



# Red Hat JBoss Fuse 6.1

## Getting Started

Learn to solve problems with Red Hat JBoss Fuse



# Red Hat JBoss Fuse 6.1 Getting Started

---

Learn to solve problems with Red Hat JBoss Fuse

JBoss A-MQ Docs Team

Content Services

[fuse-docs-support@redhat.com](mailto:fuse-docs-support@redhat.com)

## Legal Notice

Copyright © 2013 Red Hat.

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

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

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

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

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

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

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

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

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

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

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

All other trademarks are the property of their respective owners.

## Abstract

This guide provide an introduction to developing applications with Red Hat JBoss Fuse.

---

## Table of Contents

<b>CHAPTER 1. BASIC CONCEPTS FOR DEVELOPERS</b> .....	<b>3</b>
1.1. DEVELOPMENT ENVIRONMENT	3
1.2. DEVELOPMENT MODEL	3
1.3. MAVEN ESSENTIALS	5
1.4. DEPENDENCY INJECTION FRAMEWORKS	10
<b>CHAPTER 2. GETTING STARTED WITH DEVELOPING</b> .....	<b>14</b>
2.1. CREATE A WEB SERVICES PROJECT	14
2.2. CREATE A ROUTER PROJECT	20
2.3. CREATE AN AGGREGATE MAVEN PROJECT	24
2.4. DEFINE A FEATURE FOR THE APPLICATION	25
2.5. CONFIGURE THE APPLICATION	28
2.6. TROUBLESHOOTING	31
<b>CHAPTER 3. GETTING STARTED WITH DEPLOYING</b> .....	<b>33</b>
3.1. SCALABLE DEPLOYMENT WITH FUSE FABRIC	33
3.2. DEPLOYING TO A FABRIC	35
<b>APPENDIX A. EDITING PROFILES WITH THE BUILT-IN TEXT EDITOR</b> .....	<b>42</b>
A.1. EDITING AGENT PROPERTIES	42
A.2. EDITING OSGI CONFIG ADMIN PROPERTIES	45
A.3. EDITING OTHER RESOURCES	46
A.4. PROFILE ATTRIBUTES	48



# CHAPTER 1. BASIC CONCEPTS FOR DEVELOPERS

## Abstract

A typical Red Hat JBoss Fuse application is based on a specific development model, which is based around the Java language, Spring or Blueprint dependency injection frameworks, and the Apache Maven build system.

## 1.1. DEVELOPMENT ENVIRONMENT

### JDK 1.6

The basic requirement for development with Red Hat JBoss Fuse is the Java Development Kit (JDK) from Oracle. Generally, the recommended version is J2SE 1.6—for more platform-specific details, see the [Red Hat JBoss Fuse Supported Configurations](#) page.

### Apache Maven

The recommended build system for developing JBoss Fuse applications is [Apache Maven](#) version 3.0.x. See the *Installation Guide* for more details.

Maven is more than a build system, however. Just as importantly, Maven also provides an infrastructure for distributing application components (typically JAR files; formally called *artifacts*). When you build an application, Maven automatically searches repositories on the Internet to find the JAR dependencies needed by your application, and then downloads the needed dependencies. See [Section 1.3, “Maven Essentials”](#) for more details.

### Red Hat JBoss Fuse Tooling

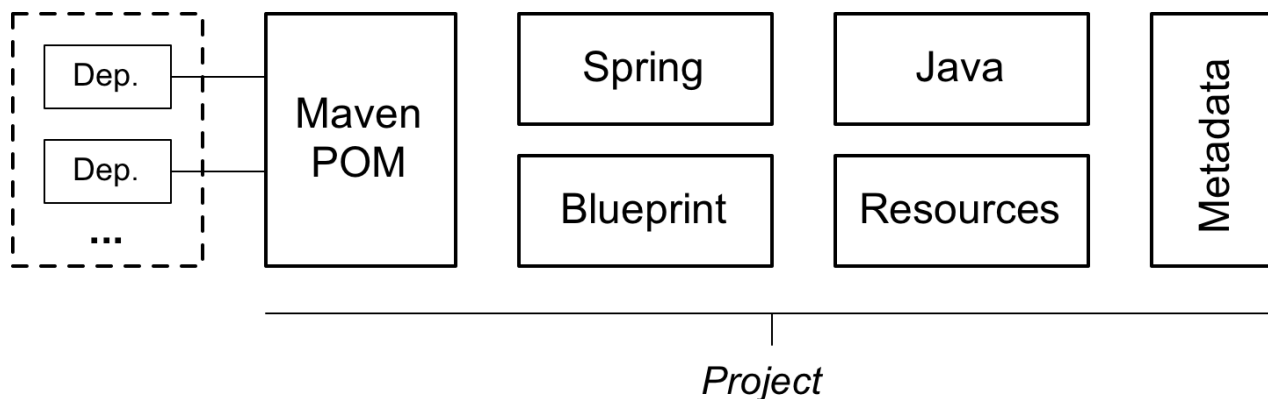
The JBoss Fuse Tooling provides a set of developer tools for developing Red Hat JBoss Fuse applications within Red Hat JBoss Development Studio. Using JBoss Fuse Tooling, you can connect and configure Enterprise Integration Patterns to build routes, browse endpoints and routes, drag and drop messages onto running routes, trace message flows, edit running routes, browse and visualize runtime processes via JMX, and deploy your project's code to Red Hat JBoss Fuse and Fabric8 containers, to Apache ServiceMix, and to Apache Karaf. You can download the Red Hat JBoss Fuse Tooling from within Red Hat JBoss Development Studio. For details, see the [Red Hat JBoss Fuse Tooling Installation Guide](#).

## 1.2. DEVELOPMENT MODEL

### Overview

[Figure 1.1, “Developing a Red Hat JBoss Fuse Project”](#) shows an overview of the development model for building an OSGi bundle or a Fuse Application Bundle that will be deployed into the Red Hat JBoss Fuse container.

Figure 1.1. Developing a Red Hat JBoss Fuse Project



## Maven

Apache Maven, which is the recommended build system for JBoss Fuse, affects the development model in the following important ways:

- *Maven directory layout*—Maven has a standard directory layout that determines where you put your Java code, associated resources, XML configuration files, unit test code, and so on.
- *Accessing dependencies through the Internet*—Maven has the ability to download dependencies automatically through the Internet, by searching through known *Maven repositories*. This implies that you must have either access to the Internet or local copies of the required Maven repositories, when building with Maven. See [the section called “Maven repositories”](#).

## Maven archetypes

An easy way to get started with development is by using a *Maven archetype*, which is analogous to using the new project wizard (except that it must be invoked from the command line). A Maven archetype typically creates a completely new Maven project, with the correct directory layout and some sample code. For example, see [Section 2.1, “Create a Web Services Project”](#) and [Section 2.2, “Create a Router Project”](#).

## Maven POM files

The Maven *Project Object Model* (POM) file, `pom.xml`, provides the description of how to build your project. The initial version of a POM is typically generated by a Maven archetype. You can then customise the POM as needed.

For larger Maven projects, there are two special kinds of POM files that you might also need:

- *Aggregator POM*—a complete application is typically composed of multiple Maven projects, which must be built in a certain order. To simplify building multi-project applications, Maven enables you to define an *aggregator POM*, which can build all of the sub-projects in a single step. For more details, see [Section 2.3, “Create an Aggregate Maven Project”](#).
- *Parent POM*—in a multi-project application, the POMs for the sub-projects typically contain a lot of the same information. Over the long term, maintaining this information, which is spread across multiple POM files, would be time-consuming and error-prone. To make the POMs more manageable, you can define a parent POM, which encapsulates all of the shared information.

## Java code and resources



Maven reserves a standard location, `src/main/java`, for your Java code, and for the associated resource files, `src/main/resources`. When Maven builds a JAR file, it automatically compiles all of the Java code and adds it to the JAR package. Likewise, all of the resource files found under `src/main/resources` are copied into the JAR package.

## Dependency injection frameworks

JBoss Fuse has built-in support for two dependency injection frameworks: Spring XML and Blueprint XML. You can use one or the other, or both at the same time. The projects underlying JBoss Fuse (Apache Camel, Apache CXF, Apache ActiveMQ, and Apache ServiceMix) all strongly support XML configuration. In fact, in many cases, it is possible to develop a complete application written in XML, without any Java code whatsoever.

For more details, see [Section 1.4, “Dependency Injection Frameworks”](#).

## Deployment metadata

Depending on how a project is packaged and deployed (as an OSGi bundle, a Fuse Application Bundle (FAB), or a WAR), there are a few different files embedded in the JAR package that can be interpreted as deployment metadata, for example:

### **META-INF/MANIFEST.MF**

The JAR manifest can be used to provide deployment metadata either for an OSGi bundle (in bundle headers) or for a FAB.

### **META-INF/maven/groupId/artifactId/pom.xml**

The POM file—which is normally embedded in any Maven-built JAR file—is the main source of deployment metadata for a FAB.

### **WEB-INF/web.xml**

The `web.xml` file is the standard descriptor for an application packaged as a Web ARchive (WAR).

## Administrative metadata

The following kinds of metadata are accessible to administrators, who can use them to customize or change the behavior of bundles at run time:

- *Apache Karaf features*—a feature specifies a related collection of packages that can be deployed together. By selecting which features to install (or uninstall), an administrator can easily control which blocks of functionality are deployed in the container.
- *OSGi Config Admin properties*—the OSGi Config Admin service exposes configuration properties to the administrator at run time, making it easy to customize application behavior (for example, by customizing the IP port numbers on a server).

## 1.3. MAVEN ESSENTIALS

### Overview

This section provides a quick introduction to some essential Maven concepts, enabling you to understand the fundamental ideas of the Maven build system.

## Build lifecycle phases

Maven defines a standard set of phases in the build lifecycle, where the precise sequence of phases depends on what type of package you are building. For example, a JAR package includes the phases (amongst others): **compile**, **test**, **package**, and **install**.

When running Maven, you normally specify the phase as an argument to the **mvn** command, in order to indicate how far you want the build to proceed. To get started, the following are the most commonly used Maven commands:

- Build the project, run the unit tests, and install the resulting package in the local Maven repository:

```
mvn install
```

- Clean the project (deleting temporary and intermediate files):

```
mvn clean
```

- Build the project and run the unit tests:

```
mvn test
```

- Build and install the project, skipping the unit tests:

```
mvn install -Dmaven.test.skip=true
```

- Build the project in offline mode:

```
mvn -o install
```

Offline mode (selected by the **-o** option) is useful in cases where you know that you already have all of the required dependencies in your local repository. It prevents Maven from (unnecessarily) checking for updates to SNAPSHOT dependencies, enabling the build to proceed more quickly.

## Maven directory structure

[Example 1.1, “Standard Maven Directory Layout”](#) shows the standard Maven directory layout. Most important is the Maven POM file, **pom.xml**, which configures the build for this Maven project.

### Example 1.1. Standard Maven Directory Layout

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

```

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

```

The project's Java source files must be stored under ***ProjectDir/src/main/java/*** and any resource files should be stored under ***ProjectDir/src/main/resources/***. In particular, Spring XML files (matching the pattern ***\*.xml***) should be stored under the following directory:

```
ProjectDir/src/main/resources/META-INF/spring/
```

Blueprint XML files (matching the pattern ***\*.xml***) should be stored under the following directory:

```
ProjectDir/src/main/resources/OSGI-INF/blueprint/
```

## Convention over configuration

An important principle of Maven is that of *convention over configuration*. What this means is that Maven's features and plug-ins are initialized with sensible default conventions, so that the basic functionality of Maven requires little or no configuration.

In particular, the location of the files within Maven's standard directory layout effectively determines how they are processed. For example, if you have a Maven project for building a JAR, all of the Java files under the ***src/main/java*** directory are automatically compiled and added to the JAR. All of the resource files under the ***src/main/resources*** directory are also added to the JAR.



### NOTE

Although it is possible to alter the default Maven conventions, this practice is *strongly discouraged*. Using non-standard Maven conventions makes your projects more difficult to configure and more difficult to understand.

## Maven packaging type

Maven defines a variety of packaging types, which determine the basic build behavior. The most common packaging types are as follows:

### jar

*(Default)* This packaging type is used for Fuse Application Bundles (FABs).

### bundle

This packaging type is used for OSGi bundles. To use this packaging type, you must also configure the ***maven-bundle-plugin*** in the POM file.

### war

This packaging type is used for WAR files. To use this packaging type, you must also configure the **maven-war-plugin** in the POM file.

## pom

When you build with this packaging type, the POM file itself gets installed into the local Maven repository. This packaging type is typically used for parent POM files.

## Maven artifacts

The end product of a Maven build is a Maven *artifact* (for example, a JAR file). Maven artifacts are normally installed into a Maven repository, from where they can be accessed and used as building blocks for other Maven projects (by declaring them as dependencies).

## Maven coordinates

Artifacts are uniquely identified by a tuple of Maven coordinates, usually consisting of **groupId:artifactId:version**. For example, when deploying a Maven artifact into the Red Hat JBoss Fuse container, you can reference it using a Maven URI of the form, **mvn:groupId/artifactId/version**.

For more details about Maven coordinates, see [Deploying into the Container](#).

## Maven dependencies

The most common modification you will need to make to your project's POM file is adding or removing Maven dependencies. A dependency is simply a reference to a Maven artifact (typically a JAR file) that is needed to build and run your project. In fact, in the context of a Maven build, managing the collection of dependencies in the POM effectively takes the place of managing the collection of JAR files in a Classpath.

The following snippet from a POM file shows how to specify a dependency on the **camel-blueprint** artifact:

```
<project ...>
  ...
  <dependencies>
    <dependency>
      <groupId>org.apache.camel</groupId>
      <artifactId>camel-blueprint</artifactId>
      <version>2.12.0.redhat-610379</version>
      <scope>provided</scope>
    </dependency>
  </dependencies>
  ...
</project>
```

## dependency element

The **dependency** element declares a dependency on the Maven artifact with coordinates (for example, on **org.apache.camel:camel-blueprint:6.1.0.redhat-379**). You can add as many **dependency** elements as you like inside the **dependencies** element.

## dependency/scope element

The **scope** element is optional and provides some additional information about when this dependency is needed. By default (with the **scope** element omitted), it is assumed that the dependency is needed at build time, at unit test time, and at run time. With **scope** set to the value, **provided**, the effect depends on what kind of artifact you are building:

- *OSGi bundle*—(when the POM's **packaging** element is specified as **bundle**) the **provided** scope setting has no effect.
- *Fuse Application Bundle (FAB)*—(when the POM's **packaging** element is specified as **jar**) the **provided** scope setting implies that this dependency is deployed as a *separate* bundle in the container and is thus shared with other applications at run time.

## Transitive dependencies

To simplify the list of dependencies in your POM and to avoid having to list every single dependency explicitly, Maven employs a recursive algorithm to figure out the dependencies needed for your project.

For example, if your project, A, depends on B1 and B2; B1 depends on C1, C2, and C3; and B2 depends on D1 and D2; Maven will automatically pull in *all* of the explicitly and implicitly required dependencies at build time, constructing a classpath that includes the dependencies, B1, B2, C1, C2, C3, D1, and D2. Of these dependencies, only B1 and B2 appear explicitly in A's POM file. The rest of the dependencies—which are figured out by Maven—are known as *transitive dependencies*.

## Maven repositories

A Maven repository is a place where Maven can go to search for artifacts. Because Maven repositories can be anywhere—and that includes anywhere on the Internet—the Maven build system is inherently distributed. The following are the main categories of Maven repository:

- *Local repository*—the local repository (by default, located at `~/ .m2/repository` on \*NIX or `C:\Documents and Settings\UserName\.m2\repository` on Windows) is used by Maven as follows:
  - *First search location*—the local repository is the first place that Maven looks when searching for a dependency.
  - *Cache of downloaded dependencies*—any artifacts that have ever been downloaded from a remote repository are stored permanently in the local repository, so that they can be retrieved quickly next time they are needed.
  - *Store of locally-built artifacts*—any time that you build a local project (using `mvn install`), the resulting artifact gets stored in your local repository.
- *Remote repository*—Maven can also search for and download artifacts from remote repositories. By default, Maven automatically tries to download an artifact from remote repositories, if it cannot find the artifact in the local repository (you can suppress this behavior by specifying the `-o` flag—for example, `mvn -o install`).
- *System repository*—(Red Hat JBoss Fuse container only; *not* used by the `mvn` command-line tool) at run time, the Red Hat JBoss Fuse container can access artifacts from the JBoss Fuse system repository, which is located at `InstallDir/system/`.

For more details about Maven repositories, see [Deploying into the Container](#).

## Specifying remote repositories

If you need to customise the remote repositories accessible to Maven, you must separately configure the build-time and runtime repository locations, as follows:

- *Build time*—to customize the remote repositories accessible at build time (when running the `mvn` command), edit the Maven `settings.xml` file, at the following location:
  - **\*Nix:** default location is `~/.m2/settings.xml`.
  - **Windows:** default location is `C:\Documents and Settings\UserName\.m2\settings.xml`.
- *Run time*—to customize the remote repositories accessible at run time (from within Red Hat JBoss Fuse container), edit the relevant property settings in the `InstallDir/etc/org.ops4j.pax.url.mvn.cfg`.

## 1.4. DEPENDENCY INJECTION FRAMEWORKS

### Overview

Red Hat JBoss Fuse offers a choice between the following built-in dependency injection frameworks:

- [the section called “Spring XML”](#).
- [the section called “Blueprint XML”](#).

### Blueprint or Spring?

When trying to decide between the Blueprint and Spring dependency injection frameworks, bear in mind that Blueprint offers one major advantage over Spring: when new dependencies are introduced in Blueprint through XML schema namespaces, Blueprint has the capability to resolve these dependencies *automatically* at run time. In contrast, when packaging your project as an OSGi bundle, Spring requires you to add new dependencies explicitly to the `maven-bundle-plugin` configuration.

### Bean registries

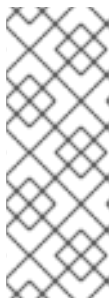
A fundamental capability of the dependency injection frameworks is the ability to create Java bean instances. Every Java bean created in a dependency injection framework is added to a *bean registry* by default. The bean registry is a map that enables you to look up a bean's object reference using the bean ID. This makes it possible to reference bean instances within the framework's XML configuration file and to reference bean instances from your Java code.

For example, when defining Apache Camel routes, you can use the `bean()` and `beanRef()` DSL commands to access the bean registry of the underlying dependency injection framework (or frameworks).

### Spring XML

[Spring](#) is fundamentally a dependency injection framework, but it also includes a suite of services and APIs that enable it to act as a fully-fledged container. A Spring XML configuration file can be used in the following ways:

- *An injection framework*—Spring is a classic injection framework, enabling you to instantiate Java objects using the **bean** element and to wire beans together, either explicitly or automatically. For details, see [The IoC Container](#) from the *Spring Reference Manual*.
- *A generic XML configuration file*—Spring has an extensibility mechanism that makes it possible to use third-party XML configuration schemas in a Spring XML file. Spring uses the schema namespace as a hook for finding an extension: it searches the classpath for a JAR file that implements that particular namespace extension. In this way, it is possible to embed the following XML configurations inside a Spring XML file:
  - *Apache Camel configuration*—usually introduced by the **camelContext** element in the schema namespace, <http://camel.apache.org/schema/spring>.
  - *Apache CXF configuration*—uses several different schema namespaces, depending on whether you are configuring the Bus, <http://cxf.apache.org/core>, a JAX-WS binding, <http://cxf.apache.org/jaxws>, a JAX-RS binding, <http://cxf.apache.org/jaxrs>, or a Simple binding, <http://cxf.apache.org/simple>.
  - *Apache ActiveMQ configuration*—usually introduced by the **broker** element in the schema namespace, <http://activemq.apache.org/schema/core>.



## NOTE

When packaging your project as an OSGi bundle, the Spring XML extensibility mechanism can introduce additional dependencies. Because the Maven bundle plug-in does *not* have the ability to scan the Spring XML file and automatically discover the dependencies introduced by schema namespaces, *it is generally necessary to add the additional dependencies explicitly to the **maven-bundle-plugin** configuration (by specifying the required Java packages).*

- *An OSGi toolkit*—Spring also has features (provided by [Spring Dynamic Modules](#)) to simplify integrating your application with the OSGi container. In particular, Spring DM provides XML elements that make it easy to export and consume OSGi services. For details, see [The Service Registry](#) from the *Spring DM Reference Manual*.
- *A provider of container services*—Spring also supports typical container services, such as security, persistence, and transactions. Before using such services, however, you should compare what is available from the JBoss Fuse container itself. In some cases, the JBoss Fuse container already layers a service on top of Spring (as with the transaction service, for example). In other cases, the JBoss Fuse container might provide an alternative implementation of the same service.

## Spring XML file location

In your Maven project, Spring XML files must be placed in the following location:

```
InstallDir/src/main/resources/META-INF/spring/*.xml
```

## Spring XML sample

The following example shows the bare outline of a Spring XML file:



```

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

  <camelContext xmlns="http://camel.apache.org/schema/spring">
    <!-- Define Camel routes here -->
    ...
  </camelContext>

</beans>

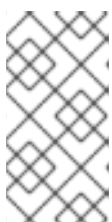
```

You can use a Spring XML file like this to configure Apache ActiveMQ, Apache CXF, and Apache Camel applications. For example, the preceding example includes a **camelContext** element, which could be used to define Apache Camel routes. For a more detailed example of Spring XML, see [the section called “Blueprint XML configuration”](#).

## Blueprint XML

Blueprint is a dependency injection framework defined in the [OSGi specification](#). Historically, Blueprint was originally sponsored by Spring and was based loosely on Spring DM. Consequently, the functionality offered by Blueprint is quite similar to Spring XML, but Blueprint is a more lightweight framework and it has been specially tailored for the OSGi container.

- *An injection framework*—Blueprint is a classic injection framework, enabling you to instantiate Java objects using the **bean** element and to wire beans together, either explicitly or automatically. For details, see [Deploying into the Container](#).
- *A generic XML configuration file*—Blueprint has an extensibility mechanism that makes it possible to use third-party XML configuration schemas in a Blueprint XML file. Blueprint uses the schema namespace as a hook for finding an extension: it searches the classpath for a JAR file that implements that particular namespace extension. In this way, it is possible to embed the following XML configurations inside a Blueprint XML file:
  - *Apache Camel configuration*—usually introduced by the **camelContext** element in the schema namespace, <http://camel.apache.org/schema/blueprint>.
  - *Apache CXF configuration*—uses several different schema namespaces, depending on whether you are configuring the Bus, <http://cxf.apache.org/blueprint/core>, a JAX-WS binding, <http://cxf.apache.org/blueprint/jaxws>, a JAX-RS binding, <http://cxf.apache.org/blueprint/jaxrs>, or a Simple binding, <http://cxf.apache.org/blueprint/simple>.
  - *Apache ActiveMQ configuration*—usually introduced by the **broker** element in the schema namespace, <http://activemq.apache.org/schema/core>.



### NOTE

When packaging your project as an OSGi bundle, the Blueprint XML extensibility mechanism can introduce additional dependencies, through the schema namespaces. *Blueprint automatically resolves the dependencies implied by the schema namespaces at run time.*



- *An OSGi toolkit*—Blueprint also has features to simplify integrating your application with the OSGi container. In particular, Blueprint provides XML elements that make it easy to export and consume OSGi services. For details, see [Deploying into the Container](#).

## Blueprint XML file location

In your Maven project, Blueprint XML files must be placed in the following location:

```
InstallDir/src/main/resources/OSGI-INF/blueprint/*.xml
```

## Blueprint XML sample

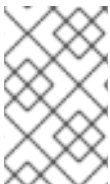
The following example shows the bare outline of a Blueprint XML file:

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

  <camelContext xmlns="http://camel.apache.org/schema/blueprint">
    <!-- Define Camel routes here -->
    ...
  </camelContext>

</blueprint>
```

You can use a Blueprint XML file like this to configure Apache ActiveMQ, Apache CXF, and Apache Camel applications. For example, the preceding example includes a **camelContext** element, which could be used to define Apache Camel routes. For a more detailed example of Blueprint XML, see [Example 2.2, “Configuring the Port Number in Blueprint XML”](#).



### NOTE

The schema namespace used for Apache Camel in Blueprint, **`http://camel.apache.org/schema/blueprint`**, is different from the namespace used for Apache Camel in Spring XML. The two schemas are almost identical, however.

## CHAPTER 2. GETTING STARTED WITH DEVELOPING

### Abstract

This chapter explains how to get started with Maven-based development, with a two-part project that illustrates how to develop applications using Apache CXF and Apache Camel.

## 2.1. CREATE A WEB SERVICES PROJECT

### Overview

This section describes how to generate a simple Web services project, which includes complete demonstration code for a server and a test client. The starting point for this project is the **servicemix-cxf-code-first-osgi-bundle** Maven archetype, which is a command-line wizard that creates the entire project from scratch. Instructions are then given to build the project, deploy the server to the Red Hat JBoss Fuse container, and run the test client.

### Prerequisites

In order to access artifacts from the Maven repository, you need to add the **fusesource** repository to Maven's **settings.xml** file. Maven looks for your **settings.xml** file in the following standard location:

- **UNIX:** `home/User/.m2/settings.xml`
- **Windows:** `Documents and Settings\User\.m2\settings.xml`

If there is currently no **settings.xml** file at this location, you need to create a new **settings.xml** file. Modify the **settings.xml** file by adding the **repository** element for **fusesource**, as highlighted in the following example:

```
<settings>
  <profiles>
    <profile>
      <id>my-profile</id>
      <activation>
        <activeByDefault>true</activeByDefault>
      </activation>
      <repositories>
        <repository>
          <id>fusesource</id>
          <url>http://repo.fusesource.com/nexus/content/groups/public/</url>
          <snapshots>
            <enabled>false</enabled>
          </snapshots>
          <releases>
            <enabled>true</enabled>
          </releases>
        </repository>
        ...
      </repositories>
```

```

        </profile>
    </profiles>
    ...
</settings>

```

## Create project from the command line

You can create a Maven project directly from the command line, by invoking the **archetype:generate** goal. First of all, create a directory to hold your getting started projects. Open a command prompt, navigate to a convenient location in your file system, and create the **get-started** directory, as follows:

```

mkdir get-started
cd get-started

```

You can now use the **archetype:generate** goal to invoke the **servicemix-cxf-code-first-osgi-bundle** archetype, which generates a simple Apache CXF demonstration, as follows:

```

mvn archetype:generate
-DarchetypeGroupId=org.apache.servicemix.tooling
-DarchetypeArtifactId=servicemix-cxf-code-first-osgi-bundle
-DarchetypeVersion=2013.01.0.redhat-610379
-DgroupId=org.fusesource.example
-DartifactId=cxf-basic
-Dversion=1.0-SNAPSHOT

```



### NOTE

The arguments of the preceding command are shown on separate lines for readability, but when you are actually entering the command, the entire command *must* be entered on a single line.

You will be prompted to confirm the project settings, with a message similar to this one:

```

[INFO] Using property: groupId = org.fusesource.example
[INFO] Using property: artifactId = cxf-basic
[INFO] Using property: version = 1.0-SNAPSHOT
[INFO] Using property: package = org.fusesource.example
Confirm properties configuration:
groupId: org.fusesource.example
artifactId: cxf-basic
version: 1.0-SNAPSHOT
package: org.fusesource.example
Y: :

```

Press the **Return** key to accept the settings and generate the project. When the command finishes, you should find a new Maven project in the **get-started/cxf-basic** directory.

## Customize the POM file

Because this project will be configured using Blueprint XML, it is necessary to edit the project's **pom.xml** file, to declare the Blueprint dependency. Edit the **cxf-basic/pom.xml** file and modify the **Import - Package** element of the **maven-bundle-plugin** by adding the lines highlighted in the following

extract:

```
<?xml version="1.0" encoding="UTF-8"?>
<project ...>
  ...
  <build>
    <defaultGoal>install</defaultGoal>
    <plugins>
      ...
      <plugin>
        <groupId>org.apache.felix</groupId>
        <artifactId>maven-bundle-plugin</artifactId>
        <version>2.3.6</version>
        <extensions>>true</extensions>
        <configuration>
          <instructions>
            <Bundle-SymbolicName>${project.artifactId}
</Bundle-SymbolicName>
            <Import-Package>
              javax.jws,
              javax.wsdl,
              javax.xml.bind,
              javax.xml.bind.annotation,
              javax.xml.namespace,
              javax.xml.ws,
              META-INF.cxf,
              META-INF.cxf.osgi,
              org.apache.cxf.bus,
              org.apache.cxf.bus.spring,
              org.apache.cxf.bus.resource,
              org.apache.cxf.configuration.spring,
              org.apache.cxf.resource,
              org.apache.cxf.jaxws,
              org.apache.cxf.transport.http,
              org.osgi.service.blueprint,
              *
            </Import-Package>
            ...
          </instructions>
        </configuration>
      </plugin>
    </plugins>
  </build>
  ...
</project>
```

## Blueprint XML configuration

We are going to use Blueprint XML to configure this project (Spring XML is not ideal in the context of OSGi, because it relies on the deprecated Spring-DM component to access OSGi services). Create the Blueprint XML file, as follows:

1. Delete the redundant Spring XML file and its parent directories:

```
cd cxf-basic
rm -rf src/main/resources/META-INF
```

2. Create a new directory to hold the Blueprint configuration files:

```
mkdir -p src/main/resources/OSGI-INF/blueprint
```

3. Create a new Blueprint XML file. Using your favourite text editor, create the file, **cxf-basic/src/main/resources/OSGI-INF/blueprint/beans.xml**, and add the following content to it:

### Example 2.1. Blueprint XML for Web Services Endpoint

```
<?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:jaxws="http://cxf.apache.org/blueprint/jaxws"
  xmlns:cxf="http://cxf.apache.org/blueprint/core"
  xsi:schemaLocation="
    http://cxf.apache.org/blueprint/core
    http://cxf.apache.org/schemas/blueprint/core.xsd
    http://cxf.apache.org/blueprint/jaxws
    http://cxf.apache.org/schemas/blueprint/jaxws.xsd">

  <jaxws:endpoint id="HTTPEndpoint"
    implementor="org.fusesource.example.PersonImpl"
    address="/PersonServiceCF"/>

</blueprint>
```

The purpose of this Blueprint XML file is to create a WS endpoint (that is, an instance of a Web service). The **jaxws:endpoint** element creates the WS endpoint and, in this example, it requires two attributes, as follows:

#### implementor

Specifies the class that implements the Service Endpoint Interface (SEI).

#### address

Specifies the WS endpoint address. In this example, instead of a HTTP URL, the address is specified as a relative path. In the context of JBoss Fuse, this is taken to mean that the Web service should be installed into the JBoss Fuse container's default Jetty container. By default, the specified path gets prefixed by **http://localhost:8181/cxf/**, so the actual address of the Web service becomes:

```
http://localhost:8181/cxf/PersonServiceCF
```

## Build the Web services project

Build the Web services project and install the generated JAR file into your local Maven repository. From a command prompt, enter the following commands:

```
cd cxf-basic
mvn install
```

## Initialize container security

If you have not already done so, create one (or more users) by adding a line of the following form to the *InstallDir/etc/users.properties* file:

```
Username=Password[,RoleA][,RoleB]...
```

At least one of the users must have the **admin** role, to enable administration of the fabric. For example:

```
admin=secretpassword,admin
```

## Start up the container

Start up the JBoss Fuse container. Open a new command prompt and enter the following commands:

```
cd InstallDir/bin
fuse
```

You will see a welcome screen similar to this:

```

  _ _ _ _ _
  | | _ \
  | | |_) | _ _ _ _ _
  _ | | _ < / _ \ / _ \ / _ \
  | | | | |_) | ( ) \_ \ \_ \ | | | | | \_ \ \_ \
  \_ \ / | \_ \ \_ \ / | \_ \ / | \_ \ \_ \ / | \_ \ \_ \ / | \_ \ \_ \

```

```
JBoss Fuse (6.1.0.redhat-379)
http://www.redhat.com/products/jbossenterprisemiddleware/fuse/
```

```
Hit '<tab>' for a list of available commands
and '[cmd] --help' for help on a specific command.
```

```
Open a browser to http://localhost:8181 to access the management console
```

```
Hit '<ctrl-d>' or 'osgi:shutdown' to shutdown JBoss Fuse.
```

```
JBossFuse:karaf@root
```

## Install prerequisite features

If you are using the *Red Hat JBoss Fuse 6.1.0 Medium Install* package or the *Red Hat JBoss Fuse 6.1.0 Minimal Install* package, you need to install additional Karaf features, as follows:

```
JBossFuse:karaf@root> features:install cxf
JBossFuse:karaf@root> features:install cxf-http-jetty
JBossFuse:karaf@root> features:install cxf-jaxws
```

**NOTE**

These features are pre-installed in the *Red Hat JBoss Fuse 6.1.0 Full Install* package.

**Deploy and start the WS server**

To install the **cxf-basic** Web service as an OSGi bundle, enter the following console command:

```
JBossFuse:karaf@root> install mvn:org.fusesource.example/cxf-basic/1.0-SNAPSHOT
```

**NOTE**

If your local Maven repository is stored in a non-standard location, you might need to customize the value of the **org.ops4j.pax.url.mvn.localRepository** property in the **InstallDir/etc/org.ops4j.pax.url.mvn.cfg** file, before you can use the **mvn:** scheme to access Maven artifacts.

If the bundle is successfully resolved and installed, the container responds by giving you the ID of the newly created bundle—for example:

```
Bundle ID: 229
```

You can now start up the Web service using the **start** console command, specifying the bundle ID, as follows:

```
JBossFuse:karaf@root> start 229
```

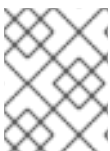
**Check that the bundle has started**

To check that the bundle has started, enter the **list** console command, which gives the status of all the bundles installed in the container:

```
JBossFuse:karaf@root> list
```

Near the end of the listing, you should see a status line like the following:

```
[ 229] [Active      ] [                ] [Started] [    60]
Apache ServiceMix :: CXF Code First OSGi Bundle (1.0.0.SNAPSHOT)
```

**NOTE**

Actually, to avoid clutter, the **list** command only shows the bundles with a start level of 50 or greater (which excludes most of the system bundles).

**Run the WS client**

The **cxf-basic** project also includes a simple WS client, which you can use to test the deployed Web service. In a command prompt, navigate to the **cxf-basic** directory and run the simple WS client as follows:

■

```
cd get-started/cxf-basic
mvn -Pclient
```

If the client runs successfully, you should see output like the following:

```
INFO: Creating Service {http://example.fusesource.org/}PersonService from
class org.fusesource.example.Person
Invoking getPerson...
getPerson._getPerson_personId=Guillaume
getPerson._getPerson_ssn=000-000-0000
getPerson._getPerson_name=Guillaume
```

## Troubleshooting

If you have trouble running the client, there is an even simpler way to connect to the Web service. Open your favorite Web browser and navigate to the following URL to contact the JBoss Fuse Jetty container:

```
http://localhost:8181/cxf?wsdl
```

To query the WSDL directly from the PersonService Web service, navigate to the following URL:

```
http://localhost:8181/cxf/PersonServiceCF?wsdl
```

## 2.2. CREATE A ROUTER PROJECT

### Overview

This section describes how to generate a router project, which acts as a proxy for the WS server described in [Section 2.1, “Create a Web Services Project”](#). The starting point for this project is the **camel-archetype-blueprint** Maven archetype.

### Prerequisites

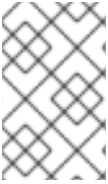
This project depends on the **cxf-basic** project and requires that you have already generated and built the **cxf-basic** project, as described in [Section 2.1, “Create a Web Services Project”](#).

### Create project from the command line

Open a command prompt and change directory to the **get-started** directory. You can now use the **archetype:generate** goal to invoke the **camel-archetype-blueprint** archetype, which generates a simple Apache Camel demonstration, as follows:

```
mvn archetype:generate
-DarchetypeGroupId=org.apache.camel.archetypes
-DarchetypeArtifactId=camel-archetype-blueprint
-DarchetypeVersion=2.12.0.redhat-610379
-DgroupId=org.fusesource.example
-DartifactId=camel-basic
-Dversion=1.0-SNAPSHOT
```



**NOTE**

The arguments of the preceding command are shown on separate lines for readability, but when you are actually entering the command, the entire command *must* be entered on a single line.

You will be prompted to confirm the project settings, with a message similar to this one:

```
[INFO] Using property: groupId = org.fusesource.example
[INFO] Using property: artifactId = camel-basic
[INFO] Using property: version = 1.0-SNAPSHOT
[INFO] Using property: package = org.fusesource.example
[INFO] Using property: camel-version = 2.9.0.fuse-7-032
[INFO] Using property: log4j-version = 1.2.16
[INFO] Using property: maven-bundle-plugin-version = 2.3.4
[INFO] Using property: maven-compiler-plugin-version = 2.3.2
[INFO] Using property: maven-surefire-plugin-version = 2.11
[INFO] Using property: slf4j-version = 1.6.1
Confirm properties configuration:
groupId: org.fusesource.example
artifactId: camel-basic
version: 1.0-SNAPSHOT
package: org.fusesource.example
camel-version: 2.9.0.fuse-7-032
log4j-version: 1.2.16
maven-bundle-plugin-version: 2.3.4
maven-compiler-plugin-version: 2.3.2
maven-surefire-plugin-version: 2.11
slf4j-version: 1.6.1
Y: :
```

Type **Return** to accept the settings and generate the project. When the command finishes, you should find a new Maven project in the **get-started/camel-basic** directory.

**Modify the route**

You are going to modify the default route generated by the archetype and change it into a route that implements a HTTP bridge. This bridge will be interposed between the WS client and Web service, enabling us to apply some routing logic to the WSDL messages that pass through the route.

Using your favorite text editor, open **camel-basic/src/main/resources/OSGI-INF/blueprint/blueprint.xml**. Remove the existing **bean** element and the **camelContext** element and replace them with the **camelContext** element highlighted in the following example:

```
<?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:camel="http://camel.apache.org/schema/blueprint"
  xsi:schemaLocation="
    http://www.osgi.org/xmlns/blueprint/v1.0.0
    http://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd
    http://camel.apache.org/schema/blueprint
    http://camel.apache.org/schema/blueprint/camel-blueprint.xsd">

  <camelContext id="blueprintContext"
```

```

        trace="false"
        xmlns="http://camel.apache.org/schema/blueprint">
    <route id="httpBridge">
        <from uri="jetty:http://0.0.0.0:8282/cxf/PersonServiceCF?
matchOnUriPrefix=true"/>
        <delay><constant>5000</constant></delay>
        <to uri="jetty:http://localhost:8181/cxf/PersonServiceCF?
bridgeEndpoint=true&throwExceptionOnFailure=false"/>
    </route>
</camelContext>

</blueprint>

```

The **from** element defines a new HTTP server port, which listens on IP port 8282. The **to** element defines a HTTP client endpoint that attempts to connect to the real Web service, which is listening on IP port 8181. To make the route a little more interesting, we add a **delay** element, which imposes a five second (5000 millisecond) delay on all requests passing through the route.

For a detailed discussion and explanation of the HTTP bridge, see [Proxying with HTTP](#).

## Disable the test

The generated project includes a built-in unit test, which employs the **camel-test** testing toolkit. The Apache Camel testing toolkit is a useful and powerful testing library, but it will not be used in this example.

To disable the test, open the **RouteTest.java** file from the **src/test/java/org/fusesource/example** directory using a text editor and look for the **@Test** annotation, as shown in the following snippet:

```

// Java
...
public class RouteTest extends CamelBlueprintTestSupport {
    ...
    @Test
    public void testRoute() throws Exception {
    ...

```

Now comment out the **@Test** annotation, as shown in the following snippet, and save the modified **RouteTest.java** file.

```

...
public class RouteTest extends CamelBlueprintTestSupport {
    ...
    // @Test
    // Disable test!
    public void testRoute() throws Exception {
    ...

```

## Add the required Maven dependency

Because the route uses the Apache Camel Jetty component, you must add a Maven dependency on the **camel-jetty** artifact, so that the requisite JAR files are added to the classpath. To add the dependency, edit the **camel-basic/pom.xml** file and add the following highlighted dependency as a

child of the **dependencies** element:

```
<?xml version="1.0" encoding="UTF-8"?>
<project ...>
  ...
  <dependencies>
    <dependency>
      <groupId>org.apache.camel</groupId>
      <artifactId>camel-core</artifactId>
      <version>2.12.0.redhat-610379</version>
    </dependency>
    <dependency>
      <groupId>org.apache.camel</groupId>
      <artifactId>camel-blueprint</artifactId>
      <version>2.12.0.redhat-610379</version>
    </dependency>
    <dependency>
      <groupId>org.apache.camel</groupId>
      <artifactId>camel-jetty</artifactId>
      <version>2.12.0.redhat-610379</version> </dependency>
    ...
  </dependencies>
  ...
</project>
```

## Build the router project

Build the router project and install the generated JAR file into your local Maven repository. From a command prompt, enter the following commands:

```
cd camel-basic
mvn install
```

## Deploy and start the route

If you have not already started the Red Hat JBoss Fuse container and deployed the Web services bundle, you should do so now—see [the section called “Deploy and start the WS server”](#).

To install the **camel-basic** route as an OSGi bundle, enter the following console command:

```
JBossFuse:karaf@root> install mvn:org.fusesource.example/camel-basic/1.0-SNAPSHOT
```

If the bundle is successfully resolved and installed, the container responds by giving you the ID of the newly created bundle—for example:

```
Bundle ID: 230
```

You can now start up the Web service using the **start** console command, specifying the bundle ID, as follows:

```
JBossFuse:karaf@root> start 230
```

## Test the route with the WS client

The **cx**f-**basic** project includes a simple WS client, which you can use to test the deployed route and Web service. In a command prompt, navigate to the **cx**f-**basic** directory and run the simple WS client as follows:

```
cd ../cx-f-basic
mvn -Pclient -Dexec.args="http://localhost:8282/cxf/PersonServiceCF"
```

If the client runs successfully, you should see output like the following:

```
INFO: Creating Service {http://example.fusesource.org/}PersonService from
class org.fusesource.example.Person
Invoking getPerson...
```

After a five second delay, you will see the following response:

```
getPerson._getPerson_personId=Guillaume
getPerson._getPerson_ssn=000-000-0000
getPerson._getPerson_name=Guillaume
```

## 2.3. CREATE AN AGGREGATE MAVEN PROJECT

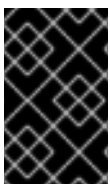
### Aggregate POM

A complete application typically consists of multiple Maven projects. As the number of projects grows larger, however, it becomes a nuisance to build each project separately. Moreover, it is usually necessary to build the projects in a certain order and the developer must remember to observe the correct build order.

To simplify building multiple projects, you can optionally create an aggregate Maven project. This consists of a single POM file (the *aggregate POM*), usually in the parent directory of the individual projects. The POM file specifies which sub-projects (or *modules*) to build and builds them in the specified order.

### Parent POM

Maven also supports the notion of a *parent POM*. A parent POM enables you to define an inheritance style relationship between POMs. POM files at the bottom of the hierarchy declare that they inherit from a specific parent POM. The parent POM can then be used to share certain properties and details of configuration.



#### IMPORTANT

The details of how to define and use a parent POM are beyond the scope of this guide, but it is important to be aware that *a parent POM and an aggregate POM are not the same thing*.

### Best practice

Quite often, you will see examples where a POM is used *both* as a parent POM and an aggregate POM. This is acceptable for small, relatively simple applications, but is not recommended as best practice. *Best practice is to define separate POM files for the parent POM and the aggregate POM.*

## Create an aggregate POM

To create an aggregate POM for your getting started application, use a text editor to create a `pom.xml` file in the `get-started` directory and add the following contents to the file:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/maven-v4_0_0.xsd">

  <groupId>org.fusesource.example</groupId>
  <artifactId>get-started</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>pom</packaging>
  <modelVersion>4.0.0</modelVersion>

  <name>Getting Started :: Aggregate POM</name>
  <description>Getting Started example</description>

  <modules>
    <module>cxf-basic</module>
    <module>camel-basic</module>
  </modules>

</project>
```

As with any other POM, the `groupId`, `artifactId`, and `version` must be defined, in order to identify this artifact uniquely. But the `packaging` must be set to `pom`. The key portion of the aggregate POM is the `modules` element, which defines the list of Maven sub-projects to build *and defines the order in which the projects are built*. The content of each `module` element is the relative path of a directory containing a Maven project.

## Building with the aggregate POM

Using the aggregate POM you can build *all* of sub-projects in one go, by entering the following at a command prompt:

```
cd get-started
mvn install
```

## 2.4. DEFINE A FEATURE FOR THE APPLICATION

### Why do you need a feature?

An OSGi bundle is *not* a convenient unit of deployment to use with the Red Hat JBoss Fuse container. Applications typically consist of multiple OSGi bundles and complex applications may consist of a very

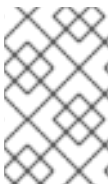
large number of bundles. Usually, you want to deploy or undeploy multiple OSGi bundles at the same time and you need a deployment mechanism that supports this.

Apache Karaf features are designed to address this problem. A feature is essentially a way of aggregating multiple OSGi bundles into a single unit of deployment. When defined as a feature, you can simultaneously deploy or undeploy a whole collection of bundles.

## What to put in a feature

At a minimum, a feature should contain the basic collection of OSGi bundles that make up the core of your application. In addition, you might need to specify some of the dependencies of your application bundles, in case those bundles are not predeployed in the container.

Ultimately, the decision about what to include in your custom feature depends on what bundles and features are predeployed in your container. Using a standardised container like Red Hat JBoss Fuse makes it easier to decide what to include in your custom feature.



### NOTE

If you decide to use Fuse Application Bundles (FABs) instead of OSGi bundles, your feature definitions can typically be much simpler. FABs are capable of finding and installing most of the dependencies that they need.

## Deployment options

You have a few different options for deploying features, as follows:

- *Hot deploy*—the simplest deployment option; just drop the XML features file straight into the hot deploy directory, **installDir/deploy**.
- *Add a repository URL*—you can tell the Red Hat JBoss Fuse container where to find your features repository file using the **features:addUrl** console command (see [Add the local repository URL to the features service](#)). You can then install the feature at any time using the **features:install** console command.
- *Through a Fuse Fabric profile*—you can use the management console to deploy a feature inside a Fuse Fabric profile.

For more details about the feature deployment options, see [Deploying a Feature](#).

## Features and Fuse Fabric

It turns out that a feature is a particularly convenient unit of deployment to use with Fuse Fabric. A Fuse Fabric profile typically consists of a list of features and a collection of related configuration settings. Hence, a Fuse Fabric profile makes it possible to deploy a completely configured application to any container in a single atomic operation.

## Create a custom features repository

Create a sub-directory to hold the features repository. Under the **get-started** project directory, create all of the directories in the following path:

```
get-started/features/src/main/resources/
```

Under the `get-started/features/src/main/resources` directory, use a text editor to create the `get-started.xml` file and add the following contents:

```
<?xml version="1.0" encoding="UTF-8"?>
<features name="get-started">
  <feature name="get-started-basic">
    <bundle>mvn:org.fusesource.example/cxf-basic/1.0-SNAPSHOT</bundle>
    <bundle>mvn:org.fusesource.example/camel-basic/1.0-SNAPSHOT</bundle>
  </feature>
  <feature name="get-started-cxf">
    <bundle>mvn:org.fusesource.example/cxf-basic/1.0-SNAPSHOT</bundle>
  </feature>
</features>
```

Under the `get-started/features/` directory, use a text editor to create the Maven POM file, `pom.xml`, and add the following contents to it:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="
    http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>

  <groupId>org.fusesource.example</groupId>
  <artifactId>get-started</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>

  <name>Getting Started Feature Repository</name>

  <build>
    <plugins>
      <!-- Attach the generated features file as an artifact,
        and publish to the maven repository -->
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>build-helper-maven-plugin</artifactId>
        <version>1.5</version>
        <executions>
          <execution>
            <id>attach-artifacts</id>
            <phase>package</phase>
            <goals>
              <goal>attach-artifact</goal>
            </goals>
            <configuration>
              <artifacts>
                <artifact>
                  <file>target/classes/get-started.xml</file>
                  <type>xml</type>
                  <classifier>features</classifier>
                </artifact>
              </artifacts>
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
```

```

        </artifacts>
      </configuration>
    </execution>
  </executions>
</plugin>
</plugins>
</build>
</project>

```

## Install the features repository

You need to install the features repository into your local Maven repository, so that it can be located by the Red Hat JBoss Fuse container. To install the features repository, open a command prompt, change directory to **get-started/features**, and enter the following command:

```

cd features
mvn install

```

## Deploy the custom feature

To deploy the **get-started-basic** feature into the container, perform the following steps:

1. If the **cxp-basic** and **camel-basic** bundles are already installed in the JBoss Fuse container, you must first uninstall them. At the console prompt, use the **list** command to discover the bundle IDs for the **cxp-basic** and **camel-basic** bundles, and then uninstall them both using the console command, **uninstall *BundleID***.
2. Before you can access features from a features repository, you must tell the container where to find the features repository. Add the features repository URL to the container, by entering the following console command:

```

JBossFuse:karaf@root> features:addurl
mvn:org.fusesource.example/get-started/1.0-SNAPSHOT/xml/features

```

You can check whether the container knows about the new features by entering the console command **features:list**. If necessary, you can use the **features:refreshurl** console command, which forces the container to re-read its features repositories.

3. To install the **get-started-basic** feature, enter the following console command:

```

JBossFuse:karaf@root> features:install get-started-basic

```

4. After waiting a few seconds for the bundles to start up, you can test the application as described in [the section called "Test the route with the WS client"](#).
5. To uninstall the feature, enter the following console command:

```

JBossFuse:karaf@root> features:uninstall get-started-basic

```

## 2.5. CONFIGURE THE APPLICATION



## OSGi Config Admin service

The OSGi Config Admin service is a standard OSGi configuration mechanism that enables administrators to modify application configuration at deployment time and at run time. This contrasts the with settings made directly in a Blueprint XML file, because these XML files are accessible only to the developer.

The OSGi Config Admin service relies on the following basic concepts:

### *Persistent ID*

A persistent ID (PID) identifies a group of related properties. Conventionally, a PID is normally written in the same format as a Java package name. For example, the `org.ops4j.pax.web` PID configures the Red Hat JBoss Fuse container's default Jetty Web server.

### *Properties*

A property is a name-value pair, which always belongs to a specific PID.

## Setting configuration properties

There are two main ways to customise the properties in the OSGi Config Admin service, as follows:

- For a given a PID, *PersistentID*, you can create a text file under the ***InstallDir/etc*** directory, which obeys the following naming convention:

```
InstallDir/etc/PersistentID.cfg
```

You can then set the properties belonging to this PID by editing this file and adding entries of the form:

```
Property=Value
```

- Fuse Fabric supports another mechanism for customising OSGi Config Admin properties. In Fuse Fabric, you set OSGi Config Admin properties in a *fabric profile* (where a profile encapsulates the data required to deploy an application). There are two alternative ways of modifying configuration settings in a profile:
  - Using the management console
  - Using the **`fabric:profile-edit`** command in a container console (see [Section 3.2.2, “Create Fabric Profiles”](#)).

## Replace IP port with a property placeholder

As an example of how the OSGi Config Admin service might be used in practice, consider the IP port used by the **PersonService** Web service from the **cxp-basic** project. By modifying the Spring XML file that defines this Web service, you can make the Web service's IP port customisable through the OSGi Config Admin service.

The IP port number in the Spring XML file is replaced by a property placeholder, which resolves the port number at run time by looking up the property in the OSGi Config Admin service.

## Blueprint XML example

In the **cxf-basic** project, any XML files from the following location are treated as Blueprint XML files (the standard Maven location for Blueprint XML files):

```
cxf-basic/src/main/resources/OSGI-INF/blueprint/*.xml
```

Edit the **beans.xml** file from the preceding directory and add the XML contents shown in [Example 2.2](#), “Configuring the Port Number in Blueprint XML”.

### Example 2.2. Configuring the Port Number in Blueprint XML

```
<?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:jaxws="http://cxf.apache.org/blueprint/jaxws"
  xmlns:cxf="http://camel.apache.org/schema/blueprint/cxf"
  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.example.get.started">
    <cm:default-properties>
      <cm:property name="portNumber" value="8181"/>
    </cm:default-properties>
  </cm:property-placeholder>

  <jaxws:endpoint id="HTTPEndpoint"
    implementor="org.fusesource.example.PersonImpl"
    address="http://0.0.0.0:${portNumber}/PersonServiceCF"/>

</blueprint>
```

The highlighted text shows the parts of the Blueprint configuration that are relevant to the OSGi Config Admin service. Apart from defining the **cm** namespace, the main changes are as follows:

1. The **cm:property-placeholder** bean contacts the OSGi Config Admin service and retrieves all of the property settings from the **org.fusesource.example.get.started** PID. The key-value pairs in the **cm:default-properties/cm:property** elements specify default values for the properties (which are overridden, if corresponding settings can be retrieved from the OSGi Config Admin service).
2. The **\${portNumber}** placeholder is used to specify the IP port number used by the **PersonService** Web service.



#### NOTE

For the Blueprint XML configuration, you must ensure that the instructions for the **maven-bundle-plugin** in the project's **pom.xml** file include the wildcard, **\***, in the packages listed in the **Import-Package** element (if the **Import-Package** element is not present, the wildcard is implied by default). Otherwise, you will get the error: **Unresolved references to [org.osgi.service.blueprint] by class(es) on the Bundle-Classpath[Jar:dot]: []**.

## Deploying the configurable application

To deploy the configurable Web service from the **cxf-basic** project, perform the following steps:

1. Edit the Blueprint XML file, **beans.xml**, to integrate the OSGi Config Admin service, as described in [Example 2.2, “Configuring the Port Number in Blueprint XML”](#).
2. Rebuild the **cxf-basic** project with Maven. Open a command prompt, change directory to the **get-started/cxf-basic** directory, and enter the following Maven command:

```
mvn clean install
```

3. Create the following configuration file in the **etc/** directory of your Red Hat JBoss Fuse installation:

```
InstallDir/etc/org.fusesource.example.get.started.cfg
```

Edit the **org.fusesource.example.get.started.cfg** file with a text editor and add the following contents:

```
portNumber=8182
```

4. If you have previously deployed the **get-started-basic** feature (as described in [Section 2.4, “Define a Feature for the Application”](#)), uninstall it now:

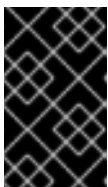
```
JBossFuse:karaf@root> features:uninstall get-started-basic
```

5. Deploy the **get-started-cxf** feature, by entering the following console command:

```
JBossFuse:karaf@root> features:install get-started-cxf
```

6. After waiting a few seconds for the bundles to start up, you can test the application by opening a command prompt, changing directory to **get-started/cxf-basic**, and entering the following command:

```
mvn -Pclient -Dexec.args="http://localhost:8182/PersonServiceCF"
```



### IMPORTANT

The URL in this command has a slightly different format from the URLs used in the previous client commands: the path part of the URL is **/PersonServiceCF**, instead of **/cxf/PersonServiceCF**.

7. To uninstall the feature, enter the following console command:

```
features:uninstall get-started-cxf
```

## 2.6. TROUBLESHOOTING

## Check the status of a deployed bundle

After deploying an OSGi bundle, you can check its status using the `osgi:list` console command. For example:

```
JBossFuse:karaf@root> osgi:list
```

The most recently deployed bundles appear at the bottom of the listing. For example, a successfully deployed `cxf-basic` bundle has a status line like the following:

```
[ 232] [Active      ] [          ] [Started] [ 60]
      Apache ServiceMix :: CXF Code First OSGi Bundle (1.0.0.SNAPSHOT)
```

The second column indicates the status of the OSGi bundle lifecycle (usually **Installed**, **Resolved**, or **Active**). A bundle that is successfully installed and started has the status **Active**. If the bundle contains a Blueprint XML file, the third column indicates whether the Blueprint context has been successfully **Created** or not. If the bundle contains a Spring XML file, the fourth column indicates whether the Spring context has been successfully **Started** or not.

## Logging

If a bundle fails to start up properly, an error message is usually sent to the log. To view the most recent messages from the log, enter the `log:display` console command. Usually, you will be able to find a stack trace for the failed bundle in the log.

You can easily change the logging level using the `log:set` console command. For example:

```
JBossFuse:karaf@root> log:set DEBUG
```

## Redeploying bundles with dev:watch

If there is an error in one of your bundles and you need to redeploy it, the best approach is to use the `dev:watch` command. For example, given that you have already deployed the `cxf-basic` bundle and it has the bundle ID, 232, you can tell the runtime to watch the bundle by entering the following console command:

```
JBossFuse:karaf@root> dev:watch 232
Watched URLs/IDs:
232
```

Now, whenever you rebuild the bundle using Maven:

```
cd cxf-basic
mvn clean install
```

The runtime automatically redeploys the bundle, as soon as it notices that the corresponding JAR in the local Maven repository has been updated. In the console window, the following message appears:

```
[Watch] Updating watched bundle: cxf-basic (1.0.0.SNAPSHOT)
```

## CHAPTER 3. GETTING STARTED WITH DEPLOYING

### Abstract

This chapter introduces the Fuse Fabric technology layer and provides a detailed example of how to deploy an application in a fabric, based on the application developed in [Chapter 2, \*Getting Started with Developing\*](#).

### 3.1. SCALABLE DEPLOYMENT WITH FUSE FABRIC

#### Why Fuse Fabric?

A single Red Hat JBoss Fuse container deployed on one host provides a flexible and sophisticated environment for deploying your applications, with support for versioning, deployment of various package types (OSGi bundle, FAB, WAR), container services and so on. But when you start to roll out a large-scale deployment of a product based on JBoss Fuse, where multiple containers are deployed on multiple hosts across a network, you are faced with an entire new set of challenges. Some of the capabilities typically needed for managing a large-scale deployment are:

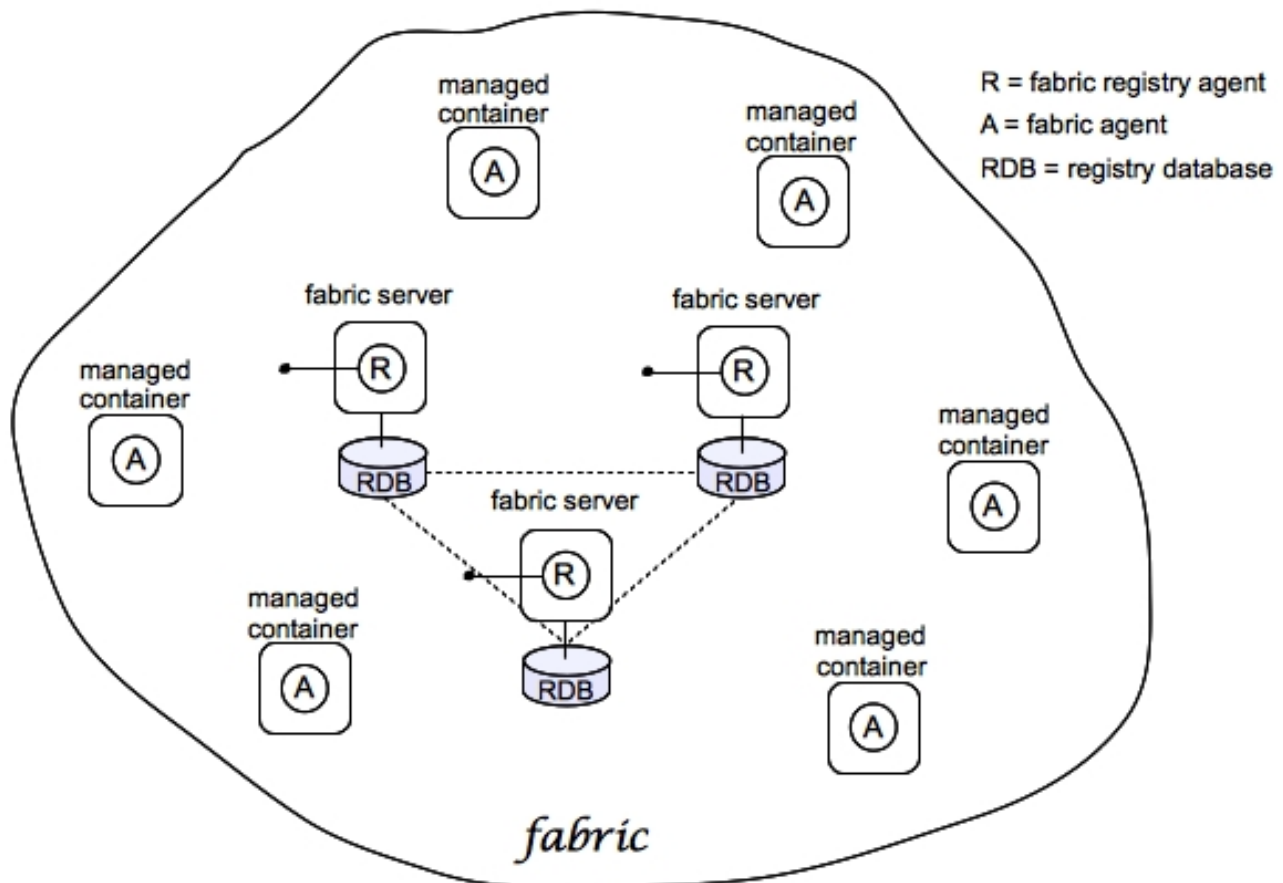
- Monitoring the state of all the containers in the network
- Starting and stopping remote containers
- Provisioning remote containers to run particular applications
- Upgrading applications and rolling out patches in a live system
- Starting up and provisioning new containers quickly—for example, to cope with an increased load on the system

The Fuse Fabric technology layer handles these kinds of challenges in a large-scale production system.

#### A sample fabric

[Figure 3.1, “Containers in a Fabric”](#) shows an example of a distributed collection of containers that belong to a single fabric.

Figure 3.1. Containers in a Fabric



## Fabric

The Fuse Fabric technology layer supports the scalable deployment of JBoss Fuse containers across a network. It enables a variety of advanced features, such as remote installation and provisioning of containers; phased rollout of new versions of libraries and applications; load-balancing and failover of deployed endpoints.

A *fabric* is a collection of containers that share a *fabric registry*, where the fabric registry is a replicated database that stores all information related to provisioning and managing the containers. A fabric is intended to manage a distributed network of containers, where the containers are deployed across multiple hosts.

## Fabric Ensemble

A *Fabric Ensemble* is a collection of Fabric Servers that collectively maintain the state of the fabric registry. The Fabric Ensemble implements a replicated database and uses a [quorum-based voting system](#) to ensure that data in the fabric registry remains consistent across all of the fabric's containers. To guard against network splits in a quorum-based system, it is a requirement that *the number of Fabric Servers in a Fabric Ensemble is always an odd number*.

The number of Fabric Servers in a fabric is typically 1, 3, or 5. A fabric with just one Fabric Server is suitable for experimentation only. A live production system should have at least 3 or 5 Fabric Servers, installed on separate hosts, to provide fault tolerance.

## Fabric Server

A Fabric Server has a special status in the fabric, because it is responsible for maintaining a replica of

the fabric registry. In each Fabric Server, a registry service is installed (labeled R in [Figure 3.1](#), “[Containers in a Fabric](#)”). The registry service (based on Apache ZooKeeper) maintains a replica of the registry database and provides a ZooKeeper server, which ordinary agents can connect to in order to retrieve registry data.

## Fabric Container

A Fabric Container is aware of the locations of all of the Fabric Servers, and it can retrieve registry data from any Fabric Server in the Fabric Ensemble. A *Fabric Agent* (labeled A in [Figure 3.1](#), “[Containers in a Fabric](#)”) is installed in each Fabric Container. The Fabric Agent actively monitors the fabric registry, and whenever a relevant modification is made to the registry, it immediately updates its container to keep the container consistent with the registry settings.

## Profile

A *Fabric profile* is an abstract unit of deployment, which is capable of holding all of the data required for deploying an application into a Fabric Container. Profiles are used exclusively in the context of fabrics. Features or bundles deployed directly to Fabric Containers are short lived.



### IMPORTANT

The presence of a Fabric Agent in a container completely changes the deployment model, *requiring you to use profiles exclusively* as the unit of deployment. Although it is still possible to deploy an individual bundle or feature (using `osgi:install` or `features:install`, respectively), these modifications are impermanent. As soon as you restart the container or refresh its contents, the Fabric Agent replaces the container's existing contents with whatever is specified by the deployed profiles.

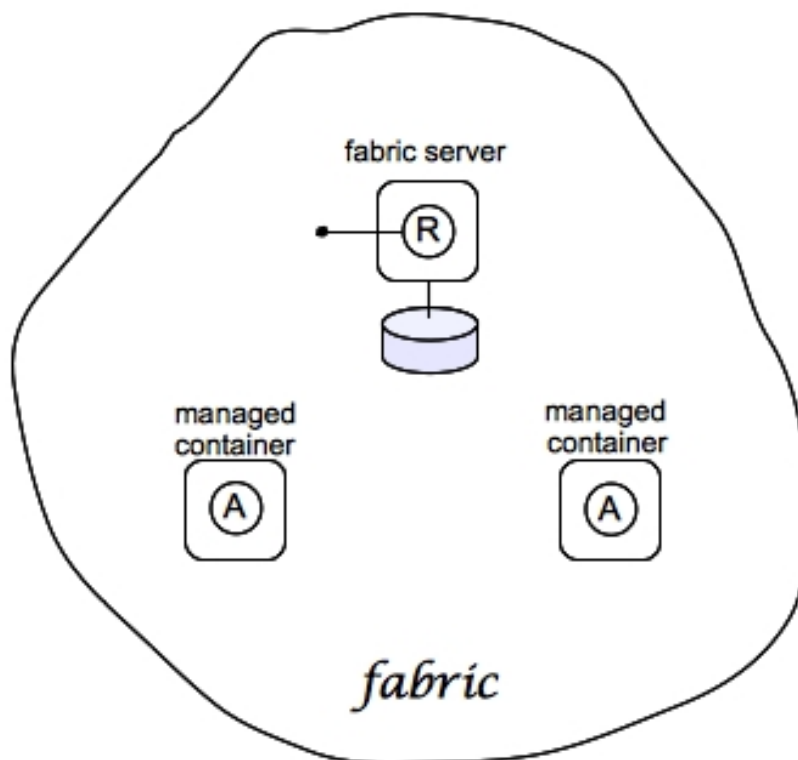
## 3.2. DEPLOYING TO A FABRIC

### 3.2.1. Create a Fabric

#### Overview

[Figure 3.2](#) shows an overview of a sample fabric that you will create. The Fabric Ensemble consists of just one Fabric Server (making this fabric suitable only for experimental use) and two managed child containers.

Figure 3.2. A Sample Fabric with Child Containers



### Fabric server

A Fabric Server (or servers) forms the backbone of a fabric. It hosts a registry service, which maintains a replicable database of information about the state of the fabric. Initially, when you create the fabric, there is just a single Fabric Server.

### Child containers

Creating one or more child containers is the simplest way to extend a fabric. As shown in [Figure 3.2, “A Sample Fabric with Child Containers”](#), the first container in the fabric is a root container, and both child containers are descended from it.

Each child container is an independent Red Hat JBoss Fuse container instance, which runs in its own JVM instance. The data files for the child containers are stored under the ***InstallDir/instances*** directory.

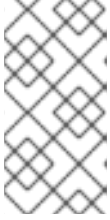
### Steps to create the fabric

To create the simple fabric shown in [Figure 3.2, “A Sample Fabric with Child Containers”](#), follow these steps:

1. (Optional) Customise the name of the root container by editing the ***InstallDir/etc/system.properties*** file and specifying a different name for this property:

```
karaf.name=root
```



**NOTE**

For the first container in your fabric, this step is optional. But at some later stage, if you want to join a root container to the fabric, you must customise the new container's name to prevent it from clashing with any existing root containers in the fabric.

- To create the first fabric container, which acts as the seed for the new fabric, enter this console command:

```
JBossFuse:karaf@root> fabric:create --new-user AdminUser --new-user-
password AdminPass
--zookeeper-password ZooPass --wait-for-provisioning
```

The current container, named **root** by default, becomes a Fabric Server with a registry service installed. Initially, this is the only container in the fabric. The **--new-user** and **--new-user-password** options specify the credentials for a new administrator user. The Zookeeper password is used to protect sensitive data in the Fabric registry service (all of the nodes under **/fabric**).

**NOTE**

Most of the time, you are not prompted to enter the Zookeeper password when accessing the registry service, because it is cached in the current session. When you *join a container to a fabric*, however, you must provide the fabric's Zookeeper password.

**NOTE**

If you use a VPN (virtual private network) on your local machine, it is advisable to log off VPN *before* you create the fabric and to stay logged off while you are using the local container. A local Fabric Server is permanently associated with a fixed IP address or hostname. If VPN is enabled when you create the fabric, the underlying Java runtime is liable to detect and use the VPN hostname instead of your permanent local hostname. This can also be an issue with multi-homed machines. To be absolutely sure about the hostname, you could specify the IP address explicitly—see [chapter "Creating a New Fabric" in "Fabric Guide"](#).

- Create two child containers. Assuming that your root container is named **root**, enter this console command:

```
JBossFuse:karaf@root> fabric:container-create-child root child 2
Jmx Login for root: admin
Jmx Password for root:
The following containers have been created successfully:
Container: child1.
Container: child2.
```

You are prompted to enter a JMX username and password (the JMX port is integrated with the JAAS authentication service). Enter one of the username/password combinations that you defined in step 2.

- Invoke the **fabric:container-list** command to see a list of all containers in your new fabric. You should see a listing something like this:

```
JBossFuse:karaf@root> fabric:container-list
[id]                [version] [alive] [profiles]
[provision status]
root                1.0      true  fabric, fabric-
ensemble-0000-1, fuse-esb-full success
  child1            1.0      true  default
success
  child2            1.0      true  default
success
```

## Shutting down the containers

Because the child containers run in their own JVMs, they do *not* automatically stop when you shut down the root container. To shut down a container and its children, first stop its children using the **fabric:container-stop** command. For example, to shut down the current fabric completely, enter these console commands:

```
JBossFuse:karaf@root> fabric:container-stop child1
JBossFuse:karaf@root> fabric:container-stop child2
JBossFuse:karaf@root> shutdown -f
```

After you restart the root container, you must explicitly restart the children using the **fabric:container-start** console command.

### 3.2.2. Create Fabric Profiles

#### Overview

A profile is the basic unit of deployment in a fabric. You can deploy one or more profiles to a container, and the content of those deployed profiles determines what is installed in the container.

#### Contents of a profile

A profile encapsulates the following kinds of information:

- The URL locations of features repositories
- A list of features to install
- A list of bundles to install (or, more generally, any suitable JAR package—including OSGi bundles, Fuse Application Bundles, and WAR files)
- A collection of configuration settings for the OSGi Config Admin service
- Java system properties that affect the Apache Karaf container (analogous to editing **etc/config.properties**)
- Java system properties that affect installed bundles (analogous to editing **etc/system.properties**)

## Base profile

Profiles support inheritance. This can be useful in cases where you want to deploy a cluster of similar servers—for example, where the servers differ only in the choice of IP port number. For this, you would typically define a *base profile*, which includes all of the deployment data that the servers have in common. Each individual server profile would inherit from the common base profile, but add configuration settings specific to its server instance.

## Create a base profile

To create the **gs-cxf-base** profile, follow these steps:

1. Create the **gs-cxf-base** profile by entering this console command:

```
JBossFuse:karaf@root> fabric:profile-create --parents feature-cxf
gs-cxf-base
```

2. Add the **get-started** features repository (see [Define a Feature for the Application](#)) to the **gs-cxf-base** profile by entering this console command:

```
JBossFuse:karaf@root> profile-edit -r
mvn:org.fusesource.example/get-started/1.0-SNAPSHOT/xml/features gs-
cxf-base
```

3. Now add the **get-started-cxf** feature (which provides the Web service example server) to the **gs-cxf-base** profile. Enter the following console command:

```
JBossFuse:karaf@root> profile-edit --features get-started-cxf gs-
cxf-base
```

## Create the derived profiles

You create two derived profiles, **gs-cxf-01** and **gs-cxf-02**, which configure different IP ports for the Web service. To do so, follow these steps:

1. Create the **gs-cxf-01** profile—which derives from **gs-cxf-base**—by entering this console command:

```
JBossFuse:karaf@root> profile-create --parents gs-cxf-base gs-cxf-01
```

2. Create the **gs-cxf-02** profile—which derives from **gs-cxf-base**—by entering this console command:

```
JBossFuse:karaf@root> profile-create --parents gs-cxf-base gs-cxf-02
```

3. In the **gs-cxf-01** profile, set the **portNumber** configuration property to 8185, by entering this console command:

```
JBossFuse:karaf@root> profile-edit -p
org.fusesource.example.get.started/portNumber=8185 gs-cxf-01
```

4. In the **gs-cxf-02** profile, set the **portNumber** configuration property to 8186, by entering this console command:

```
JBossFuse:karaf@root> profile-edit -p  
org.fusesource.example.get.started/portNumber=8186 gs-cxf-02
```

### 3.2.3. Deploy the Profiles

#### Deploy profiles to the child containers

Having created the child containers, as described in [Section 3.2.1, “Create a Fabric”](#), and the profiles, as described in [Section 3.2.2, “Create Fabric Profiles”](#), you can now deploy the profiles. To do so, follow these steps:

1. Deploy the **gs-cxf-01** profile into the **child1** container by entering this console command:

```
JBossFuse:karaf@root> fabric:container-change-profile child1 gs-cxf-  
01
```

2. Deploy the **gs-cxf-02** profile into the **child2** container by entering this console command:

```
JBossFuse:karaf@root> fabric:container-change-profile child2 gs-cxf-  
02
```

#### Test the deployed profiles

You can test the deployed profiles using the WS client from the **cxf-basic** Maven project described in [Create a Web Services Project](#). To do so, follow these steps:

1. Open a new command prompt and **cd** to **get-started/cxf-basic**.
2. Test the **gs-cxf-01** profile (which deploys a Web service listening on port 8185) by entering this command:

```
mvn -Pclient -Dexec.args="http://localhost:8185/PersonServiceCF"
```

3. Test the **gs-cxf-02** profile (which deploys a Web service listening on port 8186) by entering this command:

```
mvn -Pclient -Dexec.args="http://localhost:8186/PersonServiceCF"
```

### 3.2.4. Update a Profile

#### Atomic container upgrades

Normally, when you edit a profile that is already deployed in a container, *the modification takes effect immediately*. This is so because the Fabric Agent in the affected container (or containers) actively monitors the fabric registry in real time.

In practice, however, immediate propagation of profile modifications is often undesirable. In a production system, you typically want to roll out changes incrementally: for example, initially trying out the change

on just one container to check for problems, before you make changes globally to all containers. Moreover, sometimes several edits must be made together to reconfigure an application in a consistent way.

## Profile versioning

For quality assurance and consistency, it is typically best to modify profiles *atomically*, where several modifications are applied simultaneously. To support atomic updates, fabric implements profile versioning. Initially, the container points at version 1.0 of a profile. When you create a new profile version (for example, version 1.1), the changes are invisible to the container until you upgrade it. After you are finished editing the new profile, you can apply all of the modifications simultaneously by upgrading the container to use the new version 1.1 of the profile.

## Upgrade to a new profile

For example, to modify the **gs-cxf-01** profile, when it is deployed and running in a container, follow the recommended procedure:

1. Create a new version, 1.1, to hold the pending changes by entering this console command:

```
JBossFuse:karaf@root> fabric:version-create  
Created version: 1.1 as copy of: 1.0
```

The new version is initialised with a copy of all of the profiles from version 1.0.

2. Use the **fabric:profile-edit** command to change the **portNumber** of **gs-cxf-01** to the value 8187 by entering this console command:

```
JBossFuse:karaf@root> fabric:profile-edit -p  
org.fusesource.example.get.started/portNumber=8187 gs-cxf-01 1.1
```

Remember to specify version **1.1** to the **fabric:profile-edit** command, so that the modifications are applied to version 1.1 of the **gs-cxf-01** profile.

3. Upgrade the **child1** container to version 1.1 by entering this console command:

```
JBossFuse:karaf@root> fabric:container-upgrade 1.1 child1
```

## Roll back to an old profile

You can easily roll back to the old version of the **gs-cxf-01** profile, using the **fabric:container-rollback** command like this:

```
JBossFuse:karaf@root> fabric:container-rollback 1.0 child1
```

# APPENDIX A. EDITING PROFILES WITH THE BUILT-IN TEXT EDITOR

## Abstract

When you have a lot of changes and additions to make to a profile's configuration, it is usually more convenient to do this interactively, using the built-in text editor for profiles. The editor can be accessed by entering the `profile-edit` command with no arguments except for the profile's name (and optionally, version); or adding the `--pid` option for editing OSGi PID properties; or adding the `--resource` option for editing general resources.

## A.1. EDITING AGENT PROPERTIES

### Overview

This section explains how to use the built-in text editor to modify a profile's *agent properties*, which are mainly used to define what bundles and features are deployed by the profile.

### Open the agent properties resource

To start editing a profile's agent properties using the built-in text editor, enter the following console command:

```
JBossFuse:karaf@root> profile-edit Profile [Version]
```

Where ***Profile*** is the name of the profile to edit and you can optionally specify the profile version, *Version*, as well. The text editor opens in the console window, showing the current profile name and version in the top-left corner of the Window. The bottom row of the editor screen summarizes the available editing commands and you can use the arrow keys to move about the screen.

### Specifying feature repository locations

To specify the location of a feature repository, add a line in the following format:

```
repository.ID=URL
```

Where ***ID*** is an arbitrary unique identifier and ***URL*** gives the location of a single feature repository (only one repository URL can be specified on a line).

### Specifying deployed features

To specify a feature to deploy (which must be available from one of the specified feature repositories), add a line in the following format:

```
feature.ID=FeatureName
```

Where ***ID*** is an arbitrary unique identifier and ***FeatureName*** is the name of a feature.

### Specifying deployed bundles

To specify a bundle to deploy, add a line in the following format:

```
bundle.ID=URL
```

Where **ID** is an arbitrary unique identifier and **URL** specifies the bundle's location.



#### NOTE

A bundle entry can be used in combination with a **blueprint:** (or **spring:**) URL handler to deploy a Blueprint XML resource (or a Spring XML resource) as an OSGi bundle.

## Specifying bundle overrides

To specify a bundle override, add a line in the following format:

```
override.ID=URL
```

Where **ID** is an arbitrary unique identifier and **URL** specifies the bundle's location.



#### NOTE

A bundle override is used to override a bundle installed by a feature, replacing it with a different version of the bundle. For example, this functionality is used by the patching system to install a patched bundle in a container.

## Specifying deployed FABs

To specify a Fuse Application Bundle (FAB) to deploy, add a line in the following format:

```
fab.ID=URL
```

Where **ID** is an arbitrary unique identifier and **URL** specifies the FAB's location.

## Specifying etc/config.properties properties

To specify Java system properties that affect the Apache Karaf container (analogous to editing **etc/config.properties** in a standalone container), add a line in the following format:

```
config.Property=Value
```

## Specifying etc/system.properties properties

To specify Java system properties that affect the bundles deployed in the container (analogous to editing **etc/system.properties** in a standalone container), add a line in the following format:

```
system.Property=Value
```

If the system property, **Property**, is already set at the JVM level (for example, through the **--jvm-opts** option to the **fabric:container-create** command), the preceding **fabric:profile-edit** command *will not override the JVM level setting*. To override a JVM level setting, set the system property

as follows:

```
system.karaf.override.Property=Value
```

### Specifying libraries to add to Java runtime lib/

To specify a Java library to deploy (equivalent to adding a library to the **lib/** directory of the underlying Java runtime), add a line in the following format:

```
lib.ID=URL
```

Where **ID** is an arbitrary unique identifier and **URL** specifies the library's location.

### Specifying libraries to add to Java runtime lib/ext/

To specify a Java extension library to deploy (equivalent to adding a library to the **lib/ext/** directory of the underlying Java runtime), add a line in the following format:

```
ext.ID=URL
```

Where **ID** is an arbitrary unique identifier and **URL** specifies the extension library's location.

### Specifying libraries to add to Java runtime lib/endorsed/

To specify a Java endorsed library to deploy (equivalent to adding a library to the **lib/endorsed/** directory of the underlying Java runtime), add a line in the following format:

```
endorsed.ID=URL
```

Where **ID** is an arbitrary unique identifier and **URL** specifies the endorsed library's location.

## Example

To open the **mq-client** profile's agent properties for editing, enter the following console command:

```
JBossFuse:karaf@root> profile-edit mq-client
```

The text editor starts up, and you should see the following screen in the console window:

```
Profile:mq-client 1.0
L:1 C:1
#
# Copyright (C) Red Hat, Inc.
# http://redhat.com
#
# Licensed 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.
#

repository.activemq=mvn:org.apache.activemq/activemq-
karaf/${version:activemq}/xml/features
repository.karaf-
standard=mvn\org.apache.karaf.assemblies.features/standard/${version:kara
f}/
xml/features

      ^X Quit      ^S Save      ^Z Undo      ^R Redo      ^G Go To      ^F Find
^N Next      ^P Previous
```

Type **^X** to quit the text editor and get back to the console prompt.

## A.2. EDITING OSGI CONFIG ADMIN PROPERTIES

### Overview

This section explains how to use the built-in text editor to edit the property settings associated with a specific persistent ID.

### Persistent ID

In the context of the OSGi Config Admin service, a *persistent ID* (PID) refers to and identifies a set of related properties. In particular, when defining PID property settings in a profile, the properties associated with the **PID** persistent ID are defined in the **PID.properties** resource.

### Open the Config Admin properties resource

To start editing the properties associated with the **PID** persistent ID, enter the following console command:

```
JBossFuse:karaf@root> profile-edit --pid PID Profile [Version]
```



#### NOTE

It is also possible to edit PID properties by specifying **--resource PID.properties** in the **profile-edit** command, instead of using the **--pid PID** option.

### Specifying OSGi config admin properties

The text editor opens, showing the contents of the specified profile's **PID.properties** resource (which is actually stored in the ZooKeeper registry). To edit the properties, add, modify, or delete lines of the following form:

```
Property=Value
```

## Example

To edit the properties for the `io.fabric8.hadoop` PID in the `hadoop-base` profile, enter the following console command:

```
JBossFuse:karaf@root> profile-edit --resource io.fabric8.hadoop.properties
hadoop-base 1.0
```

The text editor starts up, and you should see the following screen in the console window:

```
Profile:hadoop-base 1.0
L:1 C:1
#
# Copyright (C) Red Hat, Inc.
# http://redhat.com
#
# Licensed 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.
#

fs.default.name=hdfs\://localhost\:9000
dfs.replication=1
mapred.job.tracker=localhost\:9001
dfs.name.dir=${karaf.data}/hadoop/dfs/name
dfs.http.address=0.0.0.0\:9002
dfs.data.dir=${karaf.data}/hadoop/dfs/data
dfs.name.edits.dir=${karaf.data}/hadoop/dfs/name

      ^X Quit      ^S Save      ^Z Undo      ^R Redo      ^G Go To      ^F Find
^N Next      ^P Previous
```

You might notice that colon characters are escaped in this example (as in `\:`). Strictly speaking, it is only necessary to escape a colon if it appears as part of a property name (left hand side of the equals sign), but the `profile-edit` command automatically escapes all colons when it writes to a resource. When manually editing resources using the text editor, however, you do not need to escape colons in URLs appearing on the right hand side of the equals sign.

Type `^X` to quit the text editor and get back to the console prompt.

## A.3. EDITING OTHER RESOURCES

### Overview

In addition to agent properties and PID properties, the built-in text editor makes it possible for you edit *any* resource associated with a profile. This is particularly useful, if you need to store additional

configuration files in a profile. The extra configuration files can be stored as profile resources (which actually correspond to ZooKeeper nodes) and then can be accessed by your applications at run time.



## NOTE

The ZooKeeper registry is designed to work with small nodes only. If you try to store a massive configuration file as a profile resource, it will severely degrade the performance of the Fuse Fabric registry.

## Creating and editing an arbitrary resource

You can create and edit arbitrary profile resources using the following command syntax:

```
JBossFuse:karaf@root> profile-edit --resource Resource Profile [Version]
```

Where **Resource** is the name of the profile resource you want to edit. If **Resource** does not already exist, it will be created.

### broker.xml example

For example, the **mq-base** profile has the **broker.xml** resource, which stores the contents of an Apache ActiveMQ broker configuration file. To edit the **broker.xml** resource, enter the following console command:

```
JBossFuse:karaf@root> profile-edit --resource broker.xml mq-base 1.0
```

The text editor starts up, and you should see the following screen in the console window:

```
Profile:mq-base 1.0
L:1 C:1
<!--
  Copyright (C) FuseSource, Inc.
  http://fusesource.com

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

```

    <!-- Allows us to use system properties and fabric as variables in
this configuration file -->
    <bean
class="org.springframework.beans.factory.config.PropertyPlaceholderConfigu
rer">
        <property name="properties">
            <bean
class="org.fusesource.mq.fabric.ConfigurationProperties"/>
        </property>
    </bean>
^X Quit      ^S Save      ^Z Undo      ^R Redo      ^G Go To      ^F Find
^N Next      ^P Previous

```

Any changes you make to this file will take effect whenever the broker restarts.

Type `^X` to quit the text editor and get back to the console prompt.

## Referencing a profile resource

In order to use an arbitrary profile resource, you must be able to reference it. Because a profile resource is ultimately stored as a ZooKeeper node, you must reference it using a ZooKeeper URL. For example, the **broker.xml** resource from the previous example is stored under the following ZooKeeper location:

```
zk:/fabric/configs/versions/1.0/profiles/mq-base/broker.xml
```

In general, you can find version, **Version**, of the **Profile** profile's **Resource** resource at the following location:

```
zk:/fabric/configs/versions/Version/profiles/Profile/Resource
```

For example, the **mq** profile's **org.fusesource.mq.fabric.server-broker** PID defines the following properties, where the **config** property references the **broker.xml** resource:

```

connectors=openwire
config=zk\:/fabric/configs/versions/1.0/profiles/mq-base/broker.xml
group=default
standby.pool=default

```

## A.4. PROFILE ATTRIBUTES

### Overview

In addition to the resources described in the other sections, a profile also has certain attributes that affect its behavior. *You cannot edit profile attributes directly using the text editor.*

For completeness, this section describes what the profile attributes are and what console commands you can use to modify them.

### parents attribute

The **parents** attribute is a list of one or more parent profiles. This attribute can be set using the **profile-change-parents** console command. For example, to assign the parent profiles **camel** and **cx** to the **my-camel-cxf-profile** profile, you would enter the following console command:

```
JBossFuse:karaf@root> profile-change-parents --version 1.0 my-camel-cxf-  
profile camel cxf
```

## abstract attribute

When a profile's **abstract** attribute is set to **true**, the profile cannot be directly deployed to a container. This is useful for profiles that are only intended to be the parents of other profiles—for example, **mq-base**. You can set the abstract attribute from the Management Console.

## locked attribute

A locked profile cannot be changed or edited until it is unlocked. You can lock or unlock a profile from the Management Console.

## hidden attribute

The **hidden** attribute is a flag that is typically set on profiles that Fuse Fabric creates automatically (for example, to customize the setup of a registry server). By default, hidden profiles are not shown when you run the **profile-list** command, but you can see them when you add the **--hidden** flag, as follows:

```
JBossFuse:karaf@root> profile-list --hidden  
...  
fabric 1 karaf  
fabric-ensemble-0000 0  
fabric-ensemble-0000-1 1 fabric-ensemble-0000  
fmc 0 default  
...
```