Red Hat Fuse 7.0

Deploying into Apache Karaf

Deploying application packages into the Apache Karaf container

Red Hat Fuse 7.0 Deploying into Apache Karaf

Deploying application packages into the Apache Karaf container

Legal Notice

Abstract

The guide describes the options for deploying applications into an Apache Karaf container.

Table of Contents

PART I. DEVELOPER GUIDE

This part contains information for developers.

CHAPTER 1. DEPLOYING USING AN OSGI BUNDLE

Abstract

The usual and most common method of deploying into Apache Karaf is using an OSGi bundle.

1.1. OSGI OVERVIEW

Apache Karaf is structured to use OSGi functionality. For more information about the structure of Apache Karaf see Chapter 2, *Introduction to OSGi*.

An OSGi bundle is a collection of JAR files with configuration files, bundled up into a JAR. For more information about creating OSGi bundles see Chapter 3, *Building an OSGi Bundle*.

1.2. PREREQUISITES

Before following the instructions make sure that you have completed the following prerequisites:

- Install Apache Karaf, following the instructions in the Red Hat Fuse Installing on Apache Karaf Guide

- Make sure you have installed and configured Maven as shown in Chapter 16, *Building with Maven*

1.3. PREPARING THE OSGI BUNDLE

For this example we will use a *quickstart*, which is a ready-prepared bundle. Quickstarts can be found in **FUSE_HOME/quickstarts**

To find out more about how to build your own OSGi bundle, see Section 3.1, "Generating a Bundle Project" .

1.4. DEPLOYING THE OSGI BUNDLE

The OSGi bundle is deployed into a running Apache Karaf instance.

1. Start Fuse from the bin direction by executing the **./fuse** script in the **FUSE_HOME/bin/** directory.
   You will see the prompt:

   ```
   karaf@root()>
   ```

2. On a separate terminal, navigate to the **FUSE_HOME/quickstarts/beginner/camel-log** directory. **camel-log** is the name of the quickstart we will use to create a bundle.

3. Compile the **camel-log** quickstart using Maven:

   ```
   $ mvn clean install
   ```

4. Return to the Karaf terminal and install the project:

```
karaf@root()> bundle:install -s
mvn:org.jboss.fuse.quickstarts/beginner-camel-log/$CURRENT_VERSION
```

You will see a bundle ID returned:

```
Bundle ID: 228
```

This is a unique identifier for this bundle on this instance of Fuse

5. To see the output of project, look in the log file at **FUSE_HOME/data/log/fuse.log** The output will look like this:

```
12:07:34.542 INFO [Camel (log-example-context) thread #1 - timer://foo]
>>> Hello from Fuse based Camel route! :
12:07:39.530 INFO [Camel (log-example-context) thread #1 - timer://foo]
>>> Hello from Fuse based Camel route! :
12:07:44.530 INFO [Camel (log-example-context) thread #1 - timer://foo]
>>> Hello from Fuse based Camel route! :
```

For more information about deploying OSGi bundles, see Chapter 4, *Hot deployment vs manual deployment*.

CHAPTER 2. INTRODUCTION TO OSGI

Abstract

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

2.1. OVERVIEW

Apache Karaf is an OSGi-based runtime container for deploying and managing bundles. Apache Karaf also provides native operating system integration, and can be integrated into the operating system as a service so that the lifecycle is bound to the operating system.

Apache Karaf has the following structure:

- Apache Karaf - 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 Fuse include hot deployment, management, and administration features.

- OSGi Framework - implements OSGi functionality, including managing dependencies and bundle lifecycles

2.2. ARCHITECTURE OF APACHE KARAF

Apache Karaf extends the OSGi layers with the following functionality:

- **Console** - the console manages services, installs and manages applications and libraries, and interacts with the Fuse runtime. It provides console commands to administer instances of Fuse. See the Apache Karaf Console Reference.

- **Logging** - the logging subsystem provides console commands to display, view and change log levels.

- **Deployment** - supports both manual deployment of OSGi bundles using the `bundle:install` and `bundle:start` commands and hot deployment of applications. See Section 4.1, "Hot Deployment".

- **Provisioning** - provides multiple mechanisms for installing applications and libraries. See Chapter 7, *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.

- **Blueprint** - is a dependency injection framework that simplifies interaction with the OSGi container. For example, providing standard XML elements to import and export OSGi services. When a Blueprint configuration file is copied to the hot deployment folder, Red Hat Fuse generates an OSGi bundle on-the-fly and instantiates the Blueprint context.

2.3. OSGI FRAMEWORK

2.3.1. 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.

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.

2.3.2. OSGi architecture

The OSGi framework contains the following:

- **Bundles** — Logical modules that make up an application. See Section 2.5, "OSGi Bundles".

- **Service layer** — Provides communication among modules and their contained components. This layer is tightly integrated with the lifecycle layer. See Section 2.4, "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.

2.4. OSGI SERVICES

2.4.1. 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.

2.4.2. OSGi service registry

In the OSGi framework, the service layer provides communication between Section 2.5, "OSGi 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 Section 2.2, "Architecture of Apache Karaf".

2.5. 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 3. 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.

3.1. GENERATING A BUNDLE PROJECT

3.1.1. 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 archetype generates a project for building OSGi bundles.

3.1.2. 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.camel.archetypes \
  -DarchetypeArtifactId=camel-archetype-blueprint \
  -DarchetypeVersion=${current-Camel-version} \
  -DgroupId=GroupId \
  -DartifactId=ArtifactId \
  -Dversion=Version
```

3.1.3. 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.

3.2. MODIFYING AN EXISTING MAVEN PROJECT

3.2.1. 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. Section 3.2.2, "Change the package type to bundle".

2. Section 3.2.3, "Add the bundle plug-in to your POM".

3. Section 3.2.4, "Customize the bundle plug-in".

4. Section 3.2.5, "Customize the JDK compiler version".

3.2.2. 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.

3.2.3. 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>
        <version>3.3.0</version>
        <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.

3.2.4. Customize the bundle plug-in

For some specific recommendations on configuring the bundle plug-in for Apache CXF, see Section 3.3, "Packaging a Web Service in a Bundle".

3.2.5. 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 1.8, 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>1.8</source>
          <target>1.8</target>
        </configuration>
      </plugin>
    </plugins>
  </build>
  ...
</project>
```

3.3. PACKAGING A WEB SERVICE IN A BUNDLE

3.3.1. 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 Fuse OSGi container. To convert the Maven project, you need to modify the project's POM file and the project's Blueprint file(s) (located in **META-INF/spring**).

3.3.2. 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 3.1, "Generating a Bundle Project".

3.3.3. 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
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
```

3.3.4. Sample Maven bundle plug-in instructions

Example 3.1, "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 3.1. Configuration of Mandatory Import Packages

```xml
<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,
                            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>
```

3.3.5. 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.

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

For details of how to use OSGi configuration properties, see Section 3.3.7, "Configuring the Bundle Plug-In" and Section 7.6, "Add OSGi configurations to the feature".

3.3.7. 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 3.2, "Setting a bundle's symbolic name".

Example 3.2. 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 3.3, "Setting a bundle's name".

Example 3.3. 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 3.4, "Setting a bundle's version".

Example 3.4. 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 default 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 3.5, "Including a private package in a bundle" shows the configuration for including a private package in a bundle

Example 3.5. 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 3.6, "Specifying the packages imported by a bundle" shows the configuration for specifying the packages imported by a bundle

Example 3.6. 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:

- olink:OsgiDependencies/OsgiDependencies

- Apache Felix documentation

- Peter Kriens' aQute Software Consultancy web site

3.3.8. OSGI configAdmin file naming convention

PID strings (symbolic-name syntax) allow hyphens in the OSGI specification. However, hyphens are interpreted by Apache **Felix.fileinstall** and **config:edit** shell commands to differentiate a "managed service" and "managed service factory". Therefore, it is recommended to not use hyphens elsewhere in a PID string.

NOTE

The Configuration file names are related to the PID and factory PID.

CHAPTER 4. HOT DEPLOYMENT VS MANUAL DEPLOYMENT

Abstract

Fuse provides two different approaches for deploying files: hot deployment or manual deployment. If you need to deploy a collection of related bundles it is recommended that you deploy them together as a *feature*, rather than singly (see Chapter 7, *Deploying Features*).

4.1. HOT DEPLOYMENT

4.1.1. Hot deploy directory

Fuse monitors files in the **FUSE_HOME/deploy** directory and hot deploys everything in this directory. Each time a file is copied to this directory, it is installed in the runtime and started. You can subsequently update or delete the files in the **FUSE_HOME/deploy** directory, 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 FUSE_HOME**/deploy** directory as follows (assuming you are working on a UNIX platform):

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

4.2. HOT UNDEPLOYING A BUNDLE

To undeploy a bundle from the hot deploy directory, simply delete the bundle file from the **FUSE_HOME/deploy** directory **while the Apache Karaf container is running**.
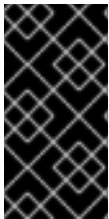
> **IMPORTANT**
>
> The hot undeploy mechanism does **not** work while the container is shut down. If you shut down the Karaf container, delete the bundle file from **FUSE_HOME/deploy** directory, and then restart the Karaf container, the bundle will **not** be undeployed after you restart the container.

You can also undeploy a bundle by using the **bundle:uninstall** console command.

4.3. MANUAL DEPLOYMENT

4.3.1. Overview

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

4.3.2. Installing a bundle

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

```
bundle: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:

```
bundle: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 12.1, "File URL Handler" for details.

4.3.3. Uninstalling a bundle

To uninstall a bundle, you must first obtain its bundle ID using the **bundle:list** command. You can then uninstall the bundle using the **bundle: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 **bundle:list** at the console prompt might produce output like the following:

```
...
[ 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:

```
bundle:uninstall 181
```

4.3.4. URL schemes for locating bundles

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

- Section 12.1, "File URL Handler".

- Section 12.2, "HTTP URL Handler".

- Section 12.3, "Mvn URL Handler".

4.4. REDEPLOYING BUNDLES AUTOMATICALLY USING BUNDLE:WATCH

In a development environment—where a developer is constantly changing and rebuilding a bundle—it is typically necessary to re-install the bundle multiple times. Using the **bundle:watch** command, you can instruct Karaf to monitor your local Maven repository and re-install a particular bundle automatically, as soon as it changes in your local Maven repository.

For example, given a particular bundle—with bundle ID, **751**—you can enable automatic redeployment by entering the command:

```
bundle:watch 751
```

Now, whenever you rebuild and install the Maven artifact into your local Maven repository (for example, by executing **mvn install** in your Maven project), the Karaf container automatically re-installs the changed Maven artifact. For more details, see Apache Karaf Console Reference.

IMPORTANT

Using the **bundle:watch** command is intended for a development environment only. It is **not** recommended for use in a production environment.

CHAPTER 5. LIFECYCLE MANAGEMENT

5.1. 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.

5.2. INSTALLING AND RESOLVING BUNDLES

When you install a bundle using the `bundle: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 **bundle:resolve** command, as follows:

```
bundle:resolve 181
```

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

5.3. STARTING AND STOPPING BUNDLES

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

```
bundle:start 181 185 186
```

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

```
bundle: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 **bundle:restart** command. For example, to restart the bundles with IDs, 181, 185, and 186, enter the following console command:

```
bundle:restart 181 185 186
```

5.4. 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.

5.5. SPECIFYING A BUNDLE'S START LEVEL

Use the **bundle:start-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:

```
bundle:start-level 181 70
```

5.6. 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 **system:start-level** in the console, as follows:

```
karaf@root()> system:start-level
Level 100
```

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

```
system:start-level 200
```

CHAPTER 6. TROUBLESHOOTING DEPENDENCIES

6.1. MISSING DEPENDENCIES

The most common issue that can arise when you deploy an OSGi bundle into the Red Hat 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 **FUSE_HOME/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.

6.2. 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.
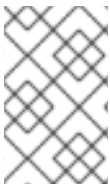
> **NOTE**
>
> Consider packaging your bundle as a feature. Using a feature, you can package your bundle together with all of its dependencies and thus ensure that they are all installed simultaneously. For details, see Chapter 7, *Deploying Features*.

6.3. 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 3.2, "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 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**.

6.4. HOW TO TRACK DOWN MISSING DEPENDENCIES

To track down missing dependencies, perform the following steps:

1. Use the **bundle:diag** console command. This will provide information about why your bundle is inactive. See Apache Karaf Console Reference for usage information.

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

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

```
karaf@root()> bundle:install MyBundleURL
```

4. Use the **bundle: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:

```
karaf@root()> bundle: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.

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

```
karaf@root()> bundle:resolve 218
```

6. Assuming your bundle is now resolved (check the bundle status using **bundle: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:

```
karaf@root()> package:imports -b 218
```

You should see a list of dependent packages in the console window:

```
Package                          | Version       | Optional
| ID  | Bundle Name
—————————————————————————————————+———————————————+———————————+—
————+————————————————————————————
org.apache.jasper.servlet        | [2.2.0,3.0.0) | resolved   |
217 | org.ops4j.pax.web.pax-web-runtime
org.jasypt.encryption.pbe        |               | resolved   |
217 | org.ops4j.pax.web.pax-web-runtime
org.ops4j.pax.web.jsp            | [7.0.0,)      | resolved   |
217 | org.ops4j.pax.web.pax-web-runtime
org.ops4j.pax.web.service.spi.model | [7.0.0,)   |            |
217 | org.ops4j.pax.web.pax-web-runtime
org.ops4j.pax.web.service.spi.util   | [7.0.0,)   |            |
217 | org.ops4j.pax.web.pax-web-runtime
...
```

7. 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.

8. 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:

```
karaf@root()> bundle:uninstall 218
```

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

addurl :experimental: :toc: :toclevels: 4 :numbered:

CHAPTER 7. 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 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.

7.1. CREATING A FEATURE

7.1.1. 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. Section 7.2, "Create a custom feature repository".

2. Section 7.3, "Add a feature to the custom feature repository".

3. Section 7.4, "Add the local repository URL to the features service".

4. Section 7.5, "Add dependent features to the feature".

5. Section 7.6, "Add OSGi configurations to the feature".

7.2. 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).

7.3. 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 Chapter 12, *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.

7.4. 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 Chapter 12, *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
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
```

7.5. 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="7.0.0.fuse-000191-redhat-1">camel-core</feature>
    <feature version="7.0.0.fuse-000191-redhat-1">camel-spring-
osgi</feature>
  </feature>
</features>
```

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

7.6. 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 overridden 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 Blueprint XML file that accesses the OSGi configuration properties:

```
<?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"
           xmlns:cm="http://aries.apache.org/blueprint/xmlns/blueprint-
cm/v1.1.0">

    <!-- osgi blueprint property placeholder -->
    <cm:property-placeholder id="placeholder"
                             persistent-
id="org.fusesource.fuseesb.example">
        <cm:default-properties>
            <cm:property name="prefix" value="DefaultValue"/>
        </cm:default-properties>
    </cm:property-placeholder>

    <bean id="myTransform"
class="org.fusesource.fuseesb.example.MyTransform">
      <property name="prefix" value="${prefix}"/>
    </bean>

</blueprint>
```

When this Blueprint 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.

7.7. AUTOMATICALLY DEPLOY AN OSGI CONFIGURATION

By adding a **configfile** element to a feature, you can ensure that an OSGi configuration file gets added to the *InstallDir/etc* directory at the same time that the feature is installed. This means that you can conveniently install a feature and its associated configuration at the same time.

For example, given that the **org.fusesource.fuseesb.example.cfg** configuration file is archived in a Maven repository at **mvn:org.fusesource.fuseesb.example/configadmin/1.0/cfg**, you could deploy the configuration file by adding the following element to the feature:

```
<configfile finalname="etc/org.fusesource.fuseesb.example.cfg">
  mvn:org.fusesource.fuseesb.example/configadmin/1.0/cfg
</configfile>
```

CHAPTER 8. DEPLOYING A FEATURE

8.1. 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!).

8.2. 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.

8.3. 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]**.

8.4. 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 Chapter 12, *URL Handlers*). The feature, **hot-example-camel-bundle**, is just a reference to the existing feature, **example-camel-bundle**.

HOT UNDEPLOYING A FEATURES FILE

To undeploy a features file from the hot deploy directory, simply delete the features file from the *InstallDir*/**deploy** directory **while the Apache Karaf container is running**.
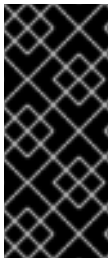
IMPORTANT

The hot undeploy mechanism does **not** work while the container is shut down. If you shut down the Karaf container, delete the features file from **deploy/**, and then restart the Karaf container, the features will **not** be undeployed after you restart the container (you can, however, undeploy the features manually using the **features:uninstall** console command).

8.5. 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.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-extras.camel-extra.karaf/camel-extra/2.21.0.fuse-
000032-redhat-2/xml/features, \
    mvn:org.apache.karaf.features/spring-legacy/4.2.0.fuse-000191-redhat-
1/xml/features, \
    mvn:org.apache.activemq/artemis-features/2.4.0.amq-710008-redhat-
1/xml/features, \
    mvn:org.jboss.fuse.modules.patch/patch-features/7.0.0.fuse-000163-
redhat-2/xml/features, \
    mvn:org.apache.karaf.features/framework/4.2.0.fuse-000191-redhat-
```

```
1/xml/features, \
    mvn:org.jboss.fuse/fuse-karaf-framework/7.0.0.fuse-000163-redhat-
2/xml/features, \
    mvn:org.apache.karaf.features/standard/4.2.0.fuse-000191-redhat-
1/xml/features, \
    mvn:org.apache.karaf.features/enterprise/4.2.0.fuse-000191-redhat-
1/xml/features, \
    mvn:org.apache.camel.karaf/apache-camel/2.21.0.fuse-000055-redhat-
2/xml/features, \
    mvn:org.apache.cxf.karaf/apache-cxf/3.1.11.fuse-000199-redhat-
1/xml/features, \
    mvn:io.hawt/hawtio-karaf/2.0.0.fuse-000145-redhat-1/xml/features

#
# Comma separated list of features to install at startup
#
featuresBoot = \
    instance/4.2.0.fuse-000191-redhat-1, \
    cxf-commands/3.1.11.fuse-000199-redhat-1, \
    log/4.2.0.fuse-000191-redhat-1, \
    pax-cdi-weld/1.0.0, \
    camel-jms/2.21.0.fuse-000055-redhat-2, \
    ssh/4.2.0.fuse-000191-redhat-1, \
    camel-cxf/2.21.0.fuse-000055-redhat-2, \
    aries-blueprint/4.2.0.fuse-000191-redhat-1, \
    cxf/3.1.11.fuse-000199-redhat-1, \
    cxf-http-undertow/3.1.11.fuse-000199-redhat-1, \
    pax-jdbc-pool-narayana/1.2.0, \
    patch/7.0.0.fuse-000163-redhat-2, \
    cxf-rs-description-swagger2/3.1.11.fuse-000199-redhat-1, \
    feature/4.2.0.fuse-000191-redhat-1, \
    camel/2.21.0.fuse-000055-redhat-2, \
    jaas/4.2.0.fuse-000191-redhat-1, \
    camel-jaxb/2.21.0.fuse-000055-redhat-2, \
    camel-paxlogging/2.21.0.fuse-000055-redhat-2, \
    deployer/4.2.0.fuse-000191-redhat-1, \
    diagnostic/4.2.0.fuse-000191-redhat-1, \
    patch-management/7.0.0.fuse-000163-redhat-2, \
    bundle/4.2.0.fuse-000191-redhat-1, \
    kar/4.2.0.fuse-000191-redhat-1, \
    camel-csv/2.21.0.fuse-000055-redhat-2, \
    package/4.2.0.fuse-000191-redhat-1, \
    scr/4.2.0.fuse-000191-redhat-1, \
    maven/4.2.0.fuse-000191-redhat-1, \
    war/4.2.0.fuse-000191-redhat-1, \
    camel-mail/2.21.0.fuse-000055-redhat-2, \
    fuse-credential-store/7.0.0.fuse-000163-redhat-2, \
    framework/4.2.0.fuse-000191-redhat-1, \
    system/4.2.0.fuse-000191-redhat-1, \
    pax-http-undertow/6.1.2, \
    camel-jdbc/2.21.0.fuse-000055-redhat-2, \
    shell/4.2.0.fuse-000191-redhat-1, \
    management/4.2.0.fuse-000191-redhat-1, \
    service/4.2.0.fuse-000191-redhat-1, \
    camel-undertow/2.21.0.fuse-000055-redhat-2, \
    camel-blueprint/2.21.0.fuse-000055-redhat-2, \
```

```
camel-spring/2.21.0.fuse-000055-redhat-2, \
hawtio/2.0.0.fuse-000145-redhat-1, \
camel-ftp/2.21.0.fuse-000055-redhat-2, \
wrap/2.5.4, \
config/4.2.0.fuse-000191-redhat-1, \
transaction-manager-narayana/5.7.2.Final
```

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 Fuse starts up. You can also modify this configuration file, if you plan to distribute Fuse with pre-installed features.
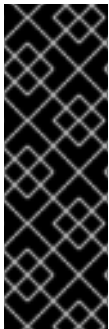
> **IMPORTANT**
>
> This method of adding a feature is only effective the **first time** a particular Apache Karaf instance boots up. Any changes made subsequently to the **featuresRepositories** setting and the **featuresBoot** setting are ignored, even if you restart the container.
>
> You could force 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 9. DEPLOYING A PLAIN JAR

Abstract

An alternative method of deploying applications into Apache Karaf is to use plain JAR files. These are usually libraries that contain **no deployment metadata**. A plain JAR is neither 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. Use the instructions in this chapter to perform the conversion process automatically, using the open source Bnd tool.

For more information on the **Bnd** tool, see Bnd tools website.

9.1. CONVERTING A JAR USING THE WRAP SCHEME

Overview

You have the option of converting a JAR into a bundle using the `wrap:` protocol, which can be used with any existing URL format. The `wrap:` protocol is based on the Bnd utility.

Syntax

The `wrap:` protocol has the following basic syntax:

```
wrap:LocationURL
```

The `wrap:` protocol can prefix any URL that locates a JAR. The locating part of the URL, *LocationURL*, is used to obtain the plain JAR and the URL handler for the `wrap:` protocol then converts the JAR automatically into a bundle.

> **NOTE**
>
> The `wrap:` protocol 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:` protocol is used just with the default settings.

Default properties

The `wrap:` protocol is based on the **Bnd** utility, so it uses exactly the same default properties to generate the bundle as Bnd does.

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, dynamically convert it into an OSGi bundle, and then install it and start it in the OSGi container:

```
karaf@root> bundle:install -s wrap:mvn:commons-logging/commons-
logging/1.1.1
```

■

Reference

The **wrap:** protocol 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:** protocol, see the Wrap Protocol reference page.

■

CHAPTER 10. 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*.

10.1. THE BLUEPRINT CONTAINER

Abstract

The Blueprint container is a dependency injection framework that simplifies interaction with the OSGi container. 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.

10.1.1. 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
<?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

Some aspects of Blueprint configuration are controlled by headers in the JAR's manifest file, **META-INF/MANIFEST.MF**, as follows:

- Custom Blueprint file locations.

- 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.

10.1.2. Defining a Service Bean

Overview

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**.

Sample beans

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:

```
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;
    }
    ...
}
```

References

For more details on defining Blueprint beans, consult the following references:

- Spring Dynamic Modules Reference Guide v2.0, Blueprint chapter.

- Section 121 **Blueprint Container Specification**, from the OSGi Compendium Services R4.2 specification.

10.1.3. Using properties to configure Blueprint

Overview

This section describes how to configure Blueprint using properties held in a file which is outside the Camel context.

Configuring Blueprint beans

Blueprint beans can be configured by using variables that can be substitued with properties from an external file. You need to declare the **ext** namespace and add the **property placeholder** bean in your Blueprint xml. Use the **Property-Placeholder** bean to declare the location of your properties file to Blueprint.

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
    xmlns:ext="http://aries.apache.org/blueprint/xmlns/blueprint-
ext/v1.2.0">

    <ext:property-placeholder>
      <ext:location>file:etc/ldap.properties</ext:location>
    </ext:property-placeholder>
    ...
    <bean ...>
        <property name="myProperty" value="${myProperty}" />
    </bean>
</blueprint>
```

The specification of **property-placeholder** configuration options can be found at http://aries.apache.org/schemas/blueprint-ext/blueprint-ext.xsd.

10.2. 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 10.1, "Sample Service Export with a Single Interface".

Example 10.1. 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 10.2, "Sample Account Classes and Interfaces"

Example 10.2. Sample Account Classes and Interfaces

```
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**.
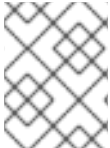
> **NOTE**
>
> The **entry** element ought to belong to the Blueprint namespace. The use of the **beans:entry** element in Spring's implementation of Blueprint is non-standard.

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:

```
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.

10.3. 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. You can use either the **reference** element or the **reference-list** element to import an OSGi service. 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:

- Reference manager.

- 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 10.1, "Reference to Stateless Service" shows an overview of the model for accessing a stateless service using the reference manager.

Figure 10.1. Reference to Stateless Service
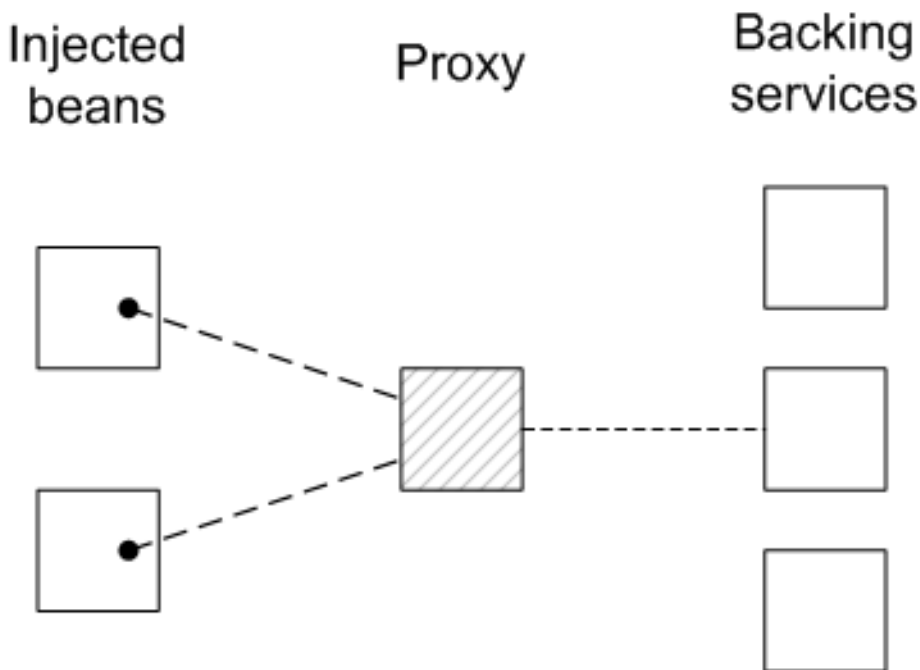
Beans in the client Blueprint container get injected with a proxy object (the *provided object*), which is backed by a service object (the *backing service*) from the OSGi service registry. This model explicitly takes advantage of the fact that stateless services are interchangeable, in the following 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 10.2, "List of References to Stateful Services" shows an overview of the model for accessing a

stateful service using the reference list manager.

Figure 10.2. List of References to Stateful Services
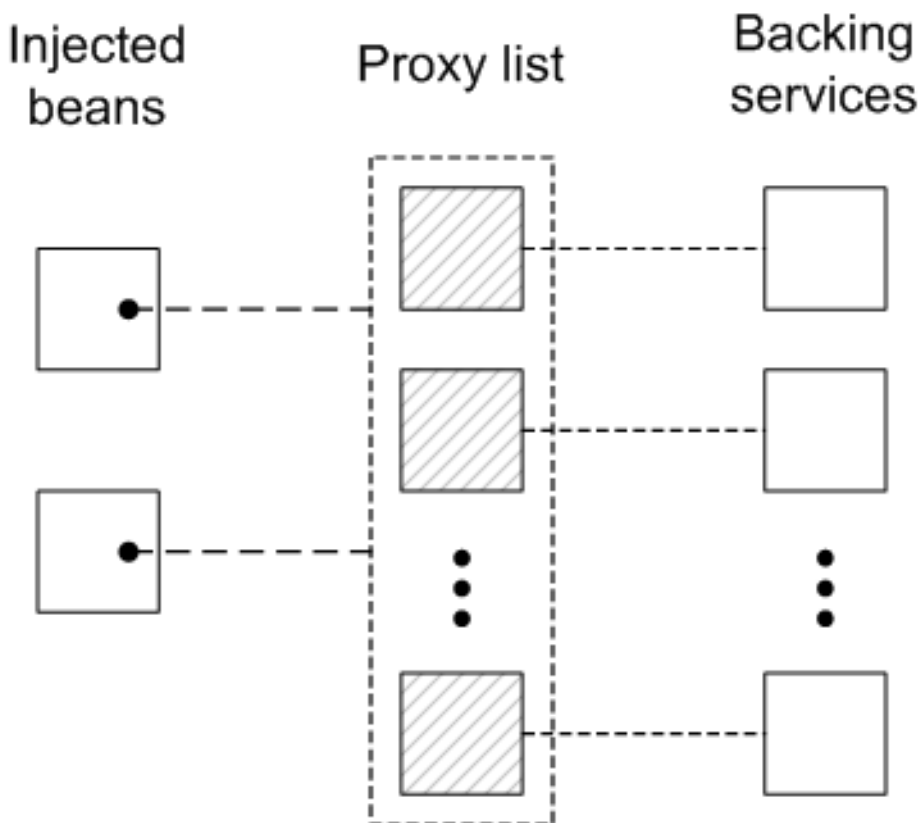
Beans in the client Blueprint container get injected with a `java.util.List` object (the *provided object*), which contains a list of proxy objects. Each proxy is backed by a unique service instance in the OSGi service registry. Unlike the stateless model, backing services are **not** considered to be 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 10.1, "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 10.1, "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 Mandatory dependencies). It is possible 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

- **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.

10.4. PUBLISHING AN OSGI SERVICE

10.4.1. 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.

10.4.2. 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.

10.4.3. 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).

10.4.4. 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 3.1, "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>
```

10.4.5. 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 10.3, "The HelloWorldSvc Interface" to it.

Example 10.3. The HelloWorldSvc Interface

```
package org.fusesource.example.service;

public interface HelloWorldSvc
{
    public void sayHello();
}
```

10.4.6. 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 10.4, "The HelloWorldSvcImpl Class" to it.

Example 10.4. 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!" );
    }

}
```

10.4.7. 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 10.5, "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 10.5, "Blueprint File for Exporting a Service".

Example 10.5. 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>
```

10.4.8. 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 Fuse console, enter the following command:

   ```
   Jkaraf@root()> bundle: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:

```
karaf@root()> bundle: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 12.1, "File URL Handler".

3. **Check that the service has been created**—to check that the bundle has started successfully, enter the following Red Hat Fuse console command:

```
karaf@root()> bundle: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)
```

10.5. ACCESSING AN OSGI SERVICE

10.5.1. 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.

10.5.2. 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.

10.5.3. 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).

10.5.4. 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 3.1, "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>
```

10.5.5. 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 10.6, "Blueprint File for Importing a Service".

Example 10.6. 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).

10.5.6. 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 10.7, "The Client Class".

Example 10.7. The Client Class

```
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.

10.5.7. 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 Fuse console, enter the following command:

   ```
   karaf@root()> bundle: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:

   ```
   karaf@root()> bundle: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 12.1, "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!
   ```

10.6. INTEGRATION WITH APACHE CAMEL

10.6.1. 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.

10.6.2. 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 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.

10.6.3. Sample OSGi service interface

Consider the OSGi service defined by the following Java interface, which defines the single method, **getGreeting()**:

```
package org.fusesource.example.hello.boston;

public interface HelloBoston {
    public String getGreeting();
}
```

10.6.4. 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).

10.6.5. 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 Apache Camel Development Guide Bean Integration .

> **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.

10.6.6. 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.

CHAPTER 11. DEPLOYING USING A JMS BROKER

Abstract

Fuse 7.0 does not ship with a default internal broker, but it is designed to interface with four external JMS brokers.

Fuse 7.0 containers contain broker client libraries for the supported external brokers.

See Supported Configurations for more information about the external brokers, client and Camel component combinations that are available for messaging on Fuse 7.0.

11.1. AMQ 7 QUICKSTART

A quickstart is provided to demonstrate the set up and deployment of apps using the AMQ 7 broker.

Setup the quickstart

1. Navigate to **FUSE_HOME/quickstarts/beginner/camel-jms**

2. Enter **mvn clean install** to build the quickstart.

3. Copy the file **org.ops4j.connectionfactory-amq7.cfg** from **FUSE_HOME/quickstarts/beginner/camel-jms/src/main** directory to the **FUSE_HOME/etc** directory in your Fuse installation. Verify its contents for the correct broker URL and credentials. By default, the broker URL is set to tcp://localhost:61616 following AMQ 7's CORE protocol. Credentials are set to admin/admin. Change these details to suit your external broker.

4. Start Fuse by running **./bin/fuse** on Linux or **bin\fuse.bat** on Windows.

5. In the Fuse console, enter the following commands:

   ```
   feature:install pax-jms-pool artemis-jms-client camel-blueprint
   camel-jms

   install -s mvn:org.jboss.fuse.quickstarts/camel-
   jms/${project.version}
   ```

   Fuse will give you a bundle ID when the bundle is deployed.

6. Enter **log:display** to see the start up log information. Check to make sure the bundle was deployed successfully.

```
12:13:50.445 INFO [Blueprint Event Dispatcher: 1] Attempting to start
Camel Context jms-example-context
12:13:50.446 INFO [Blueprint Event Dispatcher: 1] Apache Camel
2.21.0.fuse-000030 (CamelContext: jms-example-context) is starting
12:13:50.446 INFO [Blueprint Event Dispatcher: 1] JMX is enabled
12:13:50.528 INFO [Blueprint Event Dispatcher: 1] StreamCaching is not in
use. If using streams then its recommended to enable stream caching. See
more details at http://camel.apache.org/stream-caching.html
12:13:50.553 INFO [Blueprint Event Dispatcher: 1] Route: file-to-jms-route
started and consuming from: file://work/jms/input
```

```
12:13:50.555 INFO [Blueprint Event Dispatcher: 1] Route: jms-cbr-route
started and consuming from: jms://queue:incomingOrders?transacted=true
12:13:50.556 INFO [Blueprint Event Dispatcher: 1] Total 2 routes, of which
2 are started
```

Run the quickstart

1. When the Camel routes run the **FUSE_HOME/quickstarts/beginner/camel-jms/work/jms/input** directory will be created. Copy the files from the **FUSE_HOME/quickstarts/beginner/camel-jms/src/main/data** directory to the **FUSE_HOME/quickstarts/beginner/camel-jms/work/jms/input** directory.

2. The files copied into the **…/src/main/data** file are order files. Wait for a minute and then check the **FUSE_HOME/quickstarts/beginner/camel-jms/work/jms/output** directory. The files will be sorted into separate directories according to their country of destination:

   - **order1.xml**, **order2.xml** and **order4.xml** in **FUSE_HOME/quickstarts/beginner/camel-jms/work/jms/output/others/**

   - **order3.xml** and **order5.xml** in **FUSE_HOME/quickstarts/beginner/camel-jms/work/jms/output/us**

   - **order6.xml** in **FUSE_HOME/quickstarts/beginner/camel-jms/work/jms/output/fr**

3. Use **log:display** to see the log messages:

```
Receiving order order1.xml
Sending order order1.xml to another country
Done processing order1.xml
```

1. Camel commands will show details about the context:

Use **camel:context-list** to show the context details:

```
Context                Status              Total #        Failed #
Inflight #    Uptime
-------                ------              -------        --------      --
--------     ------
jms-example-context    Started                  12               0
0    3 minutes
```

Use **camel:route-list** to display the Camel routes in the context:

```
Context                Route            Status                Total #
Failed #      Inflight #    Uptime
-------                -----            ------                -------
--------      ----------    ------
jms-example-context    file-to-jms-route    Started                   6
0             0    3 minutes
jms-example-context    jms-cbr-route        Started                   6
0             0    3 minutes
```

Use **camel:route-info** to display the exchange statistics:

```
karaf@root()> camel:route-info jms-cbr-route jms-example-context
Camel Route jms-cbr-route
    Camel Context: jms-example-context
    State: Started
    State: Started

Statistics
    Exchanges Total: 6
    Exchanges Completed: 6
    Exchanges Failed: 0
    Exchanges Inflight: 0
    Min Processing Time: 2 ms
    Max Processing Time: 12 ms
    Mean Processing Time: 4 ms
    Total Processing Time: 29 ms
    Last Processing Time: 4 ms
    Delta Processing Time: 1 ms
    Start Statistics Date: 2018-01-30 12:13:50
    Reset Statistics Date: 2018-01-30 12:13:50
    First Exchange Date: 2018-01-30 12:19:47
    Last Exchange Date: 2018-01-30 12:19:47
```

11.2. USING THE ARTEMIS CORE CLIENT

The Artemis core client can be used to connect to an external broker instead of **qpid-jms-client**.

Connect using the Artemis core client

1. To enable the Artemis core client, start Fuse. Navigate to the **FUSE_HOME** directory and enter **./bin/fuse** on Linux or **bin\fuse.bat** on Windows.

2. Add the Artemis client as a feature using the following command: **feature:install artemis-core-client**

3. When you are writing your code you need to connect the Camel component with the connection factory.

Import the connection factory:

```
import org.apache.qpid.jms.JmsConnectionFactory;
```

Set up the connection:

```
ConnectionFactory connectionFactory = new
JmsConnectionFactory("amqp://localhost:5672");
    try (Connection connection = connectionFactory.createConnection()) {
```

CHAPTER 12. URL HANDLERS

Abstract

There are many contexts in Red Hat 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 Fuse's built-in URL handlers. This appendix describes the syntax for all of the available URL handlers.

12.1. 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!
```

12.2. 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.

12.3. 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.

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:

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 artifact. 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.defaultRepositories =
file:/path/to/JBossFuse/jboss-fuse-7.0.0.fuse-000191-redhat-
1/system@snapshots@id=karaf.system,file:/home/userid/.m2/repository@snapsh
ots@id=local,file:/path/to/JBossFuse/jboss-fuse-7.0.0.fuse-000191-redhat-
1/local-repo@snapshots@id=karaf.local-repo,file:/path/to/JBossFuse/jboss-
fuse-7.0.0.fuse-000191-redhat-1/system@snapshots@id=child.karaf.system
    org.ops4j.pax.url.mvn.globalChecksumPolicy = warn
    org.ops4j.pax.url.mvn.globalUpdatePolicy = daily
    org.ops4j.pax.url.mvn.localRepository = /path/to/JBossFuse/jboss-fuse-
7.0.0.fuse-000191-redhat-1/data/repository
    org.ops4j.pax.url.mvn.repositories =
http://repo1.maven.org/maven2@id=maven.central.repo,
https://maven.repository.redhat.com/ga@id=redhat.ga.repo,
https://maven.repository.redhat.com/earlyaccess/all@id=redhat.ea.repo,
https://repository.jboss.org/nexus/content/groups/ea@id=fuseearlyaccess
    org.ops4j.pax.url.mvn.settings = /path/to/jboss-fuse-7.0.0.fuse-000191-
redhat-1/etc/maven-settings.xml
    org.ops4j.pax.url.mvn.useFallbackRepositories = false
    service.pid = org.ops4j.pax.url.mvn
```

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.

12.4. 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 12.1, "Default Instructions for Wrapping a JAR".

Table 12.1. Default Instructions for Wrapping a JAR

Manifest Header	Default Value
`Import-Package`	`*;resolution:=optional`
`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.

12.5. 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.
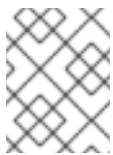
> **NOTE**
>
> Earlier versions of PAX Web used the property, **Webapp-Context**, which is now **deprecated**.

Default instructions

By default, the War URL handler adds manifest headers to the WAR's **META-INF/Manifest.mf** file as shown in Table 12.2, "Default Instructions for Wrapping a WAR File".

Table 12.2. 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.

PART II. USER GUIDE

This part contains configuration and preparation information for Apache Karaf on Red Hat Fuse.

CHAPTER 13. INTRODUCTION TO THE DEPLOYING INTO APACHE KARAF USER GUIDE

Abstract

Before you use this User Guide section of the Deploying into Apache Karaf guide, you must have installed the latest version of Red Hat Fuse, following the instructions in Installing on Apache Karaf.

13.1. INTRODUCING FUSE CONFIGURATION

Abstract

The OSGi Configuration Admin service specifies the configuration information for deployed services and ensures that the services receive that data when they are active.

13.2. OSGI CONFIGURATION

A configuration is a list of name-value pairs read from a `.cfg` file in the **FUSE_HOME/etc** directory. The file is interpreted using the Java properties file format. The filename is mapped to the persistent identifier (PID) of the service that is to be configured. In OSGi, a PID is used to identify a service across restarts of the container.

13.3. CONFIGURATION FILES

You can configure the Red Hat Fuse runtime using the following files:

Table 13.1. Fuse Configuration Files

Filename	Description
`config.properties`	The main configuration file for the container.
`custom.properties`	The main configuration file for custom properties for the container.
`keys.properties`	Lists the users who can access the Fuse runtime using the SSH key-based protocol. The file's contents take the format *username=publicKey,role*
`org.apache.karaf.features.repos.cfg`	The features repository URLs.
`org.apache.karaf.features.cfg`	Configures a list of feature repositories to be registered and a list of features to be installed when Fuse starts up for the first time.
`org.apache.karaf.jaas.cfg`	Configures options for the Karaf JAAS login module. Mainly used for configuring encrypted passwords (disabled by default).

Filename	Description
`org.apache.karaf.log.cfg`	Configures the output of the **log** console commands.
`org.apache.karaf.management.cfg`	Configures the JMX system.
`org.apache.karaf.shell.cfg`	Configures the properties of remote consoles.
`org.ops4j.pax.logging.cfg`	Configures the logging system.
`org.ops4j.pax.transx.tm.narayana.cfg`	Narayana transaction manager configuration
`org.ops4j.pax.url.mvn.cfg`	Configures additional URL resolvers.
`org.ops4j.pax.web.cfg`	Configures the default Jetty container (Web server). See olink:FMQSecurity/WebConsole.
`startup.properties`	Specifies which bundles are started in the container and their start-levels. Entries take the format *bundle=start-level*.
`system.properties`	Specifies Java system properties. Any properties set in this file are available at runtime using `System.getProperties()`.
`users.properties`	Lists the users who can access the Fuse runtime either remotely or via the web console. The file's contents take the format *username=password,role*
`setenv` or `setenv.bat`	This file is in the **/bin** directory. It is used to set JVM options. The file's contents take the format **JAVA_MIN_MEM=*512M***, where *512M* is the minimum size of Java memory. See the section called "Setting Java Options" for more information.

CONFIGURATION FILE NAMING CONVENTION

The file naming convention for configuration files depends on whether the configuration is intended for an OSGi Managed Service or for an OSGi Managed Service factory.

The configuration file for an OSGi Managed Service obeys the following naming convention:

```
<PID>.cfg
```

Where **<PID>** is the *persistent ID* of the OSGi Managed Service (as defined in the OSGi Configuration Admin specification). A persistent ID is normally dot-delimited—for example, `org.ops4j.pax.web`.

The configuration file for an OSGi Managed Service Factory obeys the following naming convention:

```
<PID>-<InstanceID>.cfg
```

Where **<PID>** is the *persistent ID* of the OSGi Managed Service Factory. In the case of a managed service factory's **<PID>,** you can append a hyphen followed by an arbitrary instance ID, **<InstanceID>**. The managed service factory then creates a unique service instance for each **<InstanceID>** that it finds.

SETTING JAVA OPTIONS

Java Options can be set using the **/bin/setenv** file in Linux, or the **bin/setenv.bat** file for Windows. Use this file to directly set a group of Java options: JAVA_MIN_MEM, JAVA_MAX_MEM, JAVA_PERM_MEM, JAVA_MAX_PERM_MEM. Other Java options can be set using the EXTRA_JAVA_OPTS variable.

For example, to allocate minimum memory for the JVM use

```
JAVA_MIN_MEM=512M # Minimum memory for the JVM
```

```
To set a Java option other than the direct options, use
```

```
EXTRA_JAVA_OPTS="Java option"
```

```
For example,
```

```
EXTRA_JAVA_OPTS="-XX:+UseG1GC"
```

13.4. CONFIG CONSOLE COMMANDS

There are a number of console commands that can be used to change or interrogate the configuration of Fuse 7.0.

See the Config section in the Apache Karaf Console Reference for more details about the **config:** commands.

13.5. JMX CONFIGMBEAN

On the JMX layer, the **MBean** is dedicated to configuration management.

The **ConfigMBean** object name is: **org.apache.karaf:type=config,name=*`**.

14.1.2.1. Attributes

The config MBean contains a list of all configuration PIDs.

14.1.2.2. Operations

Table 13.2. JMX MBean Operations

Operation name	Description

Operation name	Description
`listProperties(pid)`	returns the list of properties (property=value formatted) for the configuration pid.
`deleteProperty(pid, property)`	deletes the property from the configuration pid.
`appendProperty(pid, property, value)`	appends value at the end of the value of the property of the configuration pid.
`setProperty(pid, property, value)`	sets value for the value of the property of the configuration pid.
`delete(pid)`	deletes the configuration identified by the pid.
`create(pid)`	creates an empty (without any property) configuration with pid.
`update(pid, properties)`	updates a configuration identified with pid with the provided properties map.

13.6. USING THE CONSOLE

13.6.1. Available commands

To see a list of the available commands in the console, you can use the **help**:

```
karaf@root()> help
bundle                          Enter the subshell
bundle:capabilities             Displays OSGi capabilities of a given
bundles.
bundle:classes                  Displays a list of classes/resources
contained in the bundle
bundle:diag                     Displays diagnostic information why a
bundle is not Active
bundle:dynamic-import           Enables/disables dynamic-import for a
given bundle.
bundle:find-class               Locates a specified class in any
deployed bundle
bundle:headers                  Displays OSGi headers of a given
bundles.
bundle:id                       Gets the bundle ID.
...
```

You have the list of all commands with a short description.

You can use the tab key to get a quick list of all commands:

```
karaf@root()> Display all 294 possibilities? (y or n)
...
```

13.6.2. Subshell and completion mode

The commands have a scope and a name. For instance, the command **feature:list** has **feature** as scope, and **list** as name.

Karaf "groups" the commands by scope. Each scope form a subshell.

You can directly execute a command with its full qualified name (scope:name):

```
karaf@root()> feature:list
...
```

or enter in a subshell and type the command contextual to the subshell:

```
karaf@root()> feature
karaf@root(feature)> list
```

You can note that you enter in a subshell directly by typing the subshell name (here **feature**). You can "switch" directly from a subshell to another:

```
karaf@root()> feature
karaf@root(feature)> bundle
karaf@root(bundle)>
```

The prompt displays the current subshell between ().

The **exit** command goes to the parent subshell:

```
karaf@root()> feature
karaf@root(feature)> exit
karaf@root()>
```

The completion mode defines the behaviour of the tab key and the help command.

You have three different modes available:

- GLOBAL

- FIRST

- SUBSHELL

You can define your default completion mode using the completionMode property in **etc/org.apache.karaf.shell.cfg** file. By default, you have:

```
completionMode = GLOBAL
```

You can also change the completion mode "on the fly" (while using the Karaf shell console) using the **shell:completion** command:

```
karaf@root()> shell:completion
GLOBAL
karaf@root()> shell:completion FIRST
```

```
karaf@root()> shell:completion
FIRST
```

shell:completion can inform you about the current completion mode used. You can also provide the new completion mode that you want.

GLOBAL completion mode is the default one in Karaf 4.0.0 (mostly for transition purpose).

GLOBAL mode doesn't really use subshell: it's the same behavior as in previous Karaf versions.

When you type the tab key, whatever in which subshell you are, the completion will display all commands and all aliases:

```
karaf@root()> <TAB>
karaf@root()> Display all 273 possibilities? (y or n)
...
karaf@root()> feature
karaf@root(feature)> <TAB>
karaf@root(feature)> Display all 273 possibilities? (y or n)
```

FIRST completion mode is an alternative to the GLOBAL completion mode.

If you type the tab key on the root level subshell, the completion will display the commands and the aliases from all subshells (as in GLOBAL mode). However, if you type the tab key when you are in a subshell, the completion will display only the commands of the current subshell:

```
karaf@root()> shell:completion FIRST
karaf@root()> <TAB>
karaf@root()> Display all 273 possibilities? (y or n)
...
karaf@root()> feature
karaf@root(feature)> <TAB>
karaf@root(feature)>
info install list repo-add repo-list repo-remove uninstall version-list
karaf@root(feature)> exit
karaf@root()> log
karaf@root(log)> <TAB>
karaf@root(log)>
clear display exception-display get log set tail
```

SUBSHELL completion mode is the real subshell mode.

If you type the tab key on the root level, the completion displays the subshell commands (to go into a subshell), and the global aliases. Once you are in a subshell, if you type the TAB key, the completion displays the commands of the current subshell:

```
karaf@root()> shell:completion SUBSHELL
karaf@root()> <TAB>
karaf@root()>
* bundle cl config dev feature help instance jaas kar la ld lde log
log:list man package region service shell ssh system
karaf@root()> bundle
karaf@root(bundle)> <TAB>
karaf@root(bundle)>
capabilities classes diag dynamic-import find-class headers info install
```

```
list refresh requirements resolve restart services start start-level stop
uninstall update watch
karaf@root(bundle)> exit
karaf@root()> camel
karaf@root(camel)> <TAB>
karaf@root(camel)>
backlog-tracer-dump backlog-tracer-info backlog-tracer-start backlog-
tracer-stop context-info context-list context-start context-stop endpoint-
list route-info route-list route-profile route-reset-stats
route-resume route-show route-start route-stop route-suspend
```

13.6.3. Unix like environment

Karaf console provides a full Unix like environment.

13.6.3.1. Help or man

We already saw the usage of the **help** command to display all commands available.

But you can also use the **help** command to get details about a command or the **man** command which is an alias to the **help** command. You can also use another form to get the command help, by using the **--help** option to the command.

So these commands

```
karaf@root()> help feature:list
karaf@root()> man feature:list
karaf@root()> feature:list --help
```

All produce the same help output:

```
DESCRIPTION
        feature:list

        Lists all existing features available from the defined
repositories.

SYNTAX
        feature:list [options]

OPTIONS
        --help
                Display this help message
        -o, --ordered
                Display a list using alphabetical order
        -i, --installed
                Display a list of all installed features only
        --no-format
                Disable table rendered output
```

13.6.3.2. Completion

When you type the tab key, Karaf tries to complete:

- subshell

- commands

- aliases

- command arguments

- command options

13.6.3.3. Alias

An alias is another name associated to a given command.

The **shell:alias** command creates a new alias. For instance, to create the **list-installed-features** alias to the actual **feature:list -i** command, you can do:

```
karaf@root()> alias "list-features-installed = { feature:list -i }"
karaf@root()> list-features-installed
Name         | Version | Required | State   | Repository    | Description
---------------------------------------------------------------------------
-------------------------------------------------------------
feature      | 4.0.0   | x        | Started | standard-4.0.0 | Features
Support
shell        | 4.0.0   | x        | Started | standard-4.0.0 | Karaf Shell
deployer     | 4.0.0   | x        | Started | standard-4.0.0 | Karaf
Deployer
bundle       | 4.0.0   | x        | Started | standard-4.0.0 | Provide
Bundle support
config       | 4.0.0   | x        | Started | standard-4.0.0 | Provide OSGi
ConfigAdmin support
diagnostic   | 4.0.0   | x        | Started | standard-4.0.0 | Provide
Diagnostic support
instance     | 4.0.0   | x        | Started | standard-4.0.0 | Provide
Instance support
jaas         | 4.0.0   | x        | Started | standard-4.0.0 | Provide JAAS
support
log          | 4.0.0   | x        | Started | standard-4.0.0 | Provide Log
support
package      | 4.0.0   | x        | Started | standard-4.0.0 | Package
commands and mbeans
service      | 4.0.0   | x        | Started | standard-4.0.0 | Provide
Service support
system       | 4.0.0   | x        | Started | standard-4.0.0 | Provide
System support
kar          | 4.0.0   | x        | Started | standard-4.0.0 | Provide KAR
(KARaf archive) support
ssh          | 4.0.0   | x        | Started | standard-4.0.0 | Provide a
SSHd server on Karaf
management   | 4.0.0   | x        | Started | standard-4.0.0 | Provide a JMX
MBeanServer and a set of MBeans in
```

At login, the Apache Karaf console reads the **etc/shell.init.script** file where you can create your aliases. It's similar to a bashrc or profile file on Unix.

```
ld = { log:display $args } ;
lde = { log:exception-display $args } ;
la = { bundle:list -t 0 $args } ;
ls = { service:list $args } ;
cl = { config:list "(service.pid=$args)" } ;
halt = { system:shutdown -h -f $args } ;
help = { *:help $args | more } ;
man = { help $args } ;
log:list = { log:get ALL } ;
```

You can see here the aliases available by default:

- **ld** is a short form to display log (alias to **log:display** command)

- **lde** is a short form to display exceptions (alias to **log:exception-display** command)

- **la** is a short form to list all bundles (alias to **bundle:list -t 0** command)

- **ls** is a short form to list all services (alias to **service:list** command)

- **cl** is a short form to list all configurations (alias to **config:list** command)

- **halt** is a short form to shutdown Apache Karaf (alias to **system:shutdown -h -f** command)

- **help** is a short form to display help (alias to ***:help** command)

- **man** is the same as help (alias to **help** command)

- **log:list** displays all loggers and level (alias to **log:get ALL** command)

You can create your own aliases in the **etc/shell.init.script** file.

13.6.3.4. Key binding

Like on most Unix environment, Karaf console support some key bindings:

- the arrows key to navigate in the commands history

- CTRL-D to logout/shutdown Karaf

- CTRL-R to search previously executed command

- CTRL-U to remove the current line

13.6.3.5. Pipe

You can pipe the output of one command as input to another one. It's a pipe, using the | character:

```
karaf@root()> feature:list |grep -i war
pax-war                      | 4.1.4                                  |
| Uninstalled | org.ops4j.pax.web-4.1.4  | Provide support of a full
WebContainer
pax-war-tomcat               | 4.1.4                                  |
| Uninstalled | org.ops4j.pax.web-4.1.4  |
war                          | 4.0.0                                  |
```

```
| Uninstalled | standard-4.0.0              | Turn Karaf as a full
WebContainer
blueprint-web                   | 4.0.0                              |
| Uninstalled | standard-4.0.0              | Provides an OSGI-aware Servlet
ContextListener fo
```

13.6.3.6. Grep, more, find, …

Karaf console provides some core commands similar to Unix environment:

- **shell:alias** creates an alias to an existing command

- **shell:cat** displays the content of a file or URL

- **shell:clear** clears the current console display

- **shell:completion** displays or change the current completion mode

- **shell:date** displays the current date (optionally using a format)

- **shell:each** executes a closure on a list of arguments

- **shell:echo** echoes and prints arguments to stdout

- **shell:edit** calls a text editor on the current file or URL

- **shell:env** displays or sets the value of a shell session variable

- **shell:exec** executes a system command

- **shell:grep** prints lines matching the given pattern

- **shell:head** displays the first line of the input

- **shell:history** prints the commands history

- **shell:if** allows you to use conditions (if, then, else blocks) in script

- **shell:info** prints various information about the current Karaf instance

- **shell:java** executes a Java application

- **shell:less** file pager

- **shell:logout** disconnects shell from current session

- **shell:more** is a file pager

- **shell:new** creates a new Java object

- **shell:printf** formats and prints arguments

- **shell:sleep** sleeps for a bit then wakes up

- **shell:sort** writes sorted concatenation of all files to stdout

- **shell:source** executes commands contained in a script

- **shell:stack-traces-print** prints the full stack trace in the console when the execution of a command throws an exception

- **shell:tac** captures the STDIN and returns it as a string

- **shell:tail** displays the last lines of the input

- **shell:threads** prints the current thread

- **shell:watch** periodically executes a command and refresh the output

- **shell:wc** prints newline, words, and byte counts for each file

- **shell:while** loop while the condition is true

You don't have to use the fully qualified name of the command, you can directly use the command name as long as it is unique. So you can use 'head' instead of 'shell:head'

Again, you can find details and all options of these commands using **help** command or **--help** option.

13.6.3.7. Scripting

The Apache Karaf Console supports a complete scripting language, similar to bash or csh on Unix.

The **each** (**shell:each**) command can iterate in a list:

```
karaf@root()> list = [1 2 3]; each ($list) { echo $it }
1
2
3
```

> **NOTE**
>
> The same loop could be written with the **shell:while** command:
>
> ```
> karaf@root()> a = 0 ; while { %((a+=1) <= 3) } { echo $a }
> 1
> 2
> 3
> ```

You can create the list yourself (as in the previous example), or some commands can return a list too.

We can note that the console created a "session" variable with the name **list** that you can access with **$list**.

The **$it** variable is an implicit one corresponding to the current object (here the current iterated value from the list).

When you create a list with **[]**, Apache Karaf console creates a Java ArrayList. It means that you can use methods available in the ArrayList objects (like get or size for instance):

```
karaf@root()> list = ["Hello" world]; echo ($list get 0) ($list get 1)
```

```
Hello world
```

We can note here that calling a method on an object is directly using **(object method argument)**. Here **($list get 0)** means **$list.get(0)** where **$list** is the ArrayList.

The **class** notation will display details about the object:

```
karaf@root()> $list class
...
ProtectionDomain     ProtectionDomain  null
 null
 <no principals>
 java.security.Permissions@6521c24e (
 ("java.security.AllPermission" "<all permissions>" "<all actions>")
)


Signers              null
SimpleName           ArrayList
TypeParameters       [E]
```

You can "cast" a variable to a given type.

```
karaf@root()> ("hello world" toCharArray)
[h, e, l, l, o,   , w, o, r, l, d]
```

If it fails, you will see the casting exception:

```
karaf@root()> ("hello world" toCharArray)[0]
Error executing command: [C cannot be cast to [Ljava.lang.Object;
```

You can "call" a script using the **shell:source** command:

```
karaf@root> shell:source script.txt
True!
```

where **script.txt** contains:

```
foo = "foo"
if { $foo equals "foo" } {
  echo "True!"
}
```

NOTE

The spaces are important when writing script. For instance, the following script is not correct:

```
if{ $foo equals "foo" } ...
```

and will fail with:

```
karaf@root> shell:source script.txt
Error executing command: Cannot coerce echo "true!"() to any of
[]
```

because a space is missing after the **if** statement.

As for the aliases, you can create init scripts in the **etc/shell.init.script** file. You can also named you script with an alias. Actually, the aliases are just scripts.

See the Scripting section of the developers guide for details.

13.6.4. Security

The Apache Karaf console supports a Role Based Access Control (RBAC) security mechanism. It means that depending of the user connected to the console, you can define, depending of the user's groups and roles, the permission to execute some commands, or limit the values allowed for the arguments.

Console security is detailed in the Security section of this user guide.

CHAPTER 14. PROVISIONING

Apache Karaf supports the provisioning of applications and modules using the concept ofApache KarafFeatures.

14.1. APPLICATION

An application consists of all modules, configuration, and transitive applications required for a feature.

14.2. OSGI

Apache Karaf natively supports the deployment of OSGi applications.

An OSGi application is a set of OSGi bundles. An OSGi bundle is a regular jar file, with additional metadata in the jar MANIFEST.

In OSGi, a bundle can depend on other bundles. So, it means that to deploy an OSGi application, you may also need to firstly deploy a lot of other bundles required by the application.

You have to identify the dependant bundles first and install them. These dependency bundles may require other bundles to satisfy their own dependencies.

An application requires configuration. Before starting your application, in addition to the dependency bundles, you have to create or deploy the configuration.

14.3. FEATURE AND RESOLVER

Apache Karaf provides a simple and flexible way to provision applications.

In Apache Karaf, the application provisioning is an Apache Karaf "feature".

A feature describes an application as:

- a name

- a version

- a optional description (eventually with a long description)

- a set of bundles

- optionally a set configurations or configuration files

- optionally a set of dependency features

When you install a feature, Apache Karaf installs all resources described in the feature. Apache Karaf will automatically resolve and install all bundles, configurations, and dependency features described in the feature.

The feature resolver checks the service requirements, and install the bundles providing the services matching the requirements. The default mode enables this behavior only for "new style" features repositories (basically, the features repositories XML with schema equal or greater to 1.3.0). It doesn't apply for "old style" features repositories (coming from Apache Karaf2 or 3).

You can change the service requirements enforcement mode in **etc/org.apache.karaf.features.cfg** file, using the **serviceRequirements** property.

```
serviceRequirements=default
```

The possible values are:

- disable: service requirements are completely ignored, for both **old style** and **new style** features repositories

- default: service requirements are ignored for **old style** features repositories, and enabled for **new style** features repositories.

- enforce: service requirements are always verified, for **old style** and **new style** features repositories.

Additionally, a feature can also define requirements. In that case,Apache Karafwill automatically add additional bundles or features providing the capabilities to satisfy the requirements.

A feature has a complete lifecycle: install, start, stop, update, uninstall.

14.4. FEATURES REPOSITORIES

The features are described in a features XML descriptor. This XML file contains the description of a set of features.

A features XML descriptor is named a "features repository". Before being able to install a feature, you have to register the features repository that provides the feature (using **feature:repo-add** command or FeatureMBean as described later).

For instance, the following XML file (or "features repository") describes the **feature1** and **feature2** features:

```
<features xmlns="http://karaf.apache.org/xmlns/features/v1.3.0">
  <feature name="feature1" version="1.0.0">
    <bundle>...</bundle>
    <bundle>...</bundle>
  </feature>
  <feature name="feature2" version="1.1.0">
    <feature>feature1</feature>
    <bundle>...</bundle>
  </feature>
</features>
```

We can note that the features XML has a schema.

The **feature1** feature is available in version **1.0.0**, and contains two bundles. The **<bundle/>** element contains a URL to the bundle artifact (see [Artifacts repositories and URLs section|urls] for details). If you install the **feature1** feature (using **feature:install** or the FeatureMBean as described later),{Apache Karaf} will automatically installs the two bundles described. The **feature2** feature is available in version **1.1.0**, and contains a reference to the**feature1** feature and a bundle. The **<feature/>** element contains the name of a feature. A specific feature version can be defined using the **version** attribute to the **<feature/>** element (**<feature version="1.0.0">feature1</feature>**). If the **version** attribute is not specified, Apache Karaf

will install the latest version available. If you install the **feature2** feature (using **feature:install** or the FeatureMBean as described later), Apache Karaf will automatically installs **feature1** (if it's not already installed) and the bundle.

A feature repository is registered using the URL to the features XML file.

The features state is stored in the Apache Karaf cache (in the **KARAF_DATA** folder). You can restart Apache Karaf, the previously installed features remain installed and available after restart. If you do a clean restart or you delete the Apache Karaf cache (delete the **KARAF_DATA** folder), all previously features repositories registered and features installed will be lost: you will have to register the features repositories and install features by hand again. To prevent this behaviour, you can specify features as boot features.

14.5. BOOT FEATURES

You can describe some features as boot features. A boot feature will be automatically install by Apache Karaf, even if it has not been previously installed using **feature:install** or FeatureMBean.

Apache Karaf features configuration is located in the **etc/org.apache.karaf.features.cfg** configuration file.

This configuration file contains the two properties to use to define boot features:

- **featuresRepositories** contains a list (comma-separated) of features repositories (features XML) URLs.

- **featuresBoot** contains a list (comma-separated) of features to install at boot.

14.6. FEATURES UPGRADE

You can update a release by installing the same feature (with the same SNAPSHOT version or a different version).

Thanks to the features lifecycle, you can control the status of the feature (started, stopped, etc).

You can also use a simulation to see what the update will do.

14.7. OVERRIDES

Bundles defined in features can be overridden by using a file etc/overrides.properties. Each line in the file defines one override. The syntax is: <bundle-uri>[;range="[min,max)"] The given bundle will override all bundles in feature definitions with the same symbolic name if the version of the override is greater than the version of the overridden bundle and the range matches. If no range is given then compatibility on the micro version level is assumed.

So for example the override mvn:org.ops4j.pax.logging/pax-logging-service/1.8.5 would overide pax-logging-service 1.8.3 but not 1.8.6 or 1.7.0.

14.8. FEATURE BUNDLES

14.8.1. Start Level

By default, the bundles deployed by a feature will have a start-level equals to the value defined in the **etc/config.properties** configuration file, in the **karaf.startlevel.bundle** property.

This value can be "overrided" by the **start-level** attribute of the **<bundle/>** element, in the features XML.

```
    <feature name="my-project" version="1.0.0">
      <bundle start-level="80">mvn:com.mycompany.myproject/myproject-
dao</bundle>
      <bundle start-level="85">mvn:com.mycompany.myproject/myproject-
service</bundle>
    </feature>
```

The start-level attribute insure that the **myproject-dao** bundle is started before the bundles that use it.

Instead of using start-level, a better solution is to simply let the OSGi framework know what your dependencies are by defining the packages or services you need. It is more robust than setting start levels.

14.8.2. Simulate, Start and stop

You can simulate the installation of a feature using the **-t** option to **feature:install** command.

You can install a bundle without starting it. By default, the bundles in a feature are automatically started.

A feature can specify that a bundle should not be started automatically (the bundle stays in resolved state). To do so, a feature can specify the **start** attribute to false in the **<bundle/>** element:

```
    <feature name="my-project" version="1.0.0">
      <bundle start-level="80"
start="false">mvn:com.mycompany.myproject/myproject-dao</bundle>
      <bundle start-level="85"
start="false">mvn:com.mycompany.myproject/myproject-service</bundle>
    </feature>
```

14.8.3. Dependency

A bundle can be flagged as being a dependency, using the **dependency** attribute set to true on the **<bundle/>** element.

This information can be used by resolvers to compute the full list of bundles to be installed.

14.9. DEPENDENT FEATURES

A feature can depend on a set of other features:

```
    <feature name="my-project" version="1.0.0">
      <feature>other</feature>
      <bundle start-level="80"
start="false">mvn:com.mycompany.myproject/myproject-dao</bundle>
      <bundle start-level="85"
start="false">mvn:com.mycompany.myproject/myproject-service</bundle>
    </feature>
```

When the `my-project` feature is installed, the `other` feature will be automatically installed as well.

It's possible to define a version range for a dependent feature:

```
<feature name="spring-dm">
  <feature version="[2.5.6,4)">spring</feature>
  ...
</feature>
```

The feature with the highest version available in the range will be installed.

If a single version is specified, the range will be considered open-ended.

If nothing is specified, the highest available will be installed.

To specify an exact version, use a closed range such as `[3.1,3.1]`.

14.9.1. Feature prerequisites

A prerequisite feature is a special kind of dependency. If you add the `prerequisite` attribute to a dependant feature tag it will force installation and activation of bundles in the dependant feature before installation of the actual feature. This may be useful when bundles in the given feature are not using pre-installed URL such as `wrap` or `war`.

14.10. FEATURE CONFIGURATIONS

The `<config/>` element in a feature XML allows a feature to create and/or populate a configuration (identified by a configuration PID).

```
<config name="com.foo.bar">
  myProperty = myValue
</config>
```

The `name` attribute of the `<config/>` element corresponds to the configuration PID (see the [Configuration section|configuration] for details).

The installation of the feature will have the same effect as dropping a file named `com.foo.bar.cfg` in the `etc` folder.

The content of the `<config/>` element is a set of properties, following the key=value standard.

14.11. FEATURE CONFIGURATION FILES

Instead of using the `<config/>` element, a feature can specify `<configfile/>` elements.

```
<configfile finalname="/etc/myfile.cfg" override="false">URL</configfile>
```

Instead of directly manipulating the Apache Karaf configuration layer (as when using the `<config/>` element), the `<configfile/>` element directly takes a file specified by a URL, and copies the file in the location specified by the `finalname` attribute.

If not specified, the location is relative from the `KARAF_BASE` variable. It's also possible to use variable like ${karaf.home}, ${karaf.base}, ${karaf.etc}, or even system properties.

For instance:

```
<configfile finalname="${karaf.etc}/myfile.cfg"
override="false">URL</configfile>
```

If the file is already present at the desired location it is kept and the deployment of the configuration file is skipped, as a already existing file might contain customization. This behaviour can be overriden by **override** set to true.

The file URL is any URL supported by Apache Karaf (see the [Artifacts repositories and URLs|urls] of the user guide for details).

14.11.1. Requirements

A feature can also specify expected requirements. The feature resolver will try to satisfy the requirements. For that, it checks the features and bundles capabilities and will automatically install the bundles to satisfy the requirements.

For instance, a feature can contain:

```
<requirement>osgi.ee;filter:=&quot;(&amp;(osgi.ee=JavaSE)(!
(version&gt;=1.8)))&quot;</requirement>
```

The requirement specifies that the feature will work by only if the JDK version is not 1.8 (so basically 1.7).

The features resolver is also able to refresh the bundles when an optional dependency is satisfy, rewiring the optional import.

14.12. COMMANDS

14.12.1. `feature:repo-list`

The **feature:repo-list** command lists all registered features repository:

```
karaf@root()> feature:repo-list
Repository                | URL
----------------------------------------------------------------------
----------------
org.ops4j.pax.cdi-0.12.0 | mvn:org.ops4j.pax.cdi/pax-cdi-
features/0.12.0/xml/features
org.ops4j.pax.web-4.1.4  | mvn:org.ops4j.pax.web/pax-web-
features/4.1.4/xml/features
standard-4.0.0            |
mvn:org.apache.karaf.features/standard/4.0.0/xml/features
enterprise-4.0.0          |
mvn:org.apache.karaf.features/enterprise/4.0.0/xml/features
spring-4.0.0              |
mvn:org.apache.karaf.features/spring/4.0.0/xml/features
```

Each repository has a name and the URL to the features XML.

karaf parses the features XML when you register the features repository URL (using **feature:repo-add** command or the FeatureMBean as described later). If you want to force Apache Karaf to reload the features repository URL (and so update the features definition), you can use the **-r** option:

```
karaf@root()> feature:repo-list -r
Reloading all repositories from their urls

Repository              | URL
---------------------------------------------------------------------
---------------
org.ops4j.pax.cdi-0.12.0 | mvn:org.ops4j.pax.cdi/pax-cdi-
features/0.12.0/xml/features
org.ops4j.pax.web-4.1.4  | mvn:org.ops4j.pax.web/pax-web-
features/4.1.4/xml/features
standard-4.0.0           |
mvn:org.apache.karaf.features/standard/4.0.0/xml/features
enterprise-4.0.0         |
mvn:org.apache.karaf.features/enterprise/4.0.0/xml/features
spring-4.0.0             |
mvn:org.apache.karaf.features/spring/4.0.0/xml/features
```

14.12.2. `feature:repo-add`

To register a features repository, making new features available in Apache Karaf, you have to use the **feature:repo-add** command.

The **feature:repo-add** command requires the **name/url** argument. This argument accepts:

- a feature repository URL. It's an URL directly to the features XML file. Any URL described in the [Artifacts repositories and URLs section|urls] of the user guide is supported.

- a feature repository name defined in the **etc/org.apache.karaf.features.repos.cfg** configuration file.

The **etc/org.apache.karaf.features.repos.cfg** defines a list of "pre-installed/available" features repositories:

```
################################################################################
######
#
#    Licensed to the Apache Software Foundation (ASF) under one or more
#    contributor license agreements.  See the NOTICE file distributed with
#    this work for additional information regarding copyright ownership.
#    The ASF licenses this file to You under the Apache License, Version
2.0
#    (the "License"); you may not use this file except in compliance with
#    the License.  You may obtain a copy of the License at
#
#       http://www.apache.org/licenses/LICENSE-2.0
#
#    Unless required by applicable law or agreed to in writing, software
#    distributed under the License is distributed on an "AS IS" BASIS,
#    WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
implied.
#    See the License for the specific language governing permissions and
```

```
#     limitations under the License.
#
##############################################################################
######

#
# This file describes the features repository URL
# It could be directly installed using feature:repo-add command
#
enterprise=mvn:org.apache.karaf.features/enterprise/LATEST/xml/features
spring=mvn:org.apache.karaf.features/spring/LATEST/xml/features
cellar=mvn:org.apache.karaf.cellar/apache-karaf-cellar/LATEST/xml/features
cave=mvn:org.apache.karaf.cave/apache-karaf-cave/LATEST/xml/features
camel=mvn:org.apache.camel.karaf/apache-camel/LATEST/xml/features
camel-extras=mvn:org.apache-extras.camel-extra.karaf/camel-
extra/LATEST/xml/features
cxf=mvn:org.apache.cxf.karaf/apache-cxf/LATEST/xml/features
cxf-dosgi=mvn:org.apache.cxf.dosgi/cxf-dosgi/LATEST/xml/features
cxf-xkms=mvn:org.apache.cxf.services.xkms/cxf-services-xkms-
features/LATEST/xml
activemq=mvn:org.apache.activemq/activemq-karaf/LATEST/xml/features
jclouds=mvn:org.apache.jclouds.karaf/jclouds-karaf/LATEST/xml/features
openejb=mvn:org.apache.openejb/openejb-feature/LATEST/xml/features
wicket=mvn:org.ops4j.pax.wicket/features/LATEST/xml/features
hawtio=mvn:io.hawt/hawtio-karaf/LATEST/xml/features
pax-cdi=mvn:org.ops4j.pax.cdi/pax-cdi-features/LATEST/xml/features
pax-jdbc=mvn:org.ops4j.pax.jdbc/pax-jdbc-features/LATEST/xml/features
pax-jpa=mvn:org.ops4j.pax.jpa/pax-jpa-features/LATEST/xml/features
pax-web=mvn:org.ops4j.pax.web/pax-web-features/LATEST/xml/features
pax-wicket=mvn:org.ops4j.pax.wicket/pax-wicket-
features/LATEST/xml/features
ecf=http://download.eclipse.org/rt/ecf/latest/site.p2/karaf-features.xml
decanter=mvn:org.apache.karaf.decanter/apache-karaf-
decanter/LATEST/xml/features
```

You can directly provide a features repository name to the **feature:repo-add** command. For install, to install Apache Karaf Cellar, you can do:

```
karaf@root()> feature:repo-add cellar
Adding feature url mvn:org.apache.karaf.cellar/apache-karaf-
cellar/LATEST/xml/features
```

When you don't provide the optional **version** argument, Apache Karaf installs the latest version of the features repository available. You can specify a target version with the **version** argument:

```
karaf@root()> feature:repo-add cellar 4.0.0.RC1
Adding feature url mvn:org.apache.karaf.cellar/apache-karaf-
cellar/4.0.0.RC1/xml/features
```

Instead of providing a features repository name defined in the **etc/org.apache.karaf.features.repos.cfg** configuration file, you can directly provide the features repository URL to the **feature:repo-add** command:

```
karaf@root()> feature:repo-add mvn:org.apache.karaf.cellar/apache-karaf-
cellar/4.0.0.RC1/xml/features
```

```
Adding feature url mvn:org.apache.karaf.cellar/apache-karaf-
cellar/4.0.0.RC1/xml/features
```

By default, the **feature:repo-add** command just registers the features repository, it doesn't install any feature. If you specify the **-i** option, the **feature:repo-add** command registers the features repository and installs all features described in this features repository:

```
karaf@root()> feature:repo-add -i cellar
```

14.12.3. **feature:repo-refresh**

Apache Karaf parses the features repository XML when you register it (using **feature:repo-add** command or the FeatureMBean). If the features repository XML changes, you have to indicate to Apache Karaf to refresh the features repository to load the changes.

The **feature:repo-refresh** command refreshes the features repository.

Without argument, the command refreshes all features repository:

```
karaf@root()> feature:repo-refresh
Refreshing feature url mvn:org.ops4j.pax.cdi/pax-cdi-
features/0.12.0/xml/features
Refreshing feature url mvn:org.ops4j.pax.web/pax-web-
features/4.1.4/xml/features
Refreshing feature url
mvn:org.apache.karaf.features/standard/4.0.0/xml/features
Refreshing feature url
mvn:org.apache.karaf.features/enterprise/4.0.0/xml/features
Refreshing feature url
mvn:org.apache.karaf.features/spring/4.0.0/xml/features
```

Instead of refreshing all features repositories, you can specify the features repository to refresh, by providing the URL or the features repository name (and optionally version):

```
karaf@root()> feature:repo-refresh
mvn:org.apache.karaf.features/standard/4.0.0/xml/features
Refreshing feature url
mvn:org.apache.karaf.features/standard/4.0.0/xml/features
```

```
karaf@root()> feature:repo-refresh cellar
Refreshing feature url mvn:org.apache.karaf.cellar/apache-karaf-
cellar/LATEST/xml/features
```

14.12.4. **feature:repo-remove**

The **feature:repo-remove** command removes a features repository from the registered ones.

The **feature:repo-remove** command requires a argument:

- the features repository name (as displayed in the repository column of the **feature:repo-list** command output)

- the features repository URL (as displayed in the URL column of the **feature:repo-list** command output)

```
karaf@root()> feature:repo-remove karaf-cellar-4.0.0.RC1
```

```
karaf@root()> feature:repo-remove mvn:org.apache.karaf.cellar/apache-
karaf-cellar/LATEST/xml/features
```

By default, the **feature:repo-remove** command just removes the features repository from the registered ones: it doesn't uninstall the features provided by the features repository.

If you use **-u** option, the **feature:repo-remove** command uninstalls all features described by the features repository:

```
karaf@root()> feature:repo-remove -u karaf-cellar-4.0.0.RC1
```

14.12.5. `feature:list`

The **feature:list** command lists all available features (provided by the different registered features repositories):

```
Name                         | Version                      |
Required | State      | Repository             | Description
-----------------------------------------------------------------------
-----------------------------------------------------------------------
-----------------------------
pax-cdi                      | 0.12.0                       |
| Uninstalled | org.ops4j.pax.cdi-0.12.0 | Provide CDI support
pax-cdi-1.1                  | 0.12.0                       |
| Uninstalled | org.ops4j.pax.cdi-0.12.0 | Provide CDI 1.1 support
pax-cdi-1.2                  | 0.12.0                       |
| Uninstalled | org.ops4j.pax.cdi-0.12.0 | Provide CDI 1.2 support
pax-cdi-weld                 | 0.12.0                       |
| Uninstalled | org.ops4j.pax.cdi-0.12.0 | Weld CDI support
pax-cdi-1.1-weld             | 0.12.0                       |
| Uninstalled | org.ops4j.pax.cdi-0.12.0 | Weld CDI 1.1 support
pax-cdi-1.2-weld             | 0.12.0                       |
| Uninstalled | org.ops4j.pax.cdi-0.12.0 | Weld CDI 1.2 support
pax-cdi-openwebbeans         | 0.12.0                       |
| Uninstalled | org.ops4j.pax.cdi-0.12.0 | OpenWebBeans CDI support
pax-cdi-web                  | 0.12.0                       |
| Uninstalled | org.ops4j.pax.cdi-0.12.0 | Web CDI support
pax-cdi-1.1-web              | 0.12.0                       |
| Uninstalled | org.ops4j.pax.cdi-0.12.0 | Web CDI 1.1 support
...
```

If you want to order the features by alphabetical name, you can use the **-o** option:

```
karaf@root()> feature:list -o
Name                         | Version                      |
Required | State      | Repository             | Description
----------------------------------------------------------------------
----------------------------------------------------------------------
```

```
----------------------------
deltaspike-core           | 1.2.1                                |
| Uninstalled | org.ops4j.pax.cdi-0.12.0 | Apache Deltaspike core support
deltaspike-data           | 1.2.1                                |
| Uninstalled | org.ops4j.pax.cdi-0.12.0 | Apache Deltaspike data support
deltaspike-jpa            | 1.2.1                                |
| Uninstalled | org.ops4j.pax.cdi-0.12.0 | Apache Deltaspike jpa support
deltaspike-partial-bean   | 1.2.1                                |
| Uninstalled | org.ops4j.pax.cdi-0.12.0 | Apache Deltaspike partial bean
support
pax-cdi                   | 0.12.0                               |
| Uninstalled | org.ops4j.pax.cdi-0.12.0 | Provide CDI support
pax-cdi-1.1               | 0.12.0                               |
| Uninstalled | org.ops4j.pax.cdi-0.12.0 | Provide CDI 1.1 support
pax-cdi-1.1-web           | 0.12.0                               |
| Uninstalled | org.ops4j.pax.cdi-0.12.0 | Web CDI 1.1 support
pax-cdi-1.1-web-weld      | 0.12.0                               |
| Uninstalled | org.ops4j.pax.cdi-0.12.0 | Weld Web CDI 1.1 support
pax-cdi-1.1-weld          | 0.12.0                               |
| Uninstalled | org.ops4j.pax.cdi-0.12.0 | Weld CDI 1.1 support
pax-cdi-1.2               | 0.12.0                               |
| Uninstalled | org.ops4j.pax.cdi-0.12.0 | Provide CDI 1.2 support
...
```

By default, the **feature:list** command displays all features, whatever their current state (installed or not installed).

Using the **-i** option displays only installed features:

```
karaf@root()> feature:list -i
Name            | Version | Required | State   | Repository     | Description
-----------------------------------------------------------------------
----------------------------------------------
aries-proxy     | 4.0.0   |          | Started | standard-4.0.0 | Aries
Proxy
aries-blueprint | 4.0.0   | x        | Started | standard-4.0.0 | Aries
Blueprint
feature         | 4.0.0   | x        | Started | standard-4.0.0 |
Features Support
shell           | 4.0.0   | x        | Started | standard-4.0.0 | Karaf
Shell
shell-compat    | 4.0.0   | x        | Started | standard-4.0.0 | Karaf
Shell Compatibility
deployer        | 4.0.0   | x        | Started | standard-4.0.0 | Karaf
Deployer
bundle          | 4.0.0   | x        | Started | standard-4.0.0 | Provide
Bundle support
config          | 4.0.0   | x        | Started | standard-4.0.0 | Provide
OSGi ConfigAdmin support
diagnostic      | 4.0.0   | x        | Started | standard-4.0.0 | Provide
Diagnostic support
instance        | 4.0.0   | x        | Started | standard-4.0.0 | Provide
Instance support
jaas            | 4.0.0   | x        | Started | standard-4.0.0 | Provide
JAAS support
```

```
log               | 4.0.0   | x         | Started | standard-4.0.0 | Provide
Log support
package           | 4.0.0   | x         | Started | standard-4.0.0 | Package
commands and mbeans
service           | 4.0.0   | x         | Started | standard-4.0.0 | Provide
Service support
system            | 4.0.0   | x         | Started | standard-4.0.0 | Provide
System support
kar               | 4.0.0   | x         | Started | standard-4.0.0 | Provide
KAR (KARaf archive) support
ssh               | 4.0.0   | x         | Started | standard-4.0.0 | Provide
a SSHd server on Karaf
management        | 4.0.0   | x         | Started | standard-4.0.0 | Provide
a JMX MBeanServer and a set of MBeans in
wrap              | 0.0.0   | x         | Started | standard-4.0.0 | Wrap
URL handler
```

14.12.6. `feature:install`

The **feature:install** command installs a feature.

It requires the **feature** argument. The **feature** argument is the name of the feature, or the name/version of the feature. If only the name of the feature is provided (not the version), the latest version available will be installed.

```
karaf@root()> feature:install eventadmin
```

We can simulate an installation using **-t** or **--simulate** option: it just displays what it would do, but it doesn't do it:

```
karaf@root()> feature:install -t -v eventadmin
Adding features: eventadmin/[4.0.0,4.0.0]
No deployment change.
  Managing bundle:
    org.apache.felix.metatype / 1.0.12
```

You can specify a feature version to install:

```
karaf@root()> feature:install eventadmin/4.0.0
```

By default, the **feature:install** command is not verbose. If you want to have some details about actions performed by the **feature:install** command, you can use the **-v** option:

```
karaf@root()> feature:install -v eventadmin
Adding features: eventadmin/[4.0.0,4.0.0]
No deployment change.
Done.
```

If a feature contains a bundle which is already installed, by default, Apache Karaf will refresh this bundle. Sometime, this refresh can cause issue to other running applications. If you want to disable the auto-refresh of installed bundles, you can use the **-r** option:

```
karaf@root()> feature:install -v -r eventadmin
```

```
Adding features: eventadmin/[4.0.0,4.0.0]
No deployment change.
Done.
```

You can decide to not start the bundles installed by a feature using the **-s** or **--no-auto-start** option:

```
karaf@root()> feature:install -s eventadmin
```

14.12.7. `feature:start`

By default, when you install a feature, it's automatically started. However, you can specify the **-s** option to the **feature:install** command so that is it not started on installation.

As soon as you install a feature (started or not), all packages provided by the bundles defined in the feature will be available, and can be used for the wiring in other bundles.

When starting a feature, all bundles are started, and so, the feature also exposes the services.

14.12.8. `feature:stop`

You can also stop a feature: it means that all services provided by the feature will be stop and removed from the service registry. However, the packages are still available for the wiring (the bundles are in resolved state).

14.12.9. `feature:uninstall`

The **feature:uninstall** command uninstalls a feature. As the **feature:install** command, the **feature:uninstall** command requires the **feature** argument. The **feature** argument is the name of the feature, or the name/version of the feature. If only the name of the feature is provided (not the version), the latest version available will be installed.

```
karaf@root()> feature:uninstall eventadmin
```

The features resolver is involved during feature uninstallation: transitive features installed by the uninstalled feature can be uninstalled themselves if not used by other feature.

14.13. DEPLOYER

You can **hot deploy** a feature XML by dropping the file directly in the **deploy** folder. See Section 4.1, "Hot Deployment". When you drop a features XML in the deploy folder, the features deployer does: * register the features XML as a features repository * the features with **install** attribute set to "auto" will be automatically installed by the features deployer.

For instance, dropping the following XML in the deploy folder will automatically install feature1 and feature2, whereas feature3 won't be installed:

```
<?xml version="1.0" encoding="UTF-8"?>
<features name="my-features"
xmlns="http://karaf.apache.org/xmlns/features/v1.3.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://karaf.apache.org/xmlns/features/v1.3.0
http://karaf.apache.org/xmlns/features/v1.3.0">
```

```
        <feature name="feature1" version="1.0" install="auto">
            ...
        </feature>

        <feature name="feature2" version="1.0" install="auto">
            ...
        </feature>

        <feature name="feature3" version="1.0">
            ...
        </feature>

    </features>
```

14.14. JMX FEATUREMBEAN

On the JMX layer, you have a MBean dedicated to the management of the features and features repositories: the FeatureMBean.

The FeatureMBean object name is: **org.apache.karaf:type=feature,name=***.

14.14.1. Attributes

The FeatureMBean provides two attributes:

- **Features** is a tabular data set of all features available.

- **Repositories** is a tabular data set of all registered features repositories.

The **Repositories** attribute provides the following information:

- **Name** is the name of the features repository.

- **Uri** is the URI to the features XML for this repository.

- **Features** is a tabular data set of all features (name and version) provided by this features repository.

- **Repositories** is a tabular data set of features repositories "imported" in this features repository.

The **Features** attribute provides the following information:

- **Name** is the name of the feature.

- **Version** is the version of the feature.

- **Installed** is a boolean. If true, it means that the feature is currently installed.

- **Bundles** is a tabular data set of all bundles (bundles URL) described in the feature.

- **Configurations** is a tabular data set of all configurations described in the feature.

- **Configuration Files** is a tabular data set of all configuration files described in the feature.

- **Dependencies** is a tabular data set of all dependent features described in the feature.

14.14.2. Operations

- **addRepository(url)** adds the features repository with the **url**. The **url** can be a **name** as in the **feature:repo-add** command.

- **addRepository(url, install)** adds the features repository with the **url** and automatically installs all bundles if **install** is true. The **url** can be a **name** like in the **feature:repo-add** command.

- **removeRepository(url)** removes the features repository with the **url**. The **url** can be a **name** as in the **feature:repo-remove** command.

- **installFeature(name)** installs the feature with the **name**.

- **installFeature(name, version)** installs the feature with the **name** and **version**.

- **installFeature(name, noClean, noRefresh)** installs the feature with the **name** without cleaning the bundles in case of failure, and without refreshing already installed bundles.

- **installFeature(name, version, noClean, noRefresh)** ` **installs the feature with the ** `name** and **version** without cleaning the bundles in case of failure, and without refreshing already installed bundles.

- **uninstallFeature(name)** uninstalls the feature with the **name**.

- **uninstallFeature(name, version)** uninstalls the feature with the **name** and **version**.

14.14.3. Notifications

The FeatureMBean sends two kind of notifications (on which you can subscribe and react):

- When a feature repository changes (added or removed).

- When a feature changes (installed or uninstalled).

CHAPTER 15. REMOTE

Apache Karaf supports a complete remote mechanism allowing you to remotely connect to a running Apache Karaf instance. You can also browse, download, and upload files remotely to a running Apache Karaf instance.

Apache Karaf embeds a complete SSHd server.

15.1. SSHD SERVER

When you start Apache Karaf, it enables a remote console that can be accessed over SSH.

This remote console provides all the features of the "local" console, and gives a remote user complete control over the container and services running inside of it. As the "local" console, the remote console is secured by a RBAC mechanism See Chapter 19, *Security* for details.

In addition to the remote console, Apache Karaf also provides a remote filesystem. This remote filesystem can be accessed using a SCP/SFTP client.

15.1.1. Configuration

The configuration of the SSHd server is stored in the **etc/org.apache.karaf.shell.cfg** file:

```
################################################################################
######
#
#    Licensed to the Apache Software Foundation (ASF) under one or more
#    contributor license agreements.  See the NOTICE file distributed with
#    this work for additional information regarding copyright ownership.
#    The ASF licenses this file to You under the Apache License, Version
2.0
#    (the "License"); you may not use this file except in compliance with
#    the License.  You may obtain a copy of the License at
#
#       http://www.apache.org/licenses/LICENSE-2.0
#
#    Unless required by applicable law or agreed to in writing, software
#    distributed under the License is distributed on an "AS IS" BASIS,
#    WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
implied.
#    See the License for the specific language governing permissions and
#    limitations under the License.
#
################################################################################
######

#
# These properties are used to configure Karaf's ssh shell.
#

#
# Via sshPort and sshHost you define the address you can login into Karaf.
#
sshPort = 8101
sshHost = 0.0.0.0
```

```
#
# The sshIdleTimeout defines the inactivity timeout to logout the SSH
session.
# The sshIdleTimeout is in milliseconds, and the default is set to 30
minutes.
#
sshIdleTimeout = 1800000


#
# sshRealm defines which JAAS domain to use for password authentication.
#
sshRealm = karaf


#
# The location of the hostKey file defines where the private/public key of
the server
# is located. If no file is at the defined location it will be ignored.
#
hostKey = ${karaf.etc}/host.key


#
# Role name used for SSH access authorization
# If not set, this defaults to the ${karaf.admin.role} configured in
etc/system.properties
#
# sshRole = admin


#
# Self defined key size in 1024, 2048, 3072, or 4096
# If not set, this defaults to 4096.
#
# keySize = 4096


#
# Specify host key algorithm, defaults to RSA
#
# algorithm = RSA


#
# Defines the completion mode on the Karaf shell console. The possible
values are:
# - GLOBAL: it's the same behavior as in previous Karaf releases. The
completion displays all commands and all aliases
#           ignoring if you are in a subshell or not.
# - FIRST: the completion displays all commands and all aliases only when
you are not in a subshell. When you are
#          in a subshell, the completion displays only the commands local
to the subshell.
# - SUBSHELL: the completion displays only the subshells on the root
level. When you are in a subshell, the completion
#             displays only the commands local to the subshell.
# This property define the default value when you use the Karaf shell
console.
# You can change the completion mode directly in the shell console, using
```

```
shell:completion command.
#
completionMode = GLOBAL
```

The **etc/org.apache.karaf.shell.cfg** configuration file contains different properties to configure the SSHd server:

- **sshPort** is the port number where the SSHd server is bound (by default, it's 8101).

- **sshHost** is the address of the network interface where the SSHd server is bound. The default value is 0.0.0.0, meaning that the SSHd server is bound on all network interfaces. You can bind on a target interface providing the IP address of the network interface.

- **hostKey** is the location of the **host.key** file. By defaut, it uses **etc/host.key**. This file stores the public and private key pair of the SSHd server.

- **sshRole** is the default role used for SSH access. The default value is the value of **karaf.admin.role** property defined in **etc/system.properties**. See the [Security section|security] of this user guide for details.

- **keySize** is the key size used by the SSHd server. The possible values are 1024, 2048, 3072, or 4096. The default value is 1024.

- **algorithm** is the host key algorithm used by the SSHd server. The possible values are DSA or RSA. The default value is DSA.

The SSHd server configuration can be changed at runtime:

- by editing the **etc/org.apache.karaf.shell.cfg** configuration file

- by using the **config:*** commands

At runtime, when you change the SSHd server configuration, you have to restart the SSHd server to load the changes. You can do it with:

```
karaf@root()> bundle:restart -f org.apache.karaf.shell.ssh
```

The Apache Karaf SSHd server supports key/agent authentication and password authentication.

15.1.2. Console clients

15.1.2.1. System native clients

The Apache Karaf SSHd server is a pure SSHd server, similar to OpenSSH daemon.

It means that you can use directly a SSH client from your system.

For instance, on Unix, you can directly use OpenSSH:

```
~$ ssh -p 8101 karaf@localhost
Authenticated with partial success.
Authenticated with partial success.
Authenticated with partial success.
Password authentication
Password:
```

```
Karaf started in 7s. Bundle stats: 239 active, 239 total


    ____                       _               _         _____
   |  _ \ ___   ___| | | | | __ _| |_  | ___|   _ ___ ___
   | |_) / _ \ / _` | | |_| |/ _` | __| | |_ | | | / __|/ _ \
   |  _ <  __/ (_| | |  _  | (_| | |_  |  _|| |_| \__ \  __/
   |_| \_\___|\__,_| |_| |_|\__,_|\__| |_|   \__,_|___/___|


     Red Hat Fuse (7.0.0.fuse-000191-redhat-1)
     http://www.redhat.com/products/jbossenterprisemiddleware/fuse/



Hit '<tab>' for a list of available commands
and '[cmd] --help' for help on a specific command.
Hit 'system:shutdown' to shutdown Karaf.
Hit '<ctrl-d>' or type 'logout' to disconnect shell from current session.

karaf@root()>
```

On Windows, you can use Putty, Kitty, etc.

If you don't have SSH client installed on your machine, you can use Apache Karaf client.

15.1.2.2. `ssh:ssh` command

Apache Karaf itself provides a SSH client. When you are on the Apache Karaf console, you have the **`ssh:ssh`** command:

```
karaf@root()> ssh:ssh --help
DESCRIPTION
        ssh:ssh

        Connects to a remote SSH server

SYNTAX
        ssh:ssh [options] hostname [command]

ARGUMENTS
        hostname
                The host name to connect to via SSH
        command
                Optional command to execute

OPTIONS
        --help
                Display this help message
        -p, --port
                The port to use for SSH connection
                (defaults to 22)
        -P, --password
                The password for remote login
        -q
                Quiet Mode. Do not ask for confirmations
        -l, --username
                The user name for remote login
```

Thanks to the **ssh:ssh** command, you can connect to another running Apache Karaf instance:

```
karaf@root()> ssh:ssh -p 8101 karaf@192.168.134.2
Connecting to host 192.168.134.2 on port 8101
Connecting to unknown server. Add this server to known hosts ? (y/n)
Storing the server key in known_hosts.
Connected
Karaf started in 7s. Bundle stats: 239 active, 239 total


  ____                          _          _         _      _____
 |  _ \ ___    ___| |_  | | | | __  _| |_   |  __|    _ __   __
 | |_) / _ \/ _` | | | |_| |/ _` | __| | |_ | | | / __|/ _ \
 |  _ <  __/ (_| | | |  _  | (_| | |_  | |_|| |_| \__ \  __/
 |_| \_\___|\__,_| |_| |_| |_|\__,_|\__| |_|   \__,_|___/___|


   Red Hat Fuse (7.0.0.fuse-000191-redhat-1)
   http://www.redhat.com/products/jbossenterprisemiddleware/fuse/


Hit '<tab>' for a list of available commands
and '[cmd] --help' for help on a specific command.
Hit 'system:shutdown' to shutdown Karaf.
Hit '<ctrl-d>' or type 'logout' to disconnect shell from current session.

karaf@root()>
```

When you don't provide the **command** argument to the **ssh:ssh** command, you are in the interactive mode: you have a complete remote console available, where you can type commands, etc.

You can also provide directly a command to execute using the **command** argument. For instance, to remotely shutdown a Apache Karaf instance:

```
karaf@root()> ssh:ssh -p 8101 karaf@localhost system:shutdown -f
Connecting to host localhost on port 8101
Connected
```

As the **ssh:ssh** command is a pure SSH client, so it means that you can connect to a Unix OpenSSH daemon:

```
karaf@root()> ssh:ssh user@localhost
Connecting to host localhost on port 22
Connecting to unknown server. Add this server to known hosts ? (y/n)
Storing the server key in known_hosts.
Agent authentication failed, falling back to password authentication.
Password: Connected
Last login: Sun Sep  8 19:21:12 2013
user@server:~$
```

15.1.2.3. Apache Karaf client

The **ssh:ssh** command requires to be run into a running Apache Karaf console.

For commodity, the **ssh:ssh** command is "wrapped" as a standalone client: the **bin/client** Unix script (**bin\client.bat** on Windows).

```
bin/client --help
{karaf} client
  -a [port]     specify the port to connect to
  -h [host]     specify the host to connect to
  -u [user]     specify the user name
  --help        shows this help message
  -v            raise verbosity
  -r [attempts] retry connection establishment (up to attempts times)
  -d [delay]    intra-retry delay (defaults to 2 seconds)
  -b            batch mode, specify multiple commands via standard input
  -f [file]     read commands from the specified file
  [commands]    commands to run
If no commands are specified, the client will be put in an interactive
mode
```

For instance, to connect to local Apache Karaf instance (on the default SSHd server 8101 port), you can directly use **bin/client** Unix script (**bin\client.bat** on Windows) without any argument or option:

```
bin/client
Logging in as karaf
343 [pool-2-thread-4] WARN
org.apache.sshd.client.keyverifier.AcceptAllServerKeyVerifier - Server at
/0.0.0.0:8101 presented unverified key:
Karaf started in 7s. Bundle stats: 239 active, 239 total


  ____                         _        _          _        _____
 |  _ \ ___  __| | | | | | | __ _| |_   | |___|    _ ___ ___
 | |_) / _ \/ _` | | | |_| |/ _` | __|  | |_ | | | / __|/ _ \
 |  _ < __/ (_| | | |  _  | (_| | |_    |  _|| |_| \__ \  __/
 |_| \_\___|\__,_| |_| |_|\__,_|\__|  |_|    \__,_|___/___|


   Red Hat Fuse (7.0.0.fuse-000191-redhat-1)
   http://www.redhat.com/products/jbossenterprisemiddleware/fuse/



Hit '<tab>' for a list of available commands
and '[cmd] --help' for help on a specific command.
Hit 'system:shutdown' to shutdown Karaf.
Hit '<ctrl-d>' or type 'logout' to disconnect shell from current session.

karaf@root()>
```

When you don't provide the **command** argument to the **bin/client** Unix script (**bin\client.bat** on Windows), you are in the interactive mode: you have a complete remote console available, where you can type commands, etc.

You can also provide directly a command to execute using the **command** argument. For instance, to remotely shutdown a Apache Karaf instance:

```
bin/client "system:shutdown -f"
Logging in as karaf
330 [pool-2-thread-3] WARN
org.apache.sshd.client.keyverifier.AcceptAllServerKeyVerifier - Server at
/0.0.0.0:8101 presented unverified key:
```

As the Apache Karaf client is a pure SSH client, you can use to connect to any SSHd daemon (like Unix OpenSSH daemon):

```
bin/client -a 22 -h localhost -u user
Logging in as user
353 [pool-2-thread-2] WARN
org.apache.sshd.client.keyverifier.AcceptAllServerKeyVerifier - Server at
localhost/127.0.0.1:22 presented unverified key:
Password:
Welcome to Ubuntu 13.10 (GNU/Linux 3.11.0-13-generic x86_64)

  * Documentation:  https://help.ubuntu.com/

Last login: Tue Dec  3 18:18:31 2013 from localhost
```

15.1.2.4. Logout

When you are connected to a remote Apache Karaf console, you can logout using:

- using CTRL-D key binding. Note that CTRL-D just logout from the remote console in this case, it doesn't shutdown the Apache Karaf instance (as CTRL-D does when used on a local console).

- using **shell:logout** command (or simply **logout**)

15.1.3. Filsystem clients

Apache Karaf SSHd server also provides complete fileysystem access via SSH. For security reason, the available filesystem is limited to **KARAF_BASE** directory.

You can use this remote filesystem with any SCP/SFTP compliant clients.

15.1.3.1. Native SCP/SFTP clients

On Unix, you can directly use **scp** command to download/upload files to the Apache Karaf filesystem. For instance, to retrieve the **fuse.log** file remotely:

```
~$ scp -P 8101 karaf@localhost:/data/log/karaf.log .
Authenticated with partial success.
Authenticated with partial success.
Authenticated with partial success.
Password admin
Password:
fuse.log
```

As you have access to the complete **KARAF_BASE** directory, you can remotely change the configuration file in the **etc** folder, retrieve log files, populate the **system** folder.

On Windows, you can use WinSCP to access the Apache Karaf filesystem.

It's probably easier to use a SFTP complient client.

For instance, on Unix system, you can use **lftp** or **ncftp**:

```
$ lftp
```

```
lftp :~> open -u karaf sftp://localhost:8101
Password:
lftp karaf@localhost:~> ls
-rw-r--r--    1 jbonofre jbonofre     27754 Oct 26 10:50 LICENSE
-rw-r--r--    1 jbonofre jbonofre      1919 Dec  3 05:34 NOTICE
-rw-r--r--    1 jbonofre jbonofre      3933 Aug 18  2012 README
-rw-r--r--    1 jbonofre jbonofre    101041 Dec  3 05:34 RELEASE-NOTES
drwxr-xr-x    1 jbonofre jbonofre      4096 Dec  3 12:51 bin
drwxr-xr-x    1 jbonofre jbonofre      4096 Dec  3 18:57 data
drwxr-xr-x    1 jbonofre jbonofre      4096 Dec  3 12:51 demos
drwxr-xr-x    1 jbonofre jbonofre      4096 Dec  3 13:02 deploy
drwxr-xr-x    1 jbonofre jbonofre      4096 Dec  3 17:59 etc
drwxr-xr-x    1 jbonofre jbonofre      4096 Dec  3 13:02 instances
drwxr-xr-x    1 jbonofre jbonofre      4096 Dec  3 13:02 lib
-rw-r--r--    1 jbonofre jbonofre         0 Dec  3 13:02 lock
drwxr-xr-x    1 jbonofre jbonofre      4096 Dec  3 12:51 system
lftp karaf@localhost:/>
```

You can also use graphic client like **filezilla**, **gftp**, **nautilus**, etc.

On Windows, you can use **filezilla**, **WinSCP**, etc.

15.1.3.2. Apache Maven

The Apache Karaf **system** folder is the Karaf repository, that use a Maven directory structure. It's where Apache Karaf looks for the artifacts (bundles, features, kars, etc).

Using Apache Maven, you can populate the **system** folder using the **deploy:deploy-file** goal.

For instance, you want to add the Apache ServiceMix facebook4j OSGi bundle, you can do:

```
mvn deploy:deploy-file -Dfile=org.apache.servicemix.bundles.facebook4j-
2.0.2_1.jar -DgroupId=org.apache.servicemix.bundles -
DartifactId=org.apache.servicemix.bundles.facebook4j -Dversion=2.0.2_1 -
Dpackaging=jar -Durl=scp://localhost:8101/system
```

15.2. JMX MBEANSERVER

Apache Karaf provides a JMX MBeanServer.

This MBeanServer is available remotely, using any JMX client like **jconsole**.

You can find details on the [Monitoring section|monitoring] of the user guide.

CHAPTER 16. 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 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 Fuse.

16.1. MAVEN DIRECTORY STRUCTURE

16.1.1. 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**.

16.1.2. Standard directory layout

Example 16.1, "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 Blueprint configuration files (which are **not** defined by Maven) are also shown.

Example 16.1. Standard Maven Directory Layout

```
ProjectDir/
    pom.xml
    src/
        main/
            java/
                ...
            resources/
                META-INF/

                OSGI-INF/
                    blueprint/
                        *.xml
        test/
            java/
            resources/
    target/
        ...
```
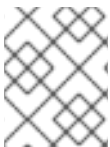
NOTE

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

16.1.3. 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
```

16.1.4. 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.

16.1.5. 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.

16.1.6. 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.

16.1.7. 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.

16.1.8. Blueprint container

OSGi R4.2 defines a Blueprint container. Red Hat 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 10, *OSGi Services*.

CHAPTER 17. MAVEN INDEXER PLUGIN

The Maven Indexer Plugin is required for the Maven plugin to enable it to quickly search Maven Central for artifacts.

To Deploy the Maven Indexer plugin use the following commands:

Prerequisites

Before deploying the Maven Indexer Plugin, make sure that you have followed the instructions in the Installing on Apache Karaf Preparing to Use Maven section.

Deploy the Maven Indexer Plugin

1. Go to the Karaf console and enter the following command to install the Maven Indexer plugin:

   ```
   features:install hawtio-maven-indexer
   ```

2. Enter the following commands to configure the Maven Indexer plugin:

   ```
   config:edit io.hawt.maven.indexer
   config:proplist
   config:propset repositories 'https://maven.oracle.com'
   config:proplist
   config:update
   ```

3. Wait for the Maven Indexer plugin to be deployed. This may take a few minutes. Look out for messages like those shown below to appear on the log tab.

❶ INFO	org.apache.felix.fileinstall	Creating configuration from io.hawt.maven.indexer.cfg
❶ INFO	io.fabric8.internal.ProfileServiceImpl	updateProfile: Profile[ver=1.0,id=fabric,atts={parents=karaf hawtio}]
❶ INFO	io.fabric8.internal.ProfileServiceImpl	updateProfile: Profile[ver=1.0,id=fabric,atts={parents=karaf hawtio}]

When the Maven Indexer plugin has been deployed, use the following commands to add further external Maven repositories to the Maven Indexer plugin configuration:

```
config:edit io.hawt.maven.indexer
config:proplist
config:propset repositories external repository
config:proplist
config:update
```

CHAPTER 18. LOG

Apache Karaf provides a very dynamic and powerful logging system.

It supports:

- the OSGi Log Service

- the Apache Log4j v1 and v2 framework

- the Apache Commons Logging framework

- the Logback framework

- the SLF4J framework

- the native Java Util Logging framework

It means that the applications can use any logging framework, Apache Karaf will use the central log system to manage the loggers, appenders, etc.

18.1. CONFIGURATION FILES

The initial log configuration is loaded from **etc/org.ops4j.pax.logging.cfg**.

This file is a standard Log4j configuration file.

You find the different Log4j element:

- loggers

- appenders

- layouts

You can add your own initial configuration directly in the file.

The default configuration is the following:

```
################################################################################
######
#
#    Licensed to the Apache Software Foundation (ASF) under one or more
#    contributor license agreements.  See the NOTICE file distributed with
#    this work for additional information regarding copyright ownership.
#    The ASF licenses this file to You under the Apache License, Version
2.0
#    (the "License"); you may not use this file except in compliance with
#    the License.  You may obtain a copy of the License at
#
#       http://www.apache.org/licenses/LICENSE-2.0
#
#    Unless required by applicable law or agreed to in writing, software
#    distributed under the License is distributed on an "AS IS" BASIS,
#    WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
implied.
```

```
#     See the License for the specific language governing permissions and
#     limitations under the License.
#
################################################################################
######

# Root logger
log4j.rootLogger=INFO, out, osgi:*
log4j.throwableRenderer=org.apache.log4j.OsgiThrowableRenderer

# CONSOLE appender not used by default
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{ISO8601} | %-5.5p | %-
16.16t | %-32.32c{1} | %X{bundle.id} - %X{bundle.name} -
%X{bundle.version} | %m%n

# File appender
log4j.appender.out=org.apache.log4j.RollingFileAppender
log4j.appender.out.layout=org.apache.log4j.PatternLayout
log4j.appender.out.layout.ConversionPattern=%d{ISO8601} | %-5.5p | %-
16.16t | %-32.32c{1} | %X{bundle.id} - %X{bundle.name} -
%X{bundle.version} | %m%n
log4j.appender.out.file=${karaf.data}/log/karaf.log
log4j.appender.out.append=true
log4j.appender.out.maxFileSize=1MB
log4j.appender.out.maxBackupIndex=10

# Sift appender
log4j.appender.sift=org.apache.log4j.sift.MDCSiftingAppender
log4j.appender.sift.key=bundle.name
log4j.appender.sift.default=karaf
log4j.appender.sift.appender=org.apache.log4j.FileAppender
log4j.appender.sift.appender.layout=org.apache.log4j.PatternLayout
log4j.appender.sift.appender.layout.ConversionPattern=%d{ISO8601} | %-5.5p
| %-16.16t | %-32.32c{1} | %m%n
log4j.appender.sift.appender.file=${karaf.data}/log/$\\{bundle.name\\}.log
log4j.appender.sift.appender.append=true
```

The default configuration only define the **ROOT** logger, with **INFO** log level, using the **out** file appender. You can change the log level to any Log4j valid values (from the most to less verbose): TRACE, DEBUG, INFO, WARN, ERROR, FATAL.

The **osgi:*** appender is a special appender to send the log message to the OSGi Log Service.

A **stdout** console appender is pre-configured, but not enabled by default. This appender allows you to display log messages directly to standard output. It's interesting if you plan to run Apache Karaf in server mode (without console).

To enable it, you have to add the **stdout** appender to the **rootLogger**:

```
log4j.rootLogger=INFO, out, stdout, osgi:*
```

The **out** appender is the default one. It's rolling file appender that maintain and rotate 10 log files of 1MB each. The log files are located in **data/log/karaf.log** by default.

The **sift** appender is not enabled by default. This appender allows you to have one log file per deployed bundle. By default, the log file name format uses the bundle symbolic name (in the **data/log** folder).

You can edit this file at runtime: any change will be reloaded and be effective immediately (no need to restart Apache Karaf).

Another configuration file is used by Apache Karaf: **etc/org.apache.karaf.log.cfg**. This files configures the Log Service used by the log commands (see later).

18.2. LOG4J V2 SUPPORT

Karaf supports log4j v2 backend.

To enable log4j v2 support you have to:

1. Edit **etc/startup.properties** to replace the line **org/ops4j/pax/logging/pax-logging-service/1.8.4/pax-logging-service-1.8.4.jar=8** with **org/ops4j/pax/logging/pax-logging-log4j2/1.8.4/pax-logging-log4j2-1.8.4.jar=8**

2. Add pax-logging-log4j2 jar file in **system/org/ops4j/pax/logging/pax-logging-log4j2/x.x/pax-logging-log4j2-x.x.jar where x.x is the version as defined in `etc/startup.properties`**

3. Edit **etc/org.ops4j.pax.logging.cfg** configuration file and add **org.ops4j.pax.logging.log4j2.config.file=${karaf.etc}/log4j2.xml**

4. Add the **etc/log4j2.xml** configuration file.

A default configuration in **etc/log4j2.xml** could be:

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="INFO">
    <Appenders>
        <Console name="console" target="SYSTEM_OUT">
            <PatternLayout pattern="%d{ABSOLUTE} | %-5.5p | %-16.16t | %-32.32c{1} | %X{bundle.id} - %X{bundle.name} - %X{bundle.version} | %m%n"/>
        </Console>
        <RollingFile name="out" fileName="${karaf.data}/log/karaf.log"
                append="true" filePattern="${karaf.data}/log/$${date:yyyy-MM}/fuse-%d{MM-dd-yyyy}-%i.log.gz">
            <PatternLayout>
              <Pattern>%d{ABSOLUTE} | %-5.5p | %-16.16t | %-32.32c{1} | %X{bundle.id} - %X{bundle.name} - %X{bundle.version} | %m%n</Pattern>
            </PatternLayout>
            <Policies>
                <TimeBasedTriggeringPolicy />
                <SizeBasedTriggeringPolicy size="250 MB"/>
            </Policies>
        </RollingFile>
        <PaxOsgi name="paxosgi" filter="VmLogAppender"/>
    </Appenders>
    <Loggers>
        <Root level="INFO">
```

```
                <AppenderRef ref="console"/>
                <AppenderRef ref="out"/>
                <AppenderRef ref="paxosgi"/>
        </Root>
    </Loggers>
</Configuration>
```

18.3. COMMANDS

Instead of changing the **etc/org.ops4j.pax.logging.cfg** file, Apache Karaf provides a set of commands allowing to dynamically change the log configuration and see the log content:

18.3.1. `log:clear`

The **log:clear** command clears the log entries.

18.3.2. `log:display`

The **log:display** command displays the log entries.

By default, it displays the log entries of the **rootLogger**:

```
karaf@root()> log:display
2015-07-01 19:12:46,208 | INFO  | FelixStartLevel | SecurityUtils
| 16 - org.apache.sshd.core - 0.12.0 | BouncyCastle not registered, using
the default JCE provider
2015-07-01 19:12:47,368 | INFO  | FelixStartLevel | core
| 68 - org.apache.aries.jmx.core - 1.1.1 | Starting JMX OSGi agent
```

You can also display the log entries from a specific logger, using the **logger** argument:

```
karaf@root()> log:display ssh
2015-07-01 19:12:46,208 | INFO  | FelixStartLevel | SecurityUtils
| 16 - org.apache.sshd.core - 0.12.0 | BouncyCastle not registered, using
the default JCE provider
```

By default, all log entries will be displayed. It could be very long if your Apache Karaf container is running since a long time. You can limit the number of entries to display using the **-n** option:

```
karaf@root()> log:display -n 5
2015-07-01 06:53:24,143 | INFO  | JMX OSGi Agent   | core
| 68 - org.apache.aries.jmx.core - 1.1.1 | Registering
org.osgi.jmx.framework.BundleStateMBean to MBeanServer
com.sun.jmx.mbeanserver.JmxMBeanServer@27cc75cb with name
osgi.core:type=bundleState,version=1.7,framework=org.apache.felix.framewor
k,uuid=5335370f-9dee-449f-9b1c-cabe74432ed1
2015-07-01 06:53:24,150 | INFO  | JMX OSGi Agent   | core
| 68 - org.apache.aries.jmx.core - 1.1.1 | Registering
org.osgi.jmx.framework.PackageStateMBean to MBeanServer
com.sun.jmx.mbeanserver.JmxMBeanServer@27cc75cb with name
osgi.core:type=packageState,version=1.5,framework=org.apache.felix.framewo
rk,uuid=5335370f-9dee-449f-9b1c-cabe74432ed1
2015-07-01 06:53:24,150 | INFO  | JMX OSGi Agent   | core
```

```
| 68 - org.apache.aries.jmx.core - 1.1.1 | Registering
org.osgi.jmx.framework.ServiceStateMBean to MBeanServer
com.sun.jmx.mbeanserver.JmxMBeanServer@27cc75cb with name
osgi.core:type=serviceState,version=1.7,framework=org.apache.felix.framewo
rk,uuid=5335370f-9dee-449f-9b1c-cabe74432ed1
2015-07-01 06:53:24,152 | INFO  | JMX OSGi Agent   | core
| 68 - org.apache.aries.jmx.core - 1.1.1 | Registering
org.osgi.jmx.framework.wiring.BundleWiringStateMBean to MBeanServer
com.sun.jmx.mbeanserver.JmxMBeanServer@27cc75cb with name
osgi.core:type=wiringState,version=1.1,framework=org.apache.felix.framewor
k,uuid=5335370f-9dee-449f-9b1c-cabe74432ed1
2015-07-01 06:53:24,501 | INFO  | FelixStartLevel   |
RegionsPersistenceImpl            | 78 - org.apache.karaf.region.persist -
4.0.0 | Loading region digraph persistence
```

You can also limit the number of entries stored and retain using the **size** property in
etc/org.apache.karaf.log.cfg file:

```
#
# The number of log statements to be displayed using log:display. It also
defines the number
# of lines searched for exceptions using log:display exception. You can
override this value
# at runtime using -n in log:display.
#
size = 500
```

By default, each log level is displayed with a different color: ERROR/FATAL are in red, DEBUG in
purple, INFO in cyan, etc. You can disable the coloring using the **--no-color** option.

The log entries format pattern doesn't use the conversion pattern define in
etc/org.ops4j.pax.logging.cfg file. By default, it uses the **pattern** property defined in
etc/org.apache.karaf.log.cfg.

```
#
# The pattern used to format the log statement when using log:display.
This pattern is according
# to the log4j layout. You can override this parameter at runtime using
log:display with -p.
#
pattern = %d{ISO8601} | %-5.5p | %-16.16t | %-32.32c{1} | %X{bundle.id} -
%X{bundle.name} - %X{bundle.version} | %m%n
```

You can also change the pattern dynamically (for one execution) using the **-p** option:

```
karaf@root()> log:display -p "%d - %c - %m%n"
2015-07-01 07:01:58,007 - org.apache.sshd.common.util.SecurityUtils -
BouncyCastle not registered, using the default JCE provider
2015-07-01 07:01:58,725 - org.apache.aries.jmx.core - Starting JMX OSGi
agent
2015-07-01 07:01:58,744 - org.apache.aries.jmx.core - Registering MBean
with ObjectName
[osgi.compendium:service=cm,version=1.3,framework=org.apache.felix.framewo
rk,uuid=6361fc65-8df4-4886-b0a6-479df2d61c83] for service with service.id
[13]
```

```
2015-07-01 07:01:58,747 - org.apache.aries.jmx.core - Registering
org.osgi.jmx.service.cm.ConfigurationAdminMBean to MBeanServer
com.sun.jmx.mbeanserver.JmxMBeanServer@27cc75cb with name
osgi.compendium:service=cm,version=1.3,framework=org.apache.felix.framewor
k,uuid=6361fc65-8df4-4886-b0a6-479df2d61c83
```

The pattern is a regular Log4j pattern where you can use keywords like %d for the date, %c for the class, %m for the log message, etc.

18.3.3. `log:exception-display`

The `log:exception-display` command displays the last occurred exception.

As for `log:display` command, the `log:exception-display` command uses the **rootLogger** by default, but you can specify a logger with the **logger** argument.

18.3.4. `log:get`

The `log:get` command show the current log level of a logger.

By default, the log level showed is the one from the root logger:

```
karaf@root()> log:get
Logger | Level
--------------
ROOT   | INFO
```

You can specify a particular logger using the **logger** argument:

```
karaf@root()> log:get ssh
Logger | Level
--------------
ssh    | INFO
```

The **logger** argument accepts the **ALL** keyword to display the log level of all logger (as a list).

For instance, if you have defined your own logger in **etc/org.ops4j.pax.logging.cfg** file like this:

```
log4j.logger.my.logger = DEBUG
```

you can see the list of loggers with the corresponding log level:

```
karaf@root()> log:get ALL
Logger    | Level
-----------------
ROOT      | INFO
my.logger | DEBUG
```

The `log:list` command is an alias to `log:get ALL`.

18.3.5. `log:log`

The `log:log` command allows you to manually add a message in the log. It's interesting when you

create Apache Karaf scripts:

```
karaf@root()> log:log "Hello World"
karaf@root()> log:display
2015-07-01 07:20:16,544 | INFO  | Local user karaf | command
| 59 - org.apache.karaf.log.command - 4.0.0 | Hello World
```

By default, the log level is INFO, but you can specify a different log level using the **-l** option:

```
karaf@root()> log:log -l ERROR "Hello World"
karaf@root()> log:display
2015-07-01 07:21:38,902 | ERROR | Local user karaf | command
| 59 - org.apache.karaf.log.command - 4.0.0 | Hello World
```

18.3.6. `log:set`

The **log:set** command sets the log level of a logger.

By default, it changes the log level of the **rootLogger**:

```
karaf@root()> log:set DEBUG
karaf@root()> log:get
Logger | Level
--------------
ROOT   | DEBUG
```

You can specify a particular logger using the **logger** argument, after the **level** one:

```
karaf@root()> log:set INFO my.logger
karaf@root()> log:get my.logger
Logger     | Level
-----------------
my.logger | INFO
```

The **level** argument accepts any Log4j log level: TRACE, DEBUG, INFO, WARN, ERROR, FATAL.

By it also accepts the DEFAULT special keyword.

The purpose of the DEFAULT keyword is to delete the current level of the logger (and only the level, the other properties like appender are not deleted) in order to use the level of the logger parent (logger are hierarchical).

For instance, you have defined the following loggers (in **etc/org.ops4j.pax.logging.cfg** file):

```
rootLogger=INFO,out,osgi:*
my.logger=INFO,appender1
my.logger.custom=DEBUG,appender2
```

You can change the level of **my.logger.custom** logger:

```
karaf@root()> log:set INFO my.logger.custom
```

Now we have:

```
rootLogger=INFO,out,osgi:*
my.logger=INFO,appender1
my.logger.custom=INFO,appender2
```

You can use the DEFAULT keyword on **my.logger.custom** logger to remove the level:

```
karaf@root()> log:set DEFAULT my.logger.custom
```

Now we have:

```
rootLogger=INFO,out,osgi:*
my.logger=INFO,appender1
my.logger.custom=appender2
```

It means that, at runtime, the **my.logger.custom** logger uses the level of its parent **my.logger**, so **INFO**.

Now, if we use DEFAULT keyword with the **my.logger** logger:

```
karaf@root()> log:set DEFAULT my.logger
```

We have:

```
rootLogger=INFO,out,osgi:*
my.logger=appender1
my.logger.custom=appender2
```

So, both **my.logger.custom** and **my.logger** use the log level of the parent **rootLogger**.

It's not possible to use DEFAULT keyword with the **rootLogger** and it doesn't have parent.

18.3.7. `log:tail`

The **log:tail** is exactly the same as **log:display** but it continuously displays the log entries.

You can use the same options and arguments as for the **log:display** command.

By default, it displays the entries from the **rootLogger**:

```
karaf@root()> log:tail
2015-07-01 07:40:28,152 | INFO  | FelixStartLevel | SecurityUtils
| 16 - org.apache.sshd.core - 0.9.0 | BouncyCastle not registered, using
the default JCE provider
2015-07-01 07:40:28,909 | INFO  | FelixStartLevel | core
| 68 - org.apache.aries.jmx.core - 1.1.1 | Starting JMX OSGi agent
2015-07-01 07:40:28,928 | INFO  | FelixStartLevel | core
| 68 - org.apache.aries.jmx.core - 1.1.1 | Registering MBean with
ObjectName
[osgi.compendium:service=cm,version=1.3,framework=org.apache.felix.framewo
rk,uuid=b44a44b7-41cd-498f-936d-3b12d7aafa7b] for service with service.id
[13]
2015-07-01 07:40:28,936 | INFO  | JMX OSGi Agent   | core
| 68 - org.apache.aries.jmx.core - 1.1.1 | Registering
```

```
org.osgi.jmx.service.cm.ConfigurationAdminMBean to MBeanServer
com.sun.jmx.mbeanserver.JmxMBeanServer@27cc75cb with name
osgi.compendium:service=cm,version=1.3,framework=org.apache.felix.framewor
k,uuid=b44a44b7-41cd-498f-936d-3b12d7aafa7b
```

To exit from the `log:tail` command, just type CTRL-C.

18.4. JMX LOGMBEAN

All actions that you can perform with the `log:*` command can be performed using the LogMBean.

The LogMBean object name is **org.apache.karaf:type=log,name=***.

18.4.1. Attributes

- **Level** attribute is the level of the ROOT logger.

18.4.2. Operations

- **getLevel(logger)** to get the log level of a specific logger. As this operation supports the ALL keyword, it returns a Map with the level of each logger.

- **setLevel(level, logger)** to set the log level of a specific logger. This operation supports the DEFAULT keyword as for the `log:set` command.

18.5. ADVANCED CONFIGURATION

18.5.1. Filters

You can use filters on appender. Filters allow log events to be evaluated to determine if or how they should be published.

Log4j provides ready to use filters:

- The DenyAllFilter (**org.apache.log4j.varia.DenyAllFilter**) drops all logging events. You can add this filter to the end of a filter chain to switch from the default "accept all unless instructed otherwise" filtering behaviour to a "deny all unless instructed otherwise" behaviour.

- The LevelMatchFilter (**org.apache.log4j.varia.LevelMatchFilter** is a very simple filter based on level matching. The filter admits two options **LevelToMatch** and **AcceptOnMatch**. If there is an exact match between the value of the **LevelToMatch** option and the level of the logging event, then the event is accepted in case the **AcceptOnMatch** option value is set to **true**. Else, if the **AcceptOnMatch** option value is set to **false**, the log event is rejected.

- The LevelRangeFilter (**org.apache.log4j.varia.LevelRangeFilter** is a very simple filter based on level matching, which can be used to reject messages with priorities outside a certain range. The filter admits three options **LevelMin**, **LevelMax** and **AcceptOnMatch**. If the log event level is between **LevelMin** and **LevelMax**, the log event is accepted if **AcceptOnMatch** is true, or rejected if **AcceptOnMatch** is false.

- The StringMatchFilter (**org.apache.log4j.varia.StringMatchFilter**) is a very simple filter based on string matching. The filter admits two options **StringToMatch** and

AcceptOnMatch. If there is a match between the **StringToMatch** and the log event message, the log event is accepted if **AcceptOnMatch** is true, or rejected if **AcceptOnMatch** is false.

The filter is defined directly on the appender, in the **etc/org.ops4j.pax.logging.cfg** configuration file.

The format to use it:

```
log4j.appender.[appender-name].filter.[filter-name]=[filter-class]
log4j.appender.[appender-name].filter.[filter-name].[option]=[value]
```

For instance, you can use the **f1** LevelRangeFilter on the **out** default appender:

```
log4j.appender.out.filter.f1=org.apache.log4j.varia.LevelRangeFilter
log4j.appender.out.filter.f1.LevelMax=FATAL
log4j.appender.out.filter.f1.LevelMin=DEBUG
```

Thanks to this filter, the log files generated by the **out** appender will contain only log messages with a level between DEBUG and FATAL (the log events with TRACE as level are rejected).

18.5.2. Nested appenders

A nested appender is a special kind of appender that you use "inside" another appender. It allows you to create some kind of "routing" between a chain of appenders.

The most used "nested compliant" appender are:

- The AsyncAppender (**org.apache.log4j.AsyncAppender**) logs events asynchronously. This appender collects the events and dispatch them to all the appenders that are attached to it.

- The RewriteAppender (**org.apache.log4j.rewrite.RewriteAppender**) forwards log events to another appender after possibly rewriting the log event.

This kind of appender accepts an **appenders** property in the appender definition:

```
log4j.appender.[appender-name].appenders=[comma-separated-list-of-
appender-names]
```

For instance, you can create a AsyncAppender named **async** and asynchronously dispatch the log events to a JMS appender:

```
log4j.appender.async=org.apache.log4j.AsyncAppender
log4j.appender.async.appenders=jms

log4j.appender.jms=org.apache.log4j.net.JMSAppender
...
```

18.5.3. Error handlers

Sometime, appenders can fail. For instance, a RollingFileAppender tries to write on the filesystem but the filesystem is full, or a JMS appender tries to send a message but the JMS broker is not there.

As log can be very critical to you, you have to be inform that the log appender failed.

It's the purpose of the error handlers. Appenders may delegate their error handling to error handlers, giving a chance to react to this appender errors.

You have two error handlers available:

- The OnlyOnceErrorHandler (**org.apache.log4j.helpers.OnlyOnceErrorHandler**) implements log4j's default error handling policy which consists of emitting a message for the first error in an appender and ignoring all following errors. The error message is printed on **System.err**. This policy aims at protecting an otherwise working application from being flooded with error messages when logging fails.

- The FallbackErrorHandler (**org.apache.log4j.varia.FallbackErrorHandler**) allows a secondary appender to take over if the primary appender fails. The error message is printed on **System.err**, and logged in the secondary appender.

You can define the error handler that you want to use for each appender using the **errorhandler** property on the appender definition itself:

```
log4j.appender.[appender-name].errorhandler=[error-handler-class]
log4j.appender.[appender-name].errorhandler.root-ref=[true|false]
log4j.appender.[appender-name].errorhandler.logger-ref=[logger-ref]
log4j.appender.[appender-name].errorhandler.appender-ref=[appender-ref]
```

18.5.4. OSGi specific MDC attributes

The **sift** appender is a OSGi oriented appender allowing you to split the log events based on MDC (Mapped Diagnostic Context) attributes.

MDC allows you to distinguish the different source of log events.

The **sift** appender provides OSGi oritend MDC attributes by default:

- **bundle.id** is the bundle ID

- **bundle.name** is the bundle symbolic name

- **bundle.version** is the bundle version

You can use these MDC properties to create a log file per bundle:

```
log4j.appender.sift=org.apache.log4j.sift.MDCSiftingAppender
log4j.appender.sift.key=bundle.name
log4j.appender.sift.default=karaf
log4j.appender.sift.appender=org.apache.log4j.FileAppender
log4j.appender.sift.appender.layout=org.apache.log4j.PatternLayout
log4j.appender.sift.appender.layout.ConversionPattern=%d{ABSOLUTE} | %-
5.5p | %-16.16t | %-32.32c{1} | %-32.32C %4L | %m%n
log4j.appender.sift.appender.file=${karaf.data}/log/$\\{bundle.name\\}.log
log4j.appender.sift.appender.append=true
```

18.5.5. Enhanced OSGi stack trace renderer

By default, Apache Karaf provides a special stack trace renderer, adding some OSGi specific specific information.

In the stack trace, in addition of the class throwing the exception, you can find a pattern **[id:name:version]** at the end of each stack trace line, where:

- **id** is the bundle ID

- **name** is the bundle name

- **version** is the bundle version

It's very helpful to diagnosing the source of an issue.

For instance, in the following IllegalArgumentException stack trace, we can see the OSGi details about the source of the exception:

```
java.lang.IllegalArgumentException: Command not found:  *:foo
 at org.apache.felix.gogo.runtime.shell.Closure.execute(Closure.java:225)
[21:org.apache.karaf.shell.console:4.0.0]
 at
org.apache.felix.gogo.runtime.shell.Closure.executeStatement(Closure.java:
162)[21:org.apache.karaf.shell.console:4.0.0]
 at org.apache.felix.gogo.runtime.shell.Pipe.run(Pipe.java:101)
[21:org.apache.karaf.shell.console:4.0.0]
 at org.apache.felix.gogo.runtime.shell.Closure.execute(Closure.java:79)
[21:org.apache.karaf.shell.console:4.0.0]
 at
org.apache.felix.gogo.runtime.shell.CommandSessionImpl.execute(CommandSess
ionImpl.java:71)[21:org.apache.karaf.shell.console:4.0.0]
 at org.apache.karaf.shell.console.jline.Console.run(Console.java:169)
[21:org.apache.karaf.shell.console:4.0.0]
 at java.lang.Thread.run(Thread.java:637)[:1.7.0_21]
```

18.5.6. Custom appenders

You can use your own appenders in Apache Karaf.

The easiest way to do that is to package your appender as an OSGi bundle and attach it as a fragment of the **org.ops4j.pax.logging.pax-logging-service** bundle.

For instance, you create **MyAppender**:

```
public class MyAppender extends AppenderSkeleton {
...
}
```

You compile and package as an OSGi bundle containing a MANIFEST looking like:

```
Manifest:
Bundle-SymbolicName: org.mydomain.myappender
Fragment-Host: org.ops4j.pax.logging.pax-logging-service
...
```

Copy your bundle in the Apache Karaf **system** folder. The **system** folder uses a standard Maven directory layout: groupId/artifactId/version.

In the `etc/startup.properties` configuration file, you define your bundle in the list before the pax-logging-service bundle.

You have to restart Apache Karaf with a clean run (purging the `data` folder) in order to reload the system bundles. You can now use your appender directly in `etc/org.ops4j.pax.logging.cfg` configuration file.

CHAPTER 19. SECURITY

Apache Karaf provides an advanced and flexible security system, powered by JAAS (Java Authentication and Authorization Service) in an OSGi compliant way.

It provides a dynamic security system.

The Apache Karaf security framework is used internally to control the access to:

- the OSGi services (described in the developer guide)

- the console commands

- the JMX layer

- the WebConsole

Your applications can also use the security framework (see the developer guide for details).

19.1. REALMS

Apache Karaf is able to manage multiple realms. A realm contains the definition of the login modules to use for the authentication and/or authorization on this realm. The login modules define the authentication and authorization for the realm.

The **jaas:realm-list** command list the current defined realms:

```
karaf@root()> jaas:realm-list
Index | Realm Name | Login Module Class Name
-----------------------------------------------------------------------
-------------
1     | karaf      |
org.apache.karaf.jaas.modules.properties.PropertiesLoginModule
2     | karaf      |
org.apache.karaf.jaas.modules.publickey.PublickeyLoginModule
```

You can see that the Apache Karaf provides a default realm named **karaf**.

This realm has two login modules:

- the **PropertiesLoginModule** uses the **etc/users.properties** file as backend for users, groups, roles and password. This login module authenticates the users and returns the users' roles.

- the **PublickeyLoginModule** is especially used by the SSHd. It uses the **etc/keys.properties** file. This file contains the users and a public key associated to each user.

Apache Karaf provides additional login modules (see the developer guide for details):

- JDBCLoginModule uses a database as backend

- LDAPLoginModule uses a LDAP server as backend

- SyncopeLoginModule uses Apache Syncope as backend

- OsgiConfigLoginModule uses a configuration as backend

- Krb5LoginModule uses a Kerberos Server as backend

- GSSAPILdapLoginModule uses a LDAP server as backend but delegate LDAP server authentication to an other backend (typically Krb5LoginModule)

You can manage an existing realm, login module, or create your own realm using the **jaas:realm-manage** command.

19.1.1. Users, groups, roles, and passwords

As we saw, by default, Apache Karaf uses a PropertiesLoginModule.

This login module uses the **etc/users.properties** file as storage for the users, groups, roles and passwords.

The initial **etc/users.properties** file contains:

```
################################################################################
######
#
#    Licensed to the Apache Software Foundation (ASF) under one or more
#    contributor license agreements.  See the NOTICE file distributed with
#    this work for additional information regarding copyright ownership.
#    The ASF licenses this file to You under the Apache License, Version
2.0
#    (the "License"); you may not use this file except in compliance with
#    the License.  You may obtain a copy of the License at
#
#       http://www.apache.org/licenses/LICENSE-2.0
#
#    Unless required by applicable law or agreed to in writing, software
#    distributed under the License is distributed on an "AS IS" BASIS,
#    WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
implied.
#    See the License for the specific language governing permissions and
#    limitations under the License.
#
################################################################################
######

#
# This file contains the users, groups, and roles.
# Each line has to be of the format:
#
# USER=PASSWORD,ROLE1,ROLE2,...
# USER=PASSWORD,_g_:GROUP,...
# _g_\:GROUP=ROLE1,ROLE2,...
#
# All users, grousp, and roles entered in this file are available after
Karaf startup
# and modifiable via the JAAS command group. These users reside in a JAAS
domain
# with the name "karaf".
```

```
#
karaf = karaf,_g_:admingroup
_g_\:admingroup = group,admin,manager,viewer
```

We can see in this file, that we have one user by default: **karaf**. The default password is **karaf**.

The **karaf** user is member of one group: the **admingroup**.

A group is always prefixed by **g:**. An entry without this prefix is an user.

A group defines a set of roles. By default, the **admingroup** defines **group**, **admin**, **manager**, and **viewer** roles.

It means that the **karaf** user will have the roles defined by the **admingroup**.

19.1.1.1. Commands

The **jaas:*** commands manage the realms, users, groups, roles in the console.

19.1.1.1.1. jaas:realm-list

We already used the **jaas:realm-list** previously in this section.

The **jaas:realm-list** command list the realm and the login modules for each realm:

```
karaf@root()> jaas:realm-list
Index | Realm Name | Login Module Class Name
----------------------------------------------------------------------
-------------
1     | karaf      |
org.apache.karaf.jaas.modules.properties.PropertiesLoginModule
2     | karaf      |
org.apache.karaf.jaas.modules.publickey.PublickeyLoginModule
```

We have here one realm (**karaf**) containing two login modules (**PropertiesLoginModule** and **PublickeyLoginModule**).

The **index** is used by the **jaas:realm-manage** command to easily identify the realm/login module that we want to manage.

19.1.1.1.2. jaas:realm-manage

The **jaas:realm-manage** command switch in realm/login module edit mode, where you can manage the users, groups, and roles in the login module.

To identify the realm and login module that you want to manage, you can use the **--index** option. The indexes are displayed by the **jaas:realm-list** command:

```
karaf@root()> jaas:realm-manage --index 1
```

Another way is to use the **--realm** and **--module** options. The **--realm** option expects the realm name, and the **--module** option expects the login module class name:

```
karaf@root()> jaas:realm-manage --realm karaf --module
```

```
org.apache.karaf.jaas.modules.properties.PropertiesLoginModule
```

19.1.1.1.3. jaas:user-list

When you are in edit mode, you can list the users in the login module using the **jaas:user-list**:

```
karaf@root()> jaas:user-list
User Name | Group       | Role
------------------------------
karaf     | admingroup | admin
karaf     | admingroup | manager
karaf     | admingroup | viewer
```

You can see the user name and the group by role.

19.1.1.1.4. jaas:user-add

The **jaas:user-add** command adds a new user (and the password) in the currently edited login module:

```
karaf@root()> jaas:user-add foo bar
```

To "commit" your change (here the user addition), you have to execute the **jaas:update** command:

```
karaf@root()> jaas:update
karaf@root()> jaas:realm-manage --index 1
karaf@root()> jaas:user-list
User Name | Group       | Role
------------------------------
karaf     | admingroup | admin
karaf     | admingroup | manager
karaf     | admingroup | viewer
foo       |            |
```

On the other hand, if you want to rollback the user addition, you can use the **jaas:cancel** command.

19.1.1.1.5. jaas:user-delete

The **jaas:user-delete** command deletes an user from the currently edited login module:

```
karaf@root()> jaas:user-delete foo
```

Like for the **jaas:user-add** command, you have to use the **jaas:update** to commit your change (or **jaas:cancel** to rollback):

```
karaf@root()> jaas:update
karaf@root()> jaas:realm-manage --index 1
karaf@root()> jaas:user-list
User Name | Group       | Role
------------------------------
karaf     | admingroup | admin
karaf     | admingroup | manager
karaf     | admingroup | viewer
```

■

19.1.1.1.6. `jaas:group-add`

The **`jaas:group-add`** command assigns a group (and eventually creates the group) to an user in the currently edited login module:

```
karaf@root()> jaas:group-add karaf mygroup
```

19.1.1.1.7. `jaas:group-delete`

The **`jaas:group-delete`** command removes an user from a group in the currently edited login module:

```
karaf@root()> jaas:group-delete karaf mygroup
```

19.1.1.1.8. `jaas:group-role-add`

The **`jaas:group-role-add`** command adds a role in a group in the currently edited login module:

```
karaf@root()> jaas:group-role-add mygroup myrole
```

19.1.1.1.9. `jaas:group-role-delete`

The **`jaas:group-role-delete`** command removes a role from a group in the currently edited login module:

```
karaf@root()> jaas:group-role-delete mygroup myrole
```

19.1.1.1.10. `jaas:update`

The **`jaas:update`** command commits your changes in the login module backend. For instance, in the case of the PropertiesLoginModule, the **`etc/users.properties`** will be updated only after the execution of the **`jaas:update`** command.

19.1.1.1.11. `jaas:cancel`

The **`jaas:cancel`** command rollback your changes and doesn't update the login module backend.

19.1.2. Passwords encryption

By default, the passwords are stored in clear form in the **`etc/users.properties`** file.

It's possible to enable encryption in the **`etc/org.apache.karaf.jaas.cfg`** configuration file:

```
################################################################################
######
#
#    Licensed to the Apache Software Foundation (ASF) under one or more
#    contributor license agreements.  See the NOTICE file distributed with
#    this work for additional information regarding copyright ownership.
```

```
#     The ASF licenses this file to You under the Apache License, Version
2.0
#     (the "License"); you may not use this file except in compliance with
#     the License.  You may obtain a copy of the License at
#
#         http://www.apache.org/licenses/LICENSE-2.0
#
#     Unless required by applicable law or agreed to in writing, software
#     distributed under the License is distributed on an "AS IS" BASIS,
#     WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
implied.
#     See the License for the specific language governing permissions and
#     limitations under the License.
#
################################################################################
######

#
# Boolean enabling / disabling encrypted passwords
#
encryption.enabled = false

#
# Encryption Service name
#   the default one is 'basic'
#   a more powerful one named 'jasypt' is available
#       when installing the encryption feature
#
encryption.name =

#
# Encryption prefix
#
encryption.prefix = {CRYPT}

#
# Encryption suffix
#
encryption.suffix = {CRYPT}

#
# Set the encryption algorithm to use in Karaf JAAS login module
# Supported encryption algorithms follow:
#   MD2
#   MD5
#   SHA-1
#   SHA-256
#   SHA-384
#   SHA-512
#
encryption.algorithm = MD5

#
# Encoding of the encrypted password.
# Can be:
#   hexadecimal
```

```
#    base64
#
encryption.encoding = hexadecimal
```

If the **encryption.enabled** y is set to true, the password encryption is enabled.

With encryption enabled, the password are encrypted at the first time an user logs in. The encrypted passwords are prefixed and suffixed with **\{CRYPT\}**. To re-encrypt the password, you can reset the password in clear (in **etc/users.properties** file), without the **\{CRYPT\}** prefix and suffix. Apache Karaf will detect that this password is in clear (because it's not prefixed and suffixed with **\{CRYPT\}**) and encrypt it again.

The **etc/org.apache.karaf.jaas.cfg** configuration file allows you to define advanced encryption behaviours:

- the **encryption.prefix** property defines the prefix to "flag" a password as encrypted. The default is **\{CRYPT\}**.

- the **encryption.suffix** property defines the suffix to "flag" a password as encrypted. The default is **\{CRYPT\}**.

- the **encryption.algorithm** property defines the algorithm to use for encryption (digest). The possible values are **MD2**, **MD5**, **SHA-1**, **SHA-256**, **SHA-384**, **SHA-512**. The default is **MD5**.

- the **encryption.encoding** property defines the encoding of the encrypted password. The possible values are **hexadecimal** or **base64**. The default value is **hexadecimal**.

19.1.3. Managing authentication by key

For the SSH layer, Karaf supports the authentication by key, allowing to login without providing the password.

The SSH client (so bin/client provided by Karaf itself, or any ssh client like OpenSSH) uses a public/private keys pair that will identify himself on Karaf SSHD (server side).

The keys allowed to connect are stored in **etc/keys.properties** file, following the format:

```
user=key,role
```

By default, Karaf allows a key for the karaf user:

```
#
karaf=AAAAB3NzaC1kc3MAAACBAP1/U4EddRIpUt9KnC7s5Of2EbdSPO9EAMMeP4C2USZpRV1A
IlH7WT2NWPq/xfW6MPbLm1Vs14E7gB00b/JmYLdrmVClpJ+f6AR7ECLCT7up1/63xhv4O1fnxq
imFQ8E+4P208UewwI1VBNaFpEy9nXzrith1yrv8iIDGZ3RSAHHAAAAFQCXYFCPFSMLzLKSuYKi
64QL8Fgc9QAAAIEA9+GghdabPd7LvKtcNrhXuXmUr7v6OuqC+VdMCz0HgmdRWVeOutRZT+ZxBx
CBgLRJFnEj6EwoFhO3zwkyjMim4TwWeotUfI0o4KOuHiuzpnWRbqN/C/ohNWLx+2J6ASQ7zKTx
vqhRkImog9/hWuWfBpKLZl6Ae1UlZAFMO/7PSSoAAACBAKKSU2PFl/qOLxIwmBZPPIcJshVe7b
VUpFvyl3BbJDow8rXfskl8wO63OzP/qLmcJM0+JbcRU/53JjTuyk31drV2qxhIOsLDC9dGCWj4
7Y7TyhPdXh/0dthTRBy6bqGtRPxGa7gJov1xm/UuYYXPIUR/3x9MAZvZ5xvE0kYXO+rx,admin
```

NOTE

For security reason, this key is disabled. We encourage to create the keys pair per client and update the **etc/keys.properties** file.

The easiest way to create key pair is to use OpenSSH.

You can create a key pair using:

```
ssh-keygen -t dsa -f karaf.id_dsa -N karaf
```

You have now the public and private keys:

```
-rw-------  1 jbonofre jbonofre    771 Jul 25 22:05 karaf.id_dsa
-rw-r--r--  1 jbonofre jbonofre    607 Jul 25 22:05 karaf.id_dsa.pub
```

You can copy in the content of the **karaf.id_dsa.pub** file in the **etc/keys.properties**:

```
karaf=AAAAB3NzaC1kc3MAAACBAJLj9vnEhu3/Q9Cvym2jRDaNWkATgQiHZxmErCmiLRuD5Klf
v+HT/+8WoYdnvj0YaXFP80phYhzZ7fbIO2LRFhYhPmGLa9nSeOsQlFuX5A9kY1120yB2kxSIZI
0fU2hy1UCgmTxdTQPSYtdWBJyvO/vczoX/8I3FziEfss07Hj1NAAAAFQD1dKEzkt4e7rBPDokP
OMZigBh4kwAAAIEAiLnpbGNbKm8SNLUEc/fJFswg4G4VjjngjbPZAjhkYe4+H2uYmynry6V+GO
TS2kaFQGZRf9XhSpSwfdxKtx7vCCaoH9bZ6S5Pe0voWmeBhJXi/Sww8f2stpitW2Oq7V7lDdDG
81+N/D7/rKDD5PjUyMsVqc1n9wCTmfqmi6XPEw8AAACAHAGwPn/Mv7P9Q9+JZRWtGq+i4pL1zs
1OluiStCN9e/Ok96t3gRVKPheQ6IwLacNjC9KkSKrLtsVyepGA+V5j/N+Cmsl6csZilnLvMUTv
L/cmHDEEhTIQnPNrDDv+tED2BFqkajQqYLgMWeGVqXsBU6IT66itZlYtrq4v6uDQG/o=,admin
```

and specify to the client to use the **karaf.id_dsa** private key:

```
bin/client -k ~/karaf.id_dsa
```

or to ssh

```
ssh -p 8101 -i ~/karaf.id_dsa karaf@localhost
```

19.1.4. RBAC

Apache Karaf uses the roles to control the access to the resources: it's a RBAC (Role Based Access Control) system.

The roles are used to control:

- access to OSGi services

- access to the console (control the execution of the commands)

- access to JMX (MBeans and/or operations)

- access to the WebConsole

19.1.4.1. OSGi services

The details about OSGi services RBAC support is explained in the developer guide.

19.1.4.2. Console

Console RBAC supports is a specialization of the OSGi service RBAC. Actually, in Apache Karaf, all console commands are defined as OSGi services.

The console command name follows the **scope:name** format.

The ACL (Access Lists) are defined in **etc/org.apache.karaf.command.acl.<scope>.cfg** configuration files, where **<scope>** is the commands scope.

For instance, we can define the ACL to the **feature:*** commands by creating a **etc/org.apache.karaf.command.acl.feature.cfg** configuration file. In this **etc/org.apache.karaf.command.acl.feature.cfg** configuration file, we can set:

```
list = viewer
info = viewer
install = admin
uninstall = admin
```

Here, we define that **feature:list** and **feature:info** commands can be executed by users with **viewer** role, whereas the **feature:install** and **feature:uninstall** commands can only be executed by users with **admin** role. Note that users in the admin group will also have viewer role, so will be able to do everything.

Apache Karaf command ACLs can control access using (inside a given command scope):

- the command name regex (e.g. **name = role**)

- the command name and options or arguments values regex (e.g. **name[/.[0-9][0-9][0-9]+./] = role** to execute name only with argument value above 100)

Both command name and options/arguments support exact matching or regex matching.

By default, Apache Karaf defines the following commands ACLs:

- **etc/org.apache.karaf.command.acl.bundle.cfg** configuration file defines the ACL for **bundle:*** commands. This ACL limits the execution of **bundle:*** commands for system bundles only to the users with **admin** role, whereas **bundle:*** commands for non-system bundles can be executed by the users with **manager** role.

- **etc/org.apache.karaf.command.acl.config.cfg** configuration file defines the ACL for **config:*** commands. This ACL limits the execution of **config:*** commands with **jmx.acl.***, **org.apache.karaf.command.acl.***, and **org.apache.karaf.service.acl.*** configuration PID to the users with **admin** role. For the other configuration PID, the users with the **manager** role can execute **config:*** commands.

- **etc/org.apache.karaf.command.acl.feature.cfg** configuration file defines the ACL for **feature:*** commands. Only the users with **admin** role can execute **feature:install** and **feature:uninstall** commands. The other **feature:*** commands can be executed by any user.

- **etc/org.apache.karaf.command.acl.jaas.cfg** configuration file defines the ACL for **jaas:*** commands. Only the users with **admin** role can execute **jaas:update** command. The other **jaas:*** commands can be executed by any user.

- **etc/org.apache.karaf.command.acl.kar.cfg** configuration file defines the ACL for **kar:*** commands. Only the users with **admin** role can execute **kar:install** and **kar:uninstall** commands. The other **kar:*** commands can be executed by any user.

- **etc/org.apache.karaf.command.acl.shell.cfg** configuration file defines the ACL for **shell:*** and "direct" commands. Only the users with **admin** role can execute **shell:edit**, **shell:exec**, **shell:new**, and **shell:java** commands. The other **shell:*** commands can be executed by any user.

You can change these default ACLs, and add your own ACLs for additional command scopes (for instance **etc/org.apache.karaf.command.acl.cluster.cfg** for Apache Karaf Cellar, **etc/org.apache.karaf.command.acl.camel.cfg** from Apache Camel, …).

You can fine tuned the command RBAC support by editing the **karaf.secured.services** property in **etc/system.properties**:

```
#
# By default, only Karaf shell commands are secured, but additional
services can be
# secured by expanding this filter
#
karaf.secured.services = (&(osgi.command.scope=*)
(osgi.command.function=*))
```

19.1.4.3. JMX

Like for the console commands, you can define ACL (AccessLists) to the JMX layer.

The JMX ACL are defined in **etc/jmx.acl<ObjectName>.cfg** configuration file, where **<ObjectName>** is a MBean object name (for instance **org.apache.karaf.bundle** represents **org.apache.karaf;type=Bundle** MBean).

The **etc/jmx.acl.cfg** is the most generic configuration file and is used when no specific ones are found. It contains the "global" ACL definition.

JMX ACLs can control access using (inside a JMX MBean):

- the operation name regex (e.g. **operation* = role**)

- the operation arguments value regex (e.g. **operation(java.lang.String, int)[/([1-4])?[0-9]/,/.*/] = role**)

By default, Apache Karaf defines the following JMX ACLs:

- **etc/jmx.acl.org.apache.karaf.bundle.cfg** configuration file defines the ACL for the **org.apache.karaf:type=bundle** MBean. This ACL limits the **setStartLevel()**, **start()**, **stop()**, and **update()** operations for system bundles for only users with **admin** role. The other operations can be performed by users with the **manager** role.

- **etc/jmx.acl.org.apache.karaf.config.cfg** configuration file defines the ACL for the **org.apache.karaf:type=config** MBean. This ACL limits the change on **jmx.acl***, **org.apache.karaf.command.acl***, and **org.apache.karaf.service.acl*** configuration PIDs for only users with **admin** role. The other operations can be performed by users with the **manager** role.

- **etc/jmx.acl.org.apache.karaf.security.jmx.cfg** configuration file defines the ACL for the **org.apache.karaf:type=security,area=jmx** MBean. This ACL limits the invocation of the **canInvoke()** operation for the users with **viewer** role.

- **etc/jmx.acl.osgi.compendium.cm.cfg** configuration file defines the ACL for the **osgi.compendium:type=cm** MBean. This ACL limits the changes on **jmx.acl***, **org.apache.karaf.command.acl***, and **org.apache.karaf.service.acl*** configuration PIDs for only users with **admin** role. The other operations can be performed by users with the **manager** role.

- **etc/jmx.acl.java.lang.Memory.cfg** configuration file defines the ACL for the core JVM Memory MBean. This ACL limits the invocation of the **gc** operation for only users with the **manager** role.

- **etc/jmx.acl.cfg** configuration file is the most generic file. The ACLs defined here are used when no other specific ACLs match (by specific ACL, it's an ACL defined in another MBean specific **etc/jmx.acl.*.cfg** configuration file). The **list*()**, **get*()**, **is*()** operations can be performed by users with the **viewer** role. The **set*()** and all other ***()** operations can be performed by users with the **admin** role.

19.1.4.4. WebConsole

The Apache Karaf WebConsole is not available by default. To enable it, you have to install the **webconsole** feature:

```
karaf@root()> feature:install webconsole
```

The WebConsole doesn't support fine grained RBAC like console or JMX for now.

All users with the **admin** role can logon the WebConsole and perform any operations.

19.1.5. SecurityMBean

Apache Karaf provides a JMX MBean to check if the current user can invoke a given MBean and/or operation.

The **canInvoke()** operation gets the roles of the current user, and check if one the roles can invoke the MBean and/or the operation, eventually with a given argument value.

19.1.5.1. Operations

- **canInvoke(objectName)** returns **true** if the current user can invoke the MBean with the **objectName**, **false** else.

- **canInvoke(objectName, methodName)** returns **true** if the current user can invoke the operation **methodName** on the MBean with the **objectName**, **false** else.

- **canInvoke(objectName, methodName, argumentTypes)** returns **true** if the current user can invoke the operation **methodName** with the array of arguments types **argumentTypes** on the MBean with **objectName**, **false** else.

- **canInvoke(bulkQuery)** returns a tabular data containing for each operation in the **bulkQuery** tabular data if **canInvoke** is **true** or **false**.

19.1.6. Security providers

Some applications require specific security providers to be available, such as [BouncyCastle|http://www.bouncycastle.org].

The JVM imposes some restrictions about the use of such jars: they have to be signed and be available on the boot classpath.

One way to deploy those providers is to put them in the JRE folder at **$JAVA_HOME/jre/lib/ext** and modify the security policy configuration (**$JAVA_HOME/jre/lib/security/java.security**) in order to register such providers.

While this approach works fine, it has a global effect and requires you to configure all your servers accordingly.

Apache Karaf offers a simple way to configure additional security providers: * put your provider jar in **lib/ext** * modify the **etc/config.properties** configuration file to add the following property

```
org.apache.karaf.security.providers = xxx,yyy
```

The value of this property is a comma separated list of the provider class names to register.

For instance, to add the bouncycastle security provider, you define:

```
org.apache.karaf.security.providers =
org.bouncycastle.jce.provider.BouncyCastleProvider
```

In addition, you may want to provide access to the classes from those providers from the system bundle so that all bundles can access those.

It can be done by modifying the **org.osgi.framework.bootdelegation** property in the same configuration file:

```
org.osgi.framework.bootdelegation = ...,org.bouncycastle*
```