Red Hat JBoss Fuse 6.3

Developing and Deploying Applications

In-depth examples of how to create, build, and run JBoss Fuse applications

Last Updated: 2020-04-17

Red Hat JBoss Fuse 6.3 Developing and Deploying Applications

In-depth examples of how to create, build, and run JBoss Fuse applications

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

Legal Notice

Abstract

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

Table of Contents

CHAPTER 1. RED HAT JBOSS FUSE OVERVIEW

Abstract

Red Hat JBoss Fuse is an open source Enterprise Service Bus (ESB) that focuses on mediating, transforming, and routing data across multiple applications, services, or devices for both internal systems and external services. By implementing the Open Source Gateway initiative (OSGi) specification it allows bundles, or modules of functionality, to be loosely coupled and highly reusable; in addition, bundles may be remotely installed, started, stopped, updated, and uninstalled without rebooting, and multiple versions of each bundle may run simultaneously.

Though no canonical definition of an ESB exists, David Chappell states in his book Enterprise Server Bus,

> An ESB is a standards-based integration platform that combines messaging, web services, data transformation, and intelligent routing to reliably connect and coordinate the interaction of significant numbers of diverse applications across extended enterprises with transactional integrity.
>
> --David A. Chapell

An ESB simplifies the complexity of integration by providing a single, standards-based infrastructure into which applications can be plugged. Once plugged into the ESB, an application or service has access to all of the infrastructure services provided by the ESB and can access any other applications that are also plugged into the ESB. For example, you could plug a billing system based on JMS into an ESB and use the ESBs transport mediation features to expose the billing system over the Web using SOAP/HTTP. You could also route internal purchase orders directly into the billing system by plugging the Purchase Order system into the ESB.

1.1. COMPONENTS

Apache Camel

Red Hat JBoss Fuse utilizes Apache Camel for building integration and routing solutions using Enterprise Integration Patterns (EIPs). Each route defines an integration path between endpoints, where a system can either send or receive messages, and while data is in these routes it may be transformed, validated, intercepted, and so on. Routes may change endpoints dynamically, either as additional bundles are activated, based on the content of the message, or through additional methods. With over 140 defined endpoints Red Hat JBoss Fuse allows for integration with a variety of services immediately upon deployment.

For further reading see the Apache Camel Development Guide .

Apache CXF

Red Hat JBoss Fuse's embedded Web and RESTful services framework is based on Apache CXF, and provides a small footprint engine for creating reusable web services along with service-enabling new and legacy applications as part of an integration solution. Red Hat JBoss Fuse supports a variety of standards and protocols for creating web services, such as SOAP and WSDL among others, and allows for Contract-first or Code-first development with JAX-WS. In addition, Red Hat JBoss Fuse provides a standard way to build RESTful services in Java with JAX-RS.

For further reading see the Apache CXF Development Guide .

Apache ActiveMQ

Red Hat JBoss Fuse's embedded messaging service is based on Apache ActiveMQ. It supports the standard JMS 1.1 features and provides a wide range of extended JMS features for building robust and reliable messaging applications. Red Hat JBoss Fuse consists of both a messaging broker and client-side libraries that enable remote communication among distributed client applications. Red Hat JBoss Fuse supports Point-to-Point and Publish/Subscribe messaging along with both persistent and nonpersistent messages; in addition, ActiveMQ can be scaled both vertically and horizontally to allow for processing of a large volume of messages for a large number of concurrently connected clients.

For further reading, see Red Hat JBoss A-MQ Product Introduction.

Fabric8

Fuse Fabric is a technology layer that allows a group of containers to form a cluster that shares a common set of configuration information and a common set of repositories from which to access runtime artifacts. This allows one to run a number of containers either on your own hardware or in the open hybrid cloud, and allows for configuration management, service discovery failover, load balancing, centralized monitoring among other benefits.

For further reading see Fabric Guide.

Switchyard

Switchyard is a lightweight service delivery framework providing full life-cycle support for developing, deploying, and managing service-oriented applications. It allows you to deploy and run services with limited dependencies, and consists of components such as composite services and composite references.

For further reading, see Switchyard Development Guide: Application Basics.

1.2. CONTAINERS

Apache Karaf

Red Hat JBoss Fuse is based on Apache Karaf, a powerful, lightweight, OSGi-based runtime container for deploying and managing bundles to facilitate componentization of applications. Red Hat JBoss Fuse also provides native OS integration and can be integrated into the operating system as a service so that the lifecycle is bound to the operating system. Furthermore, Red Hat JBoss Fuse extends the OSGi layers with an extensible console for managing applications and administering instances, a unified logging subsystem supported by Log4J, both manual and hot deployment of OSGi bundles, and multiple mechanisms for installing applications and libraries among others.

For further reading see Apache Karaf.

JBoss Enterprise Application Platform

Red Hat JBoss EAP 6 is a JEE certified container that leverages a flexible, modular architecture, and it integrates EJB components, web services, security, and clustering. By utilizing a JEE container you have full access to JEE components such as persistence and the injection framework.

For further reading, see Chapter 5, *Getting Started with Red Hat JBoss Fuse on EAP* .

1.3. USE CASES

1.3.1. Major Widgets Use Case

1.3.1.1. Major Widgets Introduction

Major Widgets Overview

Major Widgets is a single-store auto-parts company that recently decided to purchase three more auto part supply stores and are currently needing to integrate the systems located in all four stores.

Major Widgets Business Model

Major Widgets, and each of the three stores it bought, routinely supply a number of auto repair shops that are located near them. Each store delivers parts to customers free-of-charge, as long as the customer is located within twenty-five miles of the store. Each store has its own database for storing auto repair customer accounts, store inventory, and part suppliers.

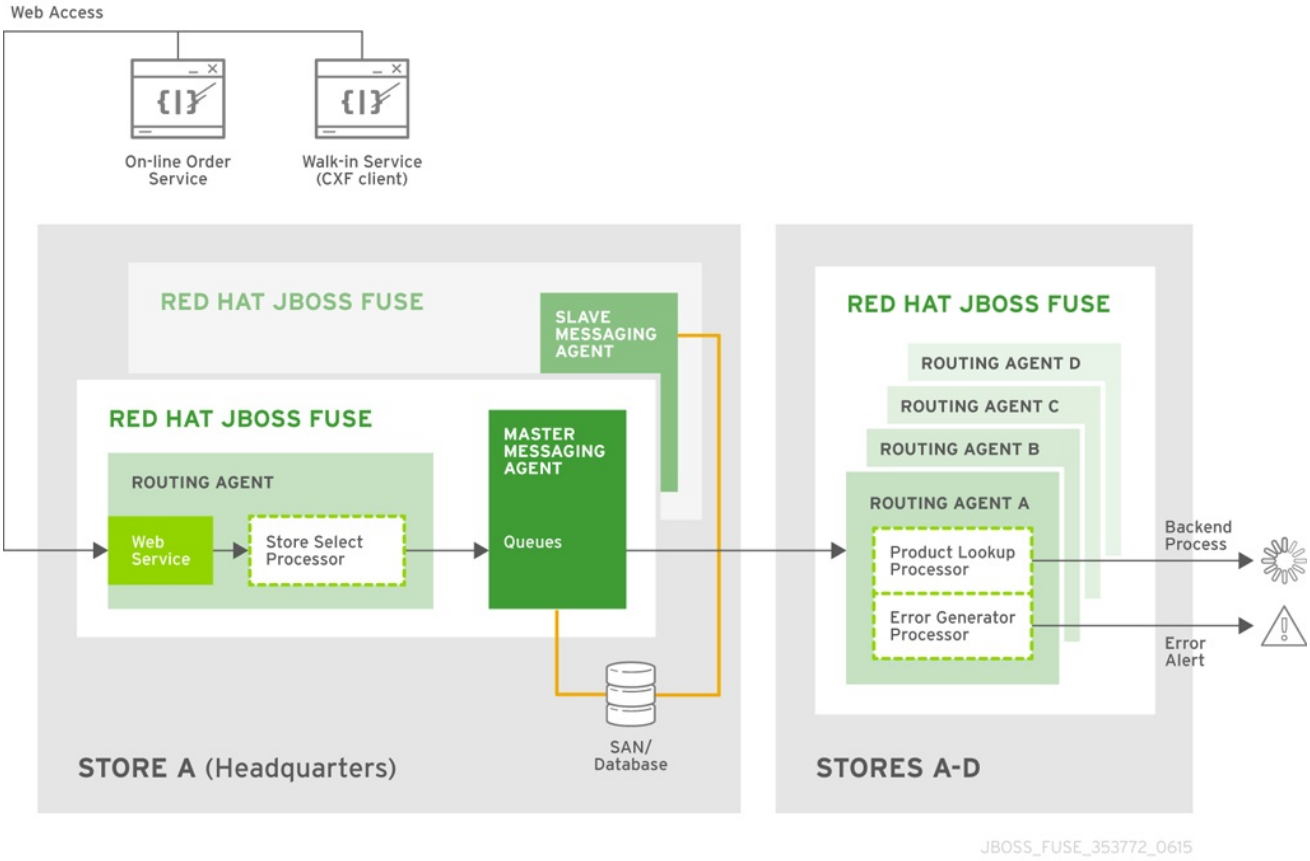
Business was done over the phone, but now Major Widgets wants to implement an online order service to enable their auto repair customers to order parts more quickly and efficiently. The Web-based service will coordinate orders and deliveries, bill customers, track and update each store's inventory, and order parts from suppliers. Customers can use it to check the status of their orders.

All four stores also sell parts over-the-counter to walk-in customers, for whom they do not typically establish customer accounts. Each of the in-store ordering systems will also tie into its store's central order processing system.

1.3.1.2. Major Widgets Integration Plan

Figure 1.1, “Major Widgets Integration Plan” shows a high-level view of how Red Hat JBoss Fuse will provide an integration solution to implement Major Widgets' new business model.

Figure 1.1. Major Widgets Integration Plan



This plan creates:

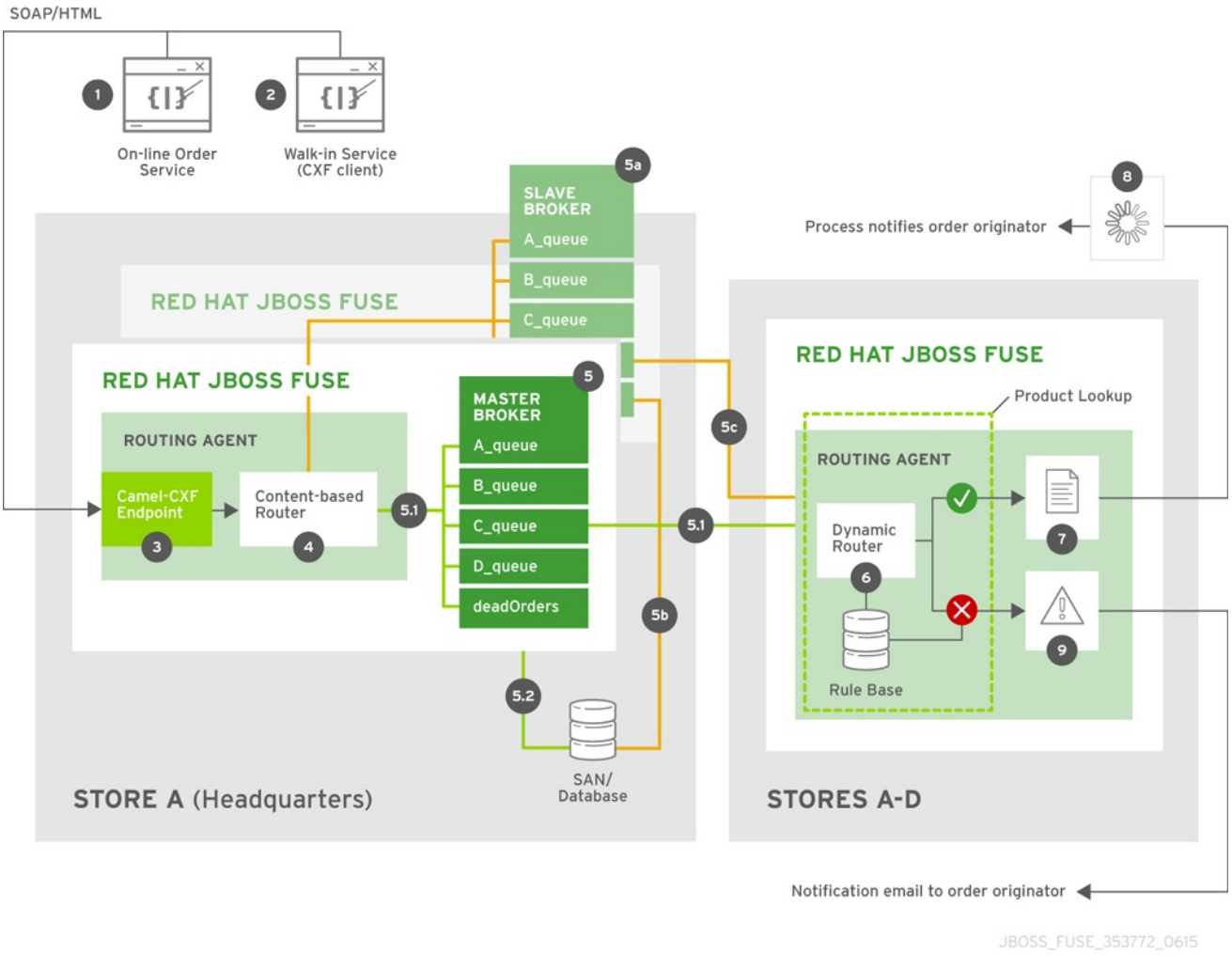
- a single order entry point into the order processing system that can be accessed via the Web and by the in-store terminals
- an intelligent order entry system that routes Web-based orders to the store closest to the delivery destination
- an order processing system (instances running locally at each store) that receives and processes orders, maintains customer accounts, and tracks and maintains inventory
- a master/slave broker cluster that provides a highly available, reliable messaging backbone for the integration solution

This plan allows each store to retain their existing internal systems, but enables them to function as a single unit.

1.3.1.3. Major Widgets Implementation

Figure 1.2, “Major Widgets Implementation Diagram” shows how a Major Widgets integration plan might be implemented.

Figure 1.2. Major Widgets Implementation Diagram



Major Widgets Components

The Red Hat JBoss Fuse kernel provides a runtime environment that provides enterprise support (management, logging, provisioning, security) for the main store **(Store A)**, where most of the integration applications run. Its embedded services provide the frameworks for implementing these components of the solution:

- RESTful service—for creating a JAX-RS application that runs on each auto repair shop terminal **(1)**, enabling customers to input part orders, via an order entry form, over the internet.
- Web service—for creating a JAX-WS front end to implement the order entry functionality on each of the in-store terminals, which receive orders from walk-in customers **(2)** who purchase parts over-the-counter.
- **camel-cxf** component—a routing and integration service component that creates an entry endpoint **(3)** that exposes Major Widgets routing logic to the outside world as a web service or a RESTful service.
- Routing and integration service—for creating routes **(4, 6)** that direct orders received from the web/RESTful service entry point through the appropriate store's order processing back end.
- Messaging service—for creating a persistent, fault-tolerant clustered messaging system **(5, 5a)**, which ensures that no order is ever lost due to failure of the system, the message broker, or the connections between the message broker and its various clients—the front end content-based router **(4)** and the back end dynamic router **(6)**.

Major Widgets Integration Flow

At Major Widgets main store **(Store A)**, the order entry front end (routing and messaging agents running inside Red Hat JBoss Fuse) is running on that store's main computer system. At each of the four stores **(Stores A-D)**, an instance of the order entry back end (routing agent and back end processing running in JBoss Fuse) is running on the local computer system.

When the front end web service **(3)** receives an online order, the routing agent passes it to a content-based router **(4)** to determine which store to route the part order for further processing. Normally, the order then enters the target store's queue **(5)**, where it waits until the target store retrieves it **(6)**. (With fault tolerance built into the system, if the master broker **(5)** fails, the system can continue to function with no loss of orders.)

In the case of auto repair shops **(1)**, the content-based router routes order requests to the store nearest the customer, based on the submitted zip code. In the case of walk-in customers **(2)**, the auto supply store submits its own zip code to the front end, so the order is always routed to the local store.

When the back end receives the submitted part order, the application employs a dynamic router **(6)** to look up the parts in the store's database to see if they are in stock. Results depend on whether the customer is an auto repair shop or a walk-in:

- Auto repair show customers

  If the parts are available, the order is submitted to the store's back end processing software **(8)**, which informs and bills the customer **(1)**, schedules delivery, updates inventory, and reorders parts accordingly.

  If the parts are unavailable, the order is submitted to a processor that generates an error message, which is emailed **(9)** to the customer **(1)**.

- Walk-in customers

  If the parts are available, the order is submitted to the store's back end processing software **(8)**, which informs the store clerk **(2)**, updates inventory, and orders parts accordingly. The store clerk retrieves the parts from stock and sells them to the customer over-the-counter.

  If the parts are unavailable, the order is submitted to a processor that generates an error message, which is emailed **(9)** to the local store's email account **(2)**. The store clerk informs the customer, who can then decide whether he wants the store clerk to search the other stores for his parts.

1.3.2. Loans Consolidated Use Case

1.3.2.1. Loans Consolidated Introduction

Loans Consolidated Overview

Loans Consolidated is a new company that focuses on consolidating other vendors' home and customer information and compare this with local schools to allow customers and vendors to view a variety of homes and compare them.

Loans Consolidated Business Model

Loans Consolidated will take in customer data and home information from several home loan vendors; these vendors will regularly provide information concerning new homes and customers along with updated information for existing entries.

Customers will be able to view a home's appraisal online, and this appraisal will be calculated by Loans Consolidated based on the home's information along with the number of surrounding schools.

The home loans vendors have requested that a service be provided that returns all of the data with the updated appraisal value back to them.

1.3.2.2. Loans Consolidated Integration Plan

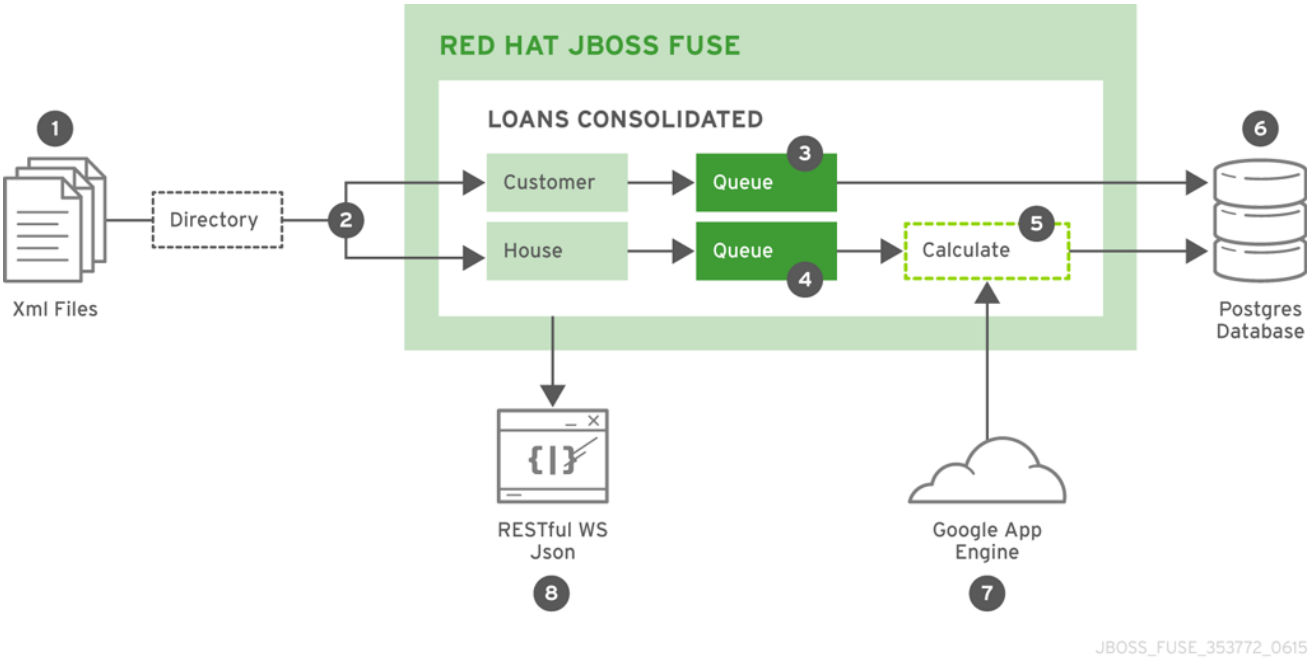
A high-level plan utilizing Red Hat JBoss Fuse will provide an integration solution to implement Loans Consolidated's new business model; this solution creates:

- a single entry point into the order processing system where files are deposited either via a FTP server or a batch job overnight.
- an intelligent system that routes the XML files and, for house files, appraises the value of the house before sending it to a messaging broker.
- a system that retrieves information from the surrounding area to provide a better appraisal.
- the ability to provide the results of the appraisal back to the vendors.

1.3.2.3. Loans Consolidated Implementation

Figure 1.3, "Loans Consolidated Implementation Diagram" shows how a Loans Consolidated plan might be implemented.

Figure 1.3. Loans Consolidated Implementation Diagram



Loans Consolidated Components

The Red Hat JBoss Fuse kernel provides a runtime environment that provides enterprise support (management, logging, provisioning, security), and its embedded services provide the framework for implementing these components:

- Routing and integration service-for creating routes that dynamically examine the contents of the deposited XML files to determine the appropriate destination.
- Integration with the Google App Engine to pull the number of surrounding schools that will be used to update each home's appraised value.
- RESTful service-for providing all of the data with the updated appraisal back to the vendors.

Loans Consolidated Integration Flow

Multiple vendors will be placing their XML files **(1)** into a directory either via a FTP server or a batch process overnight. These files will be read into a content-based router **(2)** which will separate the files based on if they contain Customer or House information.

Once separated the files containing Customer information will be placed into an in-memory queue **(3)** before being passed into a backend database for persistence **(6)**.

The files containing House information will be placed into a separate queue **(4)** before having their value estimated **(5)**. As part of this calculation the location is provided to the Google App Engine **(7)** which will look up nearby schools to determine an appraised value. The appraised value is then stored in the backend database **(6)**.

A RESTful Web Service **(8)** is used to relay the information from the database in a JSon format, so vendors may easily query it.

CHAPTER 2. 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.

2.1. DEVELOPMENT ENVIRONMENT

JDK

The basic requirement for development with Red Hat JBoss Fuse is the Java Development Kit (JDK) from Oracle. For details of which JDK version is supported and for more platform-specific details, see the Supported Configurations page.

Apache Maven

The recommended build system for developing JBoss Fuse applications is Apache Maven version 3.0.x. See "Installation on Apache Karaf", Maven Repositories.

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 2.3, "Maven Essentials" for more details.

Red Hat JBoss Fuse Tooling for Eclipse

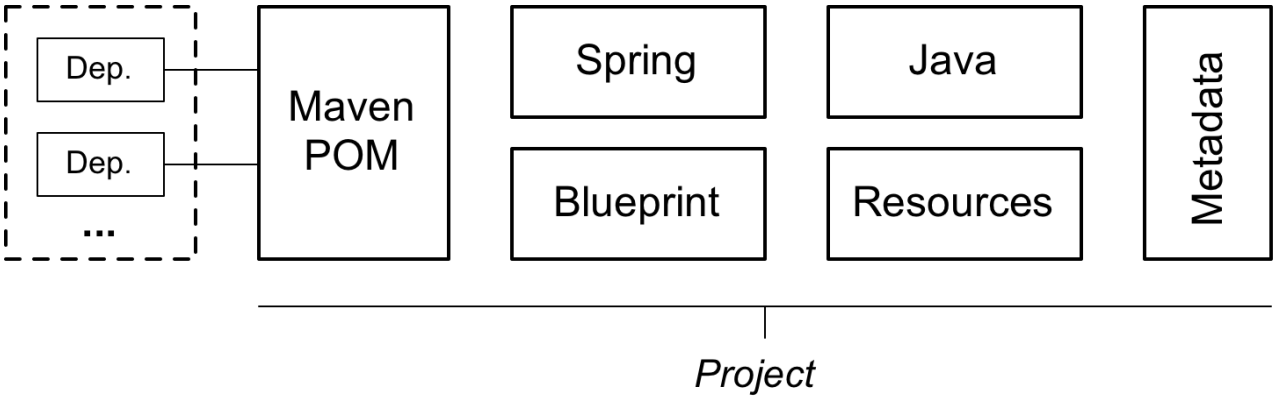
Red Hat JBoss Fuse Tooling for Eclipse is an eclipse-based development tool for developing Red Hat JBoss Fuse applications and is available either as a standalone binary or as an Eclipse plug-in. Using Red Hat JBoss Fuse Tooling for Eclipse, you can quickly create new JBoss Fuse projects with the built-in project wizard and then edit Apache Camel routes with the drag-and-drop graphical UI. For instructions on how to install the tooling, see "Installation on Apache Karaf".

2.2. DEVELOPMENT MODEL

Overview

Figure 2.1, "Developing a 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 2.1. Developing a 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 access to the Internet, when building with Maven. See the section called "Maven repositories".

Maven archetypes

An easy way to get started with development is by using *Maven archetypes*, which is analogous to a new project wizard (except that it must be invoked from the command line). A Maven archetype typically creates a complete new Maven project, with the correct directory layout and some sample code. For example, see Section 3.1, "Create a Web Services Project" and Section 3.2, "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 kind 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 3.3, "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 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 Karaf) 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 2.4, “Dependency Injection Frameworks” .

Deployment metadata

Depending on how a project is packaged and deployed (as an OSGi bundle, 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 for an OSGi bundle (in bundle headers).

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

The POM file is normally embedded in any Maven-built JAR file.

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 bundle 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 TCP port numbers on a server).

2.3. 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 2.1, "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 2.1. 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/
```

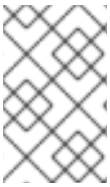
Blueprint XML files (matching the pattern ***.xml**) should be stored under the following directory:

```
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 JAR files and is the default packaging type in Maven.

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 chapter "Building with Maven" in "Deploying into Apache Karaf".

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>2.17.0.redhat-630xxx</version>
      <scope>provided</scope>
    </dependency>
  </dependencies>
  ...
</project>
```

dependency element

The **dependency** element declares a dependency on the Maven artifact with coordinates **org.apache.camel:camel-blueprint:6.3.0.redhat-xxx**. 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.

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 chapter "Building with Maven" in "Deploying into Apache Karaf".

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**.

2.4. 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. By 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 "Customize the Web client test message".

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 section "Dependency Injection Frameworks" in "Deploying into Apache Karaf".
- *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 section "Dependency Injection Frameworks" in "Deploying into Apache Karaf".

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

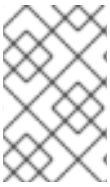
The following example shows the bare outline of a blueprint XML file:

```
<?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 3.1, “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 3. 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.

3.1. 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 **karaf-soap-archetype** 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 and the **pluginRepository** element for the Maven Red Hat repository, as shown in the following example:

```
<?xml version="1.0"?>
<settings>

  <profiles>
    <profile>
      <id>extra-repos</id>
      <activation>
        <activeByDefault>true</activeByDefault>
      </activation>
      <repositories>
       <repository>
          <id>redhat-ga-repository</id>
          <url>https://maven.repository.redhat.com/ga</url>
          <releases>
            <enabled>true</enabled>
          </releases>
          <snapshots>
            <enabled>false</enabled>
          </snapshots>
       </repository>
       <repository>
          <id>redhat-ea-repository</id>
          <url>https://maven.repository.redhat.com/earlyaccess/all</url>
```

```
        <releases>
          <enabled>true</enabled>
        </releases>
        <snapshots>
          <enabled>false</enabled>
        </snapshots>
      </repository>
      <repository>
        <id>jboss-public</id>
        <name>JBoss Public Repository Group</name>
        <url>https://repository.jboss.org/nexus/content/groups/public/</url>
      </repository>
    </repositories>
    <pluginRepositories>
      <pluginRepository>
        <id>redhat-ga-repository</id>
        <url>https://maven.repository.redhat.com/ga</url>
        <releases>
          <enabled>true</enabled>
        </releases>
        <snapshots>
          <enabled>false</enabled>
        </snapshots>
      </pluginRepository>
      <pluginRepository>
        <id>redhat-ea-repository</id>
        <url>https://maven.repository.redhat.com/earlyaccess/all</url>
        <releases>
          <enabled>true</enabled>
        </releases>
        <snapshots>
          <enabled>false</enabled>
        </snapshots>
      </pluginRepository>
      <pluginRepository>
        <id>jboss-public</id>
        <name>JBoss Public Repository Group</name>
        <url>https://repository.jboss.org/nexus/content/groups/public</url>
      </pluginRepository>
    </pluginRepositories>
  </profile>
</profiles>

<activeProfiles>
  <activeProfile>extra-repos</activeProfile>
</activeProfiles>

</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 **karaf-soap-archetype** archetype, which generates a simple Apache CXF demonstration, as follows:

```
mvn archetype:generate \
-DarchetypeGroupId=io.fabric8.archetypes \
-DarchetypeArtifactId=karaf-soap-archetype \
-DarchetypeVersion=1.2.0.redhat-630xxx \
-DgroupId=org.fusesource.example \
-DartifactId=cxf-basic \
-Dversion=1.0-SNAPSHOT \
-Dfabric8-profile=cxf-basic-profile
```

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
[INFO] Using property: fabric8-profile = cxf-basic-profile
Confirm properties configuration:
groupId: org.fusesource.example
artifactId: cxf-basic
version: 1.0-SNAPSHOT
package: org.fusesource.example
fabric8-profile: cxf-basic-profile
 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/cxf-basic** directory.

Customize the Web client test message

Customize the sample client test message, so that it uses the correct XML namespace. Edit the **cxf-basic/src/test/resources/request.xml** file, replacing the **xmlns:ns2="http://soap.quickstarts.fabric8.io/"** namespace setting by **xmlns:ns2="http://example.fusesource.org/"**.

After editing the **request.xml** file, the contents should look like the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
   <soap:Body>
      <ns2:sayHi xmlns:ns2="http://example.fusesource.org/">
         <arg0>John Doe</arg0>
```

```
        </ns2:sayHi>
    </soap:Body>
</soap:Envelope>
```

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 **Administrator** role, to enable administration of the fabric. For example:

```
admin=secretpassword,Administrator
```

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 (6.3.0.redhat-xxx)
  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>
```

Deploy and start the WS server

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

```
JBossFuse:karaf@root> install -s 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: 265
```

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:

```
[ 265] [Active     ] [Created     ] [       ] [   80] JBoss Fuse Quickstart: soap (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 -Ptest
```

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

```
Running org.fusesource.example.SoapTest
the response is ====>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"><soap:Body>
<ns2:sayHiResponse xmlns:ns2="http://example.fusesource.org/"><return>Hello John Doe</return>
```

```
</ns2:sayHiResponse></soap:Body></soap:Envelope>
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.184 sec - in
org.fusesource.example.SoapTest
```

Troubleshooting

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

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

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

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

3.2. 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 3.1, “Create a Web Services Project” . The starting point for this project is the **karaf-camel-cbr-archetype** 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 3.1, “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 **karaf-camel-cbr-archetype** archetype, which generates a simple Apache Camel demonstration, as follows:

```
mvn archetype:generate \
-DarchetypeGroupId=io.fabric8.archetypes \
-DarchetypeArtifactId=karaf-camel-cbr-archetype \
-DarchetypeVersion=1.2.0.redhat-630xxx \
-DgroupId=org.fusesource.example \
-DartifactId=camel-basic \
-Dversion=1.0-SNAPSHOT \
-Dfabric8-profile=camel-basic-profile
```

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: fabric8-profile = camel-basic-profile
Confirm properties configuration:
groupId: org.fusesource.example
artifactId: camel-basic
version: 1.0-SNAPSHOT
package: org.fusesource.example
fabric8-profile: camel-basic-profile
 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.

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:

```
<project ...>
    ...
    <dependencies>
        <dependency>
            <groupId>org.apache.camel</groupId>
            <artifactId>camel-blueprint</artifactId>
        </dependency>
        <dependency>
            <groupId>org.apache.camel</groupId>
            <artifactId>camel-jetty</artifactId>
        </dependency>
        ...
    </dependencies>
    ...
</project>
```

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 favourite text editor, open **camel-basic/src/main/resources/OSGI-INF/blueprint/cbr.xml**. Remove the existing **camelContext** element and replace it 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
https://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/HelloWorld?matchOnUriPrefix=true"/>
      <delay><constant>5000</constant></delay>
      <to uri="jetty:http://localhost:8181/cxf/HelloWorld?
bridgeEndpoint=true&amp;throwExceptionOnFailure=false"/>
    </route>
  </camelContext>

</blueprint>
```

The **from** element defines a new HTTP server port, which listens on TCP port 8282. The **to** element defines a HTTP client endpoint that attempts to connect to the real Web service, which is listening on TCP 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.

Change the port in the Web client test

By default, the Web client connects *directly* to the Web server on port 8181. In order to test the HTTP bridge, however, we want the Web client to connect to the Jetty port exposed by the HTTP bridge, which listens on port 8282. Hence, we need to edit the following file in the **cxf-basic** project:

```
cxf-basic/src/test/java/org/fusesource/example/SoapTest.java
```

Open the **SoapTest.java** file in your favourite text editor, and search for the following line:

```
URLConnection connection = new URL("http://localhost:8181/cxf/HelloWorld").openConnection();
```

Change the port number in this line from 8181 to 8282, as highlighted in the following extract:

```
URLConnection connection = new URL("http://localhost:8282/cxf/HelloWorld").openConnection();
```

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
```

Install the camel-jetty feature

Install the required **camel-jetty** feature as follows:

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

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 and start up the **camel-basic** route as an OSGi bundle, enter the following console command:

```
JBossFuse:karaf@root> install -s 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
```

Test the route with the WS client

The **cxf-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 **cxf-basic** directory and run the simple WS client as follows:

```
cd ../cxf-basic
mvn -Ptest
```

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

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running org.fusesource.example.SoapTest
```

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

```
the response is ====>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"><soap:Body>
<ns2:sayHiResponse xmlns:ns2="http://example.fusesource.org/"><return>Hello John Doe</return>
</ns2:sayHiResponse></soap:Body></soap:Envelope>
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 5.153 sec - in
org.fusesource.example.SoapTest
```

3.3. 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*.

Recommended 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. In general, it is better 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
```

3.4. 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.

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 Into Apache Karaf, Deploying Features.

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:

```
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>

</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 **cxf-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 **cxf-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
```

3.5. 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 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 4.2.2, "Create Fabric Profiles").

Replace TCP port with a property placeholder

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

The TCP port number in the Blueprint 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 **blueprint.xml** file from the preceding directory and add or modify the highlighted content shown in Example 3.1, “Configuring the Port Number in Blueprint XML” .

Example 3.1. 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://cxf.apache.org/blueprint/core"
           xmlns:cm="http://aries.apache.org/blueprint/xmlns/blueprint-cm/v1.1.0"
           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">

    <cxf:bus>
        <!--
            In this example, we're enabling the logging feature.  This will ensure that both the inbound
and outbound
            XML message are being logged for every web service invocation.
        -->
        <cxf:features>
            <cxf:logging/>
        </cxf:features>
    </cxf:bus>

    <!-- 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="helloWorld"
                    implementor="org.fusesource.example.HelloWorldImpl"
                    address="http://0.0.0.0:${portNumber}/cxf/HelloWorld">
    </jaxws:endpoint>

</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 TCP port number used by the **HelloWorld** 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, **blueprint.xml**, to integrate the OSGi Config Admin service, as described in Example 3.1, “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 3.4, “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. Deploy the **cxf-commands** feature, by entering the following console command:

```
JBossFuse:karaf@root> features:install cxf-commands
```

7. After waiting a few seconds for the bundles to start up, you can check the port used by the HelloWorld service, by entering the following console command:

```
JBossFuse:karaf@root> cxf:list-endpoints
Name                    State      Address                                    BusID
[HelloWorldImplPort     ] [Started ] [http://0.0.0.0:8182/cxf/HelloWorld                ]
[org.fusesource.example.cxf-basic-cxf1456001875]
```

   You can see from this that the HelloWorld service is listening on port **8182**.

8. If you want to run the Web client test against this Web service, you must customize the URL used by the client. Using a text editor, open the **SoapTest.java** file from the **cxf-basic/src/test/java/org/fusesource/example** directory, and change the connection URL as highlighted in the following fragment:

```
URLConnection connection = new
URL("http://localhost:8182/cxf/HelloWorld").openConnection();
```

9. You can then test the application by opening a command prompt, changing directory to **get-started/cxf-basic**, and entering the following command:

```
mvn -Ptest
```

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

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

3.6. 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]
    Fabric8 :: 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 4. 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 3, *Getting Started with Developing*.

4.1. 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:

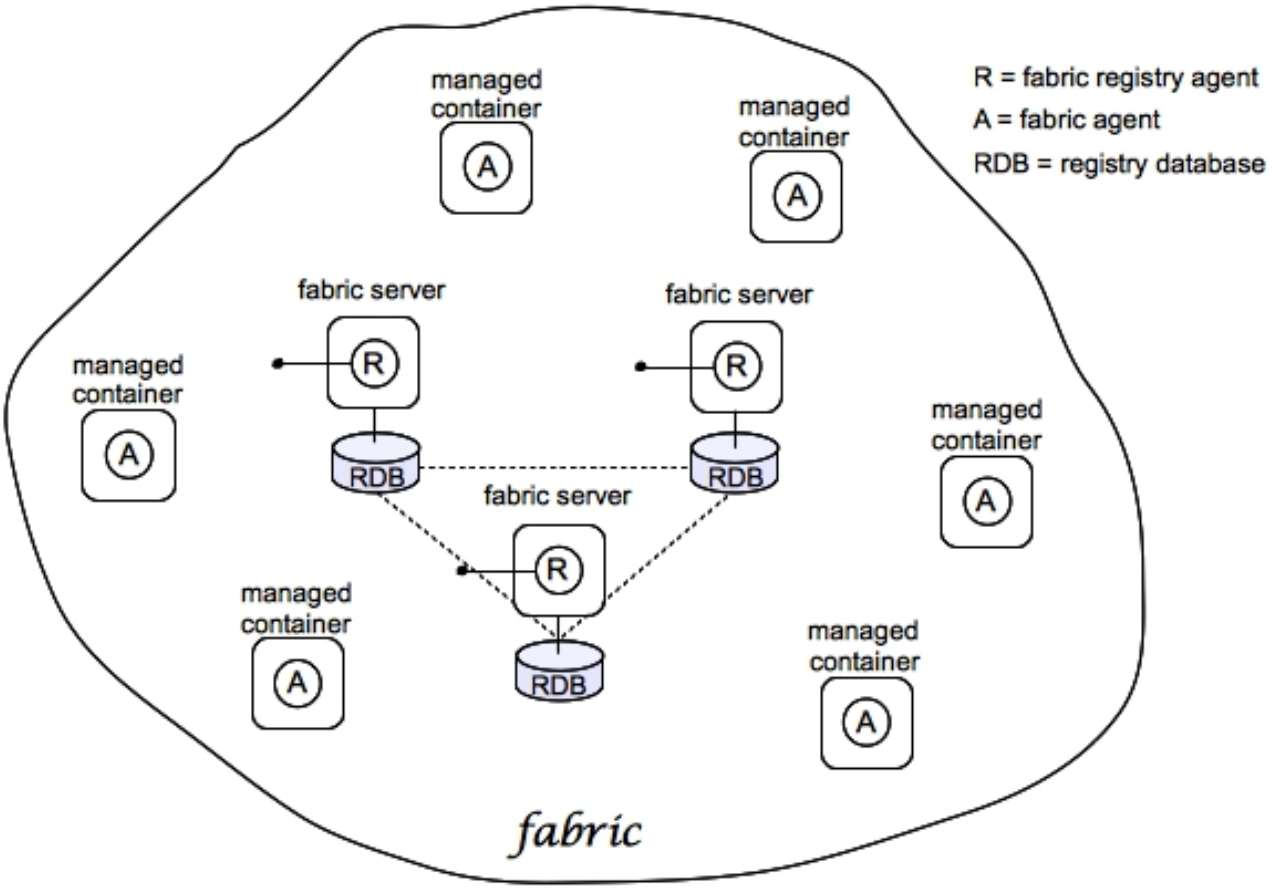
- Monitoring the state of all the containers in the network
- Starting and stopping remote containers
- Provisioning remote containers to run particular applications
- Upgrading applications and rolling out patches in a live system
- Starting up and provisioning new containers quickly—for example, to 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 4.1, "Containers in a Fabric" shows an example of a distributed collection of containers that belong to a single fabric.

Figure 4.1. 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 4.1, "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 4.1, "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.

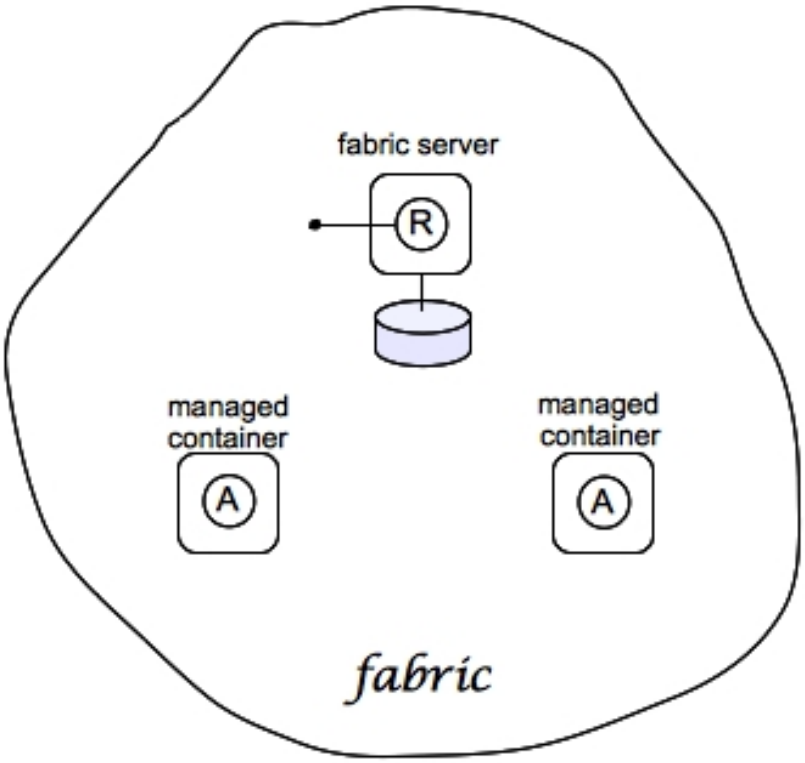
4.2. DEPLOYING TO A FABRIC

4.2.1. Create a Fabric

Overview

Figure 4.2 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 4.2. 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 4.2, "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.

Make Quickstart Examples Available

The default behavior is that profiles for quickstart examples are not available in a new fabric. To create a fabric in which you can run the quickstart examples, edit the **$FUSE_HOME/fabric/io.fabric8.import.profiles.properties** file by uncommenting the line that starts with the following:

 # importProfileURLs =

If you create a fabric without doing this and you want to run the quickstart examples, follow these steps to make them available:

1. Edit the **$FUSE_HOME/quickstarts/pom.xml** file to add a fabric I/O plugin, for example:

```
<plugin>
    <groupId>io.fabric8</groupId>
```

```
    <artifactId>fabric8-maven-plugin</artifactId>
    <version>1.2.0.redhat-630187</version>
</plugin>
```

2. In the **$FUSE_HOME/quickstarts** directory, change to the directory for the quickstart example you want to run, for example:

   cd beginner

3. In that directory, execute the following command:

   mvn fabric8:deploy

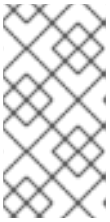
   You would need to run this command in each directory that contains a quickstart example that you want to run.

Steps to create the fabric

To create the simple fabric shown in Figure 4.2, "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.

2. 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
     --new-user-role Administrator
     --resolver manualip
     --manual-ip 127.0.0.1
     --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**, **--new-user-password**, and **--new-user-role** 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

It is recommended that you assign a static IP address to the machine that hosts a Fabric server and that you specify this port explicitly using the **--resolver** and **--manual-ip** options when you create the fabric. For simple tests and demonstrations, you can specify the loopback address, **127.0.0.1** (as here). For more details, see chapter "Creating a New Fabric" in "Fabric Guide" .

3. 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
Creating new instance on SSH port 8102 and RMI ports 1100/44445 at:
/home/jdoe/Programs/JBossFuse/jboss-fuse-6.2.0.redhat-123/instances/child2
Creating new instance on SSH port 8103 and RMI ports 1101/44446 at:
/home/jdoe/Programs/JBossFuse/jboss-fuse-6.2.0.redhat-123/instances/child
The following containers have been created successfully:
 Container: child1.
 Container: child2.
```

If you are prompted to enter a JMX username and password, enter one of the username/password combinations that you defined in step 2.

4. 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]  [type]  [connected]  [profiles]            [provision status]
root*    1.0        karaf   yes          fabric                success
                                         fabric-ensemble-0000-1
                                         jboss-fuse-full
  child1  1.0        karaf   yes          default               success
  child2  1.0        karaf   yes          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
```

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

4.2.2. 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 TCP 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 --parent 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. Add the **cxf-http-jetty** feature (which provides support for the HTTP Jetty endpoint) to the **gs-cxf-base** profile. Enter the following console command:

   ```
   JBossFuse:karaf@root> profile-edit --feature cxf-http-jetty gs-cxf-base
   ```

4. 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 --feature get-started-cxf gs-cxf-base
```

5. Add the **cxf-commands** feature (which makes the CXF console commands available) to the **gs-cxf-base** profile. Enter the following console command:

```
JBossFuse:karaf@root> profile-edit --feature cxf-commands gs-cxf-base
```

Create the derived profiles

You create two derived profiles, **gs-cxf-01** and **gs-cxf-02**, which configure different TCP 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 --parent 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 --parent 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
```

4.2.3. Deploy the Profiles

Deploy profiles to the child containers

Having created the child containers, as described in Section 4.2.1, "Create a Fabric", and the profiles, as described in Section 4.2.2, "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
```

Check that the Web service is running

To check that the Web service has successfully launched on the **child** container, perform the following steps:

1. If the child container is not already running, start it by entering the following command:

   ```
   JBossFuse:karaf@root> container-start child1
   ```

2. Wait until the **child1** container has finished starting up. You can observe the provisioning status using the **watch** command, as follows:

   ```
   JBossFuse:karaf@root> watch container-list
   ```

3. Connect to the child container, as follows:

   ```
   JBossFuse:karaf@root> container-connect child1
   ```

4. After connecting to the **child** container, list the active CXF endpoints, by entering the following command:

   ```
   JBossFuse:admin@child1> cxf:list-endpoints
   Name                  State      Address                                    BusID
   [HelloWorldImplPort     ] [Started ] [http://0.0.0.0:8185/cxf/HelloWorld                ]
   [org.fusesource.example.cxf-basic-cxf481246446]
   ```

4.2.4. Update a Profile

Upgrading containers atomically

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
```

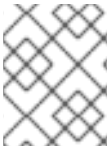
CHAPTER 5. GETTING STARTED WITH RED HAT JBOSS FUSE ON EAP

Abstract

Apache Camel in JBoss Fuse enables you to to develop an integrated application in your own way. This chapter explains how to get started with Red Hat JBoss Fuse on EAP. It illustrates several ways of developing Camel applications on EAP.

5.1. INTEGRATING APACHE CAMEL WITH JBOSS EAP

Red Hat JBoss Fuse supports Apache Camel as an EAP subsystem. Integration of Camel with EAP allows you to add Camel routes as part of the EAP configuration. You can deploy routes as a part of Java EE applications.

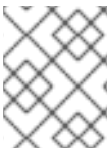


NOTE

Apache Camel is adaptable and does not force you to deploy into any particular container or JVM technology. You can choose your preferred container.

5.2. EXAMPLES OF JBOSS FUSE ON EAP

This section includes the working examples that demonstrate various features of JBoss on Fuse EAP. These examples will help you get started with the EAP Camel subsystem.



NOTE

You can access the **$JBOSS_HOME/quickstarts/camel** directory to view the full source code of all the examples.

To run the given examples, ensure that you install the following on your machine:

- Maven 3.2.3 or greater
- Java 1.7 or greater
- Red Hat JBoss Fuse 6.3
- Red Hat JBoss EAP 6.4

NOTE

To install JBoss Fuse on EAP, see chapter "Install JBoss Fuse on JBoss EAP" in "Installation on JBoss EAP".

5.2.1. Camel ActiveMQ

The following example describes how to use the ***camel-activemq*** component with JBoss Fuse on EAP, to produce and consume JMS messages.

In this example, a camel route consumes files from the **${JBOSS_HOME}/standalone/data/orders**

directory and place the content to an external ActiveMQ JMS queue. A second route consumes messages from the OrdersQueue and then via a content based router, it sorts the directory of each country that are located within the **$JBOSS_HOME/standalone/data/orders/processed** directory.

NOTE

The CLI script automatically configure the ActiveMQ resource adapter. These scripts are located within the **src/main/resources/cli** directory.

5.2.1.1. Running the Application

Before you start running the application, make sure that the following are installed on your machine:

- Maven 3.2.3 or greater
- JBoss Fuse on EAP
- An ActiveMQ broker

Procedure 5.1. To run the application

Perform the following steps:

1. Start the application server in standalone mode.

```
${JBOSS_HOME}/bin/standalone.sh -c standalone-full.xml
```

2. Run the following command to build and deploy the project.

```
mvn install -Pdeploy
```

5.2.1.2. Configuring ActiveMQ

Here are the details to configure the ActiveMQ component:

```
@Startup
@CamelAware
@ApplicationScoped
public class ActiveMQRouteBuilder extends RouteBuilder {

    /**
     * Inject the ActiveMQConnectionFactory that has been configured through the ActiveMQ Resource Adapter
     */
    @Resource(mappedName = "java:/ActiveMQConnectionFactory")
    private ConnectionFactory connectionFactory;

    @Override
    public void configure() throws Exception {

        /**
         * Configure the ActiveMQ component
         */
        ActiveMQComponent activeMQComponent = ActiveMQComponent.activeMQComponent();
```

```
        activeMQComponent.setConnectionFactory(connectionFactory);
        getContext().addComponent("activemq", activeMQComponent);

        /**
         * This route reads files placed within $JBOSS_HOME/standalone/data/orders
         * and places them to ActiveMQ queue 'ordersQueue'
         */
        from("file://{{jboss.server.data.dir}}/orders")
        .convertBodyTo(String.class)
        // Remove headers to ensure we end up with unique file names being generated in the next
route
        .removeHeaders("*")
        .to("activemq:queue:OrdersQueue");

        /**
         * This route consumes messages from the 'ordersQueue'. Then, based on the
         * message payload XML content it uses a content based router to output
         * orders into appropriate country directories
         */
        from("activemq:queue:OrdersQueue")
            .choice()
                .when(xpath("/order/customer/country = 'UK'"))
                    .log("Sending order ${file:name} to the UK")
                    .to("file:{{jboss.server.data.dir}}/orders/processed/UK")
                .when(xpath("/order/customer/country = 'US'"))
                    .log("Sending order ${file:name} to the US")
                    .to("file:{{jboss.server.data.dir}}/orders/processed/US")
                .otherwise()
                    .log("Sending order ${file:name} to another country")
                    .to("file://{{jboss.server.data.dir}}/orders/processed/Others");
    }
    }
```

5.2.1.3. Undeploy the Application

Run the following command to undeploy the application:

```
mvn clean -Pdeploy
```

It removes the ActiveMQ resource adapter configuration. However, you need to restart the application after you execute the undeploy command.

5.2.2. Camel CDI

The following example describes how to use the ***camel-cdi*** component with JBoss Fuse on EAP, to integrate CDI beans with camel routes.

In this example, a camel route takes a message payload from a servlet HTTP GET request and passes it to the direct endpoint. However, you can pass the payload to a Camel CDI bean invocation to produce a message response. It displays the message response on the web browser page.

5.2.2.1. Running the Application

Before you start running the application, make sure that the following are installed on your machine:

- Maven 3.2.3 or greater
- JBoss Fuse on EAP

Procedure 5.2. To run the application

Perform the following steps:

1. Start the application server in standalone mode.

```
${JBOSS_HOME}/bin/standalone.sh -c standalone-full.xml
```

2. Run the following command to build and deploy the project.

```
mvn install -Pdeploy
```

5.2.2.2. Configuring Camel CDI

Here are the details to configure the ***camel-cdi*** component:

```
@Startup
@CamelAware
@ApplicationScoped
public class MyRouteBuilder extends RouteBuilder {

    @Override
    public void configure() throws Exception {
     from("direct:start").bean("helloBean");
    }
}
```

```
@SuppressWarnings("serial")
@WebServlet(name = "HttpServiceServlet", urlPatterns = { "/*" }, loadOnStartup = 1)
public class SimpleServlet extends HttpServlet
{
 @Inject
 private CamelContext camelctx;

    @Override
 protected void doGet(HttpServletRequest req, HttpServletResponse res) throws ServletException,
IOException {
        String name = req.getParameter("name");
     ServletOutputStream out = res.getOutputStream();
        ProducerTemplate producer = camelctx.createProducerTemplate();
        String result = producer.requestBody("direct:start", name, String.class);
     out.print(result);
    }
}
```

5.2.2.3. Undeploy the Application

Run the following command to undeploy the application:

-

```
mvn clean -Pdeploy
```

5.2.3. Camel JMS

The following example describes how to use the ***camel-jms*** component with JBoss Fuse on EAP to produce and consume JMS messages.

In this example, a Camel route consumes files from the **${JBOSS_HOME}/standalone/data/orders** directory and place the content in the OrdersQueue. A second route consumes messages from the OrdersQueue and through a content based router.

5.2.3.1. Running the Application

Before you start running the application, make sure that the following are installed on your machine:

- Maven 3.2.3 or greater
- JBoss Fuse on EAP

Procedure 5.3. To run the application

Perform the following steps:

1. Start the application server in standalone mode.

```
${JBOSS_HOME}/bin/standalone.sh -c standalone-full.xml
```

2. Run the following command to build and deploy the project.

```
mvn install -Pdeploy
```

5.2.3.2. Configuring Camel JMS

Here are the details to configure the camel-jms component.

```
@Startup
@CamelAware
@ApplicationScoped
public class JmsRouteBuilder extends RouteBuilder {

    @Resource(mappedName = "java:/ConnectionFactory")
    private ConnectionFactory connectionFactory;

    @Override
    public void configure() throws Exception {
        /**
         * Configure the JMSComponent to use the connection factory
         * injected into this class
         */
        JmsComponent component = new JmsComponent();
        component.setConnectionFactory(connectionFactory);

        getContext().addComponent("jms", component);
```

```
        /**
         * This route reads files placed within $JBOSS_HOME/standalone/data/orders
         * and places them onto JMS queue 'ordersQueue' within the WildFly
         * internal HornetQ broker.
         */
        from("file://{{jboss.server.data.dir}}/orders")
        .convertBodyTo(String.class)
        // Remove headers to ensure we end up with unique file names being generated in the next
route
        .removeHeaders("*")
        .to("jms:queue:OrdersQueue");

        /**
         * This route consumes messages from the 'ordersQueue'. Then, based on the
         * message payload XML content it uses a content based router to output
         * orders into appropriate country directories
         */
        from("jms:queue:OrdersQueue")
            .choice()
                .when(xpath("/order/customer/country = 'UK'"))
                    .log("Sending order ${file:name} to the UK")
                    .to("file:{{jboss.server.data.dir}}/orders/processed/UK")
                .when(xpath("/order/customer/country = 'US'"))
                    .log("Sending order ${file:name} to the US")
                    .to("file:{{jboss.server.data.dir}}/orders/processed/US")
                .otherwise()
                    .log("Sending order ${file:name} to another country")
                    .to("file://{{jboss.server.data.dir}}/orders/processed/others");
    }
}
```

5.2.3.3. Undeploy the Application

Run the following command to undeploy the application:

```
mvn clean -Pdeploy
```

5.2.4. Camel JPA

The following example describes how to use the ***camel-jpa*** component with JBoss Fuse on EAP to persist entities to the in-memory database.

In this example, a camel route consumes XML files from the **${JBOSS_HOME}/standalone/data/customers** directory. Camel then uses JAXB to unmarshal the data to a Customer entity. However, the entity is then passed to the JPA endpoint and is persisted to the customer database.

5.2.4.1. Running the Application

Before you start running the application, make sure that the following are installed on your machine:

- Maven 3.2.3 or greater

- JBoss Fuse on EAP

Procedure 5.4. To run the application

Perform the following steps:

1. Start the application server in standalone mode.

```
${JBOSS_HOME}/bin/standalone.sh -c standalone-full.xml
```

2. Run the following command to build and deploy the project.

```
mvn install -Pdeploy
```

5.2.4.2. Configuring Camel JPA

Here are the details to configure the ***camel-jpa*** component.

```
@Startup
@CamelAware
@ApplicationScoped
public class JpaRouteBuilder extends RouteBuilder {

    @Inject
    private EntityManager em;

    @Inject
    UserTransaction userTransaction;

    @Override
    public void configure() throws Exception {
        // Configure our JaxbDataFormat to point at our 'model' package
        JaxbDataFormat jaxbDataFormat = new JaxbDataFormat();
        jaxbDataFormat.setContextPath(Customer.class.getPackage().getName());

        EntityManagerFactory entityManagerFactory = em.getEntityManagerFactory();

        // Configure a JtaTransactionManager by looking up the JBoss transaction manager from JNDI
        JtaTransactionManager transactionManager = new JtaTransactionManager(userTransaction);
        transactionManager.afterPropertiesSet();

        // Configure the JPA endpoint to use the correct EntityManagerFactory and
JtaTransactionManager
        JpaEndpoint jpaEndpoint = new JpaEndpoint();
        jpaEndpoint.setCamelContext(getContext());
        jpaEndpoint.setEntityType(Customer.class);
        jpaEndpoint.setEntityManagerFactory(entityManagerFactory);
        jpaEndpoint.setTransactionManager(transactionManager);

        /*
         * Simple route to consume customer record files from directory input/customers,
         * unmarshall XML file content to a Customer entity and then use the JPA endpoint
         * to persist the it to the 'ExampleDS' datasource (see standalone.camel.xml for datasource
config).
         */
```

```
        from("file://{{jboss.server.data.dir}}/customers")
                .unmarshal(jaxbDataFormat)
                .to(jpaEndpoint)
                .to("log:input?showAll=true");
    }
}
```

```
public class CustomerRepository {

    @Inject
    private EntityManager em;

    /**
     * Find all customer records
     *
     * @return A list of customers
     */
    public List<Customer> findAllCustomers() {

        CriteriaBuilder criteriaBuilder = em.getCriteriaBuilder();
        CriteriaQuery<Customer> query = criteriaBuilder.createQuery(Customer.class);
        query.select(query.from(Customer.class));

        return em.createQuery(query).getResultList();
    }
}
```

5.2.4.3. Undeploy the Application

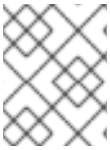
Run the following command to undeploy the application:

```
mvn clean -Pdeploy
```

5.2.5. Camel Mail

The following example describes how to use the ***camel-mail*** component with JBoss Fuse on EAP to send and receive email.

In this example, you can configure a local mail server on your machine. This eliminates the need to use any external mail services. You can access the **src/main/resources/cli** directory to see the EAP mail subsystem configuration.



NOTE

Here the mail session used is bound to JNDI at the **java:jboss/mail/** location. You can configure the server entries for SMTP and POP3 protocols.

5.2.5.1. Running the Application

Before you start running the application, make sure that the following are installed on your machine:

- Maven 3.2.3 or greater
- JBoss Fuse on EAP

Procedure 5.5. To run the application

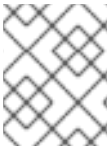
Perform the following steps:

1. Start the application server in standalone mode.

```
${JBOSS_HOME}/bin/standalone.sh -c standalone-full.xml
```

2. Run the following command to build and deploy the project.

```
mvn install -Pdeploy
```



NOTE

If you want to deploy the application multiple times, ensure that you run the undeploy command and restart the application server.

5.2.5.2. Configuring Camel Mail

Here are the configurations details to configure the ***camel-mail*** component.

```
public class MailSessionProducer {
    @Resource(lookup = "java:jboss/mail/greenmail")
    private Session mailSession;

    @Produces
    @Named
    public Session getMailSession() {
        return mailSession;
    }
}
```

```
@Startup
@CamelAware
@ApplicationScoped
public class MailRouteBuilder extends RouteBuilder {

    @Override
    public void configure() throws Exception {
        // Configure routes and endpoints to send and receive email over SMTP and POP3
        from("direct:sendmail").to("smtp://localhost:10025?session=#mailSession");

        from("pop3://user2@localhost:10110?
consumer.delay=30000&session=#mailSession").to("log:emails?showAll=true&multiline=true");
    }
}
```

5.2.5.3. Undeploy the Application

Run the following command to undeploy the application:

```
mvn clean -Pdeploy
```

5.2.6. Camel REST

The following example describes how to write the JAX-RS REST routes with JBoss Fuse on EAP.

It includes two methods of implementing Camel REST consumers. Requests made to paths under the **/example-camel-rest/camel** are handled by the Camel REST DSL and requests made to paths **>/example-camel-rest/rest** are handled by the EAP JAX-RS subsystem along with the CamelProxy.

5.2.6.1. Running the Application

Before you start running the application, make sure that the following are installed on your machine:

- Maven 3.2.3 or greater
- JBoss Fuse on EAP

Procedure 5.6. To run the application

Perform the following steps:

1. Start the application server in standalone mode.

   ```
   ${JBOSS_HOME}/bin/standalone.sh -c standalone-full.xml
   ```

2. Run the following command to build and deploy the project.

   ```
   mvn install -Pdeploy
   ```

5.2.6.2. Configuring Camel REST

Here are the configurations details to configure the Camel REST routes.

```
@Startup
@CamelAware
@ApplicationScoped
public class RestConsumerRouteBuilder extends RouteBuilder {

    /**
     * Inject a service for interacting with the EAP exampleDS in-memory database.
     */
    @Inject
    private CustomerRepository customerRepository;

    @Override
    public void configure() throws Exception {
        /**
         * Configure the Camel REST DSL to use the camel-servlet component for handling HTTP
requests.
         *
```

```
        * Whenever a POST request is made to /customer it is accompanied with a JSON string
representation
        * of a Customer object. Note that the binding mode is set to RestBindingMode.json. This will
enable
        * Camel to unmarshal JSON to the desired object type.
        *
        * Note that the contextPath setting below has no effect on how the application server handles
HTTP traffic.
        * The context root and required servlet mappings are configured in WEB-INF/jboss-web.xml and
WEB-INF/web.xml.
        *
        */
       restConfiguration().component("servlet").contextPath("/camel-example-
rest/camel").port(8080).bindingMode(RestBindingMode.json);

       /**
        * Handles requests to a base URL of /camel-example-rest/camel/customer
        */
       rest("/customer")
          /**
           * Handles GET requests to URLs such as /camel-example-rest/camel/customer/1
           */
          .get("/{id}")
             /**
              * Marshalls the response to JSON
              */
             .produces(MediaType.APPLICATION_JSON)
             .to("direct:readCustomer")
          /**
           * Handles POST requests to /camel-example-rest/camel/customer
           */
          .post()
             /**
              * Unmarshalls the JSON data sent with the POST request to a Customer object.
              */
             .type(Customer.class)
             .to("direct:createCustomer");

       /**
        * This route returns a JSON representation of any customers matching the id
        * that was sent with the GET request.
        *
        * If no customer was found, an HTTP 404 response code is returned to the calling client.
        */
       from("direct:readCustomer")
          .bean(customerRepository, "readCustomer(${header.id})")
          .choice()
             .when(simple("${body} == null"))
                .setHeader(Exchange.HTTP_RESPONSE_CODE, constant(404));

       /**
        * This route handles persistence of new customers.
        */
       from("direct:createCustomer")
          .bean(customerRepository, "createCustomer");
```

```
        /**
         * This route handles REST requests that have been made to the RESTful services defined
within
         * CustomerServiceImpl.
         *
         * These services are running under the WildFly RESTEasy JAX-RS subsystem. A CamelProxy
proxies the direct:rest
         * route so that requests can be handled from within a Camel route.
         */
        from("direct:rest")
            .process(new Processor() {
                @Override
                public void process(Exchange exchange) throws Exception {
                    /**
                     * Retrieve the message payload. Since we are using camel-proxy to proxy the direct:rest
                     * endpoint the payload will be of type BeanInvocation.
                     */
                    BeanInvocation beanInvocation = exchange.getIn().getBody(BeanInvocation.class);

                    /**
                     * Get the invoked REST service method name and build a response to send
                     * back to the client.
                     */
                    String methodName = beanInvocation.getMethod().getName();

                    if (methodName.equals("getCustomers")) {
                        /**
                         * Retrieve all customers and send back a JSON response
                         */
                        List<Customer> customers = customerRepository.findAllCustomers();
                        exchange.getOut().setBody(Response.ok(customers).build());
                    } else if(methodName.equals("updateCustomer")) {
                        /**
                         * Get the customer that was sent on this method call
                         */
                        Customer updatedCustomer = (Customer) beanInvocation.getArgs()[0];
                        Customer existingCustomer =
customerRepository.readCustomer(updatedCustomer.getId());

                        if(existingCustomer != null){
                            if(existingCustomer.equals(updatedCustomer)) {
                                /**
                                 * Nothing to be updated so return HTTP 304 - Not Modified.
                                 */
                                exchange.getOut().setBody(Response.notModified().build());
                            } else {
                                customerRepository.updateCustomer(updatedCustomer);
                                exchange.getOut().setBody(Response.ok().build());
                            }
                        } else {
                            /**
                             * No customer exists for the provided id, so return HTTP 404 - Not Found.
                             */

exchange.getOut().setBody(Response.status(Response.Status.NOT_FOUND).build());
```

```
                }
            } else if(methodName.equals("deleteCustomer")) {
                Long customerId = (Long) beanInvocation.getArgs()[0];

                Customer customer = customerRepository.readCustomer(customerId);
                if(customer != null) {
                    customerRepository.deleteCustomer(customerId);
                    exchange.getOut().setBody(Response.ok().build());
                } else {
                    /**
                     * No customer exists for the provided id, so return HTTP 404 - Not Found.
                     */

exchange.getOut().setBody(Response.status(Response.Status.NOT_FOUND).build());
                }
            } else if(methodName.equals("deleteCustomers")) {
                customerRepository.deleteCustomers();

                /**
                 * Return HTTP status OK.
                 */
                exchange.getOut().setBody(Response.ok().build());
            }
        }
    });
  }
}
```

```
@Startup
@CamelAware
@ApplicationScoped
public class RestProducerRouteBuilder extends RouteBuilder {

  @Override
  public void configure() throws Exception {
    /**
     * This route demonstrates a JAX-RS producer using the camel-restlet component.
     *
     * Every 30 seconds, a call is made to the REST API for retrieving all customers at
     * the URL http://localhost:8080/example-camel-rest/rest/customer.
     *
     * The results of the REST service call are written to a file at:
     *
     * $JBOSS_HOME/standalone/data/customer-records/customers.json
     */
    from("timer://outputCustomers?period=30000")
    .to("restlet://http://localhost:8080/example-camel-rest/rest/customer")
    .choice()
      .when(simple("${header.CamelHttpResponseCode} == 200"))
        .log("Updating customers.json")
        .setHeader(Exchange.FILE_NAME, constant("customers.json"))
        .to("file:{{jboss.server.data.dir}}/customer-records/")
      .otherwise()
        .log("REST request failed. HTTP status ${header.CamelHttpResponseCode}");
  }
}
```

5.2.6.3. Undeploy the Application

Run the following command to undeploy the application:

```
mvn clean -Pdeploy
```

5.2.7. Camel Transacted JMS

The following example describes how to use the ***camel-jms*** component with JBoss Fuse on EAP to produce and consume JMS messages in a transacted session.

In this example, a camel route consumes files from the **${JBOSS_HOME}/standalone/data/orders** directory and place the content in the OrdersQueue. A second route consumes messages from the OrdersQueue, converts the message body to the Order entity and persists it.

5.2.7.1. Running the Application

Before you start running the application, make sure that the following are installed on your machine:

- Maven 3.2.3 or greater
- JBoss Fuse on EAP

Procedure 5.7. To run the application

Perform the following steps:

1. Start the application server in standalone mode.

   ```
   ${JBOSS_HOME}/bin/standalone.sh -c standalone-full.xml
   ```

2. Run the following command to build and deploy the project.

   ```
   mvn install -Pdeploy
   ```

3. When the server starts, navigate to the **example-camel-transacted-jms/orders** directory.

The application displays the **Orders Received** page. It includes the list of processed orders.

5.2.7.2. Configuring Transacted JMS

Here are the details to configure the ***camel-jms*** component in a transacted session.

```
@Startup
@CamelAware
@ApplicationScoped
public class JmsRouteBuilder extends RouteBuilder {

  /**
   * Inject the resources required to configure the JMS and JPA Camel
   * components. The JPA EntityManager, JMS TransactionManager and a JMS
```

```
     * ConnectionFactory bound to the JNDI name java:/JmsXA
     */
    @Inject
    private EntityManager entityManager;

    @Inject
    private JmsTransactionManager transactionManager;

    @Resource(mappedName = "java:/JmsXA")
    private ConnectionFactory connectionFactory;

    @Override
    public void configure() throws Exception {
        /**
         * Create an instance of the Camel JmsComponent and configure it to support JMS
         * transactions.
         */
        JmsComponent jmsComponent =
JmsComponent.jmsComponentTransacted(connectionFactory, transactionManager);
        getContext().addComponent("jms", jmsComponent);

        /**
         * Create an instance of the Camel JpaComponent and configure it to support transactions.
         */
        JpaComponent jpaComponent = new JpaComponent();
        jpaComponent.setEntityManagerFactory(entityManager.getEntityManagerFactory());
        jpaComponent.setTransactionManager(transactionManager);
        getContext().addComponent("jpa", jpaComponent);

        /**
         * Configure JAXB so that it can discover model classes.
         */
        JaxbDataFormat jaxbDataFormat = new JaxbDataFormat();
        jaxbDataFormat.setContextPath(Order.class.getPackage().getName());

        /**
         * Configure a simple dead letter strategy. Whenever an IllegalStateException
         * is encountered this takes care of rolling back the JMS and JPA transactions. The
         * problem message is sent to the WildFly dead letter JMS queue (DLQ).
         */
        onException(IllegalStateException.class)
            .maximumRedeliveries(1)
            .handled(true)
            .to("jms:queue:DLQ")
            .markRollbackOnly();

        /**
         * This route consumes XML files from $JBOSS_HOME/standalone/data/orders and sends
         * the file content to JMS destination OrdersQueue.
         */
        from("file:{{jboss.server.data.dir}}/orders")
            .transacted()
                .to("jms:queue:OrdersQueue");

        /**
         * This route consumes messages from JMS destination OrdersQueue, unmarshalls the XML
```

```
        * message body using JAXB to an Order entity object. The order is then sent to the JPA
        * endpoint for persisting within an in-memory database.
        *
        * Whenever an order quantity greater than 10 is encountered, the route throws an
IllegalStateException
        * which forces the JMS / JPA transaction to be rolled back and the message to be delivered to
the dead letter
        * queue.
        */
       from("jms:queue:OrdersQueue")
            .unmarshal(jaxbDataFormat)
            .to("jpa:Order")
            .choice()
            .when(simple("${body.quantity} > 10"))
              .log("Order quantity is greater than 10 - rolling back transaction!")
              .throwException(new IllegalStateException())
            .otherwise()
              .log("Order processed successfully");
    }
}
```

5.2.7.3. Undeploy the Application

Run the following command to undeploy the application:

```
mvn clean -Pdeploy
```

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

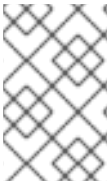
A bundle entry can be used in combination with a **blueprint:** (or **spring:**) URL handler to deploy a Blueprint XML resource (or a Spring XML resource) as an OSGi bundle.

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 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:karaf}/
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 are stored in a Fabric server's built-in Git repository) and then can be accessed by your applications at run time.

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.PropertyPlaceholderConfigurer">
        <property name="properties">
            <bean class="io.fabric8.mq.fabric.ConfigurationProperties"/>
        </property>

     ^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. You can use the profile URL to access resources stored under the current profile or parent profile. It has the following format: **profile:ResourceName** A key characteristic of the profile URL is that the location of a resource can change dynamically at run time, as follows:

- The profile URL handler first tries to find the named resource, ***ResourceName***, in the current version of the current profile (where the current version is a property of the container in which the profile is running).

- If the specified resource is not found under the current profile, the profile URL tries to find the resource in the current version of the parent profile.

For example, the default profile provides the jetty.xml resource and this resource is accessed by setting the

```
org.ops4j.pax.web.config.url=${profile:jetty.xml}
```

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 **cxf** 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 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
...
```