/nexus/content/repositories/snapshots@snapshot
```

```
s@noreleases, \
    http://repository.apache.org/content/groups/snapshots-
group@snapshots@noreleases, \
    http://repository.ops4j.org/maven2, \
    http://svn.apache.org/repos/asf/servicemix/m2-repo, \
    http://repository.springsource.com/maven/bundles/release, \
    http://repository.springsource.com/maven/bundles/external
```

You can optionally add the following suffixes to the URsL:

- **`@snapshots`**—allow snapshot versions to be read from the repository
- **`@noreleases`**—do not allow release versions to be read from the repository

4.4. LOCATING ARTIFACTS IN A FABRIC

Overview

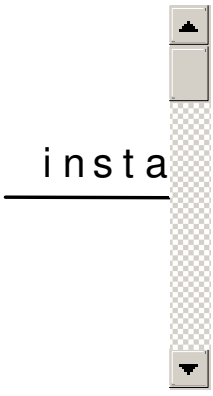
Fabric containers also use Maven to locate artifacts, however they do so in a more constrained manner than a standalone container. Fabric containers use Maven through the fabric's Maven proxy and never search the local repository of the system on which it is running.

The Maven proxy attempts to strike a similar balance between accessing artifacts locally and accessing artifacts from remote repositories, however it changes the scope of local and remote. For default features, the artifacts are always stored in the fabric's system repository which is maintained by the fabric's ensemble servers. For non-default features, third party bundles, or customer developed bundles, it is likely that Maven will need to search remote repositories that are outside of the fabric to locate the artifacts.

Procedure for locating artifacts

Figure 4.3, “How Containers Locate Artifacts in a Fabric” shows an overview of the procedure that a fabric container follows when a feature or bundle is installed.

Figure 4.3. How Containers Locate Artifacts in a Fabric



The steps followed to locate the required Maven artifacts are:

1. The container contacts the fabric Maven proxy to search for the artifacts.
2. The proxy searches for the artifacts in the fabric's system repository.

This repository contains all of the artifacts provided with the Red Hat JBoss Fuse installation.

3. If the artifacts are not available in system repository, the proxy searches the remote repositories specified in its configuration.

   See chapter "Configuring a Fabric's Maven Proxy" in "Configuring and Running Red Hat JBoss Fuse" for details on how to configure a fabric's Maven proxy.

4. The Maven proxy downloads the artifacts to the container.

Loading artifacts into the fabric's repository

Because fabric containers generally do not check a repository local to the machine on which it is running, you must load all of an application's artifacts into a repository that the fabric's Maven proxy knows about. There are two ways to do this:

- load the application's artifacts into the fabric's system repository

  Maven can upload artifacts directly to the fabric's system repository by adding a **`repository`** element defining the Maven proxy's repository to the POM's **`distributionManagement`** element. Example 4.3, “Adding a Fabric Maven Proxy to a POM” shows a POM entry for connecting to a fabric's repository when one of the Fabric Servers is running on the local machine.

  Example 4.3. Adding a Fabric Maven Proxy to a POM

  ```
  <distributionManagement>
    <repository>
      <id>fabric-maven-proxy</id>
      <name>FMC Maven Proxy</name>

  <url>http://username:password@localhost:8107/maven/upload/</url>
    </repository>
  </distributionManagement>
  ```

  You will need to modify the **`url`** element to include the connection details for your environment:

  - The username and password are the credentials used access the Fabric Server to which you are trying to connect.
  - The hostname, **`localhost`** in Example 4.3, “Adding a Fabric Maven Proxy to a POM”, is the address of the machine hosting the Fabric Server.
  - The port number, **`8107`** in Example 4.3, “Adding a Fabric Maven Proxy to a POM”, is the port number exposed by the Fabric Server. **`8107`** is the default setting.
  - The path, **`/maven/upload/`** in Example 4.3, “Adding a Fabric Maven Proxy to a POM”, is the same for all Fabric Servers.

  NOTE

  The Red Hat JBoss Fuse Plugins for Eclipse can also be used to upload artifacts to a fabric's system repository. Red Hat JBoss Fuse Plugins for Eclipse will also generate a profile for deploying the application to a container.

- load the application's artifacts to a custom repository and configure the fabric's Maven proxy to include the custom repository

  This is a good option if an application is going to be used in multiple fabrics because you will not need to install the application into a separate repository for each fabric used. All of the fabrics will use a single, centrally located version of the application.

  To configure the fabric's Maven proxy see chapter "Configuring a Fabric's Maven Proxy" in "Configuring and Running Red Hat JBoss Fuse".

4.5. GENERATING A CUSTOM OFFLINE REPOSITORY

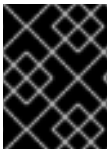
Use case for a custom offline repository

When you move from the development phase of a project to the *deployment* phase, it is typically more convenient to pre-install all of the artifacts required by your application, rather than downloading them from the Internet on demand. In this case, the ideal solution is to create a custom offline repository, which contains the artifacts needed for your deployment. Creating a custom offline repository by hand, however, would be difficult, because it would need to include *all* of the transitive dependencies associated with your application bundles and features.

The ideal way to create a custom offline repository is to generate it, with the help of the Apache Karaf **`features-maven-plugin`** plug-in.

features-maven-plugin Maven plug-in

The **`features-maven-plugin`** plug-in from Apache Karaf is a utility that is used internally by the Apache Karaf developer community and the Red Hat JBoss Fuse development team to create distributions of the Apache Karaf OSGi container. Some of the goals of this plug-in are also useful for application developers, however, and this section explains how you can use the **`add-features-to-repo`** goal to generate your own custom offline repository.



IMPORTANT

At present, only the **`add-features-to-repo`** goal of the **`features-maven-plugin`** plug-in is supported.

Steps to generate a custom repository

To generate and install a custom offline repository for specific Apache Karaf features, perform the following steps:

1. Create a POM file.
2. Add the features-maven-plugin.
3. Specify the features to download.
4. Specify the feature repositories.
5. Specify the Red Hat JBoss Fuse system repository.
6. Specify the remote repositories.

7. Generate the offline repository.

8. Install the offline repository.

Create a POM file

In a convenient location—for example, ***ProjectDir***—create a new directory, ***ProjectDir*/custom-repo** to hold the Maven project. Using a text editor, create the project's POM file, **pom.xml**, in the **custom-repo** directory and add the following contents to the file:

```
<?xml version="1.0" encoding="UTF-8"?>
<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/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <groupId>org.acme.offline-repo</groupId>
    <artifactId>custom-repo</artifactId>
    <version>1.0.0</version>
    <name>Generate offline features repository</name>

</project>
```

This is the bare bones of a Maven POM, which will be added to in the following steps. There is no need to specify a Maven package type here (it defaults to **jar**), because no package will be generated for this project.

Add the features-maven-plugin

Continue editing the **pom.xml** and add the **features-maven-plugin** as shown (where the **build** element is inserted as a child of the **project** element):

```
<project ...>
    ...
    <build>
        <plugins>
          <plugin>
            <groupId>org.apache.karaf.tooling</groupId>
            <artifactId>features-maven-plugin</artifactId>
            <version>2.2.1</version>

            <executions>
              <execution>
                <id>add-features-to-repo</id>
                <phase>generate-resources</phase>
                <goals>
                  <goal>add-features-to-repo</goal>
                </goals>
                <configuration>
                  <descriptors>
 <!-- List the URLs of required feature repositories here -->
                  </descriptors>
```

```
                    <features>
 <!-- List features you want in the offline repo here -->
                    </features>
                    <repository>target/features-repo</repository>
                  </configuration>
                </execution>
              </executions>
            </plugin>
          </plugins>
      </build>

</project>
```

Subsequent steps will explain how to specify the descriptor list (of features repositories) and the features list.

Specify the features to download

In this example scenario, it is assumed that you want to make the **camel-jms** feature and the **camel-quartz** feature available in offline mode. List all of the features you want to download and store in the offline repository in the **features** element, which is a child of the **configuration** element of the **features-maven-plugin**.

To make the **camel-jms** and **camel-quartz** features available offline, add the following **features** element as a child of the **feature-maven-plugin**'s **configuration** element:

```
<features>
  <feature>camel-jms</feature>
  <feature>camel-quartz</feature>
</features>
```

Specify the feature repositories

A *feature repository* is a location that stores feature descriptor files. Generally, because features can depend recursively on other features and because of the complexity of the dependency chains, the project normally requires access to *all* of the standard Red Hat JBoss Fuse feature repositories.

To see the full list of standard feature repositories used by your installation of JBoss Fuse, open the **etc/org.apache.karaf.features.cfg** configuration file and look at the **featuresRepository** setting, which is a comma-separated list of feature repositories, like the following:

```
...
#
# Comma separated list of feature repositories to register by default
#
featuresRepositories=mvn:org.apache.karaf/apache-karaf/2.1.3-fuse-00-
00/xml/features,
mvn:org.apache.servicemix.nmr/apache-servicemix-nmr/1.4.0-fuse-00-
00/xml/features,mvn
:org.apache.servicemix/apache-servicemix/4.3.1-fuse-00-
00/xml/features,mvn:org.apache
.camel.karaf/apache-camel/2.6.0-fuse-00-
00/xml/features,mvn:org.apache.servicemix/ode
-jbi-karaf/1.3.4/xml/features,mvn:org.apache.activemq/activemq-
```

```
karaf/5.4.2-fuse-01-00
/xml/features
...
```

Now, add the listed feature repositories to the configuration of the **features-maven-plugin** in your POM file. Open the project's **pom.xml** file and add a **descriptor** element (as a child of the **descriptors** element) for each of the standard feature repositories. For example, given the preceding value of the **featuresRepositories** list, you would define the **features-maven-plugin** descriptors list in **pom.xml** as follows:

```
<descriptors>
  <!-- List taken from featuresRepositories in
etc/org.apache.karaf.features.cfg -->
  <descriptor>mvn:org.apache.karaf/apache-karaf/2.1.3-fuse-00-
00/xml/features</descriptor>
  <descriptor>mvn:org.apache.servicemix.nmr/apache-servicemix-nmr/1.4.0-
fuse-00-00/xml/features</descriptor>
  <descriptor>mvn:org.apache.servicemix/apache-servicemix/4.3.1-fuse-00-
00/xml/features</descriptor>
  <descriptor>mvn:org.apache.camel.karaf/apache-camel/2.6.0-fuse-00-
00/xml/features</descriptor>
  <descriptor>mvn:org.apache.servicemix/ode-jbi-
karaf/1.3.4/xml/features</descriptor>
  <descriptor>mvn:org.apache.activemq/activemq-karaf/5.4.2-fuse-01-
00/xml/features</descriptor>
</descriptors>
```

Specify the Red Hat JBoss Fuse system repository

Add the Red Hat JBoss Fuse system repository, ***EsbInstallDir*/system**, to the list of repositories in the **pom.xml** file. This is necessary for two reasons: first of all, it saves you from downloading Maven artificats that are already locally available from your JBoss Fuse installation; and secondly, some of the artifacts in the system repository might not be available from any of the other repositories.

Using a text editor, open **pom.xml** and add the following **repositories** element as a child of the **project** element, *customizing the file URL to point at your local system repository*:

```
<project ...>
    ...
    <repositories>
      <repository>
        <id>esb.system.repo</id>
        <name>Red Hat JBoss Fuse internal system repo</name>
        <url>file:///E:/Programs/FUSE/apache-servicemix-6.0.0.redhat-
024/system</url>
        <snapshots>
          <enabled>false</enabled>
        </snapshots>
        <releases>
          <enabled>true</enabled>
        </releases>
      </repository>
```

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

Specify the remote repositories

Generally, the project requires access to *all* of the standard JBoss Fuse remote repositories. To see the full list of standard remote repositories, open the **etc/org.ops4j.pax.url.mvn.cfg** configuration file and look at the **org.ops4j.pax.url.mvn.repositories** setting, which is a comma-separated list of URLs like the following:

```
org.ops4j.pax.url.mvn.repositories= \
    http://repo1.maven.org/maven2, \
    http://repo.fusesource.com/maven2, \
    http://repo.fusesource.com/maven2-snapshot@snapshots@noreleases, \
    http://repo.fusesource.com/nexus/content/repositories/releases, \

http://repo.fusesource.com/nexus/content/repositories/snapshots@snapshots@
noreleases, \
    http://repository.apache.org/content/groups/snapshots-
group@snapshots@noreleases, \
    http://repository.ops4j.org/maven2, \
    http://svn.apache.org/repos/asf/servicemix/m2-repo, \
    http://repository.springsource.com/maven/bundles/release, \
    http://repository.springsource.com/maven/bundles/external
```

Each entry in this list must be converted into a **repository** element, which is then inserted as a child element of the **respositories** element in the project's **pom.xml** file. The preceding repository URLs have slightly different formats and must be converted as follows:

RepoURL

The value of the repository URL, ***RepoURL***, is inserted directly into the **url** child element of the **repository** element. For example, the **http://repo1.maven.org/maven2** repository URL translates to the following **repository** element:

```
<repository>
  <!-- 'id' can be whatever you like -->
  <id>repo1.maven.org</id>
  <!-- 'name' can be whatever you like -->
  <name>Maven central</name>
  <url>http://repo1.maven.org/maven2</url>
  <snapshots>
    <enabled>false</enabled>
  </snapshots>
  <releases>
    <enabled>true</enabled>
  </releases>
</repository>
```

***RepoURL*@snapshots**

The **@snapshots** suffix indicates that downloading snapshots should be enabled for this repository. When specifying the value of the **url** element, remove the **@snapshots** suffix from the URL. Change the **snapshots/enabled** flag to **true**, as shown in the following example:

```
<repository>
  <id>IdOfRepo</id>
  <name>LongNameOfRepo</name>
  <url>RepoURL</url>
  <snapshots>
    <enabled>true</enabled>
  </snapshots>
  <releases>
    <enabled>true</enabled>
  </releases>
</repository>
```

***RepoURL*@snapshots@noreleases**

The combination of the **@snapshots** suffix and the **@noreleases** suffix indicates that downloading snapshots should be enabled and downloading releases should be disabled for this repository. When specifying the value of the **url** element, remove both suffixes from the URL. Change the **snapshots/enabled** flag to **true** and change the **releases/enabled** flag to **false**, as shown in the following example:

```
<repository>
  <id>IdOfRepo</id>
  <name>LongNameOfRepo</name>
  <url>RepoURL</url>
  <snapshots>
    <enabled>true</enabled>
  </snapshots>
  <releases>
    <enabled>false</enabled>
  </releases>
</repository>
```

Generate the offline repository

To generate the custom offline repository, open a new command prompt, change directory to ***ProjectDir*/custom-repo**, and enter the following Maven command:

```
mvn generate-resources
```

Assuming that the Maven build completes successfully, the custom offline repository should now be available in the following location:

```
ProjectDir/custom-repo/target/features-repo
```

Install the offline repository

To install the custom offline repository in the JBoss Fuse container, edit the **etc/org.ops4j.pax.url.mvn.cfg** file and append the offline repository directory to the list of default repositories, as follows:

```
org.ops4j.pax.url.mvn.defaultRepositories=file:${karaf.home}/${karaf.defau
lt.repository}@snapshots,ProjectDir/custom-repo/target/features-
repo@snapshots
```

The **@snapshots** suffix can be added to the offline repository URL, if there is a possibility that some of the artifacts in it are snapshot versions.

PART II. THE FUSE APPLICATION BUNDLE DEPLOYMENT MODEL

Abstract

If you develop your applications using the Apache Maven build system, it is recommended that you deploy your application into the Red Hat JBoss Fuse container as a Fuse Application Bundle(FAB). The FAB model leverages metadata already present in your Maven-generated JAR, so that its required dependencies are automatically installed along with your application JAR.

CHAPTER 5. BUILDING A FAB

Abstract

A FAB is essentially a JAR file built using Maven, where the Maven **pom.xml** file declares the complete set of dependencies for the JAR. It is recommended that you adopt FABs as your standard unit of deployment for the Red Hat JBoss Fuse container, because FABs are easy to use and less likely to fail at deploy time (for example, due to missing dependencies). For more background information, see Chapter 6, *Deploying a FAB*.

5.1. GENERATING A FAB PROJECT

Generating FAB projects with Maven archetypes

To help you get started quickly, you can invoke a Maven archetype to generate the initial outline of a Maven project (a Maven archetype is analogous to a project wizard). Because FABs do not require any special configuration (apart from the presence of a **pom.xml** file, which is always present in a Maven-generated JAR), you can use almost any Maven archetype, as long as the generated Maven project has the **jar** packaging type.

Archetypes

The following Maven archetypes are useful for generating Apache Camel projects:

camel-archetype-java

Demonstrates a route defined using the Java DSL.

camel-archetype-blueprint

Demonstrates a route defined using the XML DSL in an OSGi blueprint file.

camel-archetype-activemq

Demonstrates how to use a message broker in a route.

camel-archetype-component

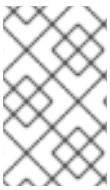
Demonstrates how to define a custom Apache Camel component.

camel-archetype-blueprint archetype

For example, consider an archetype for building an Apache Camel project. The **camel-archetype-blueprint** archetype creates a project that demonstrates a simple Apache Camel route written in the XML DSL using the Blueprint injection framework. To generate a Maven project with the coordinates, *GroupId*:*ArtifactId*:*Version*, enter the following command:

```
mvn archetype:generate
-DarchetypeGroupId=org.apache.camel.archetypes
-DarchetypeArtifactId=camel-archetype-blueprint
-DarchetypeVersion=2.10.0.redhat-60024
-DgroupId=GroupId
-DartifactId=ArtifactId
```

```
-Dversion=Version
```



NOTE

The arguments to the **mvn** command are shown on separate lines purely for the sake of readability. When you are entering the command at a command prompt, you must ensure that all of the parameters are on the same line.

Building the FAB

The archetype creates a project in a new directory whose name is that of the specified artifact ID, *ArtifactId*. To build the FAB defined by the new project, open a command prompt, go to the project directory (that is, the directory containing the **pom.xml** file), and enter the following Maven command:

```
mvn install
```

The effect of this command is to compile all of the Java source files, to generate a FAB JAR under the *ArtifactId*/**target** directory, and then to install the generated JAR in the local Maven repository.

5.2. CLASS SHARING

Private class space

By default, a FAB adds each Maven dependency to its private class space at run time (except for dependencies having Maven group ID **org.apache.camel**, **org.apache.activemq**, or **org.apache.cxf**, which are shared by default). This is a safe approach to deploying dependencies, because it reduces the risk of version inconsistencies. But it is also an expensive approach, in terms of resources, because it forces the JVM to maintain a dedicated copy of each dependency in memory, just for this FAB.

Shared class space

You can make a FAB more economical by opting to share some of its dependencies with other applications in the container. There are two approaches you can use:

- the section called “Specify shared dependencies using provided scope”.
- the section called “Specify shared dependencies using Manifest header”.

Specify shared dependencies using provided scope

To share a dependency (and its transitive dependencies), declare a dependency with the **provided** scope in your project's **pom.xml** file. For example:

```
<dependency>
  <groupId>org.acme.foo</groupId>
  <artifactId>foo-core</artifactId>
  <version>${foo-version}</version>
 <scope>provided</scope>
</dependency>
```

This is analogous to the approach you would use when building a WAR with Maven, in order to avoid including libraries like the servlet API or the JSP API.

Specify shared dependencies using Manifest header

You can also configure class sharing by setting the **`FAB-Provided-Dependency:`** manifest header. For example, to share all artifacts with the **`org.acme.foo`** Maven group ID:

```
FAB-Provided-Dependency: org.acme.foo:* org.apache.camel:*
org.apache.cxf:* org.apache.activemq:*
```

For full details of how to use this approach, see the section called “Sharing dependencies” and the section called “How to set JAR manifest headers”.

5.3. MODIFYING AN EXISTING MAVEN PROJECT

Overview

If you already have a Maven project and you want to customize it to generate a FAB, perform the following steps:

1. the section called “Ensure that the package type is JAR”.
2. the section called “Customize the JDK compiler version”.
3. the section called “Configure class sharing”.
4. the section called “Mark test artifacts with the test scope”.
5. the section called “Specify the requisite dependencies for WS applications”.
6. the section called “Prefer Blueprint over Spring”.
7. the section called “Optionally configure JAR manifest headers”.

Ensure that the package type is JAR

A FAB is packaged as a regular JAR file, which is the default package type in Maven. Ensure that the **`packaging`** element in your project's **`pom.xml`** file contains the value, **`jar`**, as shown in the following example:

```
<project ... >
  ...
  <packaging>jar</packaging>
  ...
</project>
```

Customize the JDK compiler version

It is almost always necessary to specify the JDK version in your POM file. If your code uses any modern features of the Java language—such as generics, static imports, and so on—and you have not customized the JDK version in the POM, Maven will fail to compile your source code. It is *not* sufficient to

set the **JAVA_HOME** and the **PATH** environment variables to the correct values for your JDK, you must also modify the POM file.

To configure your POM file, so that it accepts the Java language features introduced in JDK 1.6, add the following **maven-compiler-plugin** plug-in settings to your POM (if they are not already present):

```
<project ... >
  ...
  <build>
    <defaultGoal>install</defaultGoal>
    <plugins>
      ...
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
          <source>1.6</source>
          <target>1.6</target>
        </configuration>
      </plugin>
    </plugins>
  </build>
  ...
</project>
```

Configure class sharing

Any dependencies on standard artifacts already provided by the container should be declared as **provided**, to prevent the FAB runtime from unnecessarily installing those dependencies again. This typically includes artifacts whose names match the patterns, **camel-***, **cxf-***, **activemq-***, and **fabric-***.

For example, if you have an existing dependency on the **camel-http** artifact, you should modify the dependency by adding the **scope** element as follows:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-http</artifactId>
    <version>2.10.0.redhat-60024</version>
 <scope>provided</scope>
</dependency>
```

NOTE

All dependencies with Maven group ID **org.apache.camel**, **org.apache.cxf**, or **org.apache.activemq** are shared by default (equivalent to **provided** scope), so that the value of the **scope** element is ignored by default. This behavior changes, however, if the **FAB-Provided-Dependency** manifest header is specified explicitly—see the section called “Sharing dependencies” for details.

If you have a large number of dependencies to manage, it might be easier to share the standard container artifacts by adding a FAB manifest header. For details, see Example 5.1, “Configuring FAB Manifest Headers in the POM”.

Mark test artifacts with the test scope

Any artifacts needed only for testing *must* be marked with the test scope, as in the following example:

```
<dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>${log4j-version}</version>
 <scope>test</scope>
</dependency>
```

Of course, this is the standard convention in POM files. But it is particularly important to observe this convention with FAB projects. For any test artifacts that are not marked as such, *the FAB runtime will attempt to download and install the test artifact into the container at run time*.

IMPORTANT

Because the test-only artifacts are not intended to be installed in the container, it is quite likely that a FAB will fail to deploy properly if test artifacts are not marked with the **`test`** scope.

Specify the requisite dependencies for WS applications

Make sure that you declare all of the requisite dependencies for an Apache CXF application and declare these dependencies with **`provided`** scope. For example, a basic Apache CXF application that uses the JAX-WS frontend needs the following Maven dependencies:

```
<project>
    ...
    <dependencies>
        <dependency>
            <groupId>org.apache.cxf</groupId>
            <artifactId>cxf-rt-frontend-jaxws</artifactId>
            <version>2.6.0.redhat-60024</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>org.apache.cxf</groupId>
            <artifactId>cxf-rt-transports-http</artifactId>
            <version>2.6.0.redhat-60024</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>org.apache.cxf</groupId>
            <artifactId>cxf-rt-transports-http-jetty</artifactId>
            <version>2.6.0.redhat-60024</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-web</artifactId>
            <version>3.0.6.RELEASE</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
```

```
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```

Prefer Blueprint over Spring

If your project uses dependency injection or XML configuration, you should prefer the Blueprint framework over the Spring framework.

Because Blueprint is more tightly integrated with OSGi, it is usually able to find whatever dependencies it needs dynamically at deploy time. By contrast, dependencies introduced by a Spring XML file are more likely to lead to **`ClassNotFound`** exceptions at deploy time (FAB is not capable of parsing a Spring XML file to discover the dependencies it introduces).

Optionally configure JAR manifest headers

Although the FAB runtime obtains most of its deployment metadata by scanning the **`pom.xml`** file embedded in the JAR, you can also specify FAB-specific configuration settings by adding headers to the JAR's manifest. For example:

```
FAB-Version-Range-Digits: 2
FAB-Provided-Dependency: org.acme.foo:* org.apache.camel:*
org.apache.cxf:* org.apache.activemq:*
```

For detailed explanations of all the FAB manifest headers, see Section 5.4, “Configuring a FAB”.

5.4. CONFIGURING A FAB

Overview

Typically, it is not necessary to specify any additional configuration for a FAB. But you might be interested in setting some of the optional FAB manifest headers, in order to optimize the performance of your FAB at run time. In particular, it is often a good idea to share some of the bigger dependencies, so that the FAB does not consume too much system memory in the JVM at run time.

Maven artifact pattern syntax

Many of the FAB manifest headers take a value, which is a space-separated list of patterns that match Maven artifacts. Each artifact pattern has the following syntax:

```
groupId[:artifactId]
```

If the artifact ID, *artifactId*, is omitted, the pattern matches *all* of the artifacts in the specified group, *groupId*. You can also use the wildcard character, **`*`**, to match an arbitrary sequence of characters in either the group ID or the artifact ID.

For example, to match specific artifacts, you could specify a list like the following:

```
org.apache.camel:camel-core org.apache.cxf:cxf-core
```

To match all of the Apache Camel, Apache ActiveMQ, and Apache CXF artifacts, you could specify a list like the following:

```
org.apache.camel org.apache.activemq org.apache.cxf
```

To match all Apache artifacts and all Spring framework artifacts, you could specify a list like the following:

```
org.apache.* org.springframework.*
```

Supported manifest headers

To configure a FAB, you can optionally specify any of the following FAB manifest headers:

`FAB-Version-Range-Digits`

See the section called “Specifying version ranges”.

`FAB-Provided-Dependency`

See the section called “Sharing dependencies”.

`FAB-Include-Optional-Dependency`

See the section called “Including optional dependencies”.

`FAB-Dependency-Require-Bundle`

See the section called “Specifying OSGi Require-Bundle”.

`FAB-Exclude-Dependency`

See the section called “Excluding dependencies”.

`Import-Package`

See the section called “Customising import packages”.

`FAB-Skip-Matching-Feature-Detection`

See the section called “Automatic feature detection”.

`FAB-Require-Feature-URL`, **`FAB-Require-Feature`**

See the section called “Requiring features”.

`FAB-Install-Provided-Bundle-Dependencies`

See the section called “Respecting pre-installed features and bundles”.

Specifying version ranges

Maven dependencies typically specify the *exact* version of each required artifact. At deployment time, however, a little bit of flexibility is usually required. For example, you would normally prefer the FAB to be capable of accepting a patch update to one of its dependencies. For this reason, when the FAB is converted to a bundle at deploy time, each dependency version is normally converted into a range.

For example, given the following Maven dependency:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-core</artifactId>
  <version>2.8.1-fuse-00-02</version>
  <scope>provided</scope>
</dependency>
```

When the FAB is converted to a bundle, the preceding exact version is converted to a range in the generated OSGi **`Import-Package`** header, as follows:

```
Import-Package: org.apache.camel;version="[2.8.1-fuse-00-02,2.9)"
```

This reflects the default range policy, where later patch versions (**`2.8.2`**, **`2.8.3`**, **`2.8.4`**, and so on) are accepted, but minor and major upgrades are rejected.

It is possible to modify the range policy—either to be more strict or to be more lax—by setting the **`FAB-Version-Range-Digits:`** manifest header to one of the following policy values:

Value	Sample	Description
`0`	**`[2.5.6.qual,2.5.6.qual]`**	Exact only.
`1`	**`[2.5.6.qual,2.5.7)`**	Allow arbitrary qualifers (order not guaranteed).
`2`	**`[2.5.6.qual,2.6)`**	Allow patch releases *(default)*.
`3`	**`[2.5.6.qual,3)`**	Allow patch and minor releases.
`4`	**`[2.5.6.qual,)`**	*Any* version from **`2.5.6`** onwards.

Sharing dependencies

This manifest header gives you an alternative way to configure class sharing. Instead of adding **`<scope>provided</scope>`** to dependencies in the **`pom.xml`** file, you can list the shared artifacts in this manifest header. The shared artifacts are specified as a space-separated list of artifact patterns, for example:

```
FAB-Provided-Dependency: groupId1:artifactId1 groupId2:artifactId2 ...
```

You can also use a wildcard, **`*`**, for the *groupId* or the *artifactId*. For example, to share all Apache Camel and Spring dependencies (transitively) in your FAB, add the following header to your manifest:

```
FAB-Provided-Dependency: org.apache.camel:* org.springframework:*
```

If you do not explicitly configure the **`FAB-Provided-Dependency`** manifest header, the FAB runtime implicitly adds the following default header:

-

```
FAB-Provided-Dependency: org.apache.camel:* org.apache.cxf:*
org.apache.activemq:*
```

IMPORTANT

If you specify the **FAB-Provided-Dependency** manifest header explicitly, you override the default header value, which would make the **org.apache.camel**, **org.apache.cxf**, and **org.apache.activemq** dependencies unshared again. Since you normally want these dependencies to remain shared, it is good practice to include the entries, **org.apache.camel:* org.apache.cxf:* org.apache.activemq:***, in your custom **FAB-Provided-Dependency** manifest header.

Including optional dependencies

Optional dependencies In a **pom.xml** file are marked by the presence of **<optional>true</optional>** in the dependency.

By default, optional dependencies are excluded from a FAB. To force their inclusion, list them in the **FAB-Include-Optional-Dependency:** manifest header.

The header value is specified as a space-separated list of artifact patterns, with support for the wildcard, ***** . For example, to force the inclusion of *all* optional dependencies, add the following manifest header:

```
FAB-Include-Optional-Dependency: *:*
```

Specifying OSGi Require-Bundle

Specifies a list of artifacts which should not use the regular OSGi mechanism of importing packages, but instead should use the OSGi **Require-Bundle** directive.

By default, packages are imported from shared dependencies using the OSGi **Import-Package** directive for all artifacts not selected by the **FAB-Dependency-Require-Bundle** header.

The header value is specified as a space-separated list of artifacts, with support for the wildcard, ***** —for example:

```
FAB-Dependency-Require-Bundle: groupId1:artifactId1 groupId2:artifactId2
...
```

Excluding dependencies

To exclude specific dependencies, you can list them in the **FAB-Exclude-Dependency:** manifest header, as a space-separated list of artifact patterns, for example:

```
FAB-Exclude-Dependency: log4j logkit
```

Customising import packages

Specifying Java packages to import from the classpath, using the **Import-Package** manifest header, is the standard OSGi mechanism for specifying dependencies. Normally, you do *not* have to worry about imported packages when using FABs, because the FAB runtime automatically generates the **Import-**

Package manifest header with the requisite entries.

In exceptional cases, however, you might find that you need to import specific Java packages that FAB is unable to figure out by itself. For these cases, it is possible to add an **Import-Package** manifest header to your FAB package. At run time, FAB merges your customized import package list with the automatically generated list of import packages.

The **Import-Package** header is specified using the standard OSGi format. For example, to import the package, **com.acme.special**, where the version is allowed to lie in the range **[1.0, 1.1)**, you could add the following header:

```
Import-Package: com.acme.special;version="[1.0,1.1)"
```

Automatic feature detection

Red Hat JBoss Fuse comes with a collection of features that have been carefully crafted and manually adjusted so that they install *exactly* the right set of dependencies for a particular piece of functionality. For example, to install all of the required dependencies for the Apache Camel Jetty component, you can install the **camel-jetty** feature by entering the following console command:

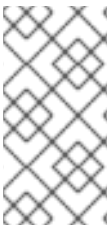
```
JBossFuse:karaf@root> features:install camel-jetty
```

If you want to use the Jetty component in a FAB, you would add the following Maven dependency to the FAB's POM:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-jetty</artifactId>
  <version>2.10.0.redhat-60024</version>
  <scope>provided</scope>
</dependency>
```

If the **camel-jetty** feature is not already installed in the runtime, however, it is unlikely that the FAB mechanism would be able to install all of the required transitive dependencies successfully (after all, the **camel-jetty** feature was created in the first place precisely because it has third-party dependencies that are not well integrated with OSGi or FAB).

This is where *automatic feature detection* comes in. By default, whenever FAB encounters a Maven dependency that matches a known feature, it automatically maps the dependency to the corresponding feature and installs the feature instead. For example, when the FAB runtime encounters the **org.apache.camel/camel-jetty** Maven dependency, it automatically maps the dependency to the **camel-jetty** feature and installs the **camel-jetty** feature.



NOTE

Automatic feature detection is not yet available for all standard features. At the time of writing, auto-detection is supported for Apache Camel component features, **camel-*ComponentName***, and some Apache CXF features, **cxf**, **cxf-sts**, and **cxf-wsn**.

Automatic feature detection is enabled by default. If you want to explicitly *disable* feature detection, you can set the **FAB-Skip-Matching-Feature-Detection**, specifying a space-separated list of artifact patterns. For example, to disable automatic feature detection for the Apache Camel components, add

the following entry to the manifest:

```
FAB-Skip-Matching-Feature-Detection: org.apache.camel
```

Requiring features

You can configure a FAB explicitly to require one or more features, so that the features are automatically installed when you deploy the FAB into the container. To identify a feature, you must specify both the location of the features repository (normally provided in a Maven repository) and the name of the feature.

For example, to require the **`cxf-sts`** feature, add the following entries to the Manifest (noting that the Maven URL is continued on a second line, because it does not fit within the 72 byte line limit):

```
FAB-Require-Feature-URL: mvn:org.apache.cxf.karaf/apache-cxf/
 2.6.0.redhat-60024/xml/features
FAB-Require-Feature: cxf-sts
```

The **`FAB-Require-Feature-URL`** header is specified as a space-separated list of URLs that give the locations of the features repositories.

The **`FAB-Require-Feature`** header is specified as a space-separated list of features, in the format, ***`FeatureName`***, or in the format, ***`FeatureName/Version`***. For example:

```
FAB-Require-Feature: cxf-sts/2.6.0.redhat-60024 cxf-wsn/2.6.0.redhat-60024
```

Respecting pre-installed features and bundles

There would not be much point in using features (which are manually adjusted to install exactly the right dependencies, for special cases that an automated tool could not copy with), if the FAB runtime simply forged ahead and installed all of the dependencies it thinks it needs, ignoring the dependencies that were already installed by the feature. For this reason, the FAB runtime adopts a respectful attitude towards previously installed bundles: *by default, FAB does not try to install dependencies for a provided bundle, if that bundle is already installed*.

The assumption is that a pre-installed bundle already has all of its transitive dependencies installed. For example, this is normally true, if the bundle was installed as part of a feature.

FAB respects pre-installed features and bundles by default. If you want to override this behavior, forcing the FAB runtime to scan the POM files in provided bundles and to install all of the transitive dependencies it finds, set the following Manifest header flag:

```
FAB-Install-Provided-Bundle-Dependencies: true
```

How to set JAR manifest headers

The Maven JAR plug-in supports the following alternative approaches to setting manifest headers in your Maven project:

- the section called “In the POM file”.
- the section called “In the META-INF/MANIFEST.MF file”.

In the POM file

You can specify manifest headers in **pom.xml**, by configuring the Maven JAR plug-in. In the JAR plug-in's **configuration/archive/manifestEntries** element, specify manifest header settings using the following syntax:

```
<ManifestHeaderName>HeaderValue</ManifestHeaderName>
```

For example, to set the **FAB-Version-Range-Digits** manifiest header and the **FAB-Provided-Dependency** manifest header in your **pom.xml** file, configure the JAR plug-in as shown in Example 5.1, “Configuring FAB Manifest Headers in the POM”.

Example 5.1. Configuring FAB Manifest Headers in the POM

```
<project ...>
  ...
  <build>
    ...
    <plugins>
      ...
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-jar-plugin</artifactId>
        <configuration>
          <archive>
            <index>true</index>
            <manifestEntries>
              <FAB-Version-Range-Digits>0</FAB-Version-Range-Digits>
              <FAB-Provided-Dependency>
                org.apache.camel:*
                org.apache.cxf:*
                org.apache.activemq:*
              </FAB-Provided-Dependency>
            </manifestEntries>
          </archive>
        </configuration>
      </plugin>
      ...
    </plugins>
  </build>

</project>
```

In the META-INF/MANIFEST.MF file

Alternatively, you can create a **MANIFEST.MF** file directly and instruct the Maven JAR plug-in to include the provided manifest in the generated JAR (the JAR plug-in does *not* include the manifest by default). Configure the JAR plug-in to include the manifest file as follows:

```
<project ...>
  ...
  <build>
    ...
```

```
    <plugins>
      ...
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-jar-plugin</artifactId>
        <configuration>
          <!-- lets use the default META-INF/MANIFEST.MF if its there -->
          <useDefaultManifestFile>true</useDefaultManifestFile>
        </configuration>
      </plugin>
      ...
    </plugins>
  </build>

</project>
```

In your Maven project directory tree, create a **MANIFEST.MF** file at the following location:

```
ProjectDir/src/main/resources/META-INF/MANIFEST.MF
```

The following example shows a correctly formatted **META-INF/MANIFEST.MF** file:

```
FAB-Version-Range-Digits: 0
FAB-Provided-Dependency: org.apache.camel:* org.apache.cxf:* org.ac
 tivemq:*
```

Note the following peculiarities of the manifest file syntax:

- Each header is terminated by a newline.
- Line length is limited to 72 bytes (not characters), including the newline.
- Line continuation is indicated by putting a space character at the start of a line.
- The manifest file *must* end with a newline character.

NOTE

Given the awkward constraints on the manifest file syntax, it is recommended that you edit the manifest using a dedicated editor, such as the Eclipse Manifest Editor Plug-In.

CHAPTER 6. DEPLOYING A FAB

Abstract

Apache Karaf provides two different approaches for deploying FABs: hot deployment and manual deployment.

6.1. THE FAB DEPLOYMENT MODEL

Overview

FABs have a fundamentally different deployment model from standard OSGi bundles. When a FAB is installed, the FAB runtime automatically figures out what dependencies are required, by scanning the Maven metadata, and these dependencies are then installed dynamically. This makes FABs easier to use than standard OSGi bundles.

What is a FAB?

A *Fuse Application Bundle* (FAB) is any JAR created using Apache Maven, or similar build tools, so that inside the JAR there is a **`pom.xml`** file at the following location (where *groupId* and *artifactId* are the Maven coordinates of the JAR):

```
META-INF/maven/groupId/artifactId/pom.xml
```

Which contains the transitive dependency information for the JAR.

IMPORTANT

A FAB is *not an OSGi bundle*. At installation time, however, a FAB results in the creation of one or more bundles, which are constructed dynamically.

Manifest.mf file

It is possible to add some additional (optional) configuration to a FAB by including FAB-specific headers in the JAR's **`Manifest.mf`** file. For example, the **`FAB-Provided-Dependency`** header can be used to specify whether some or all of the FAB's dependencies are to be shared.

For full details about FAB manifest headers, see Section 5.4, “Configuring a FAB”.

FAB deployment

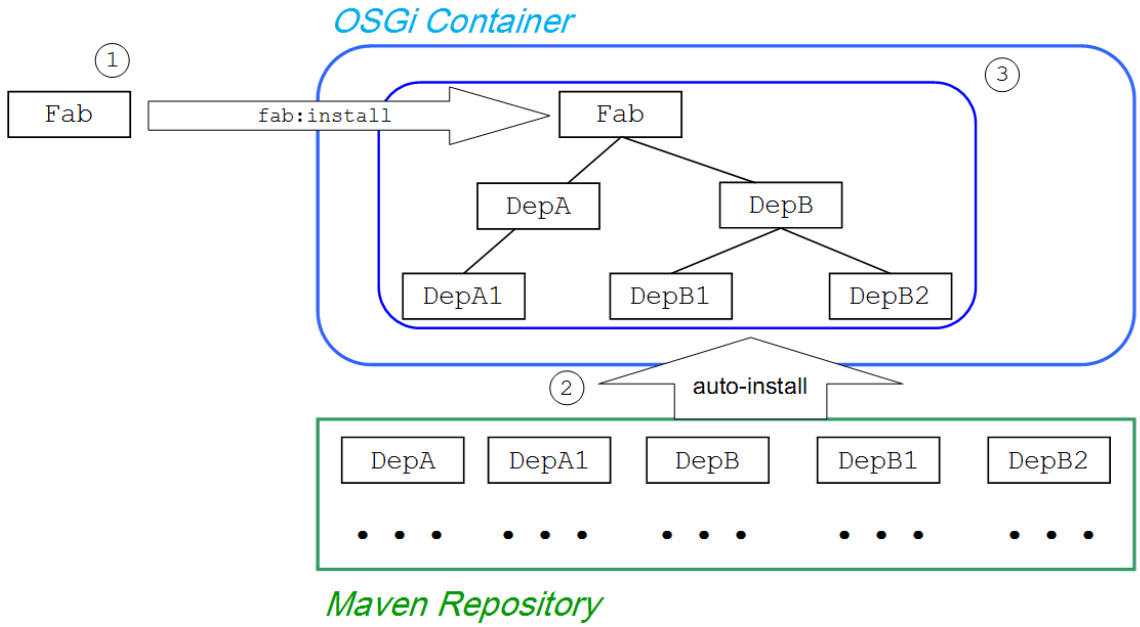
Depending on how a FAB is configured, there are two basic alternatives for deploying a FAB:

- the section called “Installing a FAB with private dependencies”.
- the section called “Installing a FAB with shared dependencies”.

Installing a FAB with private dependencies

Figure 6.1, “FAB Installed with Private Dependencies” shows how a FAB is installed when *all* of its dependencies are configured to be private.

Figure 6.1. FAB Installed with Private Dependencies



The figure shows the installation of the FAB, **Fab**, and its private dependencies, which proceeds as follows:

1. The user explicitly deploys the FAB, **Fab**, into the OSGi container (either by hot deploying or manually deploying).

2. The FAB runtime scans the embedded **pom.xml** file and figures out all of the non-optional and non-test transitive dependencies. In this case, there are five *private* dependencies: **DepA**, **DepA1**, **DepB**, **DepB1**, and **DepB2**. The FAB runtime auto-installs these dependencies by fetching them from the Maven repository.

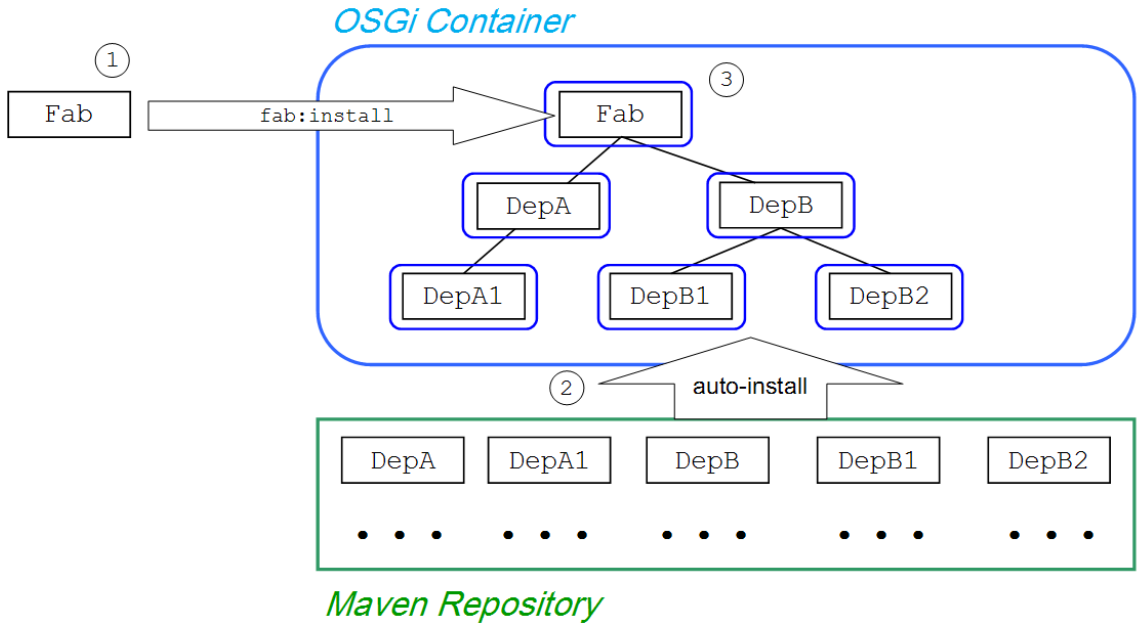
3. The dependencies are added to the FAB's private class space and the whole collection is converted into a single OSGi bundle and installed in the OSGi container.

When a FAB is configured to have all of its dependencies private, as in this example, what you end up with is very similar to a WAR deployment (which also tends to keep its dependencies private, by default). An advantage that the FAB has over the WAR, however, is that the FAB *does not need to be packaged together with its dependencies*. Thus a FAB can be much *smaller* than a typical WAR package.

Installing a FAB with shared dependencies

At the other extreme, Figure 6.2, “FAB Installed with Shared Dependencies” shows how a FAB is installed when *all* of its dependencies are configured to be shared.

Figure 6.2. FAB Installed with Shared Dependencies



The figure shows the installation of the FAB, **Fab**, and its shared dependencies, which proceeds as follows:

1. The user explicitly deploys the FAB, **Fab**, into the OSGi container (either by hot deploying or manually deploying).

2. The FAB runtime scans the embedded **pom.xml** file and figures out all of the non-optional and non-test transitive dependencies. In this case, there are five *shared* dependencies: **DepA**, **DepA1**, **DepB**, **DepB1**, and **DepB2**. The FAB runtime then auto-installs these dependencies by fetching them from the Maven repository.

3. The FAB, **Fab**, and its dependencies are separately converted into OSGi bundles and the requisite import and export bundle headers are automatically added. This collection of generated bundles is then installed in the OSGi container.

When a FAB is configured to share all of its dependencies, as in this example, what you end up with is very similar to the conventional approach using OSGi bundles and features. The advantage of the FAB approach, however, is that you do not need to write a custom features file to make sure that all of the dependencies are installed at the same time.

Optimising a FAB's shared dependencies

A FAB can have all private dependencies, some shared dependencies, or all shared dependencies. In other words, you can configure a FAB's dependencies anywhere on the spectrum from all-shared to all-private.

By default, a FAB keeps its dependencies private (except for the dependencies with Maven group ID **org.apache.camel**, **org.apache.cxf**, or **org.apache.activemq**). This deployment approach is relatively safe, because it minimizes the risk of version conflicts and so on. But it is also an expensive option, because it uses up a lot of system memory and resources. If you would like to experiment with sharing some of the FAB's dependencies, you can specify which artifacts to share using the **FAB-Provided-Dependency:** header in the FAB's manifest, **Manifest.mf**. For example:

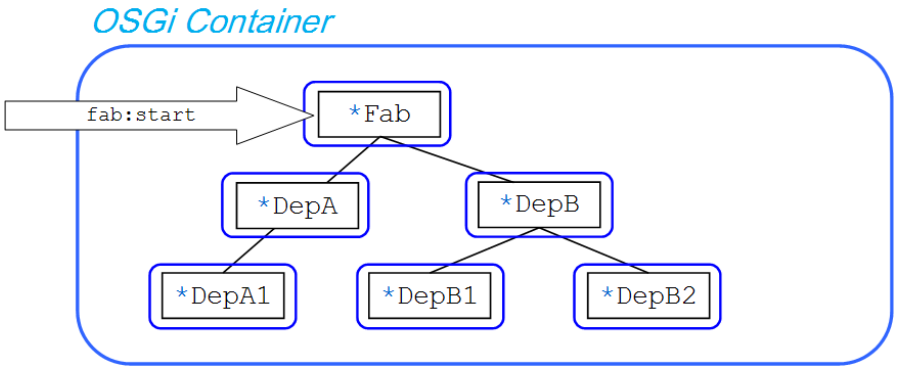
```
FAB-Provided-Dependency: org.apache.camel:* org.apache.cxf:*
org.apache.activemq:*
  org.springframework:*
```

In this way, you can easily optimize the performance of the FAB, while balancing the risk of version conflicts occurring.

Starting a FAB

Figure 6.3, “FAB Start” shows an outline of what happens when you *start* a FAB, in the case where the FAB shares its dependencies.

Figure 6.3. FAB Start



When you start a FAB using the **`fab:start`** console command, the FAB runtime iterates over the FAB's transitive dependencies and starts every single bundle in the tree of dependencies (starting with the leaves). This contrasts with the conventional **`osgi:start`** console command, which starts only the specified bundle.

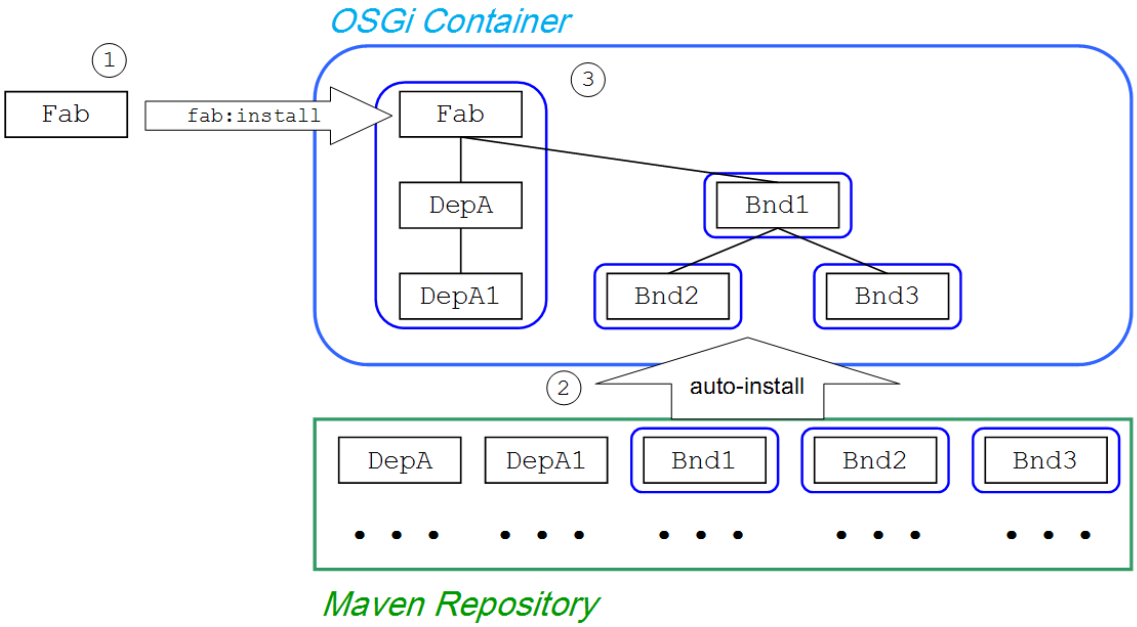
Default install behavior of a FAB

The default install behavior of a FAB is actually more complex than suggested by the all-private dependencies model (as shown in Figure 6.1, “FAB Installed with Private Dependencies”). In reality, the FAB runtime distinguishes between the following types of dependency:

- *Plain FAB dependency (has **`pom.xml`** file, but no bundle headers)*—by default, the plain FAB dependencies are added to the deployed FAB's private classpath and bundled together with the deployed FAB.
- *OSGi bundle dependency (has **`pom.xml`** file and bundle headers)*—by default, the OSGi bundle dependencies are deployed as *separate* bundles (shared dependencies).

Figure 6.4, “Default FAB Install Behavior” shows how a FAB is installed, by default, when some its dependencies are plain FABs (**DepA** and **DepA1**) and some of its dependencies are OSGi bundles (**Bnd1**, **Bnd2**, and **Bnd3**):

Figure 6.4. Default FAB Install Behavior



The FAB runtime works on the assumption that whoever built an artifact as an OSGi bundle presumably intended for the artifact to be deployed as a bundle. Another good reason for deploying OSGi bundles separately is that this guarantees that the bundles will be properly activated when you invoke **`fab:start`**. If the dependent bundles (**Bnd1**, **Bnd2**, and **Bnd3**) were merely added to the private classpath of the **Fab** bundle, they would *not* be activated when **`fab:start`** is invoked (in other words, any Spring configuration files or blueprint configuration files would fail to be activated).

6.2. FABS AND FEATURES

Overview

FABs have been designed to integrate smoothly with Apache Karaf features. Although there is some overlap in the capabilities of FABs and features, FABs cannot replace features completely. There are some cases, involving third-party dependencies, where figuring out the correct set of dependencies is inherently difficult and unavoidably it remains a manual process. Features remain the ideal mechanism for encapsulating these kinds of hand-crafted dependencies.

Requiring features

A FAB can be configured to require specific features, so that the required features are automatically installed at deploy time. For details of how to configure this, see the section called “Requiring features”.

Automatic feature detection

Automatic feature detection is a special case of requiring features. When the FAB runtime scans an embedded **`pom.xml`** file, it automatically maps certain Maven dependencies to features and installs the corresponding feature instead of the Maven dependency.

For example, if the FAB runtime finds a Maven dependency on the the **`org.apache.camel/camel-jetty`** artifact, it will automatically install the **`camel-jetty`** feature.

Respecting features

The basic FAB model of deployment is rather aggressive: the FAB runtime walks the tree of Maven

transitive dependencies, installing (and if necessary, also downloading) all of the dependencies that it finds along the way. This deployment model must be softened, however, if FABs are to integrate properly with features.

Features are manually constructed deployment units and, in many cases, they include combinations of bundles that an automated tool would never be able to figure out. For this reason, it is better if the FAB runtime respects the judgement of the developer who constructed the feature and does *not* try to install dependencies for bundles belonging to a feature. To avoid clobbering feature dependencies, the FAB runtime applies the following rule: *by default, FAB does not try to install dependencies for a provided bundle, if that bundle is already installed*.

This rule ensures that the FAB runtime does not try to second-guess the dependencies installed by a feature.

6.3. HOT DEPLOYMENT

Hot deploy directory

If you save a FAB as a file ending in **`.fab`**, you can deploy it simply by copying it into the ***`InstallDir`*`/deploy`** directory.

Red Hat JBoss Fuse monitors files in the ***`InstallDir`*`/deploy`** directory and hot deploys any files that have a recognized suffix. Each time a FAB file is copied to this directory, it is installed in the runtime and also started. You can subsequently update or delete the FABs, and the changes are handled automatically.

For example, if you have just built the FAB, *ProjectDir*`/target/fabulous-1.0.fab`, you can deploy this FAB by copying it to the *InstallDir*`/deploy` directory as follows (assuming you are working on a UNIX platform):

```
% cp ProjectDir/target/fabulous-1.0.fab InstallDir/deploy
```

6.4. MANUAL DEPLOYMENT

Overview

You can manually deploy and undeploy FABs by issuing commands at the Red Hat JBoss Fuse console.

Installing a FAB using the fab: scheme

FABs can be deployed using the **`fab:`** URL scheme, which is useful for installing FABs in a general context, such as a FAB URL embedded in an Apache Karaf features file. For example, to install a FAB using the **`osgi:install`** command, prefix the FAB's URL with the **`fab:`** scheme, as follows:

```
osgi:install fab:mvn:groupId/artifactId/version
```

Starting a FAB

To start a FAB, you must first obtain its bundle ID using the **`osgi:list`** command. You can then start the FAB using the **`fab:start`** command (which takes the bundle ID as its argument).

For example, if you have already installed the FAB named **`A Camel FAB`**, entering **`osgi:list`** at the console prompt might produce output like the following:

```
...
[ 175] [Active     ] [            ] [Started] [   60] ServiceMix :: FTP
(2009.02.0.psc-01-00RC1)
[ 180] [Installed  ] [            ] [       ] [   60] A Camel FAB
(1.0.0.SNAPSHOT)
```

You can now start the bundle with the ID, **`180`**, by entering the following console command:

```
fab:start 180
```

The **`fab:start`** command recursively starts all of the FAB's dependent bundles, which ensures that services required by the FAB are available when the FAB starts up.

Stopping a FAB

To stop a FAB, invoke the **`fab:stop`** console command (which takes the FAB's bundle ID as its argument).

For example, to stop the FAB with the bundle ID, **`180`**, enter the following console command:

```
fab:stop 180
```

The **`fab:stop`** command recursively stops all of the FAB's dependent bundles. If you would prefer to perform a shallow stop (that is, to stop only the bundle corresponding to the FAB), use the **`osgi:stop`** command instead.

Uninstalling a FAB

To uninstall a FAB, invoke the **`fab:uninstall`** console command (which takes the FAB's bundle ID as its argument).

For example, to uninstall the FAB with the bundle ID, **`180`**, enter the following console command:

```
fab:uninstall 180
```

The **`fab:uninstall`** command also uninstalls the FAB's *unused* transitive dependencies. Hence, **`fab:uninstall`** can potentially uninstall multiple bundles from the OSGi container. If you would prefer to perform a shallow uninstall (that is, to uninstall only the bundle corresponding to the FAB), use the **`osgi:uninstall`** command instead.

Bundle lifecycle management commands

Seeing as how a FAB is ultimately transformed into a bundle (after it is deployed into the OSGi container), some of the standard bundle lifecycle commands are also of interest to FAB users. For details, see Section 12.3, “Lifecycle Management”.

URL schemes for locating and installing FABs

When specifying the FAB's URL to the **osgi:install** command, you can combine **fab:** with any of the URL schemes supported by Red Hat JBoss Fuse, which includes the following scheme types:

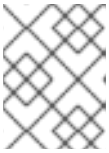
- *Maven scheme*—use the combined schemes, **fab:mvn:**, as follows:

  ```
  fab:mvn:groupId/artifactId/version
  ```

  For more details about the **mvn:** scheme, see Section A.3, “Mvn URL Handler”.

- *File scheme*—use the combined schemes, **fab:file:**, as follows:

  ```
  fab:file:PathName
  ```

  NOTE

  On Windows, use forward slashes, **/**, instead of backslashes, ****, in the pathname, *PathName*.

  For more details about the **file:** scheme, see Section A.1, “File URL Handler”.

- *HTTP scheme*—use the combined schemes, **fab:http:**, as follows:

  ```
  fab:http:Host[:Port]/[Path]
  ```

  For more details about the **http:** scheme, see Section A.2, “HTTP URL Handler”.

6.5. CONFIGURING MAVEN FOR FAB

Overview

FAB requires access to Maven repositories to download dependencies. The Maven configuration of FAB leverages the property settings of the Maven URL handler. This means that any changes to the Maven URL handler will have a ripple effect to other functions of the container. It also means that once a container is moved to a fabric, FAB uses the fabric's Maven proxy to resolve dependencies.

Maven URL handler PID

The properties used to configure a standalone container's Maven URL handler are in the **org.ops4j.pax.url.mvn** PID. See Section A.3, “Mvn URL Handler”.

A fabric container will also have a profile assigned to it that contains the **org.ops4j.pax.url.mvn** PID. This PID should not be modified for purposes of configuring FAB dependency resolution. Instead, you should modify the fabric's Maven proxy as described in chapter "Configuring a Fabric's Maven Proxy" in "Configuring and Running Red Hat JBoss Fuse".

Customize the location of the local repository

If your local Maven repository is in a non-default location, you might find it necessary to configure it explicitly in order to access Maven artifacts that you have built locally. You can change the location of the local repository for a standalone container by setting the **org.ops4j.pax.url.mvn** PID's

org.ops4j.pax.url.mvn.localRepository property to the location of your local Maven repository. Example 6.1, “Changing the Local Repository” shows the commands.

Example 6.1. Changing the Local Repository

```
JBossFuse:karaf@root> config:edit org.ops4j.pax.url.mvn
JBossFuse:karaf@root> config:propset
org.ops4j.pax.url.mvn.localRepository file:E:/Data/.m2/repository
JBossFuse:karaf@root> config:update
```

Customize the list of remote repositories

To customize the list of remote Maven repositories used by a standalone container, add or remove entries appearing in the comma-separated list in the **org.ops4j.pax.url.mvn.repositories** property value in the **org.ops4j.pax.url.mvn.cfg** PID.

CHAPTER 7. FAB TUTORIAL

Abstract

This tutorial describes a complete example of how to generate, build, run, and deploy a Apache Camel application as a FAB.

7.1. GENERATING AND RUNNING AN EIP FAB

Overview

This section explains how to generate, build, and run a complete Apache Camel example as a FAB, where the starting point code is generated with the help of a Maven archetype.

Prerequisites

In order to generate a project using an Red Hat JBoss Fuse Maven archetype, you must have the following prerequisites:

- *Maven installation*—Maven is a free, open source build tool from Apache. You can download the latest version from http://maven.apache.org/download.html (minimum is 2.0.9).
- *Internet connection*—whilst performing a build, Maven dynamically searches external repositories and downloads the required artifacts on the fly. In order for this to work, your build machine *must* be connected to the Internet.
- *fusesource Maven repository is configured*—in order to locate the archetypes, Maven's **`settings.xml`** file must be configured with the location of the **`fusesource`** Maven repository. For details of how to set this up, see the section called “Adding the Red Hat JBoss Fuse repository”.

Generating an EIP FAB

The **`camel-archetype-blueprint`** archetype creates a sample Maven project that can be deployed as a FAB. To generate a Maven project with the coordinates, **`org.fusesource.example:camel-fab`**, enter the following command:

```
mvn archetype:generate
-DarchetypeGroupId=org.apache.camel.archetypes
-DarchetypeArtifactId=camel-archetype-blueprint
-DarchetypeVersion=2.10.0.redhat-60024
-DgroupId=org.fusesource.example
-DartifactId=camel-fab
```

The result of this command is a directory, ***`ProjectDir`*`/camel-fab`**, containing the files for the generated FAB project.

Running the EIP FAB

To install and run the generated **`camel-fab`** project, perform the following steps:

1. *Build the project*—open a command prompt and change directory to ***`ProjectDir`*`/camel-fab`**.

Use Maven to build the demonstration by entering the following command:

```
mvn install
```

If this command runs successfully, the ***ProjectDir*/camel-fab/target** directory should contain the JAR file, **camel-fab-1.0-SNAPSHOT.jar** and the JAR will also be installed in the local Maven repository.

2. *Install and start the camel-fab JAR*—at the Red Hat JBoss Fuse console, enter the following command:

```
JBossFuse:karaf@root> install fab:mvn:org.fusesource.example/camel-
fab
```

If the FAB is successfully installed, its bundle ID is logged to the console window. Using this bundle ID, *bundleID*, start the FAB by entering the following console command:

```
JBossFuse:karaf@root> fab:start bundleID
```

After entering this command, you should soon see output like the following being logged to the console screen:

```
>>>> MyTransform set body:  Mon Sep 22 11:43:42 BST 2011
>>>> MyTransform set body:  Mon Sep 22 11:43:44 BST 2011
>>>> MyTransform set body:  Mon Sep 22 11:43:46 BST 2011
```

3. *Stop the camel-fab bundle*—using the FAB's bundle ID, *bundleID*, stop the **camel-fab** bundle, as follows:

```
JBossFuse:karaf@root> fab:stop bundleID
```

PART III. WAR DEPLOYMENT MODEL

Abstract

The Web application ARchive (WAR) is a tried and tested model for packaging and deploying applications. This approach is simple and reliable, thought not as flexible as the FAB model or the OSGi bundle model.

CHAPTER 8. BUILDING A WAR

Abstract

This chapter describes how to build and package a WAR using Maven.

8.1. MODIFYING AN EXISTING MAVEN PROJECT

Overview

If you already have a Maven project and you want to modify it so that it generates a WAR, perform the following steps:

1. the section called “Change the package type to WAR”.
2. the section called “Customize the JDK compiler version”.
3. the section called “Store resources under webapp/WEB-INF”.
4. the section called “Customize the Maven WAR plug-in”.

Change the package type to WAR

Configure Maven to generate a WAR by changing the package type to **`war`** in your project's **`pom.xml`** file. Change the contents of the **`packaging`** element to **`war`**, as shown in the following example:

```
<project ... >
  ...
  <packaging>war</packaging>
  ...
</project>
```

The effect of this setting is to select the Maven WAR plug-in, **`maven-war-plugin`**, to perform packaging for this project.

Customize the JDK compiler version

It is almost always necessary to specify the JDK version in your POM file. If your code uses any modern features of the Java language—such as generics, static imports, and so on—and you have not customized the JDK version in the POM, Maven will fail to compile your source code. It is *not* sufficient to set the **`JAVA_HOME`** and the **`PATH`** environment variables to the correct values for your JDK, you must also modify the POM file.

To configure your POM file, so that it accepts the Java language features introduced in JDK 1.6, add the following **`maven-compiler-plugin`** plug-in settings to your POM (if they are not already present):

```
<project ... >
  ...
  <build>
    <defaultGoal>install</defaultGoal>
    <plugins>
      ...
```

```
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
          <source>1.6</source>
          <target>1.6</target>
        </configuration>
      </plugin>
    </plugins>
  </build>
  ...
</project>
```

Store resources under webapp/WEB-INF

Resource files for the Web application are stored under the **/WEB-INF** directory in the standard WAR directory layout. In order to ensure that these resources are copied into the root of the generated WAR package, store the **WEB-INF** directory under ***ProjectDir*/src/main/webapp** in the Maven directory tree, as follows:

```
ProjectDir/
    pom.xml
    src/
        main/
            webapp/
                WEB-INF/
 web.xml
                    classes/
                    lib/
```

In particular, note that the **web.xml** file is stored at ***ProjectDir*/src/main/webapp/WEB-INF/web.xml**.

Customize the Maven WAR plug-in

It is possible to customize the Maven WAR plug-in by adding an entry to the **plugins** section of the **pom.xml** file. Most of the configuration options are concerned with adding additonal resources to the WAR package. For example, to include all of the resources under the **src/main/resources** directory (specified relative to the location of **pom.xml**) in the WAR package, you could add the following WAR plug-in configuration to your POM:

```
<project ...>
  ...
  <build>
    ...
    <plugins>
      <plugin>
        <artifactId>maven-war-plugin</artifactId>
     <version>2.1.1</version>
        <configuration>
          <!-- Optionally specify where the web.xml file comes from -->
          <webXml>src/main/webapp/WEB-INF/web.xml</webXml>
          <!-- Optionally specify extra resources to include -->
          <webResources>
```

```
            <resource>
              <directory>src/main/resources</directory>
              <targetPath>WEB-INF</targetPath>
              <includes>
                <include>**/*</include>
              </includes>
            </resource>
          </webResources>
        </configuration>
      </plugin>
      ...
    </plugins>
  </build>
</project>
```

The preceding plug-in configuration customizes the following settings:

`webXml`

Specifies where to find the **`web.xml`** file in the current Maven project, relative to the location of **`pom.xml`**. The default is **`src/main/webapp/WEB-INF/web.xml`**.

`webResources`

Specifies additional resource files that are to be included in the generated WAR package. It can contain the following sub-elements:

- **`webResources/resource`**—each resource elements specifies a set of resource files to include in the WAR.
- **`webResources/resource/directory`**—specifies the base directory from which to copy resource files, where this directory is specified relative to the location of **`pom.xml`**.
- **`webResources/resource/targetPath`**—specifies where to put the resource files in the generated WAR package.
- **`webResources/resource/includes`**—uses an Ant-style wildcard pattern to specify explicitly which resources should be *included* in the WAR.
- **`webResources/resource/excludes`**—uses an Ant-style wildcard pattern to specify explicitly which resources should be *excluded* from the WAR (exclusions have priority over inclusions).

For complete details of how to configure the Maven WAR plug-in, see http://maven.apache.org/plugins/maven-war-plugin/index.html.

NOTE

Do not use version 2.1 of the **`maven-war-plugin`** plug-in, which has a bug that causes two copies of the **`web.xml`** file to be inserted into the generated **`.war`** file.

Building the WAR

To build the WAR defined by the Maven project, open a command prompt, go to the project directory (that is, the directory containing the **`pom.xml`** file), and enter the following Maven command:

```
mvn install
```

The effect of this command is to compile all of the Java source files, to generate a WAR under the ***ProjectDir*/target** directory, and then to install the generated WAR in the local Maven repository.

8.2. BOOTSTRAPPING A CXF SERVLET IN A WAR

Overview

A simple way to bootstrap Apache CXF in a WAR is to configure **web.xml** to use the standard CXF servlet, **org.apache.cxf.transport.servlet.CXFServlet**.

Example

For example, the following **web.xml** file shows how to configure the CXF servlet, where all Web service addresses accessed through this servlet would be prefixed by **/services/** (as specified by the value of **servlet-mapping/url-pattern**):

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC
  "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
  "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
  <display-name>cxf</display-name>
  <description>cxf</description>

  <servlet>
    <servlet-name>cxf</servlet-name>
    <display-name>cxf</display-name>
    <description>Apache CXF Endpoint</description>
    <servlet-class>org.apache.cxf.transport.servlet.CXFServlet</servlet-
class>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>cxf</servlet-name>
    <url-pattern>/services/*</url-pattern>
  </servlet-mapping>

  <session-config>
    <session-timeout>60</session-timeout>
  </session-config>

</web-app>
```

cxf-servlet.xml file

In addition to configuring the **web.xml** file, it is also necessary to configure your Web services by defining a **cxf-servlet.xml** file, which must be copied into the root of the generated WAR.

Alternatively, if you do not want to put **cxf-servlet.xml** in the default location, you can customize its

name and location, by setting the **contextConfigLocation** context parameter in the **web.xml** file. For example, to specify that Apache CXF configuration is located in **WEB-INF/cxf-servlet.xml**, set the following context parameter in **web.xml**:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC
  "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
  "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>
    ...
 <context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>WEB-INF/cxf-servlet.xml</param-value>
 </context-param>
    ...
</web-app>
```

Reference

For full details of how to configure the CXF servlet, see .

8.3. BOOTSTRAPPING A SPRING CONTEXT IN A WAR

Overview

You can bootstrap a Spring context in a WAR using Spring's ContextLoaderListener class.

Bootstrapping a Spring context in a WAR

For example, the following **web.xml** file shows how to boot up a Spring application context that is initialized by the XML file, **/WEB-INF/applicationContext.xml** (which is the location of the context file in the generated WAR package):

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

    <display-name>Camel Routes</display-name>

    <!-- location of spring xml files -->
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/applicationContext.xml</param-value>
    </context-param>

    <!-- the listener that kick-starts Spring -->
    <listener>
        <listener-
class>org.springframework.web.context.ContextLoaderListener</listener-
class>
```

```
    </listener>

</web-app>
```

CHAPTER 9. DEPLOYING A WAR

Abstract

This chapter explains how to deploy a Web archive (WAR) file as a bundle in the OSGi container. Conversion to a bundle is performed automatically by the PAX War URL, which is based on the open source Bnd tool. The presence of a **web.xml** file in the bundle signals to the container that the bundle should be deployed as a Web application.

9.1. CONVERTING THE WAR USING THE WAR SCHEME

Overview

To convert a WAR file into a bundle suitable for deployment in the OSGi container, add the war: prefix to the WAR URL. The PAX War URL handler acts as a wrapper, which adds the requisite manifest headers to the WAR file.

Syntax

The **war** scheme has the following basic syntax:

```
war:LocationURL[?Options]
```

The location URL, *LocationURL*, can be any of the location URLs described in Appendix A, *URL Handlers* (for example, an **mvn:** or a **file:** URL). Options can be appended to the URL in the following format:

```
?Option=Value&Option=Value&...
```

Or if the **war** URL appears in an XML file:

```
?Option=Value&amp;Option=Value&amp;...
```

Prerequisite

The Apache Karaf **war** feature is required to convert and deploy WARs using the war: scheme. It can be installed from the container's command console using **features:install war**.

Deploying a WAR file

If the WAR file is stored in a Maven repository, you can deploy it into the OSGi container using the **osgi:install** command, taking a **war:mvn:** URL as its argument. For example, to deploy the wicket-example WAR file from a Maven repository, where the application should be accessible from the **wicket** Web application context, enter the following console command:

```
JBossFuse:karaf@root> install war:mvn:org.apache.wicket/wicket-
examples/1.4.7/war?Web-ContextPath=wicket
```

Alternatively, if the WAR file is stored on the filesystem, you can deploy it into the OSGi container by specifying a **`war:file:`** URL. For example, to deploy the WAR file, **`wicket-example-1.4.6.war`**, enter the following console command:

```
JBossFuse:karaf@root> install war:file://wicket-examples-1.4.7.war?Web-
ContextPath=wicket
```

Accessing the Web application

The WAR file is automatically installed into a Web container, which listens on the IP port 8181 by default, and the Web container uses the Web application context specified by the **`Web-ContextPath`** option. For example, the **`wicket-example`** WAR deployed in the preceding examples, would be accessible from the following URL:

```
http://localhost:8181/wicket
```

Default conversion parameters

The PAX War URL handler converts a WAR file to a special kind of OSGi bundle, which includes additional Manifest headers to support WAR deployment (for example, the **`Web-ContextPath`** Manifest header). By default, the deployed WAR is configured as an isolated bundle (neither importing nor exporting any packages). This mimics the deployment model of a WAR inside a J2EE container, where the WAR is completely self-contained, including all of the JAR files it needs.

For details of the default conversion parameters, see Table A.2, “Default Instructions for Wrapping a WAR File”.

Customizing the conversion parameters

The PAX War URL handler is layered over Bnd. If you want to customize the bundle headers in the Manifest file, you can either add a Bnd instruction as a URL option or you can specify a Bnd instructions file for the War URL handler to use—for details, see Section A.5, “War URL Handler”.

In particular, you might sometimes find it necessary to customize the entry for the **`Bundle-ClassPath`**, because the default value of **`Bundle-ClassPath`** does *not* include all of the resources in the WAR file (see Table A.2, “Default Instructions for Wrapping a WAR File”).

References

Support for running WARs in the OSGi container is provided by the PAX WAR Extender, which monitors each bundle as it starts and, if the bundle contains a **`web.xml`** file, automatically deploys the WAR in a Web container. The War Protocol page has the original reference documentation for the War URL handler.

9.2. CONFIGURING THE WEB CONTAINER

Overview

Red Hat JBoss Fuse automatically deploys WAR files into a Web container, which is implemented by the PAX Web library. You can configure the Web container through the OSGi Configuration Admin service.

Configuration file

The Web container uses the following configuration file:

```
EsbInstallDir/etc/org.ops4j.pax.web.cfg
```

You must create this file, if it does not already exist in the ***EsbInstallDir*/etc/** directory.

Customizing the HTTP port

By default, the Web container listens on the IP port, 8181. You can change this value by editing the **etc/org.ops4j.pax.web.cfg** file and setting the value of the **org.osgi.service.http.port** property, as follows:

```
# Configure the Web container
org.osgi.service.http.port=8181
```

Enabling SSL/TLS security

The Web container is also used for deploying the JBoss Fuse Web console. The instructions for securing the Web container with SSL/TLS are identical to the instructions for securing the Web console with SSL/TLS. See chapter "Securing the Web Console" in "Security Guide" for details.

Reference

The properties that you can set in the Web container's configuration file are defined by the PAX Web library. You can set the following kinds of property:

- Basic
- SSL
- JSP

PART IV. OSGI BUNDLE DEPLOYMENT MODEL

Abstract

The OSGi bundle is the underlying unit of deployment for the Red Hat JBoss Fuse container. You can either package and deploy your applications directly as bundles or use one of the alternative deployment models (FAB or WAR).

CHAPTER 10. INTRODUCTION TO OSGI

Abstract

The OSGi specification supports modular application development by defining a runtime framework that simplifies building, deploying, and managing complex applications.

10.1. RED HAT JBOSS FUSE

Overview

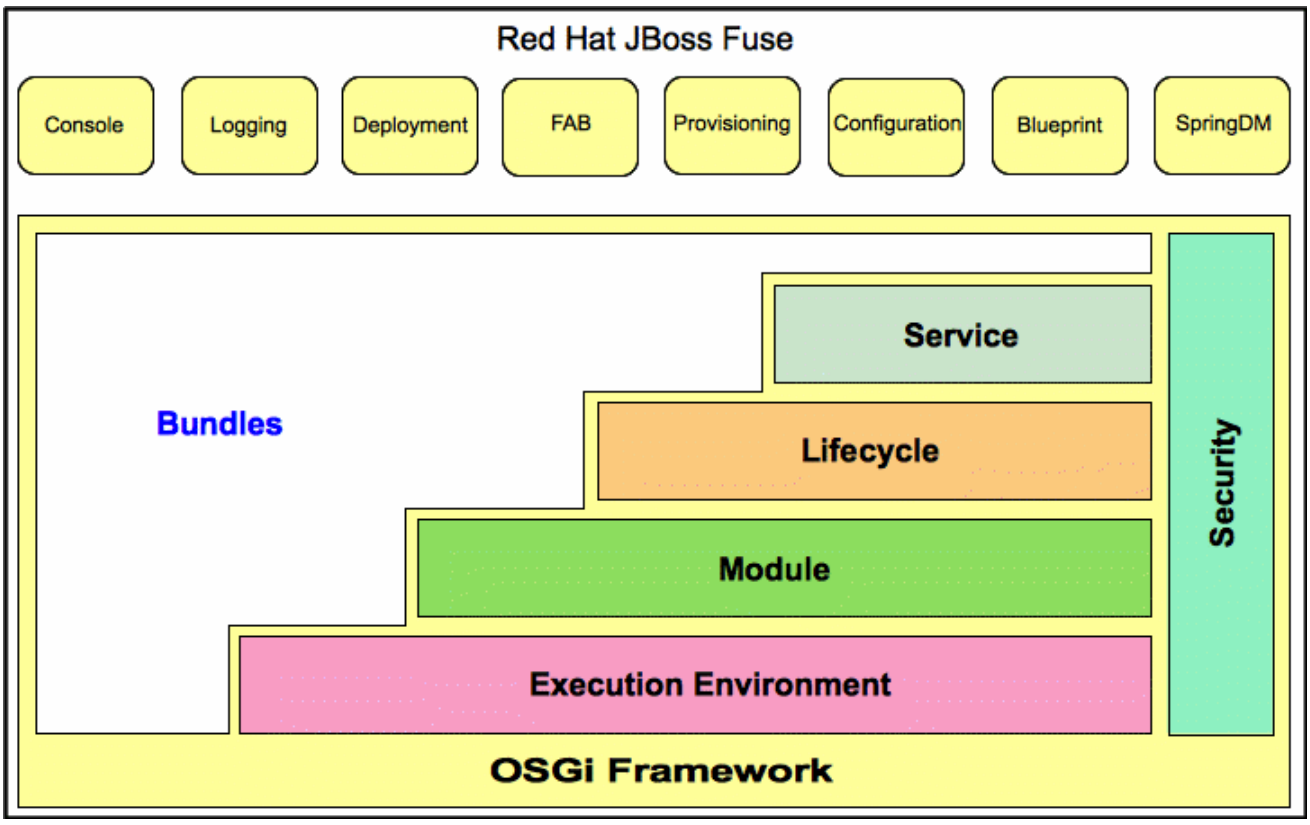
Red Hat JBoss Fuse has the following layered architecture:

- **Technology layer**—includes technologies such as JBI, JAX-WS, JAX-RS, JMS, Spring, and JEE
- **the section called "Red Hat JBoss Fuse"**—a wrapper layer around the OSGi container implementation, which provides support for deploying the OSGi container as a runtime server. Runtime features provided by the Red Hat JBoss Fuse include hot deployment, management, and administration features.
- **OSGi framework** —implements OSGi functionality, including managing dependencies and bundle lifecycles

Red Hat JBoss Fuse

Figure 10.1 shows the architecture of Red Hat JBoss Fuse.

Figure 10.1. Red Hat JBoss Fuse Architecture



Red Hat JBoss Fuse is based on Apache Karaf, a powerful, lightweight, OSGi-based runtime container for deploying and managing bundles to facilitate componentization of applications. Red Hat JBoss Fuse also provides native OS integration and can be integrated into the operating system as a service so that the lifecycle is bound to the operating system.

As shown in Figure 10.1, Red Hat JBoss Fuse extends the OSGi layers with:

- **Console**— an extensible Gogo console manages services, installs and manages applications and libraries, and interacts with the Red Hat JBoss Fuse runtime. It provides console commands to administer instances of Red Hat JBoss Fuse. See the "Console Reference".
- **Logging**—a powerful, unified logging subsystem provides console commands to display, view and change log levels. See "Configuring and Running Red Hat JBoss Fuse".
- **Deployment**—supports both manual deployment of OSGi bundles using the `osgi:install` and `osgi:start` commands and hot deployment of applications. When a JAR file, WAR file, FAB file, or OSGi bundle is copied into the hot deployment folder *InstallDir*`/deploy`, it's automatically installed on-the-fly inside the Red Hat JBoss Fuse runtime. When you update or delete these files or bundles, the changes are made automatically. See Section 12.1, “Hot Deployment”.
- **Fuse Application Bundle (FAB)**—FABs automate the creation and maintenance of OSGi bundles, freeing application developers to focus on building their applications. See Part II, “The Fuse Application Bundle Deployment Model”.
- **Provisioning**—provides multiple mechanisms for installing applications and libraries. See Chapter 13, *Deploying Features*.
- **Configuration**—the properties files stored in the *InstallDir*`/etc` folder are continuously monitored, and changes to them are automatically propagated to the relevant services at configurable intervals.
- **Spring DM**—simplifies building Spring applications that run in an OSGi framework. When a Spring configuration file is copied to the hot deployment folder, Red Hat JBoss Fuse generates and OSGi bundle on-the-fly and instantiates the Spring application context.
- **Blueprint**—is essentially a standardized version of Spring DM. It is a dependency injection framework that simplifies interaction with the OSGi container—for example, providing standard XML elements to import and export OSGi services.

10.2. OSGI FRAMEWORK

Overview

The OSGi Alliance is an independent organization responsible for defining the features and capabilities of the OSGi Service Platform Release 4. The OSGi Service Platform is a set of open specifications that simplify building, deploying, and managing complex software applications.

OSGi technology is often referred to as the dynamic module system for Java. OSGi is a framework for Java that uses bundles to modularly deploy Java components and handle dependencies, versioning, classpath control, and class loading. OSGi's lifecycle management allows you to load, start, and stop bundles without shutting down the JVM.

OSGi provides the best runtime platform for Java, a superior class loading architecture, and a registry for services. Bundles can export services, run processes, and have their dependencies managed. Each bundle can have its requirements managed by the OSGi container.

Red Hat JBoss Fuse uses Apache Felix as its default OSGi implementation. The framework layers form the container where you install bundles. The framework manages the installation and updating of bundles in a dynamic, scalable manner, and manages the dependencies between bundles and services.

OSGi architecture

As shown in Figure 10.1, " Red Hat JBoss Fuse Architecture", the OSGi framework contains the following:

- **Bundles** — Logical modules that make up an application. See Section 10.4, "OSGi Bundles".
- **Service layer** — Provides communication among modules and their contained components. This layer is tightly integrated with the lifecycle layer. See Section 10.3, "OSGi Services".
- **Lifecycle layer** — Provides access to the underlying OSGi framework. This layer handles the lifecycle of individual bundles so you can manage your application dynamically, including starting and stopping bundles.
- **Module layer** — Provides an API to manage bundle packaging, dependency resolution, and class loading.
- **Execution environment** — A configuration of a JVM. This environment uses profiles that define the environment in which bundles can work.
- **Security layer** — Optional layer based on Java 2 security, with additional constraints and enhancements.

Each layer in the framework depends on the layer beneath it. For example, the lifecycle layer requires the module layer. The module layer can be used without the lifecycle and service layers.

10.3. OSGI SERVICES

Overview

An OSGi service is a Java class or service interface with service properties defined as name/value pairs. The service properties differentiate among service providers that provide services with the same service interface.

An OSGi service is defined semantically by its service interface, and it is implemented as a service object. A service's functionality is defined by the interfaces it implements. Thus, different applications can implement the same service.

Service interfaces allow bundles to interact by binding interfaces, not implementations. A service interface should be specified with as few implementation details as possible.

OSGi service registry

In the OSGi framework, the service layer provides communication between bundles and their contained components using the publish, find, and bind service model. The service layer contains a service registry where:

- Service providers register services with the framework to be used by other bundles
- Service requesters find services and bind to service providers

Services are owned by, and run within, a bundle. The bundle registers an implementation of a service with the framework service registry under one or more Java interfaces. Thus, the service's functionality is available to other bundles under the control of the framework, and other bundles can look up and use the service. Lookup is performed using the Java interface and service properties.

Each bundle can register multiple services in the service registry using the fully qualified name of its interface and its properties. Bundles use names and properties with LDAP syntax to query the service registry for services.

A bundle is responsible for runtime service dependency management activities including publication, discovery, and binding. Bundles can also adapt to changes resulting from the dynamic availability (arrival or departure) of the services that are bound to the bundle.

Event notification

Service interfaces are implemented by objects created by a bundle. Bundles can:

- Register services
- Search for services
- Receive notifications when their registration state changes

The OSGi framework provides an event notification mechanism so service requesters can receive notification events when changes in the service registry occur. These changes include the publication or retrieval of a particular service and when services are registered, modified, or unregistered.

Service invocation model

When a bundle wants to use a service, it looks up the service and invokes the Java object as a normal Java call. Therefore, invocations on services are synchronous and occur in the same thread. You can use callbacks for more asynchronous processing. Parameters are passed as Java object references. No marshalling or intermediary canonical formats are required as with XML. OSGi provides solutions for the problem of services being unavailable.

OSGi framework services

In addition to your own services, the OSGi framework provides the following optional services to manage the operation of the framework:

- **Package Admin service**—allows a management agent to define the policy for managing Java package sharing by examining the status of the shared packages. It also allows the management agent to refresh packages and to stop and restart bundles as required. This service enables the management agent to make decisions regarding any shared packages when an exporting bundle is uninstalled or updated.

  The service also provides methods to refresh exported packages that were removed or updated since the last refresh, and to explicitly resolve specific bundles. This service can also trace dependencies between bundles at runtime, allowing you to see what bundles might be affected by upgrading.

- **Start Level service**—enables a management agent to control the starting and stopping order of bundles. The service assigns each bundle a start level. The management agent can modify the start level of bundles and set the active start level of the framework, which starts and stops the appropriate bundles. Only bundles that have a start level less than, or equal to, this active start level can be active.

- **URL Handlers service**—dynamically extends the Java runtime with URL schemes and content handlers enabling any component to provide additional URL handlers.
- **Permission Admin service**—enables the OSGi framework management agent to administer the permissions of a specific bundle and to provide defaults for all bundles. A bundle can have a single set of permissions that are used to verify that it is authorized to execute privileged code. You can dynamically manipulate permissions by changing policies on the fly and by adding new policies for newly installed components. Policy files are used to control what bundles can do.
- **Conditional Permission Admin service**—extends the Permission Admin service with permissions that can apply when certain conditions are either true or false at the time the permission is checked. These conditions determine the selection of the bundles to which the permissions apply. Permissions are activated immediately after they are set.

The OSGi framework services are described in detail in separate chapters in the *OSGi Service Platform Release 4* specification available from the release 4 download page on the OSGi Alliance web site.

OSGi Compendium services

In addition to the OSGi framework services, the OSGi Alliance defines a set of optional, standardized compendium services. The OSGi compendium services provide APIs for tasks such as logging and preferences. These services are described in the *OSGi Service Platform, Service Compendium* available from the release 4 download page on the OSGi Alliance Web site.

The *Configuration Admin* compendium service is like a central hub that persists configuration information and distributes it to interested parties. The Configuration Admin service specifies the configuration information for deployed bundles and ensures that the bundles receive that data when they are active. The configuration data for a bundle is a list of name-value pairs. See the section called “Red Hat JBoss Fuse”.

10.4. OSGI BUNDLES

Overview

With OSGi, you modularize applications into bundles. Each bundle is a tightly coupled, dynamically loadable collection of classes, JARs, and configuration files that explicitly declare any external dependencies. In OSGi, a bundle is the primary deployment format. Bundles are applications that are packaged in JARs, and can be installed, started, stopped, updated, and removed.

OSGi provides a dynamic, concise, and consistent programming model for developing bundles. Development and deployment are simplified by decoupling the service's specification (Java interface) from its implementation.

The OSGi bundle abstraction allows modules to share Java classes. This is a static form of reuse. The shared classes must be available when the dependent bundle is started.

A bundle is a JAR file with metadata in its OSGi manifest file. A bundle contains class files and, optionally, other resources and native libraries. You can explicitly declare which packages in the bundle are visible externally (exported packages) and which external packages a bundle requires (imported packages).

The module layer handles the packaging and sharing of Java packages between bundles and the hiding of packages from other bundles. The OSGi framework dynamically resolves dependencies among bundles. The framework performs bundle resolution to match imported and exported packages. It can also manage multiple versions of a deployed bundle.

Class Loading in OSGi

OSGi uses a graph model for class loading rather than a tree model (as used by the JVM). Bundles can share and re-use classes in a standardized way, with no runtime class-loading conflicts.

Each bundle has its own internal classpath so that it can serve as an independent unit if required.

The benefits of class loading in OSGi include:

- Sharing classes directly between bundles. There is no requirement to promote JARs to a parent class-loader.
- You can deploy different versions of the same class at the same time, with no conflict.

CHAPTER 11. BUILDING AN OSGI BUNDLE

Abstract

This chapter describes how to build an OSGi bundle using Maven. For building bundles, the Maven bundle plug-in plays a key role, because it enables you to automate the generation of OSGi bundle headers (which would otherwise be a tedious task). Maven archetypes, which generate a complete sample project, can also provide a starting point for your bundle projects.

11.1. GENERATING A BUNDLE PROJECT

Generating bundle projects with Maven archetypes

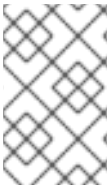
To help you get started quickly, you can invoke a Maven archetype to generate the initial outline of a Maven project (a Maven archetype is analogous to a project wizard). The following Maven archetypes can generate projects for building OSGi bundles:

- the section called “Apache CXF code-first archetype”.
- the section called “Apache CXF WSDL-first archetype”.
- the section called “Apache Camel archetype”.

Apache CXF code-first archetype

The Apache CXF code-first archetype creates a project for building a service from Java. To generate a Maven project with the coordinates, *GroupId*:*ArtifactId*:*Version*, enter the following command:

```
mvn archetype:generate
-DarchetypeGroupId=org.apache.servicemix.tooling
-DarchetypeArtifactId=servicemix-cxf-code-first-osgi-bundle
-DarchetypeVersion=2012.01.0.redhat-60024
-DgroupId=GroupId
-DartifactId=ArtifactId
-Dversion=Version
```



NOTE

The arguments to the **mvn** command are shown on separate lines purely for the sake of readability. When you are entering the command at a command prompt, you must ensure that all of the parameters are on the same line.

Apache CXF WSDL-first archetype

The Apache CXF WSDL-first archetype creates a project for building a service from WSDL. To generate a Maven project with the coordinates, *GroupId*:*ArtifactId*:*Version*, enter the following command:

```
mvn archetype:generate
-DarchetypeGroupId=org.apache.servicemix.tooling
-DarchetypeArtifactId=servicemix-cxf-wsdl-first-osgi-bundle
-DarchetypeVersion=2012.01.0.redhat-60024
```

```
-DgroupId=GroupId
-DartifactId=ArtifactId
-Dversion=Version
```

Apache Camel archetype

The Apache Camel OSGi archetype creates a project for building a route that can be deployed into the OSGi container. To generate a Maven project with the coordinates, *GroupId*:*ArtifactId*:*Version*, enter the following command:

```
mvn archetype:generate
-DarchetypeGroupId=org.apache.servicemix.tooling
-DarchetypeArtifactId=servicemix-camel-osgi-bundle
-DarchetypeVersion=2012.01.0.redhat-60024
-DgroupId=GroupId
-DartifactId=ArtifactId
-Dversion=Version
```

Building the bundle

By default, the preceding archetypes create a project in a new directory, whose names is the same as the specified artifact ID, *ArtifactId*. To build the bundle defined by the new project, open a command prompt, go to the project directory (that is, the directory containing the **`pom.xml`** file), and enter the following Maven command:

```
mvn install
```

The effect of this command is to compile all of the Java source files, to generate a bundle JAR under the *ArtifactId*/**`target`** directory, and then to install the generated JAR in the local Maven repository.

11.2. MODIFYING AN EXISTING MAVEN PROJECT

Overview

If you already have a Maven project and you want to modify it so that it generates an OSGi bundle, perform the following steps:

1. the section called “Change the package type to bundle”.
2. the section called “Add the bundle plug-in to your POM”.
3. the section called “Customize the bundle plug-in”.
4. the section called “Customize the JDK compiler version”.

Change the package type to bundle

Configure Maven to generate an OSGi bundle by changing the package type to **`bundle`** in your project's **`pom.xml`** file. Change the contents of the **`packaging`** element to **`bundle`**, as shown in the following example:

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

```
  ...
  <packaging>bundle</packaging>
  ...
</project>
```

The effect of this setting is to select the Maven bundle plug-in, **maven-bundle-plugin**, to perform packaging for this project. This setting on its own, however, has no effect until you explicitly add the bundle plug-in to your POM.

Add the bundle plug-in to your POM

To add the Maven bundle plug-in, copy and paste the following sample **plugin** element into the **project/build/plugins** section of your project's **pom.xml** file:

```
<project ... >
  ...
  <build>
    <defaultGoal>install</defaultGoal>
    <plugins>
      ...
      <plugin>
        <groupId>org.apache.felix</groupId>
        <artifactId>maven-bundle-plugin</artifactId>
        <extensions>true</extensions>
        <configuration>
          <instructions>
            <Bundle-SymbolicName>${project.groupId}.${project.artifactId}
</Bundle-SymbolicName>
            <Import-Package>*</Import-Package>
          </instructions>
        </configuration>
      </plugin>
    </plugins>
  </build>
  ...
</project>
```

Where the bundle plug-in is configured by the settings in the **instructions** element.

Customize the bundle plug-in

For some specific recommendations on configuring the bundle plug-in for Apache CXF and Apache Camel, see Section 11.3, “Packaging a Web Service in a Bundle” and the section called “Spring example”.

For an in-depth discussion of bundle plug-in configuration, in the context of the OSGi framework and versioning policy, see "Managing OSGi Dependencies".

Customize the JDK compiler version

It is almost always necessary to specify the JDK version in your POM file. If your code uses any modern features of the Java language—such as generics, static imports, and so on—and you have not customized the JDK version in the POM, Maven will fail to compile your source code. It is *not* sufficient to

set the **JAVA_HOME** and the **PATH** environment variables to the correct values for your JDK, you must also modify the POM file.

To configure your POM file, so that it accepts the Java language features introduced in JDK 1.6, add the following **maven-compiler-plugin** plug-in settings to your POM (if they are not already present):

```
<project ... >
  ...
  <build>
    <defaultGoal>install</defaultGoal>
    <plugins>
      ...
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
          <source>1.6</source>
          <target>1.6</target>
        </configuration>
      </plugin>
    </plugins>
  </build>
  ...
</project>
```

11.3. PACKAGING A WEB SERVICE IN A BUNDLE

Overview

This section explains how to modify an existing Maven project for a Apache CXF application, so that the project generates an OSGi bundle suitable for deployment in the Red Hat JBoss Fuse OSGi container. To convert the Maven project, you need to modify the project's POM file and the project's Spring XML file(s) (located in **META-INF/spring**).

Modifying the POM file to generate a bundle

To configure a Maven POM file to generate a bundle, there are essentially two changes you need to make: change the POM's package type to **bundle**; and add the Maven bundle plug-in to your POM. For details, see Section 11.1, “Generating a Bundle Project”.

Required bundle

The Apache CXF runtime components are included in JBoss Fuse as an OSGi bundle called **org.apache.cxf.bundle**. The dependency on this bundle can conveniently be expressed by adding the **Require-Bundle** element to the Maven bundle plug-in's instructions, as highlighted in the following sample POM:

```
<project ... >
    ...
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.felix</groupId>
```

```
                <artifactId>maven-bundle-plugin</artifactId>
                <extensions>true</extensions>
                <configuration>
                    <instructions>
                        ...
                        <Require-Bundle>org.apache.cxf.bundle</Require-
Bundle>
                        ...
                    </instructions>
                </configuration>
            </plugin>
        </plugins>
    </build>
    ...
</project>
```

Mandatory import packages

In order for your application to use the Apache CXF components, you need to import their packages into the application's bundle. Because of the complex nature of the dependencies in Apache CXF, you cannot rely on the Maven bundle plug-in, or the **bnd** tool, to automatically determine the needed imports. You will need to explicitly declare them.

You need to import the following packages into your bundle:

```
javax.jws
javax.wsdl
javax.xml.bind
javax.xml.bind.annotation
javax.xml.namespace
javax.xml.ws
META-INF.cxf
META-INF.cxf.osgi
org.apache.cxf.bus
org.apache.cxf.bus.spring
org.apache.cxf.bus.resource
org.apache.cxf.configuration.spring
org.apache.cxf.resource
org.apache.cxf.jaxws
org.springframework.beans.factory.config
```

Sample Maven bundle plug-in instructions

Example 11.1, “Configuration of Mandatory Import Packages” shows how to configure the Maven bundle plug-in in your POM to import the mandatory packages. The mandatory import packages appear as a comma-separated list inside the **`Import-Package`** element. Note the appearance of the wildcard, **`*`**, as the last element of the list. The wildcard ensures that the Java source files from the current bundle are scanned to discover what additional packages need to be imported.

Example 11.1. Configuration of Mandatory Import Packages

```
<project ... >
    ...
    <build>
```

```
        <plugins>
            <plugin>
                <groupId>org.apache.felix</groupId>
                <artifactId>maven-bundle-plugin</artifactId>
                <extensions>true</extensions>
                <configuration>
                    <instructions>
                        ...
                        <Import-Package>
                            javax.jws,
                            javax.wsdl,
                            javax.xml.bind,
                            javax.xml.bind.annotation,
                            javax.xml.namespace,
                            javax.xml.ws,
                            META-INF.cxf,
                            META-INF.cxf.osgi,
                            org.apache.cxf.bus,
                            org.apache.cxf.bus.spring,
                            org.apache.cxf.bus.resource,
                            org.apache.cxf.configuration.spring,
                            org.apache.cxf.resource,
                            org.apache.cxf.jaxws,
                            org.springframework.beans.factory.config,
                            *
                        </Import-Package>
                        ...
                    </instructions>
                </configuration>
            </plugin>
        </plugins>
    </build>
    ...
</project>
```

Add a code generation plug-in

A Web services project typically requires code to be generated. Apache CXF provides two Maven plug-ins for the JAX-WS front-end, which enable tyou to integrate the code generation step into your build. The choice of plug-in depends on whether you develop your service using the Java-first approach or the WSDL-first approach, as follows:

- *Java-first approach*—use the **`cxf-java2ws-plugin`** plug-in.
- *WSDL-first approach*—use the **`cxf-codegen-plugin`** plug-in.

OSGi configuration properties

The OSGi Configuration Admin service defines a mechanism for passing configuration settings to an OSGi bundle. You do not have to use this service for configuration, but it is typically the most convenient way of configuring bundle applications. Both Spring DM and Blueprint provide support for OSGi configuration, enabling you to substitute variables in a Spring XML file or a Blueprint file using values obtained from the OSGi Configuration Admin service.

For details of how to use OSGi configuration properties, see Section 11.4, “Configuring the Bundle Plug-In” and the section called “Add OSGi configurations to the feature”.

11.4. CONFIGURING THE BUNDLE PLUG-IN

Overview

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

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

Configuration properties

Some of the commonly used configuration properties are:

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

Setting a bundle's symbolic name

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

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

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

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

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

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

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

To specify your own value for the bundle's symbolic name, add a **`Bundle-SymbolicName`** child in the plug-in's **`instructions`** element, as shown in Example 11.2.

> **Example 11.2. Setting a bundle's symbolic name**

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

Setting a bundle's name

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

To specify your own value for the bundle's name, add a **`Bundle-Name`** child to the plug-in's **`instructions`** element, as shown in Example 11.3.

Example 11.3. Setting a bundle's name

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

Setting a bundle's version

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

To specify your own value for the bundle's version, add a **`Bundle-Version`** child to the plug-in's **`instructions`** element, as shown in Example 11.4.

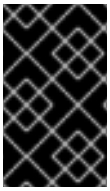
Example 11.4. Setting a bundle's version

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

```
      </instructions>
    </configuration>
  </plugin>
```

Specifying exported packages

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



IMPORTANT

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

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

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

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

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

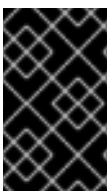
For example, to include all packages starting with com.fuse.demo except the package com.fuse.demo.private, list the packages using:

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

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

Specifying private packages

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



IMPORTANT

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

The **Private-Package** element works similarly to the **Export-Package** element in that you specify a

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

Example 11.5 shows the configuration for including a private package in a bundle

Example 11.5. Including a private package in a bundle

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

Specifying imported packages

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

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

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

IMPORTANT

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

Example 11.6 shows the configuration for specifying the packages imported by a bundle

Example 11.6. Specifying the packages imported by a bundle

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

```
            org.apache.cxf.bus.resource,
            org.apache.cxf.configuration.spring,
            org.apache.cxf.resource,
            org.springframework.beans.factory.config,
            *
        </Import-Package>
        ...
      </instructions>
    </configuration>
  </plugin>
```

More information

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

- "Managing OSGi Dependencies"
- Apache Felix documentation
- Peter Kriens' aQute Software Consultancy web site

CHAPTER 12. DEPLOYING AN OSGI BUNDLE

Abstract

Apache Karaf provides two different approaches for deploying a single OSGi bundle: hot deployment or manual deployment. If you need to deploy a collection of related bundles, on the other hand, it is recommended that you deploy them together as a *feature*, rather than singly (see Chapter 13, *Deploying Features*).

12.1. HOT DEPLOYMENT

Hot deploy directory

Red Hat JBoss Fuse monitors JAR files in the ***InstallDir*/deploy** directory and hot deploys everything in this directory. Each time a JAR file is copied to this directory, it is installed in the runtime and also started. You can subsequently update or delete the JARs, and the changes are handled automatically.

For example, if you have just built the bundle, *ProjectDir*/**target/foo-1.0-SNAPSHOT.jar**, you can deploy this bundle by copying it to the *InstallDir*/**deploy** directory as follows (assuming you are working on a UNIX platform):

```
% cp ProjectDir/target/foo-1.0-SNAPSHOT.jar InstallDir/deploy
```

12.2. MANUAL DEPLOYMENT

Overview

You can manually deploy and undeploy bundles by issuing commands at the Red Hat JBoss Fuse console.

Installing a bundle

Use the **osgi:install** command to install one or more bundles in the OSGi container. This command has the following syntax:

```
osgi:install [-s] [--start] [--help] UrlList
```

Where *UrlList* is a whitespace-separated list of URLs that specify the location of each bundle to deploy. The following command arguments are supported:

-s

Start the bundle after installing.

--start

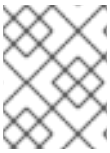
Same as **-s**.

--help

Show and explain the command syntax.

For example, to install and start the bundle, *ProjectDir*/**target/foo-1.0-SNAPSHOT.jar**, enter the following command at the Karaf console prompt:

```
osgi:install -s file:ProjectDir/target/foo-1.0-SNAPSHOT.jar
```



NOTE

On Windows platforms, you must be careful to use the correct syntax for the **file** URL in this command. See Section A.1, “File URL Handler” for details.

Uninstalling a bundle

To uninstall a bundle, you must first obtain its bundle ID using the **osgi:list** command. You can then uninstall the bundle using the **osgi:uninstall** command (which takes the bundle ID as its argument).

For example, if you have already installed the bundle named **A Camel OSGi Service Unit**, entering **osgi:list** at the console prompt might produce output like the following:

```
...
[ 175] [Active     ] [            ] [Started] [   60] ServiceMix :: FTP
(2009.02.0.psc-01-00RC1)
[ 181] [Resolved   ] [            ] [       ] [   60] A Camel OSGi
Service Unit (1.0.0.SNAPSHOT)
```

You can now uninstall the bundle with the ID, **181**, by entering the following console command:

```
osgi:uninstall 181
```

URL schemes for locating bundles

When specifying the location URL to the **osgi:install** command, you can use any of the URL schemes supported by Red Hat JBoss Fuse, which includes the following scheme types:

- Section A.1, “File URL Handler”.
- .Section A.2, “HTTP URL Handler”.
- .Section A.3, “Mvn URL Handler”.

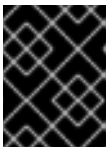
12.3. LIFECYCLE MANAGEMENT

Bundle lifecycle states

Applications in an OSGi environment are subject to the lifecycle of its bundles. Bundles have six lifecycle states:

1. **Installed** — All bundles start in the installed state. Bundles in the installed state are waiting for all of their dependencies to be resolved, and once they are resolved, bundles move to the resolved state.
2. **Resolved** — Bundles are moved to the resolved state when the following conditions are met:

- The runtime environment meets or exceeds the environment specified by the bundle.
- All of the packages imported by the bundle are exposed by bundles that are either in the resolved state or that can be moved into the resolved state at the same time as the current bundle.
- All of the required bundles are either in the resolved state or they can be resolved at the same time as the current bundle.



IMPORTANT

All of an application's bundles must be in the resolved state before the application can be started.

If any of the above conditions ceases to be satisfied, the bundle is moved back into the installed state. For example, this can happen when a bundle that contains an imported package is removed from the container.

3. **Starting** — The starting state is a transitory state between the resolved state and the active state. When a bundle is started, the container must create the resources for the bundle. The container also calls the **`start()`** method of the bundle's bundle activator when one is provided.
4. **Active** — Bundles in the active state are available to do work. What a bundle does in the active state depends on the contents of the bundle. For example, a bundle containing a JAX-WS service provider indicates that the service is available to accept requests.
5. **Stopping** — The stopping state is a transitory state between the active state and the resolved state. When a bundle is stopped, the container must clean up the resources for the bundle. The container also calls the **`stop()`** method of the bundle's bundle activator when one is provided.
6. **Uninstalled** — When a bundle is uninstalled it is moved from the resolved state to the uninstalled state. A bundle in this state cannot be transitioned back into the resolved state or any other state. It must be explicitly re-installed.

The most important lifecycle states for application developers are the starting state and the stopping state. The endpoints exposed by an application are published during the starting state. The published endpoints are stopped during the stopping state.

Installing and resolving bundles

When you install a bundle using the **`osgi:install`** command (without the **`-s`** flag), the kernel installs the specified bundle and attempts to put it into the resolved state. If the resolution of the bundle fails for some reason (for example, if one of its dependencies is unsatisfied), the kernel leaves the bundle in the installed state.

At a later time (for example, after you have installed missing dependencies) you can attempt to move the bundle into the resolved state by invoking the **`osgi:resolve`** command, as follows:

```
osgi:resolve 181
```

Where the argument (**`181`**, in this example) is the ID of the bundle you want to resolve.

Starting and stopping bundles

You can start one or more bundles (from either the installed or the resolved state) using the **osgi:start** command. For example, to start the bundles with IDs, 181, 185, and 186, enter the following console command:

```
osgi:start 181 185 186
```

You can stop one or more bundles using the **osgi:stop** command. For example, to stop the bundles with IDs, 181, 185, and 186, enter the following console command:

```
osgi:stop 181 185 186
```

You can restart one or more bundles (that is, moving from the started state to the resolved state, and then back again to the started state) using the **osgi:restart** command. For example, to restart the bundles with IDs, 181, 185, and 186, enter the following console command:

```
osgi:restart 181 185 186
```

Bundle start level

A *start level* is associated with every bundle. The start level is a positive integer value that controls the order in which bundles are activated/started. Bundles with a low start level are started before bundles with a high start level. Hence, bundles with the start level, **1**, are started first and bundles belonging to the kernel tend to have lower start levels, because they provide the prerequisites for running most other bundles.

Typically, the start level of user bundles is 60 or higher.

Specifying a bundle's start level

Use the **osgi:bundle-level** command to set the start level of a particular bundle. For example, to configure the bundle with ID, **181**, to have a start level of **70**, enter the following console command:

```
osgi:bundle-level 181 70
```

System start level

The OSGi container itself has a start level associated with it and this *system start level* determines which bundles can be active and which cannot: only those bundles whose start level is *less than or equal* to the system start level can be active.

To discover the current system start level, enter **osgi:start-level** in the console, as follows:

```
JBossFuse:karaf@root> osgi:start-level
Level 100
```

If you want to change the system start level, provide the new start level as an argument to the **osgi:start-level** command, as follows:

```
osgi:start-level 200
```

12.4. TROUBLESHOOTING DEPENDENCIES

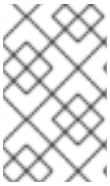
Missing dependencies

The most common issue that can arise when you deploy an OSGi bundle into the Red Hat JBoss Fuse container is that one or more dependencies are missing. This problem shows itself when you try to resolve the bundle in the OSGi container, usually as a side effect of starting the bundle. The bundle fails to resolve (or start) and a **`ClassNotFound`** error is logged (to view the log, use the **`log:display`** console command or look at the log file in the ***`InstallDir`*`/data/log`** directory).

There are two basic causes of a missing dependency: either a required feature or bundle is not installed in the container; or your bundle's **`Import-Package`** header is incomplete.

Required features or bundles are not installed

Evidently, all features and bundles required by your bundle must *already* be installed in the OSGi container, before you attempt to resolve your bundle. In particular, because Apache Camel has a modular architecture, where each component is installed as a separate feature, it is easy to forget to install one of the required components.



NOTE

Consider packaging your bundle as a feature. Using a feature, you can package your bundle together with all of its dependencies and thus ensure that they are all installed simultaneously. For details, see Chapter 13, *Deploying Features*.

Import-Package header is incomplete

If all of the required features and bundles are already installed and you are still getting a **`ClassNotFound`** error, this means that the **`Import-Package`** header in your bundle's **`MANIFEST.MF`** file is incomplete. The **`maven-bundle-plugin`** (see Section 11.2, "Modifying an Existing Maven Project") is a great help when it comes to generating your bundle's **`Import-Package`** header, but you should note the following points:

- Make sure that you include the wildcard, **`*`**, in the **`Import-Package`** element of the Maven bundle plug-in configuration. The wildcard directs the plug-in to scan your Java source code and automatically generates a list of package dependencies.
- The Maven bundle plug-in is *not* able to figure out dynamic dependencies. For example, if your Java code explicitly calls a class loader to load a class dynamically, the bundle plug-in does not take this into account and the required Java package will not be listed in the generated **`Import-Package`** header.
- If you define a Spring XML file (for example, in the **`META-INF/spring`** directory), the Maven bundle plug-in is *not* able to figure out dependencies arising from the Spring XML configuration. Any dependencies arising from Spring XML must be added manually to the bundle plug-in's **`Import-Package`** element.
- If you define a blueprint XML file (for example, in the **`OSGI-INF/blueprint`** directory), any dependencies arising from the blueprint XML file are *automatically resolved at run time*. This is an important advantage of blueprint over Spring.

How to track down missing dependencies

To track down missing dependencies, perform the following steps:

1. Perform a quick check to ensure that all of the required bundles and features are actually installed in the OSGi container. You can use **`osgi:list`** to check which bundles are installed and **`features:list`** to check which features are installed.

2. Install (but do not start) your bundle, using the **`osgi:install`** console command. For example:

   ```
   JBossFuse:karaf@root> osgi:install MyBundleURL
   ```

3. Use the **`dev:dynamic-import`** console command to enable dynamic imports on the bundle you just installed. For example, if the bundle ID of your bundle is 218, you would enable dynamic imports on this bundle by entering the following command:

   ```
   JBossFuse:karaf@root> dev:dynamic-import 218
   ```

   This setting allows OSGi to resolve dependencies using *any* of the bundles already installed in the container, effectively bypassing the usual dependency resolution mechanism (based on the **`Import-Package`** header). This is *not* recommemded for normal deployment, because it bypasses version checks: you could easily pick up the wrong version of a package, causing your application to malfunction.

4. You should now be able to resolve your bundle. For example, if your bundle ID is 218, enter the followng console command:

   ```
   JBossFuse:karaf@root> osgi:resolve 218
   ```

5. Assuming your bundle is now resolved (check the bundle status using **`osgi:list`**), you can get a complete list of all the packages wired to your bundle using the **`package:imports`** command. For example, if your bundle ID is 218, enter the following console command:

   ```
   JBossFuse:karaf@root> package:imports 218
   ```

   You should see a list of dependent packages in the console window (where the package names are highlighted in this example):

   ```
   Spring Beans (67): org.springframework.beans.factory.xml;
   version=3.0.5.RELEASE
   Apache ServiceMix :: Specs :: JAXB API 2.2 (87):
   javax.xml.bind.annotation; version=2.2.1
   Apache ServiceMix :: Specs :: JAXB API 2.2 (87): javax.xml.bind;
   version=2.2.1
   Web Services Metadata 2.0 (104): javax.jws; version=2.0.0
   Apache ServiceMix :: Specs :: JAXWS API 2.2 (105):
   javax.xml.ws.handler; version=2.2.0
   Apache ServiceMix :: Specs :: JAXWS API 2.2 (105): javax.xml.ws;
   version=2.2.0
   Apache CXF Bundle Jar (125): org.apache.cxf.helpers;
   version=2.4.2.fuse-00-08
   Apache CXF Bundle Jar (125): org.apache.cxf.transport.jms.wsdl11;
   version=2.4.2.fuse-00-08
   ...
   ```

6. Unpack your bundle JAR file and look at the packages listed under the **`Import-Package`** header in the **`META-INF/MANIFEST.MF`** file. Compare this list with the list of packages found in

the previous step. Now, compile a list of the packages that are missing from the manifest's **`Import-Package`** header and add these package names to the **`Import-Package`** element of the Maven bundle plug-in configuration in your project's POM file.

7. To cancel the dynamic import option, you must uninstall the old bundle from the OSGi container. For example, if your bundle ID is 218, enter the following command:

```
JBossFuse:karaf@root> osgi:uninstall 218
```

8. You can now rebuild your bundle with the updated list of imported packages and test it in the OSGi container.

CHAPTER 13. DEPLOYING FEATURES

Abstract

Because applications and other tools typically consist of multiple OSGi bundles, it is often convenient to aggregate inter-dependent or related bundles into a larger unit of deployment. Red Hat JBoss Fuse therefore provides a scalable unit of deployment, the *feature*, which enables you to deploy multiple bundles (and, optionally, dependencies on other features) in a single step.

Alternatively, you could switch to using the FAB model of deployment, which automatically aggregates related bundles at deployment time, without requiring any additional configuration.

13.1. CREATING A FEATURE

Overview

Essentially, a feature is created by adding a new **`feature`** element to a special kind of XML file, known as a *feature repository*. To create a feature, perform the following steps:

1. the section called “Create a custom feature repository”.
2. the section called “Add a feature to the custom feature repository”.
3. the section called “Add the local repository URL to the features service”.
4. the section called “Add dependent features to the feature”.
5. the section called “Add OSGi configurations to the feature”.

Create a custom feature repository

If you have not already defined a custom feature repository, you can create one as follows. Choose a convenient location for the feature repository on your file system—for example, **`C:\Projects\features.xml`**—and use your favorite text editor to add the following lines to it:

```
<?xml version="1.0" encoding="UTF-8"?>
<features name="CustomRepository">
</features>
```

Where you must specify a name for the repository, *CustomRepository*, by setting the **`name`** attribute.

NOTE

In contrast to a Maven repository or an OBR, a feature repository does *not* provide a storage location for bundles. A feature repository merely stores an aggregate of references to bundles. The bundles themselves are stored elsewhere (for example, in the file system or in a Maven repository).

Add a feature to the custom feature repository

To add a feature to the custom feature repository, insert a new **`feature`** element as a child of the root **`features`** element. You must give the feature a name and you can list any number of bundles

belonging to the feature, by inserting **bundle** child elements. For example, to add a feature named **example-camel-bundle** containing the single bundle, **C:\Projects\camel-bundle\target\camel-bundle-1.0-SNAPSHOT.jar**, add a **feature** element as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<features name="MyFeaturesRepo">
  <feature name="example-camel-bundle">
    <bundle>file:C:/Projects/camel-bundle/target/camel-bundle-1.0-
SNAPSHOT.jar</bundle>
  </feature>
</features>
```

The contents of the **bundle** element can be any valid URL, giving the location of a bundle (see Appendix A, *URL Handlers*). You can optionally specify a **version** attribute on the feature element, to assign a non-zero version to the feature (you can then specify the version as an optional argument to the **features:install** command).

To check whether the features service successfully parses the new feature entry, enter the following pair of console commands:

```
JBossFuse:karaf@root> features:refreshUrl
JBossFuse:karaf@root> features:list
...
[uninstalled] [0.0.0                ] example-camel-bundle
MyFeaturesRepo
...
```

The **features:list** command typically produces a rather long listing of features, but you should be able to find the entry for your new feature (in this case, **example-camel-bundle**) by scrolling back through the listing. The **features:refreshUrl** command forces the kernel to reread all the feature repositories: if you did not issue this command, the kernel would not be aware of any recent changes that you made to any of the repositories (in particular, the new feature would not appear in the listing).

To avoid scrolling through the long list of features, you can **grep** for the **example-camel-bundle** feature as follows:

```
JBossFuse:karaf@root> features:list | grep example-camel-bundle
[uninstalled] [0.0.0                ] example-camel-bundle
MyFeaturesRepo
```

Where the **grep** command (a standard UNIX pattern matching utility) is built into the shell, so this command also works on Windows platforms.

Add the local repository URL to the features service

In order to make the new feature repository available to Apache Karaf, you must add the feature repository using the **features:addUrl** console command. For example, to make the contents of the repository, **C:\Projects\features.xml**, available to the kernel, you would enter the following console command:

```
features:addUrl file:C:/Projects/features.xml
```

Where the argument to **features:addUrl** can be specified using any of the supported URL formats (see Appendix A, *URL Handlers*).

You can check that the repository's URL is registered correctly by entering the **features:listUrl** console command, to get a complete listing of all registered feature repository URLs, as follows:

```
JBossFuse:karaf@root> features:listUrl
mvn:org.apache.servicemix.nmr/apache-servicemix-nmr/1.1.0-fuse-01-
00/xml/features
mvn:org.apache.servicemix.camel/features/6.0.0.redhat-024/xml/features
file:C:/Projects/features.xml
mvn:org.apache.ode/ode-jbi-karaf/1.3.3-fuse-01-00/xml/features
mvn:org.apache.felix.karaf/apache-felix-karaf/1.2.0-fuse-01-
00/xml/features
mvn:org.apache.servicemix/apache-servicemix/6.0.0.redhat-024/xml/features
```

Add dependent features to the feature

If your feature depends on other features, you can specify these dependencies by adding **feature** elements as children of the original **feature** element. Each child **feature** element contains the name of a feature on which the current feature depends. When you deploy a feature with dependent features, the dependency mechanism checks whether or not the dependent features are installed in the container. If not, the dependency mechanism automatically installs the missing dependencies (and any recursive dependencies).

For example, for the custom Apache Camel feature, **example-camel-bundle**, you can specify explicitly which standard Apache Camel features it depends on. This has the advantage that the application could now be successfully deployed and run, even if the OSGi container does not have the required features pre-deployed. For example, you can define the **example-camel-bundle** feature with Apache Camel dependencies as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<features name="MyFeaturesRepo">
  <feature name="example-camel-bundle">
    <bundle>file:C:/Projects/camel-bundle/target/camel-bundle-1.0-
SNAPSHOT.jar</bundle>
    <feature version="6.0.0.redhat-024">camel-core</feature>
    <feature version="6.0.0.redhat-024">camel-spring-osgi</feature>
    <feature version="6.0.0.redhat-024">servicemix-camel</feature>
  </feature>
</features>
```

Specifying the **version** attribute is optional. When present, it enables you to select the specified version of the feature.

Add OSGi configurations to the feature

If your application uses the *OSGi Configuration Admin* service, you can specify configuration settings for this service using the **config** child element of your feature definition. For example, to specify that the **prefix** property has the value, **MyTransform**, add the following **config** child element to your feature's configuration:

```
<?xml version="1.0" encoding="UTF-8"?>
<features name="MyFeaturesRepo">
```

```
  <feature name="example-camel-bundle">
    <config name="org.fusesource.fuseesb.example">
      prefix=MyTransform
    </config>
  </feature>
</features>
```

Where the **name** attribute of the **config** element specifies the *persistent ID* of the property settings (where the persistent ID acts effectively as a name scope for the property names). The content of the **config** element is parsed in the same way as a Java properties file.

The settings in the **config** element can optionally be overriden by the settings in the Java properties file located in the ***InstallDir*/etc** directory, which is named after the persistent ID, as follows:

```
InstallDir/etc/org.fusesource.fuseesb.example.cfg
```

As an example of how the preceding configuration properties can be used in practice, consider the following Spring XML file that accesses the OSGi configuration properties using Spring DM:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:ctx="http://www.springframework.org/schema/context"
       xmlns:osgi="http://camel.apache.org/schema/osgi"
       xmlns:osgix="http://www.springframework.org/schema/osgi-compendium"
...>
    ...
    <bean id="myTransform"
class="org.fusesource.fuseesb.example.MyTransform">
      <property name="prefix" value="${prefix}"/>
    </bean>

    <osgix:cm-properties id="preProps" persistent-
id="org.fusesource.fuseesb.example">
        <prop key="prefix">DefaultValue</prop>
    </osgix:cm-properties>

    <ctx:property-placeholder properties-ref="preProps" />

</beans>
```

When this Spring XML file is deployed in the **example-camel-bundle** bundle, the property reference, **${prefix}**, is replaced by the value, **MyTransform**, which is specified by the **config** element in the feature repository.

13.2. DEPLOYING A FEATURE

Overview

You can deploy a feature in one of the following ways:

- Install at the console, using **features:install**.
- Use hot deployment.

- Modify the boot configuration (first boot only!).

Installing at the console

After you have created a feature (by adding an entry for it in a feature repository and registering the feature repository), it is relatively easy to deploy the feature using the **`features:install`** console command. For example, to deploy the **`example-camel-bundle`** feature, enter the following pair of console commands:

```
JBossFuse:karaf@root> features:refreshUrl
JBossFuse:karaf@root> features:install example-camel-bundle
```

It is recommended that you invoke the **`features:refreshUrl`** command before calling **`features:install`**, in case any recent changes were made to the features in the feature repository which the kernel has not picked up yet. The **`features:install`** command takes the feature name as its argument (and, optionally, the feature version as its second argument).

NOTE

Features use a flat namespace. So when naming your features, be careful to avoid name clashes with existing features.

Uninstalling at the console

To uninstall a feature, invoke the **`features:uninstall`** command as follows:

```
JBossFuse:karaf@root> features:uninstall example-camel-bundle
```

NOTE

After uninstalling, the feature will still be visible when you invoke **`features:list`**, but its status will now be flagged as **`[uninstalled]`**.

Hot deployment

You can hot deploy *all* of the features in a feature repository simply by copying the feature repository file into the ***`InstallDir`*`/deploy`** directory.

As it is unlikely that you would want to hot deploy an entire feature repository at once, it is often more convenient to define a reduced feature repository or *feature descriptor*, which references only those features you want to deploy. The feature descriptor has exactly the same syntax as a feature repository, but it is written in a different style. The difference is that a feature descriptor consists only of references to existing features from a feature repository.

For example, you could define a feature descriptor to load the **`example-camel-bundle`** feature as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<features name="CustomDescriptor">
  <repository>RepositoryURL</repository>
  <feature name="hot-example-camel-bundle">
    <feature>example-camel-bundle</feature>
  </feature>
```

```
</features>
```

The repository element specifies the location of the custom feature repository, *RepositoryURL* (where you can use any of the URL formats described in Appendix A, *URL Handlers*). The feature, **hot-example-camel-bundle**, is just a reference to the existing feature, **example-camel-bundle**.

Adding a feature to the boot configuration

If you want to provision copies of Apache Karaf for deployment on multiple hosts, you might be interested in adding a feature to the boot configuration, which determines the collection of features that are installed when Apache Karaf boots up for the very first time.

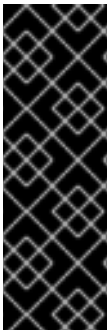
The configuration file, **/etc/org.apache.felix.karaf.features.cfg**, in your install directory contains the following settings:

```
...
#
# Comma separated list of features repositories to register by default
#
featuresRepositories=mvn:org.apache.felix.karaf/apache-felix-
karaf/1.1.0.3-fuse-SNAPSHOT/xml/features,
mvn:org.apache.servicemix.nmr/apache-servicemix-nmr/1.1.0-fuse-
SNAPSHOT/xml/features,
mvn:org.apache.servicemix/apache-servicemix/4.2.0-fuse-
SNAPSHOT/xml/features,
mvn:org.apache.camel.karaf/apache-camel/2.x-fuse-SNAPSHOT/xml/features,
mvn:org.apache.ode/ode-jbi-karaf/1.3.3-fuse-SNAPSHOT/xml/features

#
# Comma separated list of features to install at startup
#

# Will put these back in when we decide to include these components
# servicemix-smpp,servicemix-snmp,servicemix-vfs,

featuresBoot=activemq,activemq-broker,camel,jbi-cluster,web,servicemix-
cxf-bc,servicemix-file,
servicemix-ftp,servicemix-http,servicemix-jms,servicemix-mail,servicemix-
bean,servicemix-camel,
servicemix-cxf-se,servicemix-drools,servicemix-eip,servicemix-
osworkflow,servicemix-quartz,
servicemix-scripting,servicemix-validation,servicemix-saxon,servicemix-
wsn2005,camel-cxf,camel-jms
```

This configuration file has two properties:

- **featuresRepositories**—Comma separated list of feature repositories to load at startup.
- **featuresBoot**—Comma separated list of features to install at startup.

You can modify the configuration to customize the features that are installed as Red Hat JBoss Fuse starts up. You can also modify this configuration file, if you plan to distribute Red Hat JBoss Fuse with pre-installed features.



IMPORTANT

This method of adding a feature is only effective the *first time* a particular Apache Karaf instance boots up. Any changes made subsequently to the **`featuresRepositories`** setting and the **`featuresBoot`** setting are ignored, even if you restart the container.

You could force the container to revert back to its initial state, however, by deleting the complete contents of the **`InstallDir/data/cache`** (thereby losing all of the container's custom settings).

CHAPTER 14. DEPLOYING A PLAIN JAR

Abstract

This chapter explains how to deal with plain JAR files (typically libraries) that contain *no deployment metadata whatsoever*. That is, a plain JAR is neither a FAB, nor a WAR, nor an OSGi bundle.

If the plain JAR occurs as a dependency of a bundle, you must add bundle headers to the JAR . If the JAR exposes a public API, typically the best solution is to convert the existing JAR into a bundle, enabling the JAR to be shared with other bundles. This chapter describes how to perform the conversion process automatically, using the open source Bnd tool.

14.1. BUNDLE TOOL (BND)

About the bnd tool

The Bnd tool is a an open source utility for creating and diagnosing OSGi bundles. It has been developed by Peter Kriens and is freely downloadable from the aQute Web site (subject to an Apache version 2.0 open source license). The key feature of the Bnd tool is that it automatically generates Manifest headers for the OSGi bundle, thus relieving you of this tedious task. The main tasks that Bnd can perform are:

- Print the manifiest and show the package dependencies of a JAR file or bundle file.
- Wrap a vanilla JAR file, converting it into a bundle.
- Build a bundle from the class path, based on specifications in a **`.bnd`** file.
- Validate manifest entries.

You have the option of invoking Bnd in any of the following ways: from the command line; as an Ant task; or through the Maven bundle plug-in, **`maven-bundle-plugin`**. In fact, the approach to building bundles described in Chapter 3, *Building with Maven* is based on the Maven bundle plug-in and therefore, implicitly, is also based on the Bnd tool.

Downloading and installing bnd

You can download Bnd from the aQute Web site at the following location:

```
http://www.aqute.biz/Code/Download#bnd
```

Download the Bnd JAR file, **`bnd-`*`Version`*`.jar`**, to a convenient location. There is no installation involved: the JAR file is all that you need to use the Bnd tool. For convenience, however, it is advisable to rename the JAR file to **`bnd.jar`**, so you won't have to do so much typing when you invoke it from the command line (for example, as in **`java -jar bnd.jar `*`Cmd Options`***)

References

To learn more about the Bnd tool, consult the following references:

- Bnd tool documentation.

- Creating OSGi bundles (SpringSource blog).

14.2. CONVERTING A JAR USING BND

Overview

This section describes how to convert a vanilla JAR file into an OSGi bundle using the Bnd tool's **wrap** command. You can choose either to perform the conversion with default settings (which works in most cases) or to perform a custom conversion with the help of a Bnd properties file.

Sample JAR file

To demonstrate how to convert a plain JAR into a bundle, we will consider the example of the commons-logging-Version.jar, which is available from the Apache Commons project and can be downloaded from the following location:

```
http://commons.apache.org/downloads/download_logging.cgi
```

NOTE

Actually, this is a rather artificial example, because the Apache Commons logging API is *not* intended to be deployed as an OSGi bundle (which is why it does not have the requisite Manifest headers in the first place). Most of the other JARs from Apache Commons are already provided as bundles.

Bnd print command

The Bnd **print** command is a useful diagnostic tool that displays most of the information about a JAR that is relevant to bundle creation. For example, to print the Manifest headers and package dependencies of the commons logging JAR, you can invoke the Bnd **print** command as follows:

```
java -jar bnd.jar print commons-logging-1.1.1.jar
```

The preceding command produces the following output:

```
[MANIFEST commons-logging-1.1.1.jar]
Archiver-Version                        Plexus Archiver
Build-Jdk                               1.4.2_16
Built-By                                dlg01
Created-By                              Apache Maven
Extension-Name                          org.apache.commons.logging
Implementation-Title                    Commons Logging
Implementation-Vendor                   Apache Software Foundation
Implementation-Vendor-Id                org.apache
Implementation-Version                  1.1.1
Manifest-Version                        1.0
Specification-Title                     Commons Logging
Specification-Vendor                    Apache Software Foundation
Specification-Version                   1.0
X-Compile-Source-JDK                    1.2
X-Compile-Target-JDK                    1.1

[IMPEXP]
```

```
[USES]
org.apache.commons.logging              org.apache.commons.logging.impl
org.apache.commons.logging.impl         javax.servlet

org.apache.avalon.framework.logger
                                        org.apache.commons.logging
                                        org.apache.log
                                        org.apache.log4j
One error
1 : Unresolved references to [javax.servlet,
org.apache.avalon.framework.logger,
 org.apache.log, org.apache.log4j] by class(es) on the Bundle-
Classpath[Jar:comm
ons-logging-1.1.1.jar]:
[org/apache/commons/logging/impl/AvalonLogger.class, org
/apache/commons/logging/impl/ServletContextCleaner.class,
org/apache/commons/log
ging/impl/LogKitLogger.class,
org/apache/commons/logging/impl/Log4JLogger.class]
```

From this output, you can see that the JAR does not define any bundle manifest headers. The output consists of the following sections:

[MANIFEST *JarFileName*]

Lists all of the header settings from the JAR file's **META-INF/Manifest.mf** file.

[IMPEXP]

Lists any Java packages that are imported or exported through the **Import-Package** or **Export-Package** Manifest headers.

[USES]

Shows the JAR's package dependencies. The left column lists all of the packages defined in the JAR, while the right column lists the dependent packages for each of the packages in the left column.

Errors

Lists any errors—for example, any unresolved dependencies.

Bnd wrap command

To convert the plain commons logging JAR into an OSGi bundle, invoke the Bnd **wrap** command as follows:

```
java -jar bnd.jar wrap commons-logging-1.1.1.jar
```

The result of running this command is a bundle file, **commons-logging-1.1.1.bar**, which consists of the original JAR augmented by the Manifest headers required by a bundle.

Checking the new bundle headers

To display the Manifest headers and package dependencies of the newly created bundle JAR, enter the following Bnd **`print`** command:

```
java -jar bnd.jar print commons-logging-1.1.1.bar
```

The preceding command should produce output like the following:

```
[MANIFEST commons-logging-1.1.1-bnd.jar]
Archiver-Version                        Plexus Archiver
Bnd-LastModified                        1263987809524
Build-Jdk                               1.4.2_16
Built-By                                dlg01
Bundle-ManifestVersion                  2
Bundle-Name                             commons-logging-1.1.1
Bundle-SymbolicName                     commons-logging-1.1.1
Bundle-Version                          0
Created-By                              1.5.0_08 (Sun Microsystems Inc.)
Export-Package
org.apache.commons.logging;uses:="org.ap
ache.commons.logging.impl",org.apache.commons.logging.impl;uses:="org.apac
he.ava
lon.framework.logger,org.apache.commons.logging,org.apache.log4j,org.apach
e.log,
javax.servlet"
Extension-Name                          org.apache.commons.logging
Implementation-Title                    Commons Logging
Implementation-Vendor                   Apache Software Foundation
Implementation-Vendor-Id                org.apache
Implementation-Version                  1.1.1
Import-Package
javax.servlet;resolution:=optional,org.a
pache.avalon.framework.logger;resolution:=optional,org.apache.commons.logg
ing;re
solution:=optional,org.apache.commons.logging.impl;resolution:=optional,or
g.apac
he.log;resolution:=optional,org.apache.log4j;resolution:=optional
Manifest-Version                        1.0
Originally-Created-By                   Apache Maven
Specification-Title                     Commons Logging
Specification-Vendor                    Apache Software Foundation
Specification-Version                   1.0
Tool                                    Bnd-0.0.384
X-Compile-Source-JDK                    1.2
X-Compile-Target-JDK                    1.1


[IMPEXP]
Import-Package
  javax.servlet                         {resolution:=optional}
  org.apache.avalon.framework.logger    {resolution:=optional}
  org.apache.log                        {resolution:=optional}
  org.apache.log4j                      {resolution:=optional}
Export-Package
  org.apache.commons.logging
  org.apache.commons.logging.impl
```

```
[USES]
org.apache.commons.logging                org.apache.commons.logging.impl
org.apache.commons.logging.impl           javax.servlet

org.apache.avalon.framework.logger
                                          org.apache.commons.logging
                                          org.apache.log
                                          org.apache.log4j
```

Default property file

By default, the Bnd **`wrap`** command behaves as if it was configured to use the following Bnd property file:

```
Export-Package: *
Import-Package: AllReferencedPackages
```

The result of this configuration is that the new bundle imports all of the packages referenced by the JAR (which is almost always what you need) and all of the packages defined in the JAR are exported. Sometimes you might want to hide some of the packages in the JAR, however, in which case you would need to define a custom property file.

Defining a custom property file

If you want to have more control over the way the Bnd **`wrap`** command generates a bundle, you can define a Bnd properties file to control the conversion process. For a detailed description of the syntax and capabilities of the Bnd properties file, see the Bnd tool documentation.

For example, in the case of the commons logging JAR, you might decide to hide the **`org.apache.commons.logging.impl`** package, while exporting the **`org.apache.commons.logging`** package. You could do this by creating a Bnd properties file called **`commons-logging-1.1.1.bnd`** and inserting the following lines using a text editor:

```
version=1.1.1
Export-Package: org.apache.commons.logging;version=${version}
Private-Package: org.apache.commons.logging.impl
Bundle-Version: ${version}
```

Notice how a version number is assigned to the exported package by substituting the **`version`** variable (any properties starting with a lowercase letter are interpreted as variables).

Wrapping with the custom property file

To wrap a JAR file using the custom property file, specify the Bnd properties file using the **`-properties`** option of the **`wrap`** command. For example, to wrap the vanilla commons logging JAR using the instructions contained in the **`commons-logging-1.1.1.bnd`** properties file, enter the following command:

```
java -jar bnd.jar wrap -properties commons-logging-1.1.1.bnd commons-
logging-1.1.1.jar
```

14.3. CONVERTING A JAR USING THE WRAP SCHEME

Overview

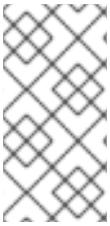
You also have the option of converting a JAR into a bundle using the **wrap** scheme, which can be prefixed to any existing URL format. The **wrap** scheme is also based on the Bnd utility.

Syntax

The **wrap** scheme has the following basic syntax:

```
wrap:LocationURL
```

The **wrap** scheme can prefix any URL that locates a JAR. The locating part of the URL, *LocationURL*, is used to obtain the (non-bundlized) JAR and the URL handler for the **wrap** scheme then converts the JAR automatically into a bundle.



NOTE

The **wrap** scheme also supports a more elaborate syntax, which enables you to customize the conversion by specifying a Bnd properties file or by specifying individual Bnd properties in the URL. Typically, however, the **wrap** scheme is used just with its default settings.

Default properties

Because the **wrap** scheme is based on the Bnd utility, it uses exactly the same default properties to generate the bundle as Bnd does—see the section called “Default property file”.

Wrap and install

The following example shows how you can use a single console command to download the plain **commons-logging** JAR from a remote Maven repository, convert it into an OSGi bundle on the fly, and then install it and start it in the OSGi container:

```
JBossFuse:karaf@root> osgi:install -s wrap:mvn:commons-logging/commons-
logging/1.1.1
```

Feature example

Example 14.1, “The example-jpa-osgi Feature” shows how the **example-jpa-osgi** feature combines the **mvn** scheme and the **wrap** scheme in order to download the plain HyperSQL JAR file and convert it to an OSGi bundle on the fly.

Example 14.1. The example-jpa-osgi Feature

```
<feature name="examples-jpa-osgi" version="6.0.0.redhat-024">
    <feature version="6.0.0.redhat-024">jpa-hibernate</feature>
    <bundle>wrap:mvn:hsqldb/hsqldb/1.8.0.7</bundle>
    <bundle>mvn:org.apache.servicemix.examples.jpa-osgi/wsdl-first-
cxfbc-bundle/6.0.0.redhat-024</bundle>
    <bundle>mvn:org.apache.servicemix.examples.jpa-osgi/wsdl-first-
cxfse-bundle/6.0.0.redhat-024</bundle>
</feature>
```

Reference

The **wrap** scheme is provided by the Pax project, which is the umbrella project for a variety of open source OSGi utilities. For full documentation on the **wrap** scheme, see the Wrap Protocol reference page.

CHAPTER 15. OSGI BUNDLE TUTORIALS

Abstract

This chapter presents tutorials for Apache Camel and Apache CXF applications. Each tutorial describes how to generate, build, run, and deploy an application as an OSGi bundle.

15.1. GENERATING AND RUNNING AN EIP BUNDLE

Overview

This section explains how to generate, build, and run a complete Apache Camel example as an OSGi bundle, where the starting point code is generated with the help of a Maven archetype.

Prerequisites

In order to generate a project using an Red Hat JBoss Fuse Maven archetype, you must have the following prerequisites:

- *Maven installation*—Maven is a free, open source build tool from Apache. You can download the latest version from http://maven.apache.org/download.html (minimum is 2.0.9).
- *Internet connection*—whilst performing a build, Maven dynamically searches external repositories and downloads the required artifacts on the fly. In order for this to work, your build machine *must* be connected to the Internet.
- *fusesource Maven repository is configured*—in order to locate the archetypes, Maven's **`settings.xml`** file must be configured with the location of the **`fusesource`** Maven repository. For details of how to set this up, see the section called “Adding the Red Hat JBoss Fuse repository”.

Generating an EIP bundle

The **`servicemix-camel-osgi-bundle`** archetype creates a router project, which is configured to deploy as a bundle. To generate a Maven project with the coordinates, **`org.fusesource.example:camel-bundle`**, enter the following command:

```
mvn archetype:generate
-DarchetypeGroupId=org.apache.servicemix.tooling
-DarchetypeArtifactId=servicemix-camel-osgi-bundle
-DarchetypeVersion=2012.01.0.redhat-60024
-DgroupId=org.fusesource.example
-DartifactId=camel-bundle
```

The result of this command is a directory, ***`ProjectDir`*`/camel-bundle`**, containing the files for the generated bundle project.

Running the EIP bundle

To install and run the generated **`camel-bundle`** project, perform the following steps:

1. *Build the project*—open a command prompt and change directory to ***ProjectDir*/camel-bundle**. Use Maven to build the demonstration by entering the following command:

   ```
   mvn install
   ```

   If this command runs successfully, the ***ProjectDir*/camel-bundle/target** directory should contain the bundle file, **camel-bundle-1.0-SNAPSHOT.jar** and the bundle will also be installed in the local Maven repository.

2. *Install prerequisite features (optional)*—by default, the **camel-core** feature and some related features are pre-installed in the OSGi container. But many of the Apache Camel components are *not* installed by default. To check which features are available and whether or not they are installed, enter the following console command:

   ```
   JBossFuse:karaf@root> features:list
   ```

   Apache Camel features are identifiable by the **camel-** prefix. For example, if one of your routes requires the HTTP component, you can make sure that it is installed in the OSGi container by issuing the following console command:

   ```
   JBossFuse:karaf@root> features:install camel-http
   ```

3. *Install and start the camel-bundle bundle*—at the Red Hat JBoss Fuse console, enter the following command:

   ```
   JBossFuse:karaf@root> osgi:install -s file:ProjectDir/camel-
   bundle/target/camel-bundle-1.0-SNAPSHOT.jar
   ```

   Where *ProjectDir* is the directory containing your Maven projects and the **-s** flag directs the container to start the bundle right away. For example, if your project directory is **C:\Projects** on a Windows machine, you would enter the following command:

   ```
   JBossFuse:karaf@root> osgi:install -s file:C:/Projects/camel-
   bundle/target/camel-bundle-1.0-SNAPSHOT.jar
   ```

   Alternatively, you could install the bundle from the local Maven repository using an Mvn URL (see Section A.3, “Mvn URL Handler”) as follows:

   ```
   JBossFuse:karaf@root> osgi:install -s
   mvn:org.fusesource.example/camel-bundle
   ```

   After entering this command, you should soon see output like the following being logged to the console screen:

   ```
   >>>> MyTransform set body:  Mon Sep 22 11:43:42 BST 2008
   >>>> MyTransform set body:  Mon Sep 22 11:43:44 BST 2008
   >>>> MyTransform set body:  Mon Sep 22 11:43:46 BST 2008
   ```

NOTE

On Windows machines, be careful how you format the **`file`** URL—for details of the syntax understood by the **`file`** URL handler, see Section A.1, “File URL Handler”.

4. *Stop the camel-bundle bundle*—to stop the **`camel-bundle`** bundle, you first need to discover the relevant bundle number. To find the bundle number, enter the following console command (this might look a bit confusing, because the text you are typing will intermingle with the output that is being logged to the screen):

```
JBossFuse:karaf@root> osgi:list
```

At the end of the listing, you should see an entry like the following:

```
[ 189] [Active     ] [            ] [       ] [   60] A Camel OSGi
Service Unit (1.0.0.SNAPSHOT)
```

Where, in this example, the bundle number is 189. To stop this bundle, enter the following console command:

```
JBossFuse:karaf@root> osgi:stop 189
```

15.2. GENERATING AND RUNNING A WEB SERVICES BUNDLE

Overview

This section explains how to generate, build, and run a complete Apache CXF example as a *bundle* in the OSGi container, where the starting point code is generated with the help of a Maven archetype.

Prerequisites

In order to generate a project using a Red Hat JBoss Fuse Maven archetype, you must have the following prerequisites:

- *Maven installation*—Maven is an open source build tool from Apache. You can download the latest version from http://maven.apache.org/download.html (minimum is 2.0.9).
- *Internet connection*—whilst performing a build, Maven dynamically searches external repositories and downloads the required artifacts on the fly. In order for this to work, your build machine *must* be connected to the Internet.
- *fusesource Maven repository is configured*—in order to locate the archetypes, Maven's **`settings.xml`** file must be configured with the location of the **`fusesource`** Maven repository. For details of how to set this up, see the section called “Adding the Red Hat JBoss Fuse repository”.

Generating a Web services bundle

The **`servicemix-cxf-code-first-osgi-bundle`** archetype creates a project for building a Java-first JAX-WS application that can be deployed into the OSGi container. To generate a Maven project with the coordinates, **`org.fusesource.example:cxf-code-first-bundle`**, enter the following

command:

```
mvn archetype:generate
-DarchetypeGroupId=org.apache.servicemix.tooling
-DarchetypeArtifactId=servicemix-cxf-code-first-osgi-bundle
-DarchetypeVersion=2012.01.0.redhat-60024
-DgroupId=org.fusesource.example
-DartifactId=cxf-code-first-bundle
```

The result of this command is a directory, ***ProjectDir*/cxf-code-first-bundle**, containing the files for the generated bundle project.

Modifying the bundle instructions

Typically, you will need to modify the instructions for the Maven bundle plug-in in the POM file. In particular, the default **Import-Package** element generated by the **servicemix-cxf-code-first-osgi-bundle** archetype is not configured to scan the project's Java source files. In most cases, however, you would want the Maven bundle plug-in to perform this automatic scanning in order to ensure that the bundle imports all of the packages needed by your code.

To enable the **Import-Package** scanning feature, simply add the wildcard, *****, as the last item in the comma-separated list inside the **Import-Package** element, as shown in the following example:

```
<project ... >
    ...
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.felix</groupId>
                <artifactId>maven-bundle-plugin</artifactId>
                <extensions>true</extensions>
                <configuration>
                    <instructions>
                        ...
                        <Import-Package>
                            javax.jws,
                            javax.wsdl,
                            javax.xml.bind,
                            javax.xml.bind.annotation,
                               javax.xml.namespace,
                               javax.xml.ws,
                               META-INF.cxf,
                            META-INF.cxf.osgi,
                            org.apache.cxf.bus,
                            org.apache.cxf.bus.spring,
                            org.apache.cxf.bus.resource,
                            org.apache.cxf.configuration.spring,
                            org.apache.cxf.resource,
                            org.apache.cxf.jaxws,
                            org.apache.cxf.transport.http_osgi,
                            org.springframework.beans.factory.config,
                            *
                        </Import-Package>
                        ...
                    </instructions>
```

```
                </configuration>
            </plugin>
        </plugins>
    </build>
    ...
</project>
```

Running the Web services bundle

To install and run the generated **cxf-code-first-bundle** project, perform the following steps:

1. *Build the project*—open a command prompt and change directory to ***ProjectDir*/cxf-code-first-bundle**. Use Maven to build the demonstration by entering the following command:

   ```
   mvn install
   ```

   If this command runs successfully, the ***ProjectDir*/cxf-code-first-bundle/target** directory should contain the bundle file, **cxf-code-first-bundle-1.0-SNAPSHOT.jar**.

2. *Install and start the cxf-code-first-bundle bundle*—at the Red Hat JBoss Fuse console, enter the following command:

   ```
   JBossFuse:karaf@root> osgi:install -s file:ProjectDir/cxf-code-
   first-bundle/target/cxf-code-first-bundle-1.0-SNAPSHOT.jar
   ```

   Where *ProjectDir* is the directory containing your Maven projects and the **-s** flag directs the container to start the bundle right away. For example, if your project directory is **C:\Projects** on a Windows machine, you would enter the following command:

   ```
   JBossFuse:karaf@root> osgi:install -s file:C:/Projects/cxf-code-
   first-bundle/target/cxf-code-first-bundle-1.0-SNAPSHOT.jar
   ```

   NOTE

   On Windows machines, be careful how you format the **file** URL—for details of the syntax understood by the **file** URL handler, see Section A.1, “File URL Handler”.

   Alternatively, you could install the bundle from your local Maven repository, using the following PAX **mvn** URL:

   ```
   JBossFuse:karaf@root> osgi:install -s
   mvn:org.fusesource.example/cxf-code-first-bundle/1.0-SNAPSHOT
   ```

3. *Test the Web serivce*—to test the Web service deployed in the previous step, you can use a web browser to query the service's WSDL. Open your favorite web browser and navigate to the following URL:

   ```
   http://localhost:8181/cxf/PersonServiceCF?wsdl
   ```

   When the web service receives the query, **?wsdl**, it returns a WSDL description of the running service.

4. *Stop the cxf-code-first-bundle bundle*—to stop the **`cxf-code-first-bundle`** bundle, you first need to discover the relevant bundle number. To find the bundle number, enter the following console command:

```
JBossFuse:karaf@root> osgi:list
```

At the end of the listing, you should see an entry like the following:

```
[ 191] [Active     ] [            ] [       ] [   60] A CXF Code
First OSGi Project (1.0.0.SNAPSHOT)
```

Where, in this example, the bundle number is 191. To stop this bundle, enter the following console command:

```
JBossFuse:karaf@root> osgi:stop 191
```

PART V. OSGI SERVICE LAYER

Abstract

In Red Hat JBoss Fuse, the natural way to communicate between deployed bundles is to use *OSGi services*. An OSGi service exposes Java methods that can be invoked by other bundles in the container.

CHAPTER 16. OSGI SERVICES

Abstract

The OSGi core framework defines the *OSGi Service Layer*, which provides a simple mechanism for bundles to interact by registering Java objects as services in the *OSGi service registry*. One of the strengths of the OSGi service model is that *any* Java object can be offered as a service: there are no particular constraints, inheritance rules, or annotations that must be applied to the service class. This chapter describes how to deploy an OSGi service using the OSGi *blueprint container*.

16.1. THE BLUEPRINT CONTAINER

Abstract

The *blueprint container* is a dependency injection framework that simplifies interaction with the OSGi container. In particular, the blueprint container supports a configuration-based approach to using the OSGi service registry—for example, providing standard XML elements to import and export OSGi services.

16.1.1. Blueprint Configuration

Location of blueprint files in a JAR file

Relative to the root of the bundle JAR file, the standard location for blueprint configuration files is the following directory:

```
OSGI-INF/blueprint
```

Any files with the suffix, **`.xml`**, under this directory are interpreted as blueprint configuration files; in other words, any files that match the pattern, **`OSGI-INF/blueprint/*.xml`**.

Location of blueprint files in a Maven project

In the context of a Maven project, *ProjectDir*, the standard location for blueprint configuration files is the following directory:

```
ProjectDir/src/main/resources/OSGI-INF/blueprint
```

Blueprint namespace and root element

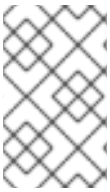
Blueprint configuration elements are associated with the following XML namespace:

```
http://www.osgi.org/xmlns/blueprint/v1.0.0
```

The root element for blueprint configuration is **`blueprint`**, so a blueprint XML configuration file normally has the following outline form:

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">
```

```
  ...
</blueprint>
```



NOTE

In the **blueprint** root element, there is no need to specify the location of the blueprint schema using an **xsi:schemaLocation** attribute, because the schema location is already known to the blueprint framework.

Blueprint Manifest configuration

There are a few aspects of blueprint configuration that are controlled by headers in the JAR's manifest file, **META-INF/MANIFEST.MF**, as follows:

- the section called “Custom Blueprint file locations”.
- the section called “Mandatory dependencies”.

Custom Blueprint file locations

If you need to place your blueprint configuration files in a non-standard location (that is, somewhere other than **OSGI-INF/blueprint/*.xml**), you can specify a comma-separated list of alternative locations in the **Bundle-Blueprint** header in the manifest file—for example:

```
Bundle-Blueprint: lib/account.xml, security.bp, cnf/*.xml
```

Mandatory dependencies

Dependencies on an OSGi service are mandatory by default (although this can be changed by setting the **availability** attribute to **optional** on a **reference** element or a **reference-list** element). Declaring a dependency to be mandatory means that the bundle cannot function properly without that dependency and the dependency must be available at all times.

Normally, while a blueprint container is initializing, it passes through a *grace period*, during which time it attempts to resolve all mandatory dependencies. If the mandatory dependencies cannot be resolved in this time (the default timeout is 5 minutes), container initialization is aborted and the bundle is not started. The following settings can be appended to the **Bundle-SymbolicName** manifest header to configure the grace period:

blueprint.graceperiod

If **true** (the default), the grace period is enabled and the blueprint container waits for mandatory dependencies to be resolved during initialization; if **false**, the grace period is skipped and the container does not check whether the mandatory dependencies are resolved.

blueprint.timeout

Specifies the grace period timeout in milliseconds. The default is 300000 (5 minutes).

For example, to enable a grace period of 10 seconds, you could define the following **Bundle-SymbolicName** header in the manifest file:

```
Bundle-SymbolicName: org.fusesource.example.osgi-client;
 blueprint.graceperiod:=true;
 blueprint.timeout:= 10000
```

The value of the **Bundle-SymbolicName** header is a semi-colon separated list, where the first item is the actual bundle symbolic name, the second item, **blueprint.graceperiod:=true**, enables the grace period and the third item, **blueprint.timeout:= 10000**, specifies a 10 second timeout.

16.1.2. Defining a Service Bean

Overview

Similarly to the Spring container, the blueprint container enables you to instantiate Java classes using a **bean** element. You can create all of your main application objects this way. In particular, you can use the **bean** element to create a Java object that represents an OSGi service instance.

Blueprint bean element

The blueprint **bean** element is defined in the blueprint schema namespace, **http://www.osgi.org/xmlns/blueprint/v1.0.0**. The blueprint **{http://www.osgi.org/xmlns/blueprint/v1.0.0}bean** element should not be confused with the Spring **{http://www.springframework.org/schema/beans}bean** selement, which has a similar syntax but is defined in a different namespace.

NOTE

The Spring DM specification version 2.0 or later, allows you to mix both kinds of **bean** element under the **beans** root element, (as long as you define each **bean** elements using the appropriate namespace prefix).

Sample beans

The blueprint **bean** element enables you to create objects using a similar syntax to the conventional Spring **bean** element. One significant difference, however, is that blueprint constructor arguments are specified using the **argument** child element, in contrast to Spring's **constructor-arg** child element. The following example shows how to create a few different types of bean using blueprint's **bean** element:

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">

  <bean id="label" class="java.lang.String">
    <argument value="LABEL_VALUE"/>
  </bean>

  <bean id="myList" class="java.util.ArrayList">
    <argument type="int" value="10"/>
  </bean>

  <bean id="account" class="org.fusesource.example.Account">
    <property name="accountName" value="john.doe"/>
    <property name="balance" value="10000"/>
  </bean>

</blueprint>
```

Where the **Account** class referenced by the last bean example could be defined as follows:

```
// Java
package org.fusesource.example;

public class Account
{
    private String accountName;
    private int balance;

    public Account () { }

    public void setAccountName(String name) {
        this.accountName = name;
    }

    public void setBalance(int bal) {
        this.balance = bal;
    }
    ...
}
```

Differences between Blueprint and Spring

Althought the syntax of the blueprint **bean** element and the Spring **bean** element are similar, there are a few differences, as you can see from Table 16.1, “Comparison of Spring bean with Blueprint bean”. In this table, the XML tags (identifiers enclosed in angle brackets) refer to child elements of **bean** and the plain identifiers refer to attributes.

Table 16.1. Comparison of Spring bean with Blueprint bean

Spring DM Attributes/Tags	Blueprint Attributes/Tags
id	**id**
name/**<alias>**	*N/A*
class	**class**
scope	**scope=("singleton"|"prototype")**
lazy-init=("true"|"false")	**activation=("eager"|"lazy")**
depends-on	**depends-on**
init-method	**init-method**
destroy-method	**destroy-method**
factory-method	**factory-bean**

Spring DM Attributes/Tags	Blueprint Attributes/Tags
`factory-bean`	`factory-ref`
`<constructor-arg>`	`<argument>`
`<property>`	`<property>`

Where the default value of the blueprint **`scope`** attribute is **`singleton`** and the default value of the blueprint **`activation`** attribute is **`eager`**.

References

For more details on defining blueprint beans, consult the following references:

- Spring Dynamic Modules Reference Guide v2.0 (see the blueprint chapters).
- Section 121 *Blueprint Container Specification*, from the OSGi Compendium Services R4.2 specification.

16.1.3. Exporting a Service

Overview

This section describes how to export a Java object to the OSGi service registry, thus making it accessible as a service to other bundles in the OSGi container.

Exporting with a single interface

To export a service to the OSGi service registry under a single interface name, define a **`service`** element that references the relevant service bean, using the **`ref`** attribute, and specifies the published interface, using the **`interface`** attribute.

For example, you could export an instance of the **`SavingsAccountImpl`** class under the **`org.fusesource.example.Account`** interface name using the blueprint configuration code shown in Example 16.1, “Sample Service Export with a Single Interface”.

Example 16.1. Sample Service Export with a Single Interface

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">

  <bean id="savings" class="org.fusesource.example.SavingsAccountImpl"/>

 <service ref="savings" interface="org.fusesource.example.Account"/>

</blueprint>
```

Where the **`ref`** attribute specifies the ID of the corresponding bean instance and the **`interface`** attribute specifies the name of the public Java interface under which the service is registered in the OSGi service registry. The classes and interfaces used in this example are shown in Example 16.2, “Sample

Account Classes and Interfaces”

Example 16.2. Sample Account Classes and Interfaces

```
// Java
package org.fusesource.example

public interface Account { ... }

public interface SavingsAccount { ... }

public interface CheckingAccount { ... }

public class SavingsAccountImpl implements SavingsAccount
{
    ...
}

public class CheckingAccountImpl implements CheckingAccount
{
    ...
}
```

Exporting with multiple interfaces

To export a service to the OSGi service registry under multiple interface names, define a **`service`** element that references the relevant service bean, using the **`ref`** attribute, and specifies the published interfaces, using the **`interfaces`** child element.

For example, you could export an instance of the **`SavingsAccountImpl`** class under the list of public Java interfaces, **`org.fusesource.example.Account`** and **`org.fusesource.example.SavingsAccount`**, using the following blueprint configuration code:

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">
  <bean id="savings" class="org.fusesource.example.SavingsAccountImpl"/>
  <service ref="savings">
    <interfaces>
      <value>org.fusesource.example.Account</value>
      <value>org.fusesource.example.SavingsAccount</value>
    </interfaces>
  </service>
  ...
</blueprint>
```

NOTE

The **`interface`** attribute and the **`interfaces`** element cannot be used simultaneously in the same **`service`** element. You must use either one or the other.

Exporting with auto-export

If you want to export a service to the OSGi service registry under *all* of its implemented public Java interfaces, there is an easy way of accomplishing this using the **auto-export** attribute.

For example, to export an instance of the **SavingsAccountImpl** class under all of its implemented public interfaces, use the following blueprint configuration code:

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">
  <bean id="savings" class="org.fusesource.example.SavingsAccountImpl"/>
  <service ref="savings" auto-export="interfaces"/>
  ...
</blueprint>
```

Where the **interfaces** value of the **auto-export** attribute indicates that blueprint should register all of the public interfaces implemented by **SavingsAccountImpl**. The **auto-export** attribute can have the following valid values:

disabled

Disables auto-export. This is the default.

interfaces

Registers the service under all of its implemented public Java interfaces.

class-hierarchy

Registers the service under its own type (class) and under all super-types (super-classes), except for the **Object** class.

all-classes

Like the **class-hierarchy** option, but including all of the implemented public Java interfaces as well.

Setting service properties

The OSGi service registry also allows you to associate *service properties* with a registered service. Clients of the service can then use the service properties to search for or filter services. To associate service properties with an exported service, add a **service-properties** child element that contains one or more **beans:entry** elements (one **beans:entry** element for each service property).

For example, to associate the **bank.name** string property with a savings account service, you could use the following blueprint configuration:

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
           xmlns:beans="http://www.springframework.org/schema/beans"
           ...>
  ...
  <service ref="savings" auto-export="interfaces">
    <service-properties>
      <beans:entry key="bank.name" value="HighStreetBank"/>
    </service-properties>
  </service>
  ...
</blueprint>
```

Where the **bank.name** string property has the value, **HighStreetBank**. It is possible to define service properties of type other than string: that is, primitive types, arrays, and collections are also supported. For details of how to define these types, see Controlling the Set of Advertised Properties. in the *Spring Reference Guide*.

NOTE

Strictly speaking, the **entry** element ought to belong to the blueprint namespace. The use of the **beans:entry** element in Spring's implementation of blueprint is non-standard.

Default service properties

There are two service properties that might be set automatically when you export a service using the **service** element, as follows:

- **osgi.service.blueprint.compname**—is always set to the **id** of the service's **bean** element, unless the bean is inlined (that is, the bean is defined as a child element of the **service** element). Inlined beans are always anonymous.
- **service.ranking**—is automatically set, if the ranking attribute is non-zero.

Specifying a ranking attribute

If a bundle looks up a service in the service registry and finds more than one matching service, you can use ranking to determine which of the services is returned. The rule is that, whenever a lookup matches multiple services, the service with the highest rank is returned. The service rank can be any non-negative integer, with **0** being the default. You can specify the service ranking by setting the **ranking** attribute on the **service** element—for example:

```
<service ref="savings" interface="org.fusesource.example.Account"
ranking="10"/>
```

Specifying a registration listener

If you want to keep track of service registration and unregistration events, you can define a *registration listener* callback bean that receives registration and unregistration event notifications. To define a registration listener, add a **registration-listener** child element to a **service** element.

For example, the following blueprint configuration defines a listener bean, **listenerBean**, which is referenced by a **registration-listener** element, so that the listener bean receives callbacks whenever an **Account** service is registered or unregistered:

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0" ...>
  ...
  <bean id="listenerBean" class="org.fusesource.example.Listener"/>

  <service ref="savings" auto-export="interfaces">
    <registration-listener
        ref="listenerBean"
        registration-method="register"
        unregistration-method="unregister"/>
```

```
  </service>
  ...
</blueprint>
```

Where the **registration-listener** element's **ref** attribute references the **id** of the listener bean, the **registration-method** attribute specifies the name of the listener method that receives the registration callback, and **unregistration-method** attribute specifies the name of the listener method that receives the unregistration callback.

The following Java code shows a sample definition of the **Listener** class that receives notifications of registration and unregistration events:

```
// Java
package org.fusesource.example;

public class Listener
{
    public void register(Account service, java.util.Map serviceProperties)
{
        ...
    }

    public void unregister(Account service, java.util.Map
serviceProperties) {
        ...
    }
}
```

The method names, **register** and **unregister**, are specified by the **registration-method** and **unregistration-method** attributes respectively. The signatures of these methods must conform to the following syntax:

- *First method argument*—any type T that is assignable from the service object's type. In other words, any supertype class of the service class or any interface implemented by the service class. This argument contains the service instance, unless the service bean declares the **scope** to be **prototype**, in which case this argument is **null** (when the scope is **prototype**, no service instance is available at registration time).
- *Second method argument*—must be of either **java.util.Map** type or **java.util.Dictionary** type. This map contains the service properties associated with this service registration.

16.1.4. Importing a Service

Overview

This section describes how to obtain and use references to OSGi services that have been exported to the OSGi service registry. Essentially, you can use either the **reference** element or the **reference-list** element to import an OSGi service. The key difference between these elements is *not* (as you might at first be tempted to think) that **reference** returns a single service reference, while **reference-list** returns a list of service references. Rather, the real difference is that the**reference** element is suitable for accessing *stateless* services, while the **reference-list** element is suitable for accessing *stateful* services.

Managing service references

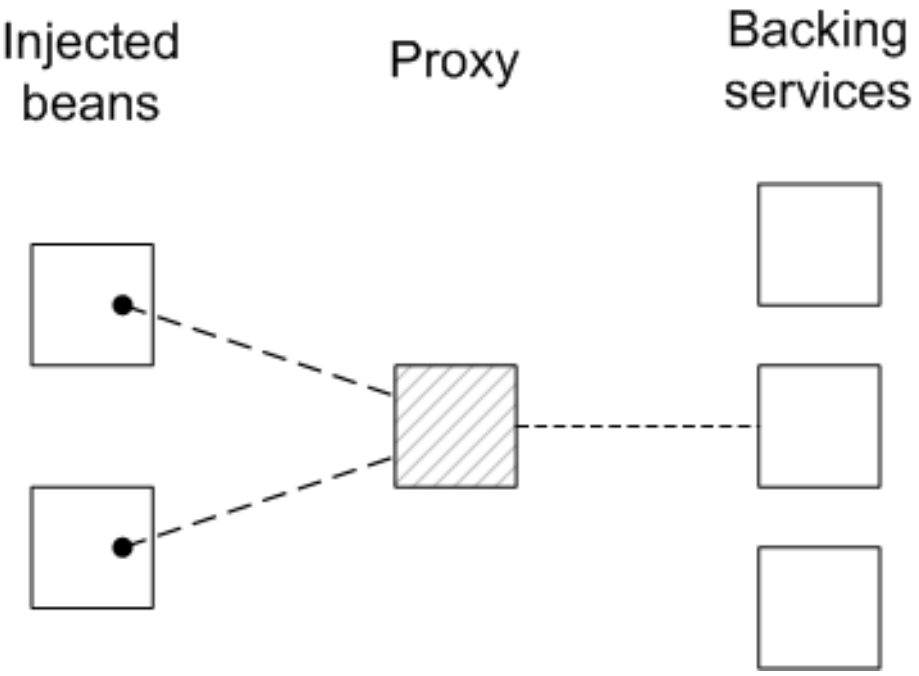
The following models for obtaining OSGi services references are supported:

- the section called “Reference manager”.
- the section called “Reference list manager”.

Reference manager

A *reference manager* instance is created by the blueprint **`reference`** element. This element returns a single service reference and is the preferred approach for accessing *stateless* services. Figure 16.1, “Reference to Stateless Service” shows an overview of the model for accessing a stateless service using the reference manager.

Figure 16.1. Reference to Stateless Service



Beans in the client blueprint container get injected with a proxy object (the *provided object*), which is backed by a service object (the *backing service*) from the OSGi service registry. This model explicitly takes advantage of the fact that stateless services are interchangeable, in the following ways:

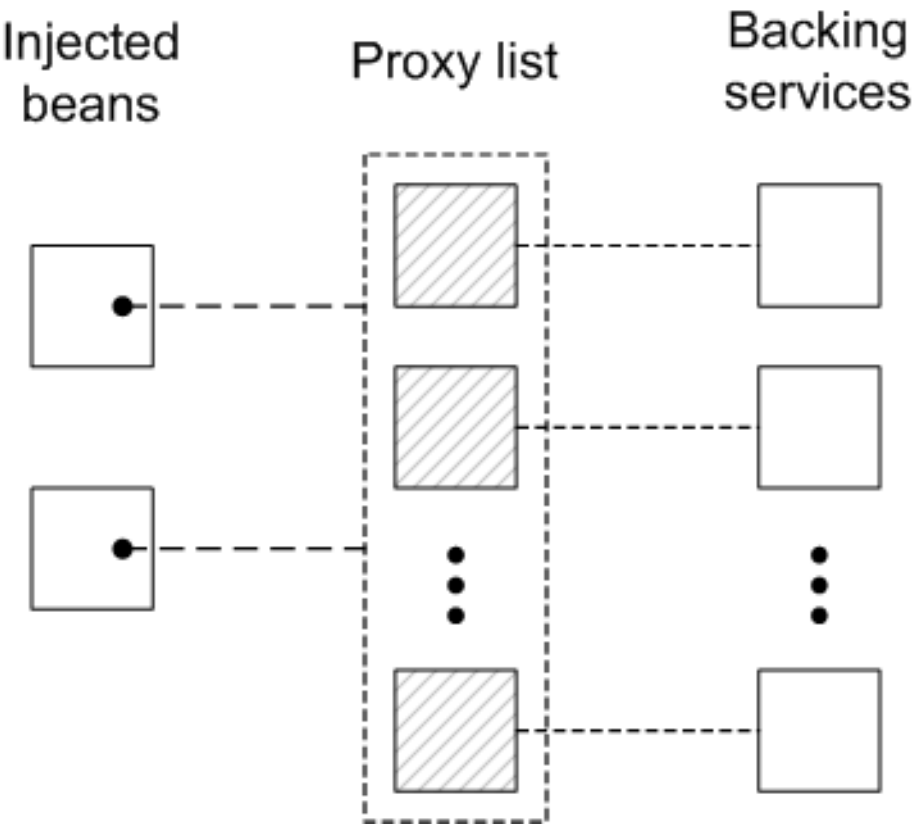
- If multiple services instances are found that match the criteria in the **`reference`** element, the reference manager can arbitrarily choose one of them as the backing instance (because they are interchangeable).
- If the backing service disappears, the reference manager can immediately switch to using one of the other available services of the same type. Hence, there is no guarantee, from one method invocation to the next, that the proxy remains connected to the same backing service.

The contract between the client and the backing service is thus *stateless*, and the client must *not* assume that it is always talking to the same service instance. If no matching service instances are available, the proxy will wait for a certain length of time before throwing the **`ServiceUnavailable`** exception. The length of the timeout is configurable by setting the **`timeout`** attribute on the **`reference`** element.

Reference list manager

A *reference list manager* instance is created by the blueprint **`reference-list`** element. This element returns a list of service references and is the preferred approach for accessing *stateful* services. Figure 16.2, “List of References to Stateful Services” shows an overview of the model for accessing a stateful service using the reference list manager.

Figure 16.2. List of References to Stateful Services



Beans in the client blueprint container get injected with a **`java.util.List`** object (the *provided object*), which contains a list of proxy objects. Each proxy is backed by a unique service instance in the OSGi service registry. Unlike the stateless model, backing services are *not* considered to be interchangeable here. In fact, the lifecycle of each proxy in the list is tightly linked to the lifecycle of the corresponding backing service: when a service gets registered in the OSGi registry, a corresponding proxy is synchronously created and added to the proxy list; and when a service gets unregistered from the OSGi registry, the corresponding proxy is synchronously removed from the proxy list.

The contract between a proxy and its backing service is thus *stateful*, and the client may assume when it invokes methods on a particular proxy, that it is always communicating with the *same* backing service. It could happen, however, that the backing service becomes unavailable, in which case the proxy becomes stale. Any attempt to invoke a method on a stale proxy will generate the **`ServiceUnavailable`** exception.

Matching by interface (stateless)

The simplest way to obtain a *stateles* service reference is by specifying the interface to match, using the **`interface`** attribute on the **`reference`** element. The service is deemed to match, if the **`interface`** attribute value is a super-type of the service or if the attribute value is a Java interface implemented by the service (the **`interface`** attribute can specify either a Java class or a Java interface).

For example, to reference a stateless **`SavingsAccount`** service (see Example 16.1, “Sample Service Export with a Single Interface”), define a **`reference`** element as follows:

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">
```

```
  <reference id="savingsRef"
             interface="org.fusesource.example.SavingsAccount"/>

  <bean id="client" class="org.fusesource.example.client.Client">
    <property name="savingsAccount" ref="savingsRef"/>
  </bean>

</blueprint>
```

Where the **reference** element creates a reference manager bean with the ID, **savingsRef**. To use the referenced service, inject the **savingsRef** bean into one of your client classes, as shown.

The bean property injected into the client class can be any type that is assignable from **SavingsAccount**. For example, you could define the **Client** class as follows:

```
package org.fusesource.example.client;

import org.fusesource.example.SavingsAccount;

public class Client {
    SavingsAccount savingsAccount;

    // Bean properties
    public SavingsAccount getSavingsAccount() {
        return savingsAccount;
    }

    public void setSavingsAccount(SavingsAccount savingsAccount) {
        this.savingsAccount = savingsAccount;
    }
    ...
}
```

Matching by interface (stateful)

The simplest way to obtain a *stateful* service reference is by specifying the interface to match, using the **interface** attribute on the **reference-list** element. The reference list manager then obtains a list of all the services, whose **interface** attribute value is either a super-type of the service or a Java interface implemented by the service (the **interface** attribute can specify either a Java class or a Java interface).

For example, to reference a stateful **SavingsAccount** service (see Example 16.1, “Sample Service Export with a Single Interface”), define a **reference-list** element as follows:

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">

  <reference-list id="savingsListRef"
                  interface="org.fusesource.example.SavingsAccount"/>

  <bean id="client" class="org.fusesource.example.client.Client">
    <property name="savingsAccountList" ref="savingsListRef"/>
  </bean>

</blueprint>
```

Where the **reference-list** element creates a reference list manager bean with the ID, **savingsListRef**. To use the referenced service list, inject the **savingsListRef** bean reference into one of your client classes, as shown.

By default, the **savingsAccountList** bean property is a list of service objects (for example, **java.util.List<SavingsAccount>**). You could define the client class as follows:

```
package org.fusesource.example.client;

import org.fusesource.example.SavingsAccount;

public class Client {
    java.util.List<SavingsAccount> accountList;

    // Bean properties
    public java.util.List<SavingsAccount> getSavingsAccountList() {
        return accountList;
    }

    public void setSavingsAccountList(
                    java.util.List<SavingsAccount> accountList
    ) {
        this.accountList = accountList;
    }
    ...
}
```

Matching by interface and component name

To match both the interface and the component name (bean ID) of a *stateless* service, specify both the **interface** attribute and the **component-name** attribute on the **reference** element, as follows:

```
<reference id="savingsRef"
           interface="org.fusesource.example.SavingsAccount"
           component-name="savings"/>
```

To match both the interface and the component name (bean ID) of a *stateful* service, specify both the **interface** attribute and the **component-name** attribute on the **reference-list** element, as follows:

```
<reference-list id="savingsRef"
           interface="org.fusesource.example.SavingsAccount"
           component-name="savings"/>
```

Matching service properties with a filter

You can select services by matching service properties against a filter. The filter is specified using the **filter** attribute on the **reference** element or on the **reference-list** element. The value of the **filter** attribute must be an *LDAP filter expression*. For example, to define a filter that matches when the **bank.name** service property equals **HighStreetBank**, you could use the following LDAP filter expression:

```
(bank.name=HighStreetBank)
```

To match two service property values, you can use **&** conjunction, which combines expressions with a logical **and**.For example, to require that the **foo** property is equal to **FooValue** and the **bar** property is equal to **BarValue**, you could use the following LDAP filter expression:

```
(&(foo=FooValue)(bar=BarValue))
```

For the complete syntax of LDAP filter expressions, see section 3.2.7 of the *OSGi Core Specification*.

Filters can also be combined with the **interface** and **component-name** settings, in which case all of the specified conditions are required to match.

For example, to match a *stateless* service of **SavingsAccount** type, with a **bank.name** service property equal to **HighStreetBank**, you could define a **reference** element as follows:

```
<reference id="savingsRef"
           interface="org.fusesource.example.SavingsAccount"
           filter="(bank.name=HighStreetBank)"/>
```

To match a *stateful* service of **SavingsAccount** type, with a **bank.name** service property equal to **HighStreetBank**, you could define a **reference-list** element as follows:

```
<reference-list id="savingsRef"
           interface="org.fusesource.example.SavingsAccount"
           filter="(bank.name=HighStreetBank)"/>
```

Specifying whether mandatory or optional

By default, a reference to an OSGi service is assumed to be mandatory (see the section called “Mandatory dependencies”). It is possible, however, to customize the dependency behavior of a **reference** element or a **reference-list** element by setting the **availability** attribute on the element. There are two possible values of the **availability** attribute: **mandatory** (the default), means that the dependency *must* be resolved during a normal blueprint container initialization; and **optional**, means that the dependency need *not* be resolved during initialization.

The following example of a **reference** element shows how to declare explicitly that the reference is a mandatory dependency:

```
<reference id="savingsRef"
           interface="org.fusesource.example.SavingsAccount"
           availability="mandatory"/>
```

Specifying a reference listener

To cope with the dynamic nature of the OSGi environment—for example, if you have declared some of your service references to have **optional** availability—it is often useful to track when a backing service gets bound to the registry and when it gets unbound from the registry. To receive notifications of service binding and unbinding events, you can define a **reference-listener** element as the child of either the **reference** element or the **reference-list** element.

For example, the following blueprint configuration shows how to define a reference listener as a child of the reference manager with the ID, **savingsRef**:

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">

  <reference id="savingsRef"
             interface="org.fusesource.example.SavingsAccount"
             >
    <reference-listener bind-method="onBind" unbind-method="onUnbind">
      <bean class="org.fusesource.example.client.Listener"/>
    </reference-listener>
  </reference>

  <bean id="client" class="org.fusesource.example.client.Client">
    <property name="savingsAcc" ref="savingsRef"/>
  </bean>

</blueprint>
```

The preceding configuration registers an instance of **`org.fusesource.example.client.Listener`** type as a callback that listens for **`bind`** and **`unbind`** events. Events are generated whenever the **`savingsRef`** reference manager's backing service binds or unbinds.

The following example shows a sample implementation of the **`Listener`** class:

```
package org.fusesource.example.client;

import org.osgi.framework.ServiceReference;

public class Listener {

    public void onBind(ServiceReference ref) {
        System.out.println("Bound service: " + ref);
    }

    public void onUnbind(ServiceReference ref) {
        System.out.println("Unbound service: " + ref);
    }

}
```

The method names, **`onBind`** and **`onUnbind`**, are specified by the **`bind-method`** and **`unbind-method`** attributes respectively. Both of these callback methods take an **`org.osgi.framework.ServiceReference`** argument.

16.2. PUBLISHING AN OSGI SERVICE

Overview

This section explains how to generate, build, and deploy a simple OSGi service in the OSGi container. The service is a simple *Hello World* Java class and the OSGi configuration is defined using a blueprint configuration file.

Prerequisites

In order to generate a project using the Maven Quickstart archetype, you must have the following prerequisites:

- *Maven installation*—Maven is a free, open source build tool from Apache. You can download the latest version from http://maven.apache.org/download.html (minimum is 2.0.9).
- *Internet connection*—whilst performing a build, Maven dynamically searches external repositories and downloads the required artifacts on the fly. In order for this to work, your build machine *must* be connected to the Internet.

Generating a Maven project

The **`maven-archetype-quickstart`** archetype creates a generic Maven project, which you can then customize for whatever purpose you like. To generate a Maven project with the coordinates, **`org.fusesource.example:osgi-service`**, enter the following command:

```
mvn archetype:create
-DarchetypeArtifactId=maven-archetype-quickstart
-DgroupId=org.fusesource.example
-DartifactId=osgi-service
```

The result of this command is a directory, ***`ProjectDir`*`/osgi-service`**, containing the files for the generated project.

NOTE

Be careful not *to choose a group ID for your artifact that clashes with the group ID of an existing product!* This could lead to clashes between your project's packages and the packages from the existing product (because the group ID is typically used as the root of a project's Java package names).

Customizing the POM file

You must customize the POM file in order to generate an OSGi bundle, as follows:

1. Follow the POM customization steps described in Section 11.1, “Generating a Bundle Project”.
2. In the configuration of the Maven bundle plug-in, modify the bundle instructions to export the **`org.fusesource.example.service`** package, as follows:

```
<project ... >
  ...
  <build>
    ...
    <plugins>
      ...
      <plugin>
        <groupId>org.apache.felix</groupId>
        <artifactId>maven-bundle-plugin</artifactId>
        <extensions>true</extensions>
        <configuration>
          <instructions>
            <Bundle-SymbolicName>${pom.groupId}.${pom.artifactId}
</Bundle-SymbolicName>
 <Export-Package>org.fusesource.example.service</Export-Package>
```

```
          </instructions>
        </configuration>
      </plugin>
    </plugins>
  </build>
  ...
</project>
```

Writing the service interface

Create the ***ProjectDir*/osgi-service/src/main/java/org/fusesource/example/service** sub-directory. In this directory, use your favorite text editor to create the file, **HelloWorldSvc.java**, and add the code from Example 16.3, "The HelloWorldSvc Interface" to it.

Example 16.3. The HelloWorldSvc Interface

```
// Java
package org.fusesource.example.service;

public interface HelloWorldSvc
{
    public void sayHello();
}
```

Writing the service class

Create the ***ProjectDir*/osgi-service/src/main/java/org/fusesource/example/service/impl** sub-directory. In this directory, use your favorite text editor to create the file, **HelloWorldSvcImpl.java**, and add the code from Example 16.4, "The HelloWorldSvcImpl Class" to it.

Example 16.4. The HelloWorldSvcImpl Class

```
package org.fusesource.example.service.impl;

import org.fusesource.example.service.HelloWorldSvc;


public class HelloWorldSvcImpl implements HelloWorldSvc {

    public void sayHello()
    {
        System.out.println( "Hello World!" );
    }

}
```

Writing the blueprint file

The blueprint configuration file is an XML file stored under the **OSGI-INF/blueprint** directory on the class path. To add a blueprint file to your project, first create the following sub-directories:

```
ProjectDir/osgi-service/src/main/resources
ProjectDir/osgi-service/src/main/resources/OSGI-INF
ProjectDir/osgi-service/src/main/resources/OSGI-INF/blueprint
```

Where the **src/main/resources** is the standard Maven location for all JAR resources. Resource files under this directory will automatically be packaged in the root scope of the generated bundle JAR.

Example 16.5, "Blueprint File for Exporting a Service" shows a sample blueprint file that creates a **HelloWorldSvc** bean, using the **bean** element, and then exports the bean as an OSGi service, using the **service** element.

Under the ***ProjectDir*/osgi-service/src/main/resources/OSGI-INF/blueprint** directory, use your favorite text editor to create the file, **config.xml**, and add the XML code from Example 16.5, "Blueprint File for Exporting a Service".

Example 16.5. Blueprint File for Exporting a Service

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">

  <bean id="hello"
class="org.fusesource.example.service.impl.HelloWorldSvcImpl"/>

  <service ref="hello"
interface="org.fusesource.example.service.HelloWorldSvc"/>

</blueprint>
```

Running the service bundle

To install and run the **osgi-service** project, perform the following steps:

1. *Build the project*—open a command prompt and change directory to ***ProjectDir*/osgi-service**. Use Maven to build the demonstration by entering the following command:

   ```
   mvn install
   ```

   If this command runs successfully, the ***ProjectDir*/osgi-service/target** directory should contain the bundle file, **osgi-service-1.0-SNAPSHOT.jar**.

2. *Install and start the osgi-service bundle*—at the Red Hat JBoss Fuse console, enter the following command:

   ```
   JBossFuse:karaf@root> osgi:install -s file:ProjectDir/osgi-
   service/target/osgi-service-1.0-SNAPSHOT.jar
   ```

   Where *ProjectDir* is the directory containing your Maven projects and the **-s** flag directs the container to start the bundle right away. For example, if your project directory is **C:\Projects** on a Windows machine, you would enter the following command:

```
JBossFuse:karaf@root> osgi:install -s file:C:/Projects/osgi-
service/target/osgi-service-1.0-SNAPSHOT.jar
```

NOTE

On Windows machines, be careful how you format the **`file`** URL—for details of the syntax understood by the **`file`** URL handler, see Section A.1, “File URL Handler”.

3. *Check that the service has been created*—to check that the bundle has started successfully, enter the following Red Hat JBoss Fuse console command:

   ```
   JBossFuse:karaf@root> osgi:list
   ```

   Somewhere in this listing, you should see a line for the **`osgi-service`** bundle, for example:

   ```
   [ 236] [Active     ] [Created     ] [       ] [   60] osgi-service
   (1.0.0.SNAPSHOT)
   ```

   To check that the service is registered in the OSGi service registry, enter a console command like the following:

   ```
   JBossFuse:karaf@root> osgi:ls 236
   ```

   Where the argument to the preceding command is the **`osgi-service`** bundle ID. You should see some output like the following at the console:

   ```
   osgi-service (236) provides:
   ----------------------------
   osgi.service.blueprint.compname = hello
   objectClass = org.fusesource.example.service.HelloWorldSvc
   service.id = 272
   ----
   osgi.blueprint.container.version = 1.0.0.SNAPSHOT
   osgi.blueprint.container.symbolicname = org.fusesource.example.osgi-
   service
   objectClass =
   org.osgi.service.blueprint.container.BlueprintContainer
   service.id = 273
   ```

16.3. ACCESSING AN OSGI SERVICE

Overview

This section explains how to generate, build, and deploy a simple OSGi client in the OSGi container. The client finds the simple Hello World service in the OSGi registry and invokes the **`sayHello()`** method on it.

Prerequisites

In order to generate a project using the Maven Quickstart archetype, you must have the following prerequisites:

- *Maven installation*—Maven is a free, open source build tool from Apache. You can download the latest version from http://maven.apache.org/download.html (minimum is 2.0.9).
- *Internet connection*—whilst performing a build, Maven dynamically searches external repositories and downloads the required artifacts on the fly. In order for this to work, your build machine *must* be connected to the Internet.

Generating a Maven project

The **maven-archetype-quickstart** archetype creates a generic Maven project, which you can then customize for whatever purpose you like. To generate a Maven project with the coordinates, **org.fusesource.example:osgi-client**, enter the following command:

```
mvn archetype:create
-DarchetypeArtifactId=maven-archetype-quickstart
-DgroupId=org.fusesource.example
-DartifactId=osgi-client
```

The result of this command is a directory, ***ProjectDir*/osgi-client**, containing the files for the generated project.

NOTE

Be careful not *to choose a group ID for your artifact that clashes with the group ID of an existing product!* This could lead to clashes between your project's packages and the packages from the existing product (because the group ID is typically used as the root of a project's Java package names).

Customizing the POM file

You must customize the POM file in order to generate an OSGi bundle, as follows:

1. Follow the POM customization steps described in Section 11.1, “Generating a Bundle Project”.
2. Because the client uses the **HelloWorldSvc** Java interface, which is defined in the **osgi-service** bundle, it is necessary to add a Maven dependency on the **osgi-service** bundle. Assuming that the Maven coordinates of the **osgi-service** bundle are **org.fusesource.example:osgi-service:1.0-SNAPSHOT**, you should add the following dependency to the client's POM file:

```
<project ... >
  ...
  <dependencies>
    ...
    <dependency>
        <groupId>org.fusesource.example</groupId>
        <artifactId>osgi-service</artifactId>
        <version>1.0-SNAPSHOT</version>
    </dependency>
```

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

Writing the Blueprint file

To add a blueprint file to your client project, first create the following sub-directories:

```
ProjectDir/osgi-client/src/main/resources
ProjectDir/osgi-client/src/main/resources/OSGI-INF
ProjectDir/osgi-client/src/main/resources/OSGI-INF/blueprint
```

Under the ***ProjectDir*/osgi-client/src/main/resources/OSGI-INF/blueprint** directory, use your favorite text editor to create the file, **config.xml**, and add the XML code from Example 16.6, "Blueprint File for Importing a Service".

Example 16.6. Blueprint File for Importing a Service

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">

  <reference id="helloWorld"
        interface="org.fusesource.example.service.HelloWorldSvc"/>

  <bean id="client"
        class="org.fusesource.example.client.Client"
        init-method="init">
    <property name="helloWorldSvc" ref="helloWorld"/>
  </bean>

</blueprint>
```

Where the **reference** element creates a reference manager that finds a service of **HelloWorldSvc** type in the OSGi registry. The **bean** element creates an instance of the **Client** class and injects the service reference as the bean property, **helloWorldSvc**. In addition, the **init-method** attribute specifies that the **Client.init()** method is called during the bean initialization phase (that is, *after* the service reference has been injected into the client bean).

Writing the client class

Under the ***ProjectDir*/osgi-client/src/main/java/org/fusesource/example/client** directory, use your favorite text editor to create the file, **Client.java**, and add the Java code from Example 16.7, "The Client Class".

Example 16.7. The Client Class

```
// Java
package org.fusesource.example.client;

import org.fusesource.example.service.HelloWorldSvc;

public class Client {
```

```
    HelloWorldSvc helloWorldSvc;

    // Bean properties
    public HelloWorldSvc getHelloWorldSvc() {
        return helloWorldSvc;
    }

    public void setHelloWorldSvc(HelloWorldSvc helloWorldSvc) {
        this.helloWorldSvc = helloWorldSvc;
    }

    public void init() {
        System.out.println("OSGi client started.");
        if (helloWorldSvc != null) {
            System.out.println("Calling sayHello()");
            helloWorldSvc.sayHello();  // Invoke the OSGi service!
        }
    }

}
```

The **Client** class defines a getter and a setter method for the **helloWorldSvc** bean property, which enables it to receive the reference to the Hello World service by injection. The **init()** method is called during the bean initialization phase, after property injection, which means that it is normally possible to invoke the Hello World service within the scope of this method.

Running the client bundle

To install and run the **osgi-client** project, perform the following steps:

1. *Build the project*—open a command prompt and change directory to ***ProjectDir*/osgi-client**. Use Maven to build the demonstration by entering the following command:

   ```
   mvn install
   ```

   If this command runs successfully, the ***ProjectDir*/osgi-client/target** directory should contain the bundle file, **osgi-client-1.0-SNAPSHOT.jar**.

2. *Install and start the osgi-service bundle*—at the Red Hat JBoss Fuse console, enter the following command:

   ```
   JBossFuse:karaf@root> osgi:install -s file:ProjectDir/osgi-
   client/target/osgi-client-1.0-SNAPSHOT.jar
   ```

   Where *ProjectDir* is the directory containing your Maven projects and the **-s** flag directs the container to start the bundle right away. For example, if your project directory is **C:\Projects** on a Windows machine, you would enter the following command:

   ```
   JBossFuse:karaf@root> osgi:install -s file:C:/Projects/osgi-
   client/target/osgi-client-1.0-SNAPSHOT.jar
   ```

NOTE

On Windows machines, be careful how you format the **`file`** URL—for details of the syntax understood by the **`file`** URL handler, see Section A.1, “File URL Handler”.

3. *Client output*—f the client bundle is started successfully, you should immediately see output like the following in the console:

```
Bundle ID: 239
OSGi client started.
Calling sayHello()
Hello World!
```

16.4. INTEGRATION WITH APACHE CAMEL

Overview

Apache Camel provides a simple way to invoke OSGi services using the Bean language. This feature is automatically available whenever a Apache Camel application is deployed into an OSGi container and requires no special configuration.

Registry chaining

When a Apache Camel route is deployed into the OSGi container, the **`CamelContext`** automatically sets up a registry chain for resolving bean instances: the registry chain consists of the OSGi registry, followed by the blueprint (or Spring) registry. Now, if you try to reference a particular bean class or bean instance, the registry resolves the bean as follows:

1. Look up the bean in the OSGi registry first. If a class name is specified, try to match this with the interface or class of an OSGi service.
2. If no match is found in the OSGi registry, fall back on the blueprint registry (or the Spring registry, if you are using the Spring-DM container).

Sample OSGi service interface

Consider the OSGi service defined by the following Java interface, which defines the single method, **`getGreeting()`**:

```
// Java
package org.fusesource.example.hello.boston;

public interface HelloBoston {
    public String getGreeting();
}
```

Sample service export

When defining the bundle that implements the **`HelloBoston`** OSGi service, you could use the following blueprint configuration to export the service:

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">

  <bean id="hello"
class="org.fusesource.example.hello.boston.HelloBostonImpl"/>

  <service ref="hello"
interface="org.fusesource.example.hello.boston.HelloBoston"/>

</blueprint>
```

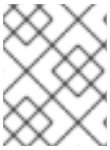
Where it is assumed that the **HelloBoston** interface is implemented by the **HelloBostonImpl** class (not shown).

Invoking the OSGi service from Java DSL

After you have deployed the bundle containing the **HelloBoston** OSGi service, you can invoke the service from a Apache Camel application using the Java DSL. In the Java DSL, you invoke the OSGi service through the Bean language, as follows:

```
from("timer:foo?period=5000")
 .bean(org.fusesource.example.hello.boston.HelloBoston.class,
"getGreeting")
  .log("The message contains: ${body}")
```

In the **bean** command, the first argument is the OSGi interface or class, which must match the interface exported from the OSGi service bundle. The second argument is the name of the bean method you want to invoke. For full details of the **bean** command syntax, see section "Bean Integration" in "Implementing Enterprise Integration Patterns".



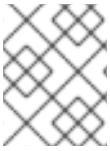
NOTE

When you use this approach, the OSGi service is implicitly imported. It is *not* necessary to import the OSGi service explicitly in this case.

Invoking the OSGi service from XML DSL

In the XML DSL, you can also use the Bean language to invoke the **HelloBoston** OSGi service, but the syntax is slightly different. In the XML DSL, you invoke the OSGi service through the Bean language, using the **method** element, as follows:

```
<beans ...>
  <camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
      <from uri="timer:foo?period=5000"/>
      <setBody>
          <method ref="org.fusesource.example.hello.boston.HelloBoston"
                  method="getGreeting"/>
      </setBody>
      <log message="The message contains: ${body}"/>
    </route>
  </camelContext>
</beans>
```



NOTE

When you use this approach, the OSGi service is implicitly imported. It is *not* necessary to import the OSGi service explicitly in this case.

PART VI. ASYNCHRONOUS COMMUNICATION

Abstract

For asynchronous communication between applications deployed in the Red Hat JBoss Fuse container, there are two alternative mechanisms available: the Apache ActiveMQ and the NMR channel.

CHAPTER 17. JMS BROKER

5/24/12

Updated to include content about how the broker works in enterprise.

5/28/12

Moved information about creating new brokers in enterprise to the broker documentation.

5/28/12

Hid the management section in enterprise because that content is already in the broker documentation.

Abstract

Red Hat JBoss Fuse supports the deployment of JMS brokers. By default, it deploys an Apache ActiveMQ JMS broker. It includes all of the required bundles to deploy additional Apache ActiveMQ instances by deploying a new broker configuration.

17.1. WORKING WITH THE DEFAULT BROKER

5/29/12

Added directions for disabling the default the default broker.

Abstract

Red Hat JBoss Fuse starts up with a message broker by default. You can use this broker as it is for your application, or you can update its configuration to suite the needs of your application.

Overview

When you deploy a Red Hat JBoss Fuse instance, whether as a standalone container or as a part of a fabric, the default behavior is for a Apache ActiveMQ instance to be started in the container. The default broker creates an Openwire port that listens on port **61616**. The broker remains installed in the container and activates whenever you restart the container.

Broker configuration

The default broker's configuration is controlled by two files:

- **etc/activemq.xml**—a standard Apache ActiveMQ configuration file that serves as a template for the default broker's configuration. It contains property place holders, specified using the syntax **${*propName*}**, that allow you to set the values of the actual property using the OSGi Admin service.
- **etc/org.fusesource.mq.fabric.server-default.cfg**—the OSGi configuration file that specifies the values for the properties in the broker's template configuration file.

For details on how to edit the default broker's configuration see the JBoss A-MQ documentation.

Broker data

The default broker's data is stored in **data/activemq**. You can change this location using the **config** command to change the broker's data property as shown in Example 17.1, “Configuring the Default Broker's Data Directory”.

Example 17.1. Configuring the Default Broker's Data Directory

```
JBossFuse:karaf@root> config:edit
org.fusesource.mq.fabric.server.3e3d0055-1c5f-40e3-987e-024c1fac1c3f
JBossFuse:karaf@root> config:propset data dataStore
JBossFuse:karaf@root> config:exit
```

Disabling the default broker

If you decide that you don't want to use the default broker, you can disable it by removing it's OSGi configuration file:

1. From the JBoss Fuse command console, delete the configuration PID using the **`config:delete`** command as shown in Example 17.2, "Deleting the Default Broker Configuration".

   Example 17.2. Deleting the Default Broker Configuration

   ```
   JBossFuse:karaf@root> config:delete
   org.fusesource.mq.fabric.server.xxx
   ```

   xxx is the system generated ID for the broker. You can find this value using the **`config:list`** command.

2. From the system terminal, delete the actual configuration file **`etc/org.fusesource.mq.fabric.server-default.cfg`**.

It is important to do both steps. Step 1 removes the broker from the running container and its cached deployment information. Step 2 removes the broker's configuration from the file system and ensures that it will not be automatically reloaded if the container is restarted.

17.2. JMS ENDPOINTS IN A ROUTER APPLICATION

Overview

The following example shows how you can integrate a JMS broker into a router application. The example generates messages using a timer; sends the messages through the **`camel.timer`** queue in the JMS broker; and then writes the messages to a specific directory in the file system.

Prerequisites

In order to run the sample router application, you need to have the **`camel-activemq`** feature installed in the OSGi container. The **`camel-activemq`** component is needed for defining Apache ActiveMQ-based JMS endpoints in Apache Camel. This feature is *not* installed by default, so you must install it using the following console command:

```
JBossFuse:karaf@root> features:install camel-activemq
```

You also need the **`activemq`** feature, but this feature is normally available, because Red Hat JBoss Fuse installs it by default.

TIP

Most of the Apache Camel components are *not* installed by default. Whenever you are about to define an endpoint in a Apache Camel route, remember to check whether the corresponding component feature is installed. Apache Camel component features generally have the same name as the corresponding Apache Camel component artifact ID, **`camel-ComponentName`**.

Router configuration

Example 17.3, “Sample Route with JMS Endpoints” gives an example of a Apache Camel route defined using the Spring XML DSL. Messages generated by the timer endpoint are propagated through the JMS broker and then written out to the file system.

Example 17.3. Sample Route with JMS Endpoints

```
<?xml version="1.0"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:util="http://www.springframework.org/schema/util"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:camel="http://camel.apache.org/schema/spring"
  xmlns:osgi="http://www.springframework.org/schema/osgi"
  xsi:schemaLocation="
         http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans.xsd
         http://www.springframework.org/schema/util
         http://www.springframework.org/schema/util/spring-util.xsd
         http://www.springframework.org/schema/osgi
         http://www.springframework.org/schema/osgi/spring-osgi.xsd
         http://www.springframework.org/schema/osgi-compendium
         http://www.springframework.org/schema/osgi-compendium/spring-
osgi-compendium.xsd
         http://camel.apache.org/schema/spring
         http://camel.apache.org/schema/spring/camel-spring.xsd
         http://www.springframework.org/schema/osgi
         http://www.springframework.org/schema/osgi/spring-osgi.xsd">

  <bean id="activemq"
class="org.apache.activemq.camel.component.ActiveMQComponent">
    <property name="brokerURL" value="tcp://localhost:61616"/>
  </bean>

  <camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
      <from uri="timer://MyTimer?fixedRate=true&amp;period=4000"/>
      <setBody><constant>Hello World!</constant></setBody>
      <to uri="activemq:camel.timer"/>
    </route>
    <route>
      <from uri="activemq:camel.timer"/>
      <to uri="file:C:/temp/sandpit/timer"/>
    </route>
  </camelContext>

</beans>
```

Camel activemq component

In general, it is necessary to create a custom instance of the Apache Camel **activemq** component, because you need to specify the connection details for connecting to the broker. The preceding example uses Spring syntax to instantiate the **activemq** bean which connects to the broker URL, **tcp://localhost:61616**. The broker URL must correspond to one of the transport connectors defined in the broker configuration file, **deploy/test-broker.xml**.

Sample routes

Example 17.3, "Sample Route with JMS Endpoints" defines two routes, as follows:

1. The first route uses a **timer** endpoint to generate messages at four-second intervals. The **setBody** element places a dummy string in the body of the message (which would otherwise be **null**). The messages are then sent to the **camel.timer** queue on the broker (the **activemq:camel.timer** endpoint).

   NOTE

   The **activemq** scheme in **activemq:camel.timer** is resolved by looking up **activemq** in the bean registry, which resolves to the locally instantiated bean with ID, **activemq**.

2. The second route pulls messages off the **camel.timer** queue and then writes the messages to the specified directory, **C:\temp\sandpit\timer**, in the file system.

Steps to run the example

To run the sample router application, perform the following steps:

1. Using your favorite text editor, copy and paste the router configuration from Example 17.3, "Sample Route with JMS Endpoints" into a file called **camel-timer.xml**.

   Edit the file endpoint in the second route, in order to change the target directory to a suitable location on your file system:

   ```
   <route>
     <from uri="activemq:camel.timer"/>
     <to uri="file:YourDirectoryHere!"/>
   </route>
   ```

2. Start up a local instance of the Red Hat JBoss Fuse runtime by entering the following at a command prompt:

   ```
   servicemix
   ```

3. Make sure the requisite features are installed in the OSGi container. To install the **camel-activemq** feature, enter the following command at the console:

   ```
   JBossFuse:karaf@root> features:install camel-activemq
   ```

To ensure that the **activemq-broker** feature is *not* installed, enter the following command at the console:

```
JBossFuse:karaf@root> features:uninstall activemq-broker
```

4. Use one of the following alternatives to obtain a broker instance for this demonstration:

   - *Use the default broker*—assuming you have not disabled the default broker, you can use it for this demonstration, because it is listening on the correct port, 61616.
   - *Create a new broker instance using the console*—if you prefer not to use the default broker, you can disable it (as described in Section 17.1, "Working with the Default Broker") and then create a new JMS broker instance by entering the following command at the console:

     ```
     JBossFuse:karaf@root> activemq:create-broker --name test
     ```

     After executing this command, you should see the broker configuration file, **test-broker.xml**, in the ***InstallDir*/deploy** directory.

5. Hot deploy the router configuration you created in step 1. Copy the **camel-timer.xml** file into the ***InstallDir*/deploy** directory.

6. Within a few seconds, you should start to see files appearing in the target directory (which is **C:\temp\sandpit\timer**, by default). The file component automatically generates a unique filename for each message that it writes.

   It is also possible to monitor activity in the JMS broker by connecting to the Red Hat JBoss Fuse runtime's JMX port. To monitor the broker using JMX, perform the following steps:

   a. To monitor the JBoss Fuse runtime, start a JConsole instance (a standard Java utility) by entering the following command:

      ```
      jconsole
      ```

   b. Initially, a **JConsole: Connect to Agent** dialog prompts you to connect to a JMX port. From the **Local** tab, select the **org.apache.felix.karaf.main.Bootstrap** entry and click **Connect**.

   c. In the main JConsole window, click on the **MBeans** tab and then drill down to **org.apache.activemq|test|Queue** in the MBean tree (assuming that **test** is the name of your broker).

   d. Under the **Queue** folder, you should see the **camel.timer** queue. Click on the **camel.timer** queue to view statistics on the message throughput of this queue.

7. To shut down the router application, delete the **camel-timer.xml** file from the ***InstallDir*/deploy** directory.

CHAPTER 18. INTER-BUNDLE COMMUNICATION WITH THE NMR

Abstract

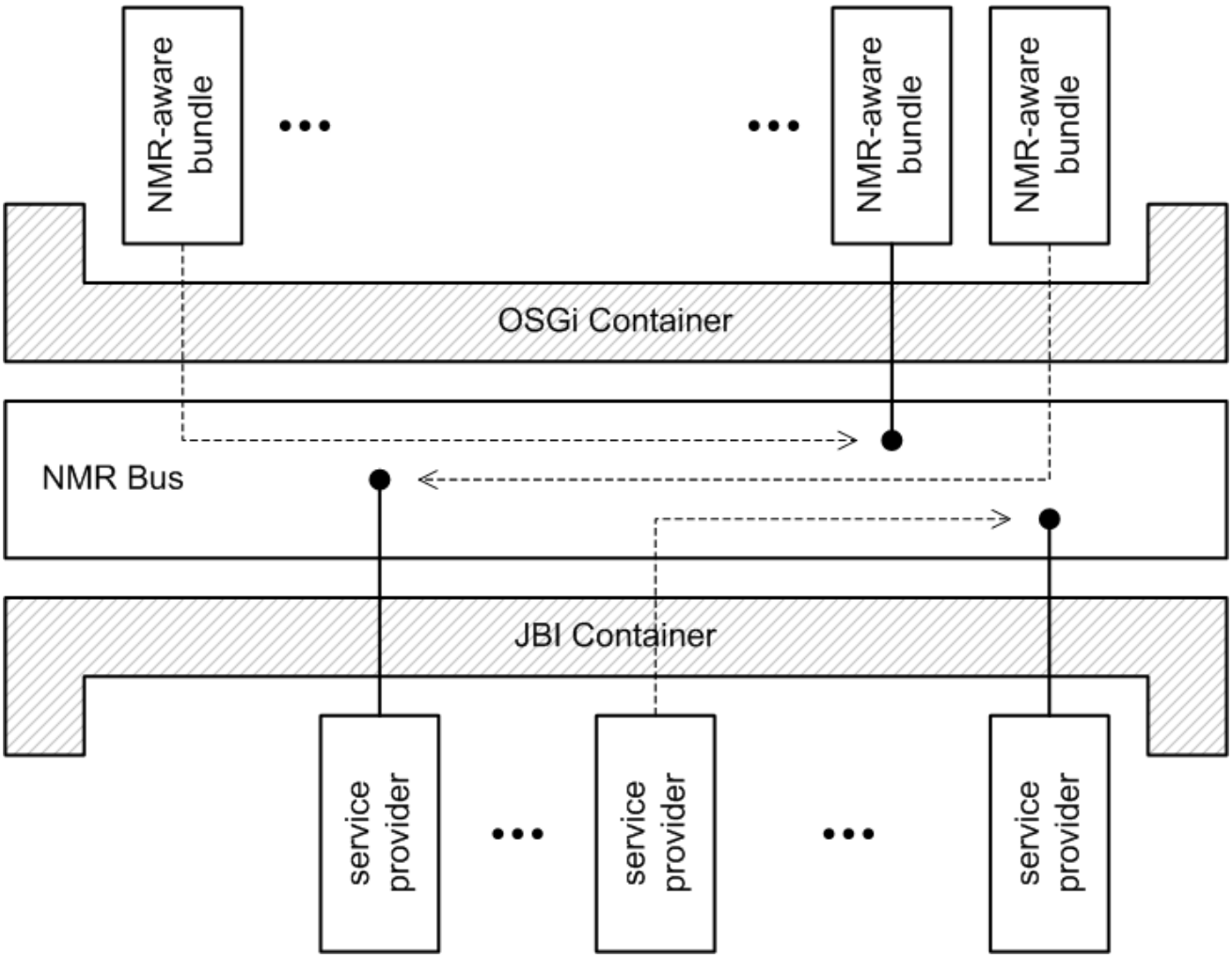
Red Hat JBoss Fuse provides a non-standard mechanism to support asynchronous messaging, known as the *Normalized Message Router* (NMR), which is loosely based on the NMR defined in the JBI standard. The NMR has the advantage that it spans both the OSGi container and the JBI container. By contrast, the *OSGi Event Admin Service*, which also provides asynchronous communication between bundles, can only be used within the OSGi container.

18.1. ARCHITECTURE OF THE NMR

Overview

Figure 18.1, “NMR Architecture” shows a general overview of the NMR architecture, which spans both the OSGi container and the JBI container.

Figure 18.1. NMR Architecture



In Figure 18.1, “NMR Architecture”, the NMR is represented as a horizontal graphical element in order to emphasize its role linking together various application bundles. In practice, however, the NMR is deployed as a collection of bundles, just like any other application in the OSGi container.

NMR

The Red Hat JBoss Fuse NMR is a general-purpose message bus used for transmitting messages between bundles in the OSGi container. It is modelled on the Normalized Message Router (NMR) defined in the Java Business Integration (JBI) specification. Hence, the JBoss Fuse NMR can be used to transmit XML messages, optionally augmented with properties and attachments.

NMR for OSGi

Unlike the standard NMR, however, the JBoss Fuse NMR is *not* restricted to the JBI container. You can use the NMR to transmit messages inside the OSGi container or, if the JBI container is also deployed, to transmit messages between the two containers.

Normalized messages in the JBI container

A key feature of the NMR message bus is that messages are transmitted in a standard, *normalized* form. The JBI standard defines a *normalized message*, which is based on the Web Services Description Language (WSDL) message format (both WSDL 1.1 and WSDL 2.0 formats are supported). A complete normalized message has the following aspects:

- *Content*—the main content of a normalized message must be in XML format, where the layout of a particular message is defined in a WSDL service description.
- *Attachments*—for sending binary content, you can add attachments to the normalized message.
- *Properties*—consist of name/value pairs.
- *Security subject*—identifies the sender, if security features are enabled.

Normalized messages in the OSGi container

In the OSGi container, normalized messages have a standard layout, as follows:

- *Content*—the main content of a normalized message, which can be in any format.
- *Attachments*—for sending binary attachments.
- *Properties*—consist of name/value pairs.
- *Security subject*—identifies the sender, if security features are enabled.

NOTE

When transmitting messages solely *within* the OSGi container, normalization of message content is not enforced. That is, the OSGi container does *not* impose any restrictions on the format of the message content. If messages are transmitted to an endpoint in the JBI container, however, message normalization must be observed.

NMR API

Red Hat JBoss Fuse provides a simple Java API for accessing the NMR. You can use this API to define endpoints that process messages received from the NMR and you can write clients that send messages to NMR endpoints. To see how to use this API in practice, take a look at the **`examples/nmr`** demonstration code.

NMR component for Apache Camel

To enable integration with the NMR, Apache Camel provides an NMR component, which lets you define NMR endpoints either at the beginning (for example, as in **`from("nmr:ExampleEndpoint")`**) or at the end (for example, **`to("nmr:ExampleEndpoint")`**) of a route. For full details of how to use the NMR component, see Section 18.2, “The Apache Camel NMR Component”.

NOTE

The NMR component is designed specifically for integrating Apache Camel with the NMR *within the OSGi container*. If you deploy a Apache Camel application in the JBI container, however, NMR integration is provided by the JBI component. The NMR component (for use in an OSGi context) is conventionally identified by the **`nmr`** URI scheme, whereas the JBI component (for use in a JBI context) is conventionally identified by the **`jbi`** URI scheme.

18.2. THE APACHE CAMEL NMR COMPONENT

Overview

The NMR component is an adapter to the NMR, enabling Apache Camel applications to send messages to other bundles within the OSGi container or to components within the JBI container.

Installing the NMR feature

Normally, the NMR feature is pre-installed in the OSGi container. If you need to install the NMR feature, however, you can do so by entering the following console command:

```
JBossFuse:karaf@root> features:install nmr
```

Instantiating the NMR component

To make NMR endpoints available to your Apache Camel application, you need to create an instance of the NMR component. Add the code shown in Example 18.1, “Creating the NMR Component Bean” to your bundle's Spring configuration file (located in **`META-INF/spring/*.xml`**) in order to instantiate the NMR component.

Example 18.1. Creating the NMR Component Bean

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:osgi="http://www.springframework.org/schema/osgi"
       xmlns:camel-osgi="http://camel.apache.org/schema/osgi"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
       http://www.springframework.org/schema/osgi
http://www.springframework.org/schema/osgi/spring-osgi.xsd
       http://camel.apache.org/schema/spring
http://camel.apache.org/camel/schema/spring/camel-spring.xsd
       http://camel.apache.org/schema/osgi
http://camel.apache.org/schema/osgi/camel-osgi.xsd">
```

```
    <bean id="nmr"
class="org.apache.servicemix.camel.nmr.ServiceMixComponent">
        <property name="nmr">
            <osgi:reference
interface="org.apache.servicemix.nmr.api.NMR" />
        </property>
    </bean>

</beans>
```

The **bean** element creates an instance of the NMR component with the bean ID, **nmr**, where this bean ID can then be used as the scheme prefix to create or reference NMR endpoints in your Apache Camel routes. The bean definition references two external Java packages —**org.apache.servicemix.camel.nmr** and **org.apache.servicemix.nmr.api**—which must therefore be imported by this bundle. Because the packages do not occur in Java source code, you must add them explicitly to the list of imported packages in the bundle instructions in the POM—see the section called "Configuring the bundle instructions" for details.

URI format for the NMR component

The NMR component enables you to create endpoints with the following URI format:

```
nmr:EndpointId
```

Where *EndpointId* is a string that identifies the endpoint uniquely. In particular, when used within the OSGi container, the endpoint ID string is not restricted to have any particular format. The scheme prefix, **nmr**, is actually determined by the ID of the bean that instantiates the NMR component—for example, see Example 18.1, "Creating the NMR Component Bean".

Addressing JBI endpoints

If you want to use the NMR component to send messages between the OSGi container and the JBI container, you need to be aware that NMR endpoints inside the JBI container requires a special syntax for the endpoint IDs. You can address an NMR endpoint inside the JBI container using the following URI format:

```
nmr:JBIAddressingURI
```

Where *JBIAddressingURI* conforms to the URI format described in ServiceMix URIs.

Determining the message exchange pattern

An NMR consumer endpoint automatically determines the message exchange pattern (for example, *In* or *InOut*) from the incoming message and sets the message exchange pattern in the current exchange accordingly.

camel-nmr demonstration

The **camel-nmr** demonstration is located in the following directory:

```
InstallDir/examples/camel-nmr
```

The demonstration defines two routes in XML, where the routes are joined together using an NMR endpoint, as follows:

1. The first route is defined as follows:
   a. At the start of the route is a **timer** endpoint, which generates a heartbeat event every two seconds.
   b. At the end of the route is an NMR endpoint, which transmits the messages to the next route.
2. The second route is defined as follows:
   a. At the start of the second route is an NMR endpoint, which receives the messages sent by the first route.
   b. Next comes a callout to a transformer bean (implemented in Java), which transforms the hearbeat into a message containing the current date and time.
   c. At the end of the route is a **log** endpoint, which sends the transformed message to Jakarta commons logger.

The route is deployed into the Red Hat JBoss Fuse container as an OSGi bundle.

Defining the route in a Spring XML file

Example 18.2, "Spring XML Defining a Route with an NMR Endpoint" shows the routes for the **camel-nmr** demonstration, taken from the Spring XML configuration file, **META-INF/spring/beans.xml**.

Example 18.2. Spring XML Defining a Route with an NMR Endpoint

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:osgi="http://www.springframework.org/schema/osgi"
       xmlns:camel-osgi="http://camel.apache.org/schema/osgi"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
       http://www.springframework.org/schema/osgi
http://www.springframework.org/schema/osgi/spring-osgi.xsd
       http://camel.apache.org/schema/spring
http://camel.apache.org/schema/spring/camel-spring.xsd
       http://camel.apache.org/schema/osgi
http://camel.apache.org/schema/osgi/camel-osgi.xsd">

  <import resource="classpath:org/apache/servicemix/camel/nmr/camel-
(1)nmr.xml" />

  <camel-osgi:camelContext
xmlns="http://camel.apache.org/schema/spring">
    <!-- Route periodically sent events into the NMR -->
    <route>
      <from uri="timer://myTimer?fixedRate=true&amp;period=2000"/>
(2)       <to uri="nmr:ExampleRouter"/>
      </route>
```

```
    <!-- Route exchange from the NMR endpoint to a log endpoint -->
    <route>
3       <from uri="nmr:ExampleRouter"/>
        <bean ref="myTransform" method="transform"/>
      <to uri="log:ExampleRouter"/>
    </route>
  </camel-osgi:camelContext>

  <bean id="myTransform"
class="org.apache.servicemix.examples.camel.MyTransform">
    <property name="prefix" value="MyTransform"/>
  </bean>

</beans>
```

1. This Spring **`import`** element imports a snippet of XML that instantiates and initializes the NMR component. In fact, the content of this snippet is identical to the XML code shown in Example 18.1, “Creating the NMR Component Bean”.
2. At the end of the first route, messages are sent to the NMR endpoint, **`nmr:ExampleRouter`**.
3. When you specify an NMR endpoint in the **`uri`** attribute of the **`<from>`** tag, a new NMR endpoint is created by the NMR component. In this example, the **`<from>`** tag implicitly creates the NMR endpoint, **`nmr:ExampleRouter`**, which is then capable of receiving the messages sent by the first route.

Configuring the bundle instructions

Not all of the packages required by the NMR component can be automatically detected by the Maven bundle plug-in. Some of the package dependencies arise from settings in the Spring configuration file (see Example 18.1, “Creating the NMR Component Bean”), which are not automatically taken into account by the bundle plug-in. In particular, you must ensure that the following additional packages are imported by the bundle:

- **`org.apache.servicemix.camel.nmr`**
- **`org.apache.servicemix.nmr.api`**

For example, the following sample configuration of the Maven bundle plug-in shows how to add an **`Import-Package`** element that contains a list of the packages required for the NMR component:

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

```
Package>org.apache.servicemix.camel.nmr,org.apache.servicemix.nmr.api,*
</Import-Package>
                    </instructions>
                </configuration>
            </plugin>
        </plugins>
    </build>

</project>
```

The **Import-Package** list also includes the wildcard, *****, which instructs the bundle plug-in to scan the Java source code in order to discover further package dependencies.

APPENDIX A. URL HANDLERS

Abstract

There are many contexts in Red Hat JBoss Fuse where you need to provide a URL to specify the location of a resource (for example, as the argument to a console command). In general, when specifying a URL, you can use any of the schemes supported by JBoss Fuse's built-in URL handlers. This appendix describes the syntax for all of the available URL handlers.

A.1. FILE URL HANDLER

Syntax

A file URL has the syntax, **`file:`***PathName*, where *PathName* is the relative or absolute pathname of a file that is available on the Classpath. The provided *PathName* is parsed by Java's built-in file *URL handler*. Hence, the *PathName* syntax is subject to the usual conventions of a Java pathname: in particular, on Windows, each backslash must either be escaped by another backslash or replaced by a forward slash.

Examples

For example, consider the pathname, **`C:\Projects\camel-bundle\target\foo-1.0-SNAPSHOT.jar`**, on Windows. The following example shows the *correct* alternatives for the file URL on Windows:

```
file:C:/Projects/camel-bundle/target/foo-1.0-SNAPSHOT.jar
file:C:\\Projects\\camel-bundle\\target\\foo-1.0-SNAPSHOT.jar
```

The following example shows some *incorrect* alternatives for the file URL on Windows:

```
file:C:\Projects\camel-bundle\target\foo-1.0-SNAPSHOT.jar        // WRONG!
file://C:/Projects/camel-bundle/target/foo-1.0-SNAPSHOT.jar      // WRONG!
file://C:\\Projects\\camel-bundle\\target\\foo-1.0-SNAPSHOT.jar  // WRONG!
```

A.2. HTTP URL HANDLER

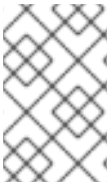
Syntax

A HTTP URL has the standard syntax, **`http:`*`Host`*`[:`*`Port`*`]/[`*`Path`*`][#`*`AnchorName`*`][?`*`Query`*`]`**. You can also specify a secure HTTP URL using the **`https`** scheme. The provided HTTP URL is parsed by Java's built-in HTTP URL handler, so the HTTP URL behaves in the normal way for a Java application.

A.3. MVN URL HANDLER

Overview

If you use Maven to build your bundles or if you know that a particular bundle is available from a Maven repository, you can use the Mvn handler scheme to locate the bundle.



NOTE

To ensure that the Mvn URL handler can find local and remote Maven artifacts, you might find it necessary to customize the Mvn URL handler configuration. For details, see the section called “Configuring the Mvn URL handler”.

Syntax

An Mvn URL has the following syntax:

```
mvn:[repositoryUrl!]groupId/artifactId[/[version][/[packaging]
[/[classifier]]]]
```

Where *repositoryUrl* optionally specifies the URL of a Maven repository. The *groupId*, *artifactId*, *version*, *packaging*, and *classifier* are the standard Maven coordinates for locating Maven artifacts (see the section called “Maven coordinates”).

Omitting coordinates

When specifying an Mvn URL, only the *groupId* and the *artifactId* coordinates are required. The following examples reference a Maven bundle with the *groupId*, **`org.fusesource.example`**, and with the *artifactId*, **`bundle-demo`**:

```
mvn:org.fusesource.example/bundle-demo
mvn:org.fusesource.example/bundle-demo/1.1
```

When the *version* is omitted, as in the first example, it defaults to **LATEST**, which resolves to the latest version based on the available Maven metadata.

In order to specify a *classifier* value without specifying a *packaging* or a *version* value, it is permissible to leave gaps in the Mvn URL. Likewise, if you want to specify a *packaging* value without a *version* value. For example:

```
mvn:groupId/artifactId///classifier
mvn:groupId/artifactId/version//classifier
mvn:groupId/artifactId//packaging/classifier
mvn:groupId/artifactId//packaging
```

Specifying a version range

When specifying the *version* value in an Mvn URL, you can specify a version range (using standard Maven version range syntax) in place of a simple version number. You use square brackets—**`[`** and **`]`**—to denote inclusive ranges and parentheses—**`(`** and **`)`**—to denote exclusive ranges. For example, the range, **`[1.0.4,2.0)`**, matches any version, **`v`**, that satisfies **`1.0.4 <= v < 2.0`**. You can use this version range in an Mvn URL as follows:

```
mvn:org.fusesource.example/bundle-demo/[1.0.4,2.0)
```

Configuring the Mvn URL handler

Before using Mvn URLs for the first time, you might need to customize the Mvn URL handler settings, as follows:

1. the section called “Check the Mvn URL settings”.
2. the section called “Edit the configuration file”.
3. the section called “Customize the location of the local repository”.

Check the Mvn URL settings

The Mvn URL handler resolves a reference to a local Maven repository and maintains a list of remote Maven repositories. When resolving an Mvn URL, the handler searches first the local repository and then the remote repositories in order to locate the specified Maven artifiact. If there is a problem with resolving an Mvn URL, the first thing you should do is to check the handler settings to see which local repository and remote repositories it is using to resolve URLs.

To check the Mvn URL settings, enter the following commands at the console:

```
JBossFuse:karaf@root> config:edit org.ops4j.pax.url.mvn
JBossFuse:karaf@root> config:proplist
```

The **config:edit** command switches the focus of the **config** utility to the properties belonging to the **org.ops4j.pax.url.mvn** persistent ID. The **config:proplist** command outputs all of the property settings for the current persistent ID. With the focus on **org.ops4j.pax.url.mvn**, you should see a listing similar to the following:

```
   org.ops4j.pax.url.mvn.localRepository = file:E:/Data/.m2/repository
   service.pid = org.ops4j.pax.url.mvn
   org.ops4j.pax.url.mvn.defaultRepositories =
file:E:/Programs/FUSE/apache-serv
icemix-4.2.0-fuse-SNAPSHOT/system@snapshots
   felix.fileinstall.filename = org.ops4j.pax.url.mvn.cfg
   org.ops4j.pax.url.mvn.repositories = http://repo1.maven.org/maven2,
http://re
po.fusesource.com/maven2, http://repo.fusesource.com/maven2-
snapshot@snapshots@n
oreleases, http://repository.apache.org/content/groups/snapshots-
group@snapshots
@noreleases, http://repository.ops4j.org/maven2,
http://svn.apache.org/repos/asf
/servicemix/m2-repo,
http://repository.springsource.com/maven/bundles/release, h
ttp://repository.springsource.com/maven/bundles/external
```

Where the **localRepository** setting shows the local repository location currently used by the handler and the **repositories** setting shows the remote repository list currently used by the handler.

Edit the configuration file

To customize the property settings for the Mvn URL handler, edit the following configuration file:

```
InstallDir/etc/org.ops4j.pax.url.mvn.cfg
```

The settings in this file enable you to specify explicitly the location of the local Maven repository, remove Maven repositories, Maven proxy server settings, and more. Please see the comments in the configuration file for more details about these settings.

Customize the location of the local repository

In particular, if your local Maven repository is in a non-default location, you might find it necessary to configure it explicitly in order to access Maven artifacts that you build locally. In your **org.ops4j.pax.url.mvn.cfg** configuration file, uncomment the **org.ops4j.pax.url.mvn.localRepository** property and set it to the location of your local Maven repository. For example:

```
# Path to the local maven repository which is used to avoid downloading
# artifacts when they already exist locally.
# The value of this property will be extracted from the settings.xml file
# above, or defaulted to:
#     System.getProperty( "user.home" ) + "/.m2/repository"
#
org.ops4j.pax.url.mvn.localRepository=file:E:/Data/.m2/repository
```

Reference

For more details about the **mvn** URL syntax, see the original Pax URL Mvn Protocol documentation.

A.4. WRAP URL HANDLER

Overview

If you need to reference a JAR file that is not already packaged as a bundle, you can use the Wrap URL handler to convert it dynamically. The implementation of the Wrap URL handler is based on Peter Krien's open source Bnd utility.

Syntax

A Wrap URL has the following syntax:

```
wrap:locationURL[,instructionsURL][$instructions]
```

The *locationURL* can be any URL that locates a JAR (where the referenced JAR is *not* formatted as a bundle). The optional *instructionsURL* references a Bnd properties file that specifies how the bundle conversion is performed. The optional *instructions* is an ampersand, **&**, delimited list of Bnd properties that specify how the bundle conversion is performed.

Default instructions

In most cases, the default Bnd instructions are adequate for wrapping an API JAR file. By default, Wrap adds manifest headers to the JAR's **META-INF/Manifest.mf** file as shown in Table A.1, “Default Instructions for Wrapping a JAR”.

Table A.1. Default Instructions for Wrapping a JAR

Manifest Header	Default Value
Import-Package	***;resolution:=optional**

Manifest Header	Default Value
`Export-Package`	All packages from the wrapped JAR.
`Bundle-SymbolicName`	The name of the JAR file, where any characters not in the set **`[a-zA-Z0-9_-]`** are replaced by underscore, _.

Examples

The following Wrap URL locates version 1.1 of the **`commons-logging`** JAR in a Maven repository and converts it to an OSGi bundle using the default Bnd properties:

```
wrap:mvn:commons-logging/commons-logging/1.1
```

The following Wrap URL uses the Bnd properties from the file, **`E:\Data\Examples\commons-logging-1.1.bnd`**:

```
wrap:mvn:commons-logging/commons-
logging/1.1,file:E:/Data/Examples/commons-logging-1.1.bnd
```

The following Wrap URL specifies the **`Bundle-SymbolicName`** property and the **`Bundle-Version`** property explicitly:

```
wrap:mvn:commons-logging/commons-logging/1.1$Bundle-SymbolicName=apache-
comm-log&Bundle-Version=1.1
```

If the preceding URL is used as a command-line argument, it might be necessary to escape the dollar sign, **`\$`**, to prevent it from being processed by the command line, as follows:

```
wrap:mvn:commons-logging/commons-logging/1.1\$Bundle-SymbolicName=apache-
comm-log&Bundle-Version=1.1
```

Reference

For more details about the **`wrap`** URL handler, see the following references:

- The Bnd tool documentation, for more details about Bnd properties and Bnd instruction files.
- The original Pax URL Wrap Protocol documentation.

A.5. WAR URL HANDLER

Overview

If you need to deploy a WAR file in an OSGi container, you can automatically add the requisite manifest headers to the WAR file by prefixing the WAR URL with **`war:`**, as described here.

Syntax

A War URL is specified using either of the following syntaxes:

```
war:warURL
warref:instructionsURL
```

The first syntax, using the **war** scheme, specifies a WAR file that is converted into a bundle using the default instructions. The *warURL* can be any URL that locates a WAR file.

The second syntax, using the **warref** scheme, specifies a Bnd properties file, *instructionsURL*, that contains the conversion instructions (including some instructions that are specific to this handler). In this syntax, the location of the referenced WAR file does *not* appear explicitly in the URL. The WAR file is specified instead by the (mandatory) **WAR-URL** property in the properties file.

WAR-specific properties/instructions

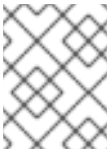
Some of the properties in the **.bnd** instructions file are specific to the War URL handler, as follows:

WAR-URL

(Mandatory) Specifies the location of the War file that is to be converted into a bundle.

Web-ContextPath

Specifies the piece of the URL path that is used to access this Web application, after it has been deployed inside the Web container.



NOTE

Earlier versions of PAX Web used the property, **Webapp-Context**, which is now *deprecated*.

Default instructions

By default, the War URL handler adds manifest headers to the WAR's **META-INF/Manifest.mf** file as shown in Table A.2, “Default Instructions for Wrapping a WAR File”.

Table A.2. Default Instructions for Wrapping a WAR File

Manifest Header	Default Value
Import-Package	**javax.*,org.xml.*,org.w3c.***
Export-Package	No packages are exported.
Bundle-SymbolicName	The name of the WAR file, where any characters not in the set **[a-zA-Z0-9_-\.]** are replaced by period, **.** .
Web-ContextPath	No default value. But the *WAR extender* will use the value of **Bundle-SymbolicName** by default.

Manifest Header	Default Value
`Bundle-ClassPath`	In addition to any class path entries specified explicitly, the following entries are added automatically: • **`.`** • **`WEB-INF/classes`** • All of the JARs from the **`WEB-INF/lib`** directory.

Examples

The following War URL locates version 1.4.7 of the **`wicket-examples`** WAR in a Maven repository and converts it to an OSGi bundle using the default instructions:

```
war:mvn:org.apache.wicket/wicket-examples/1.4.7/war
```

The following Wrap URL specifies the **`Web-ContextPath`** explicitly:

```
war:mvn:org.apache.wicket/wicket-examples/1.4.7/war?Web-ContextPath=wicket
```

The following War URL converts the WAR file referenced by the **`WAR-URL`** property in the **`wicket-examples-1.4.7.bnd`** file and then converts the WAR into an OSGi bundle using the other instructions in the **`.bnd`** file:

```
warref:file:E:/Data/Examples/wicket-examples-1.4.7.bnd
```

Reference

For more details about the **`war`** URL syntax, see the original Pax URL War Protocol documentation.

APPENDIX B. OSGI BEST PRACTICES

Abstract

The combination of Maven and the OSGi framework provides a sophisticated framework for building and deploying enterprise applications. In order to use this framework effectively, however, it is necessary to adopt certain conventions and best practices. The practices described in this appendix are intended to optimize the manageability and scalability of your OSGi applications.

B.1. OSGI TOOLING

Overview

The following best practices are recommended for OSGi related tools and utilities:

- the section called “Use the Maven bundle plug-in to generate the Manifest”.
- the section called “Avoid using the OSGi API directly”.
- the section called “Prefer Blueprint over Spring-DM”.
- the section called “Use Apache Karaf features to group bundles together”.
- the section called “Use the OSGi Configuration Admin service”.
- the section called “Use PAX-Exam for testing”.

Use the Maven bundle plug-in to generate the Manifest

Even for a moderately sized bundle project, it is usually impractical to create and maintain a bundle Manifest by hand. The Maven bundle plug-in is the most effective tool for automating the generation of bundle Manifests in a Maven project. See Section B.2, “Building OSGi Bundles”.

Avoid using the OSGi API directly

Avoid using the OSGi Java API directly. Prefer a higher level technology, for example: Blueprint (from the OSGi Compendium Specification), Spring-DM, Declarative Services (DS), iPojo, and so on.

Prefer Blueprint over Spring-DM

The Blueprint container is now the preferred framework for instantiating, registering, and referencing OSGi services, because this container has now been adopted as an OSGi standard. This ensures greater portability for your OSGi service definitions in the future.

Spring Dynamic Modules (Spring-DM) provided much of the original impetus for the definition of the Blueprint standard, but should now be regarded as obsolescent. Using the Blueprint container does *not* prevent you from using the Spring framework: the latest version of Spring is compatible with Blueprint.

Use Apache Karaf features to group bundles together

When an application is composed of a large number of bundles, it becomes essential to group bundles together in order to deploy them efficiently. Apache Karaf *features* is a mechanism that is designed just

for this purpose. It is easy to use and supported by a variety of different tools. See Chapter 13, *Deploying Features* for details.

Use the OSGi Configuration Admin service

The OSGi Configuration Admin service is the preferred mechanism for providing configuration properties to your application. This configuration mechanism enjoys better tooling support than other approaches. For example, in Red Hat JBoss Fuse the OSGi Configuration Admin service is supported in the following ways:

- Properties integrated with Spring XML files.
- Properties automatically read from configuration files, **`etc/persistendId.cfg`**
- Properties can be set in feature repositories.

Use PAX-Exam for testing

In order for testing to be really effective, you should run at least some of your tests in an OSGi container. This requires you to start an OSGi container, configure its environment, install prerequisite bundles, and install the actual test. Performing these steps manually for every test would make testing prohibitively difficult and time-consuming. Pax-Exam solves this problem by providing a testing framework that is capable of automatically initializing an OSGi container before running tests in the container.

See Appendix C, *Pax-Exam Testing Framework* for more details.

B.2. BUILDING OSGI BUNDLES

Overview

The following best practices are recommended when building OSGi bundles using Maven:

- the section called “Use package prefix as the bundle symbolic name”.
- the section called “Artifact ID should be derived from the bundle symbolic name”.
- the section called “Always export packages with a version”.
- the section called “Use naming convention for private packages”.
- the section called “Import packages with version ranges”.
- the section called “Avoid importing packages that you export”.
- the section called “Use optional imports with caution”.
- the section called “Avoid using the Require-Bundle header”.

Use package prefix as the bundle symbolic name

Use your application's package prefix as the bundle symbolic name. For example, if all of your Java source code is located in sub-packages of **`org.fusesource.fooProject`**, use **`org.fusesource.fooProject`** as the bundle symbolic name.

Artifact ID should be derived from the bundle symbolic name

It makes sense to identify a Maven artifact with an OSGi bundle. To show this relationship as clearly as possible, you should use base the artifact ID on the bundle symbolic name. Two conventions are commonly used:

- *The artifact ID is identical to the bundle symbolic name*—this enables you to define the bundle symbolic name in terms of the artifact ID, using the following Maven bundle instruction:

  ```
  <Bundle-SymbolicName>${project.artifactId}</Bundle-SymbolicName>
  ```

- *The bundle symbolic name is composed of the group ID and the artifact ID, joined by a dot*—this enables you to define the bundle symbolic name in terms of the group ID and the artifact ID, using the following Maven bundle instruction:

  ```
  <Bundle-SymbolicName>${project.groupId}.${project.artifactId}
  </Bundle-SymbolicName>
  ```

NOTE

Properties of the form **`project.*`** can be used to reference the value of *any* element in the current POM. To construct the name of a POM property, take the XPath name of any POM element (for example, **`project/artifactId`**) and replace occurrences of **`/`** with the **`.`** character. Hence, **`${project.artifactId}`** references the **`artifactId`** element from the current POM.

Always export packages with a version

One of the key advantages of the OSGi framework is its ability to manage bundle versions and the possibility of deploying multiple versions of a bundle in the same container. In order to take advantage of this capability, however, it is essential that you associate a version with any packages that you export.

For example, you can configure the **`maven-bundle-plugin`** plug-in to export packages with the current artifact version (given by the **`project.version`** property) as follows:

```
<Export-Package>
  ${project.artifactId}*;version=${project.version}
</Export-Package>
```

Notice how this example exploits the convention that packages use the artifact ID, **`project.artifactId`**, as their package prefix. The combination of package prefix and wildcard, **`${project.artifactId}*`**, enables you to reference all of the source code in your bundle.

Use naming convention for private packages

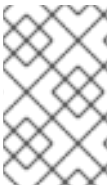
If you define any private packages in your bundle (packages that you do not want to export), it is recommended that you identify these packages using a strict naming convention. For example, if your bundle includes implementation classes that you do not want to export, you should place these classes in packages prefixed by **`${project.artifactId}.impl`** or **`${project.artifactId}.internal`**.

NOTE

If you do not specify any **Export-Package** instruction, the default behavior of the Maven bundle plug-in is to exclude any packages that contain a path segment equal to **impl** or **internal**.

To ensure that the private packages are *not* exported, you can add an entry of the form **!*PackagePattern*** to the Maven bundle plug-in's export instructions. The effect of this entry is to exclude any matching packages. For example, to exclude any packages prefixed by **${project.artifactId}.impl**, you could add the following instruction to the Maven bundle plug-in configuration:

```
<Export-Package>
  !${project.artifactId}.impl.*,
  ${project.artifactId}*;version=${project.version}
</Export-Package>
```



NOTE

The order of entries in the **Export-Package** element is significant. The first match in the list determines whether a package is included or excluded. Hence, in order for exclusions to be effective, they should appear at the *start* of the list.

Import packages with version ranges

In order to benefit from OSGi version management capabilities, it is important to restrict the range of acceptable versions for imported packages. You can use either of the following approaches:

- *Manual version ranges*—you can manually specify the version range for an imported package using the **version** qualifier, as shown in the following example:

  ```
  <Import-Package>
    org.springframework.*;version="[2.5,4)",
    org.apache.commons.logging.*;version="[1.1,2)",
    *
  </Import-Package>
  ```

  Version ranges are specified using the standard OSGi version range syntax, where square brackets—that is, **[** and **]**—denote inclusive ranges and parentheses—that is, **(** and **)**—denote exclusive ranges. Hence the range, **[2.5,4)**, means that the version, **v**, is restricted to the range, **2.5 <= v < 4**. Note the special case of a range written as a simple number—for example, **version="2.5"**, which is equivalent to the range, **[2.5,*infinity*)**.

- *Automatic version ranges*—if packages are imported from a Maven dependency and if the dependency is packaged as an OSGi bundle, the Maven bundle plug-in automatically adds the version range to the import instructions.

  The default behavior is as follows. If your POM depends on a bundle that is identified as version 1.2.4.8, the generated manifest will import version 1.2 of the bundle's exported packages (that is, the imported version number is truncated to the first two parts, major and minor).

  It is also possible to customize how imported version ranges are generated from the bundle dependency. When setting the **version** property, you can use the **${@}** macro (which returns

the original export version) and the **${version}** macro (which modifies a version number) to generate a version range. For example, consider the following **version** settings:

***;version="${@}"**

If a particular package has export version **1.2.4.8**, the generated import version resolves to **1.2.4.8**.

***;version="${version;==;${@}}"**

If a particular package has export version **1.2.4.8**, the generated import version resolves to **1.2**.

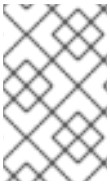
***;version="[${version;==;${@}},${version;=+;${@}})"**

If a particular package has export version **1.2.4.8**, the generated import version range resolves to **[1.2,1.3)**.

***;version="[${version;==;${@}},${version;+;${@}})"**

If a particular package has export version **1.2.4.8**, the generated import version range resolves to **[1.2,2)**.

The middle part of the version macro—for example, **==** or **=+**—formats the returned version number. The equals sign, **=**, returns the corresponding version part unchanged; the plus sign, **+**, returns the corresponding version part plus one; and the minus sign, **-**, returns the corresponding version part minus one. For more details, consult the Bnd documentation for the version macro and the -versionpolicy option.



NOTE

In practice, you are likely to find that the majority of imported packages can be automatically versioned by Maven. It is, typically, only occasionally necessary to specify a version manually.

Avoid importing packages that you export

Normally, it is not good practice to import the packages that you export (though there are exceptions to this rule). Here are some guidelines to follow:

- If the bundle is a pure library (providing interfaces and classes, but *not* instantiating any classes or OSGi services), do not import the packages that you export.
- If the bundle is a pure API (providing interfaces and abstract classes, but no implementation classes), do not import the packages that you export.
- If the bundle is a pure implementation (implementing and registering an OSGi service, but not providing any API), you do not need to export any packages at all.

NOTE

The registered OSGi service must be accessible through an API interface or class, but it is presumed that this API is provided in a *separate* API bundle. The implementation bundle therefore needs to import the corresponding API packages.

- A special case arises, if an implementation and its corresponding API are combined into the same bundle. In this case, the API packages must be listed amongst the export packages *and* amongst the import packages. This configuration is interpreted in a special way by the OSGi framework: it actually means that the API packages will either be exported *or* imported at run time (but not both).

  The reason for this special configuration is that, in a complex OSGi application, it is possible that an API package might be provided by more than one bundle. But you do not want multiple copies of an API to be exported into OSGi, because that can lead to technical problems like class cast exceptions. When a package is listed both in the exports and in the imports, the OSGi resolver proceeds as follows:

  1. First of all, the resolver checks whether the package has already been exported from another bundle. If so, the resolver imports the package, but *does not export it*.

  2. Otherwise, the resolver uses the local API package and exports this package, but it *does not import the package*.

Assuming you want to avoid importing the packages that you export, there are two alternative approaches you can take, as follows:

- *(Recommended)* The most effective way of suppressing the import of exported packages is to append the **`-noimport:=true`** setting to package patterns in the **`Export-Package`** instruction. For example:

  ```
  <Export-Package>
    ${project.artifactId}*;version=${project.version};-noimport:=true
  </Export-Package>
  ```

  The marked packages are now *not* imported, irrespective of what is contained in the **`Import-Package`** instruction.

- An alternative way of avoiding the import is to add one or more package exclusions to the Maven bundle plug-in's **`Import-Package`** element (this was the only possibility in earlier versions of the Maven bundle plug-in). For example, the following **`Import-Package`** element instructs the Maven bundle plug-in to exclude all packages prefixed by the artifact ID, **`${project.artifactId}`**:

  ```
  <Import-Package>
    !${project.artifactId}*,
    org.springframework.*;version="[2.5,4)",
    org.apache.commons.logging.*;version="[1.1,2)",
    *
  </Import-Package>
  ```

Use optional imports with caution

When an imported package is specified with *optional* resolution, this allows the bundle to be resolved *without* resolving the optional package. This affects the resolution order of the bundles which, in turn, can affect the runtime behavior. You should therefore be careful with optional imports, in case they have some unintended side effects.

A package is optional when it appears in the **`Import-Package`** manifest header with the **`resolution:="optional"`** setting appended to it. For example, the following example shows an **`Import-Package`** instruction for the Maven bundle plug-in that specifies an optional import:

```
<Import-Package>
  org.springframework.*;version="[2.5,4)",
  org.apache.commons.logging.*;version="[1.1,2)";resolution:="optional",
  *
</Import-Package>
```

Avoid using the Require-Bundle header

Avoid using the **`Require-Bundle`** header, if possible. The trouble with using the **`Require-Bundle`** header is that it *forces* the OSGi resolver to use packages from the specified bundle. Importing at the granularity of packages, on the other hand, allows the resolver to be more flexible, because there are fewer constraints: if a package is already available and resolved from another bundle, the resolver could use that package instead.

B.3. SAMPLE POM FILE

POM file

Example B.1, “Sample POM File Illustrating Best Practices” shows a sample POM that illustrates the best practices for building an OSGi bundle using Maven.

Example B.1. Sample POM File Illustrating Best Practices

```
<?xml version="1.0" encoding="UTF-8"?>
<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>

  <groupId>org.fusesource</groupId>
  <artifactId>org.fusesource.fooProject</artifactId>
  <packaging>bundle</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>A fooProject OSGi Bundle</name>
  <url>http://www.myorganization.org</url>

  <dependencies>...</dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.felix</groupId>
        <artifactId>maven-bundle-plugin</artifactId>
        <configuration>
          <instructions>
            <Bundle-SymbolicName>${project.artifactId}</Bundle-
SymbolicName>
            <Export-Package>
              !${project.artifactId}.impl.*,
              ${project.artifactId}*;version=${project.version};-
noimport:=true
            </Export-Package>
```

```
            <Import-Package>
              org.springframework.*;version="[2.5,4)",
              org.apache.commons.logging.*;version="[1.1,2)",
              *
            </Import-Package>
          </instructions>
        </configuration>
      </plugin>
    </plugins>
  </build>

</project>
```

APPENDIX C. PAX-EXAM TESTING FRAMEWORK

Abstract

When it is time to start testing your application bundles in the OSGi container, it is recommended that you perform the testing using the Pax-Exam testing framework.

C.1. INTRODUCTION TO PAX-EXAM

Overview

Pax-Exam is an automated testing framework for running tests in an OSGi container. Consider the manual steps that would be needed to run tests in an OSGi container:

1. Set up the environment and initial options for the OSGi container.
2. Start the OSGi container.
3. Install the prerequisite bundles into the OSGi container (provisioning).
4. Install and run the test classes.

Using Pax-Exam you can automate and simplify this testing procedure. Initialization options and provisioning options are performed by a configuration method in the test class. The Pax-Exam framework takes care of starting the OSGi container and before running the test class bundle.

This section gives a brief introduction to the Pax-Exam testing framework and explains how to write a basic example using Apache Karaf.

JUnit 4 framework

JUnit 4 is the latest version of the JUnit Java testing suite. What distinguishes JUnit 4 from earlier versions is that JUnit 4 defines a test by applying *Java annotations* (in contrast to earlier versions of JUnit, which used inherited classes and naming conventions).

The simplest JUnit tests require just two steps:

1. Specify which methods are the test methods by annotating them with the **`@org.junit.Test`** annotation.
2. At any point in the test method, define an assertion by calling **`assertTrue()`** with a boolean argument (you also need to include a static import of **`org.junit.Assert.assertTrue()`** in the file). The test succeeds, if the specified assertions all evaluate to true.

Integration with Pax-Exam

To integrate JUnit 4 with Pax-Exam, perform the following steps:

1. Customize JUnit to run the test in the Pax-Exam test runner class—JUnit allows you to delegate control over a test run to a custom runner class (by defining a runner class that inherits from **`org.junit.runner.Runner`**). In order to integrate JUnit with Pax-Exam, add a **`@RunWith`** annotation to the test class as follows:

```
import org.junit.runner.RunWith;
import org.ops4j.pax.exam.junit.JUnit4TestRunner;
...
@RunWith(JUnit4TestRunner.class)
public class MyTest {
    ...
}
```

2. Add a Pax-Exam configuration method to the test class—in order to run a test in an OSGi framework, you need to initialize the OSGi container properly and install any prerequisite bundles. These essential steps are performed by returning the appropriate options from the Pax-Exam configuration method. This method is identified by the **`@Configuration`** annotation as follows:

```
import org.ops4j.pax.exam.junit.Configuration;
...
@Configuration
public static Option[] configuration() throws Exception {
    ...
}
```

3. Use the Pax-Exam fluent API to configure the OSGi framework—there are a fairly large number of settings and options that you can return from the Pax-Exam configuration method. In order to define these options efficiently, Pax-Exam provides a fluent API, which is defined mainly by the following classes:

   `org.ops4j.pax.exam.CoreOptions`

   Provides basic options for setting up the OSGi container. For example, this class provides options to set Java system properties (**`systemProperty()`**), define URLs (as a string, **`url()`**, or as an Mvn URL, **`maven()`**), and select OSGi frameworks (for example, **`felix()`** or **`equinox()`**).

   `org.ops4j.pax.exam.OptionUtils`

   Provides utilities for manipulating arrays of options and composite options.

   `org.ops4j.pax.exam.container.def.PaxRunnerOptions`

   Provides options for starting the OSGi container and for provisioning features and bundles. For example, the **`scanFeatures()`** and **`scanBundle()`** methods can be used to find and install features and bundles in the OSGi container before running the test.

Integration with Apache Karaf

Theoretically, Pax-Exam provides all of the features that are needed to run an OSGi framework embedded in Apache Karaf. In practice, however, there are a lot of Java system properties and configuration options that need to be set in order to initialize Apache Karaf. It would be a nuisance, if all of these properties and options needed to be specified explicitly in the Pax-Exam configuration method.

In order to simplify running Pax-Exam in Apache Karaf, helper classes are provided, which automatically take care of initializing the OSGi framework for you. The following classes are provided:

`org.apache.karaf.testing.AbstractIntegrationTest`

Provides some helper methods, particularly the **`getOsgiService()`** methods which make it easy to find an OSGi service, by specifying the Java type of the service or by specifying service properties.

`org.apache.karaf.testing.Helper`

Provides the **`Helper.getDefaultOptions()`** method, which configures all of the settings needed to start up Apache Karaf in a default configuration.

Maven dependencies

Example C.1, "Pax-Exam and Related Maven Dependencies" shows the Maven dependencies you need in order to run the Pax-Exam testing framework. You must specify dependencies on JUnit 4, Pax-Exam, and Apache Karaf tooling.

Example C.1. Pax-Exam and Related Maven Dependencies

```
<project ...>
  ...
  <properties>
    <junit-version>4.4</junit-version>
    <pax-exam-version>1.2.0</pax-exam-version>
    <felix.karaf.version>1.4.0-fuse-01-00</felix.karaf.version>
    ...
  </properties>

  <dependencies>
      <!-- Pax-Exam dependencies -->
      <dependency>
        <groupId>org.ops4j.pax.exam</groupId>
        <artifactId>pax-exam</artifactId>
        <version>${pax-exam-version}</version>
      </dependency>
      <dependency>
        <groupId>org.ops4j.pax.exam</groupId>
        <artifactId>pax-exam-junit</artifactId>
        <version>${pax-exam-version}</version>
      </dependency>
      <dependency>
        <groupId>org.ops4j.pax.exam</groupId>
        <artifactId>pax-exam-container-default</artifactId>
        <version>${pax-exam-version}</version>
      </dependency>
      <dependency>
        <groupId>org.ops4j.pax.exam</groupId>
        <artifactId>pax-exam-junit-extender-impl</artifactId>
        <version>${pax-exam-version}</version>
      </dependency>

      <!-- JUnit dependencies -->
      <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>${junit-version}</version>
      </dependency>
```

```
      <!-- Apache Karaf integration -->
      <dependency>
        <groupId>org.apache.karaf.tooling</groupId>
        <artifactId>org.apache.karaf.tooling.testing</artifactId>
        <version>${felix.karaf.version}</version>
        <scope>test</scope>
      </dependency>
  </dependencies>
  ...
</project>
```

This example uses custom properties to specify the versions of the various Maven artifacts.

References

To learn more about using the Pax-Exam testing framework, consult the following references:

- JUnit 4 testing framework on the JUnit Web site.
- The Pax-Exam reference guide on the Pax-Exam Web site.
- Source code for the sample FeaturesText class.

C.2. SAMPLE PAX-EXAM TEST CLASS

Sample test class

Example C.2, “FeaturesText Class” shows an example of how to write a test class for Apache Karaf in the Pax-Exam testing framework. The **`FeaturesText`** class configures the Apache Karaf environment, installs the **`obr`** and **`wrapper`** features, and then runs a test against the two features (where the **`obr`** and **`wrapper`** features implement particular sets of commands in the command console).

Example C.2. FeaturesText Class

```
// Java
/*
 * Licensed to the Apache Software Foundation (ASF)
 * ...
 */
package org.apache.karaf.shell.itests;

import org.apache.karaf.testing.AbstractIntegrationTest;
import org.apache.karaf.testing.Helper;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.ops4j.pax.exam.Option;
import org.ops4j.pax.exam.junit.Configuration;
import org.ops4j.pax.exam.junit.JUnit4TestRunner;
import org.osgi.service.blueprint.container.BlueprintContainer;
import org.osgi.service.command.CommandProcessor;
import org.osgi.service.command.CommandSession;

import static org.junit.Assert.assertNotNull;
```

```
import static org.ops4j.pax.exam.CoreOptions.felix;
import static org.ops4j.pax.exam.CoreOptions.maven;
import static org.ops4j.pax.exam.CoreOptions.systemProperty;
import static org.ops4j.pax.exam.CoreOptions.waitForFrameworkStartup;
import static org.ops4j.pax.exam.OptionUtils.combine;
import static
org.ops4j.pax.exam.container.def.PaxRunnerOptions.scanFeatures;

import static
org.ops4j.pax.exam.container.def.PaxRunnerOptions.workingDirectory;

1 @RunWith(JUnit4TestRunner.class)
2 public class FeaturesTest extends AbstractIntegrationTest {

3     @Test
      public void testFeatures() throws Exception {
        // Make sure the command services are available
        assertNotNull(getOsgiService(BlueprintContainer.class,
"osgi.blueprint.container.symbolicname=org.apache.karaf.shell.obr",
20000));
        assertNotNull(getOsgiService(BlueprintContainer.class,
"osgi.blueprint.container.symbolicname=org.apache.karaf.shell.wrapper",
20000));
        // Run some commands to make sure they are installed properly
        CommandProcessor cp = getOsgiService(CommandProcessor.class);
4         CommandSession cs = cp.createSession(System.in, System.out,
  System.err);
        cs.execute("obr:listUrl");
        cs.execute("wrapper:install --help");
        cs.close();
    }

5     @Configuration
      public static Option[] configuration() throws Exception{
6         return combine(
              // Default karaf environment
7             Helper.getDefaultOptions(
                  // this is how you set the default log level when
using pax logging (logProfile)

systemProperty("org.ops4j.pax.logging.DefaultServiceLog.level").value("D
EBUG")),

            // add two features
8           scanFeatures(

maven().groupId("org.apache.karaf").artifactId("apache-felix-
karaf").type("xml").classifier("features").versionAsInProject(),
                    "obr", "wrapper"
            ),

9           workingDirectory("target/paxrunner/features/"),

10          waitForFrameworkStartup(),

            // Test on the felix OSGi framework
```

```
            felix()
        );
    }

}
```

1. The **@RunWith** annotation instructs JUnit 4 to run the following test with the Pax-Exam test runner class, **JUnit4TestRunner**. This is the key step to integrate JUnit 4 with the Pax-Exam testing framework.

2. In order to integrate this JUnit test properly with Apache Karaf, you are required to derive this test class from **org.apache.karaf.testing.AbstractIntegrationTest**.

   The **AbstractIntegrationTest** base class also provides some helper methods that access the bundle context: **getOsgiService()** methods, for obtaining a reference to an OSGi service, and the **getInstalledBundle()** method, for obtaining a reference to an **org.osgi.framework.bundle** object.

3. The **@Test** annotation is a standard JUnit 4 annotation that identifies the following method as a test method that is to be executed in the testing framework.

4. This line gives an example of how to use the **getOsgiService()** helper method to obtain an OSGi service from the OSGi container. In this example, the service is identified by specifying its Java type, **org.osgi.service.command.CommandProcessor**. The **CommandProcessor** service is the Apache Karaf service that has the capability to process console commands.

5. The **@Configuration** annotation is a Pax-Exam-specific annotation that marks the following the method as the configuration method that sets Pax-Exam testing options. The configuration method must be declared as **public static** and must have a return value of type, **org.ops4j.pax.exam.Option[]**.

6. The **OptionUtils.combine()** method combines a given options array (of **Option[]** type) in the first argument with the options in the remaining arguments, returning an options array that contains all of the options.

7. The **getDefaultOptions()** method from the **org.apache.karaf.testing.Helper** class returns an options array containing all of the system property settings and option settings required to initialize Apache Karaf for the Pax-Exam testing framework.

   If there are any Java system properties in the Apache Karaf environment that you would like to customize, you can pass the properties as optional arguments to the **getDefaultOptions()** method. In the example shown here, the system property for the Pax logging level is set to **DEBUG**.

8. The Pax-Exam framework supports the concept of Apache Karaf features (see Chapter 13, *Deploying Features*). You can use the **PaxRunnerOptions.scanFeatures()** method to install specific features in the OSGi container before the test is run.

   The location of the relevant *features repository* is specified by passing a Pax URL as the first argument to **scanFeatures()**. In this example, the URL is constructed by creating a Pax Mvn URL (see Section A.3, “Mvn URL Handler”) with the fluent API from the **CoreOptions** class. Subsequent arguments specify which features to install—in this example, the **obr** and **wrapper** features.

9.

The **`workingDirectory()`** option specifies the directory where the Pax Runner provisioning module looks for OSGi bundles.

10 The **`waitForFrameworkStartup()`** specifies that the testing framework should wait for a default length of time (five minutes) for the OSGi framework to start up before timing out. To specify the timeout explicitly, you could use the **`waitForFrameworkStartupFor(long millis)`** method instead, where the timeout is specified in milliseconds.

11 The **`felix()`** option is used to specify that the test should be run in the Felix OSGI framework.

INDEX

U