Red Hat Reference Architecture Series

Designing, developing, and deploying integration solutions with JBoss Fuse

Red Hat JBoss Fuse 6.1

Babak Mozaffari
Consulting Software Engineer
Systems Engineering

Version 1.1
December 2014

redhat.

100 East Davie Street
Raleigh NC 27601 USA
Phone: +1 919 754 3700
Phone: 888 733 4281
Fax: +1 919 754 3701
PO Box 13588
Research Triangle Park NC 27709 USA

The GPG fingerprint of the security@redhat.com key is:
CA 20 86 86 2B D6 9D FC 65 F6 EC C4 21 91 80 CD DB 42 A6 0E

Comments and Feedback

In the spirit of open source, we invite anyone to provide feedback and comments on any reference architectures. Although we review our papers internally, sometimes issues or typographical errors are encountered. Feedback allows us to not only improve the quality of the papers we produce, but allows the reader to provide their thoughts on potential improvements and topic expansion to the papers.

Feedback on the papers can be provided by emailing refarch-feedback@redhat.com. Please refer to the title within the email.

Staying In Touch

Join us on some of the popular social media sites where we keep our audience informed on new reference architectures as well as offer related information on things we find interesting.

Like us on Facebook:

https://www.facebook.com/rhrefarch

Follow us on Twitter:

https://twitter.com/RedHatRefArch

Plus us on Google+:

https://plus.google.com/114152126783830728030/

Table of Contents

1 Executive Summary

Red Hat JBoss Fuse is a small-footprint, open source Enterprise Service Bus (ESB). It delivers a robust, cost-effective, and open integration platform that lets enterprises easily connect their disparate applications, services, or devices in real time. An integrated enterprise is able to provide better products and services to its customers. A flexible architecture coupled with popular and proven integration tools enables Red Hat JBoss Fuse to provide integration everywhere.[1]

This reference architecture reviews **Red Hat JBoss Fuse 6.1** and walks through the design, implementation and deployment of a sample application. The section on design and development starts with a very simple project, assuming little to no prior experience. Once deployed and successfully tested, further requirements are defined and iteratively addressed, leading to a more complex and comprehensive solution. Without attempting to provide a thorough tutorial or educational content, this reference architecture allows a new user with the adequate technical background to quickly get up to speed with the basics of Fuse 6 and start creating real-world integration solutions.

Requirements for the sample application include support for receiving requests in various legacy and modern formats, message validation, auditing and throttling, as well as the use of **ActiveMQ** messaging as an asynchronous bridge to the back-end order processing servers. Some of the components from **Apache Camel** and **Apache CXF** are used in the Fuse application to satisfy these requirements and while covering every product feature is not feasible within the time and scope constraints of this effort, these cherry-picked examples serve to demonstrate some of the product capabilities and establish patterns that can help developers moving forward. Build, packaging and dependency management is discussed and the application is deployed on a Fabric ensemble across three machines to provide better horizontal scalability and eliminate a single point of failure.

1 https://access.redhat.com/products/red-hat-jboss-fuse/

2.3 Dependency Injection

Red Hat JBoss Fuse offers a choice between the **`Spring XML`** and **`Blueprint XML`** dependency injection frameworks. 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.[7] The use of blueprint is normally preferred in an OSGi environment but Spring may be adopted as the framework of choice to enable deployment on Java EE containers in the future.

2.3.1 Spring XML

While primarily a dependency injection framework, Spring also includes a suite of services and APIs that enable it to act as a container.

Refer to the IoC Container documentation from the Spring Reference Manual for details on how to use Spring for dependency injection.[8]

The Spring extensibility mechanism allows it to include XML configuration for Apache Camel, Apache CXF and ActiveMQ, while Spring Dynamic Modules support integration of Spring applications with OSGi containers.

When using Spring as the dependency injection model, the configuration files for your application should reside in the following location: *src/main/resources/META-INF/spring/*.xml*

2.3.2 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.[9]

The Blueprint extensibility mechanism allows it to include XML configuration for Apache Camel, Apache CXF and ActiveMQ. Injection is supported through beans that are either automatically or explicitly wired togethet. The XML syntax allows OSGi services to be easily exported or consumed.

In a Maven project, blueprint configuration files for your application should reside in the following location: *src/main/resources/META-INF/blueprint/*.xml*

---

7 https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Fuse/6.1/html/Getting_Started/files/Concepts-Injection.html

8 http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/html/beans.html

9 https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Fuse/6.1/html/Getting_Started/Concepts-Injection.html

2.4 Fuse Fabric

Fuse Fabric is a technology layer that allows a group of containers to form a cluster that shares a common set of configuration information and a common set of repositories from which to access runtime artifacts. Fabric containers are managed by a Fabric Agent that installs a set of bundles that are specified in the profiles assigned to the container. The agent requests artifacts from the Fabric Ensemble. The ensemble has a list of repositories that it can access. These repositories are managed using a Maven proxy and include a repository that is local to the ensemble.[10]

In a fabric container, you cannot directly deploy bundles to a container, whether directly or as part of a feature. A container's configuration is managed by a Fabric Agent that updates its contents and configuration based on one or more profiles. So to deploy to a container, you must either add the bundle or install the feature to an existing profile or create a new profile that is configured to include the bundle or feature in question.

10 https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Fuse/6.1/html/Deploying_into_the_Container/FESBIntroFabric.html

2.5 Components

2.5.1 Apache Camel

Apache Camel is an open-source project that provides the EIP-based routing technology used by JBoss Fuse. Camel supports both a Java-based and XML-based syntax for building routes. This reference architecture uses the XML syntax within blueprint configuration files.

The router schema for blueprint, which defines the XML DSL, is defined in the following XML schema namespace: *http://camel.apache.org/schema/blueprint*

To define a Camel route in XML syntax using blueprint dependency injection, use a configuration file that resembles the following:

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

    <camelContext xmlns="http://camel.apache.org/schema/blueprint">
          <route>
                ...
          </route>
    </camelContext>
</blueprint>
```

A local routing rule always starts with a <from> element, which specifies the source of messages (consumer endpoint) for the routing rule. You can then add an arbitrarily long chain of processors to the rule (for example, <filter>). You typically finish off the rule with a <to> element, which specifies the target (producer endpoint) for the messages that pass through the rule. However, it is not always necessary to end a rule with <to>. There are alternative ways of specifying the message target in a rule.

Refer to the Apache Camel Development Guide as part of the official Red Hat documentation for further details on Camel development.[11]

Camel includes a large number of components that are described in the Apache Camel Component Reference as part of the official Red Hat Documentation.[12]

11 https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Fuse/6.1/html/Apache_Camel_Development_Guide/

12 https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Fuse/6.1/html/Apache_Camel_Component_Reference/index.html

2.5.2 Apache CXF

Apache CXF is an open source services framework. JBoss Fuse primarily leverages the capabilities and features of Apache CXF to provide comprehensive support for SOAP-based and RESTful Web Services.

Apache CXF is often used within a Camel route to specify either a consumer or producer endpoint. For example, to expose a WSDL for a SOAP-based web service using Apache CXF and Camel, it is enough to simply write a simple Java interface with a single method and no annotations, and then use this interface to generate the endpoint:

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

    <cxf:cxfEndpoint id="myWS" address="/myWS/"
         serviceClass="..." />

    <camelContext trace="false"
xmlns="http://camel.apache.org/schema/blueprint">
         <route>
              <from uri="cxf:bean:myWS" />
              <log message="My Web Service was invoked..." />
              ...
         </route>
    </camelContext>
</blueprint>
```

Refer to the official Apache CXF Development Guide as part of the Red Hat documentation for further details.[13]

13 https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Fuse/6.1/html/Apache_CXF_Development_Guide/index.html

2.5.3 Apache ActiveMQ

ActiveMQ is a fast and powerful messaging server. Within Camel, the ActiveMQ component allows messages to be sent to a JMS Queue or Topic, or messages to be consumed from a JMS Queue or Topic using Apache ActiveMQ.

The following example includes two Camel routes where the first one simply picks up the contents of a file and places it on a JMS queue. The second route creates a queue consumer and process messages on this queue:

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

    <camelContext xmlns="http://camel.apache.org/schema/blueprint"
          xmlns:order="http://fusesource.com/examples/order/v7" id="jms-
example-context">

          <route id="file-to-jms-queue">
                <from uri="file:..." />
                <to uri="amq:incoming" />
          </route>

          <route id="jms-route">
                <from uri="amq:incoming" />
                ...
          </route>
    </camelContext>
</blueprint>
```

2.6 Deployment

Red Hat JBoss Fuse is a multi-faceted container that supports a variety of deployment models. However because the Red Hat JBoss Fuse container is fundamentally an OSGi container, the OSGi bundle is also the native format for the container and after deployment, all of the other deployment unit types are converted into OSGi bundles.

2.6.1 OSGi Bundle

An OSGi bundle is a tightly coupled, dynamically loadable collection of classes, JARs, and configuration files that explicitly declare any external dependencies. In OSGi, a bundle is the primary deployment format. Bundles are applications that are packaged in JARs, and can be installed, started, stopped, updated, and removed.[14]

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

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

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

The benefits of class loading in OSGi include:

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

It is fairly easy to change a simple Maven-based Java project to build a bundle. It is simply enough to:

1. Change the Maven packaging from *jar* to *bundle*
2. Add *maven-bundle-plugin* as a build plugin

For example:

```
    <packaging>bundle</packaging>
...
    <build>
          <plugins>
                <plugin>
                      <groupId>org.apache.felix</groupId>
                      <artifactId>maven-bundle-plugin</artifactId>
                      <version>2.3.7</version>
...
```

14 https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Fuse/6.1/html/Deploying_into_the_Container/bundles.html

2.6.2 Fuse Feature

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

JBoss Fuse Features are the preferred and recommended deployment approach. It is a concept that is simple and convenient, without being too abstract to understand the bundles and dependencies that form the backbone of the deployment.

To create a feature, start by creating a feature repository. A feature repository is a location that stores feature descriptor files. Generally, because features can depend recursively on other features and because of the complexity of the dependency chains, the project normally requires access to all of the standard Red Hat JBoss Fuse feature repositories.[15]

A feature repository is itself an XML file:

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

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

The contents of the bundle element can be any valid URL, including a direct file reference to a built bundle JAR file, however using Maven references to bundles can streamline the build process:

```
<?xml version="1.0" encoding="UTF-8"?>
<features name="fulfillment-feature_repository">
  <feature name="fulfillment-feature">
    <bundle>mvn:com.redhat.refarch.fuse/fulfillment/1.0.0</bundle>
    <bundle>file:/opt/myBundle.jar</bundle>
     ...

    <feature>camel-blueprint</feature>
     ...
  </feature>
</features>
```

Each feature builds upon the bundles and dependencies of other features to satisfy its requirements. JBoss Fuse includes a number of preconfigured features, made available in system feature repositories in the Fuse installation.

15 https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Fuse/6.1/html/Deploying_into_the_Container/Locate-CustomRepo.html

2.6.3 Fabric Profile

A profile is a description of how to provision a logical group of containers. Each profile can have none, one, or more parents, which allows you to have profile hierarchies. A container can be assigned one or more profiles. Profiles are also versioned, which enables you to maintain different versions of each profile, and then upgrade or roll back containers, by changing the version of the profiles they use.

JBoss Fuse Fabric includes a number of profiles by default, each with a set of bundles and features associated with them. An easy to get started with a profile is to inherit one that includes the required features. Alternatively, the default profile can be inherited and set up with any needed features. One advantage of inheriting from the default profile as opposed to creating one from scratch is that the provided feature repositories would be preconfigured.

2.6.4 Other Options

2.6.4.1 Fuse Application Bundle

The Fuse Application Bundle (FAB) deployment model is fundamentally different from that of standard OSGi bundles. When a FAB is installed, the FAB runtime automatically figures out what dependencies are required, by scanning the Maven metadata, and these dependencies are then installed dynamically.

However, Fuse Application Bundles have been *DEPRECATED* in JBoss Fuse 6.1 and will be removed in future versions.[16]

2.6.4.2 Web Application Archive

The Web Application Archive (WAR) format is supported in two ways:

Web Applications built and packaged as WAR files may be deployed to JBoss Fuse through the PAX War URL handler, acting as a wrapper.[17]

It is also possible to create and deploy web applications based on Fuse technology, including Camel and CXF. This would involve deploying the Camel servlet or bootstrapping a CXF Servlet in a WAR. Under the WAR deployment model, all of the requisite JBoss Fuse libraries are packaged into your application's WAR file.

16 https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Fuse/6.1/html/Release_Notes/FMQReleaseNotesNew.html#idp108848

17 https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Fuse/6.1/html/Deploying_into_the_Container/DeployWar.html#DeployWar-Convert

3 Reference Architecture Environment

3.1 Overview

This reference architecture consists of a Fuse Fabric ensemble with three member nodes distributed over separate logical machines. Red Hat JBoss Fuse 6.1 is installed on three machines running Red Hat Enterprise Linux 6.1 and each installation is used to start a root and a child container. Assume that the nodes have host names of *fuse-node1*, *fuse-node2* and *fuse-node3*.

An instance of **`MySQL`** database is installed on a fourth node called *fuse-web* and accessed from the ensemble. Avoiding a single point of failure for the database is outside the scope of this reference architecture and it is assumed that a proper high availability strategy is employed to avoid system downtime due to database failure.

3.1.1 Order Fulfillment

In this reference architecture, JBoss Fuse 6.1 is used to design and deploy an order fulfillment system, showcasing the capabilities of the product in each step as various common challenges are discussed and resolved by applying appropriate solutions.

It is assumed that legacy systems not equipped to use web services provide order data in the form of flat files. Processing these files presents several challenges, including picking up the files, aggregating the information of an order that is split in two separate files and making sure this aggregation is consistent throughout the distributed ensemble. Apache Camel is used to implement file polling and provides a reliable and efficient way to process these files with little development effort. Apache Camel also provides the capability of aggregating multiple requests, in this case two separate but related flat files, into a single business request. The database aggregation feature is used to support a distributed deployment.

Apache CXF is used to expose the order fulfillment service as JSON and XML RESTful services as well as a SOAP over HTTP web service. ActiveMQ is leveraged as a JMS implementation and support asynchronous messaging, allowing for fire and forget, parallel processing, and seamless distribution of work.

3.2 Fabric Ensemble

Red Hat JBoss Fuse 6.1 is installed on three separate Linux machines running RHEL 6.1.

After setup and minimal configuration, an instance of JBoss Fuse is started on each of 3 machines. A Fuse Fabric is created with all three nodes joining it as members. A child container is set up on each Fuse instance.

The Order Fulfillment application includes a Fuse Feature which describes the application bundle along with all of its dependencies. A Fabric profile is configured with this Feature and applied to the child containers in order to deploy the application.

3.3 MySQL Database

While it is very likely that an enterprise database would be required for every large-scale deployment, the only database requirement in this reference application is to coordinate and aggregate parts of the same order that may potentially be picked up and processed by two separate nodes of the Fuse ensemble.

Apache Camel provides a JDBC-based implementation of the aggregation repository as part of its SQL Component. Using this feature requires two tables to be created for each aggregation repository, where one stores pieces of an aggregation waiting for completion and the other records completed aggregations. The name of these tables depend on the configuration of the aggregation repository and each is very simple, with a text ID column serving as its primary key and a blob column called exchange to store the individual and completed aggregation parts.

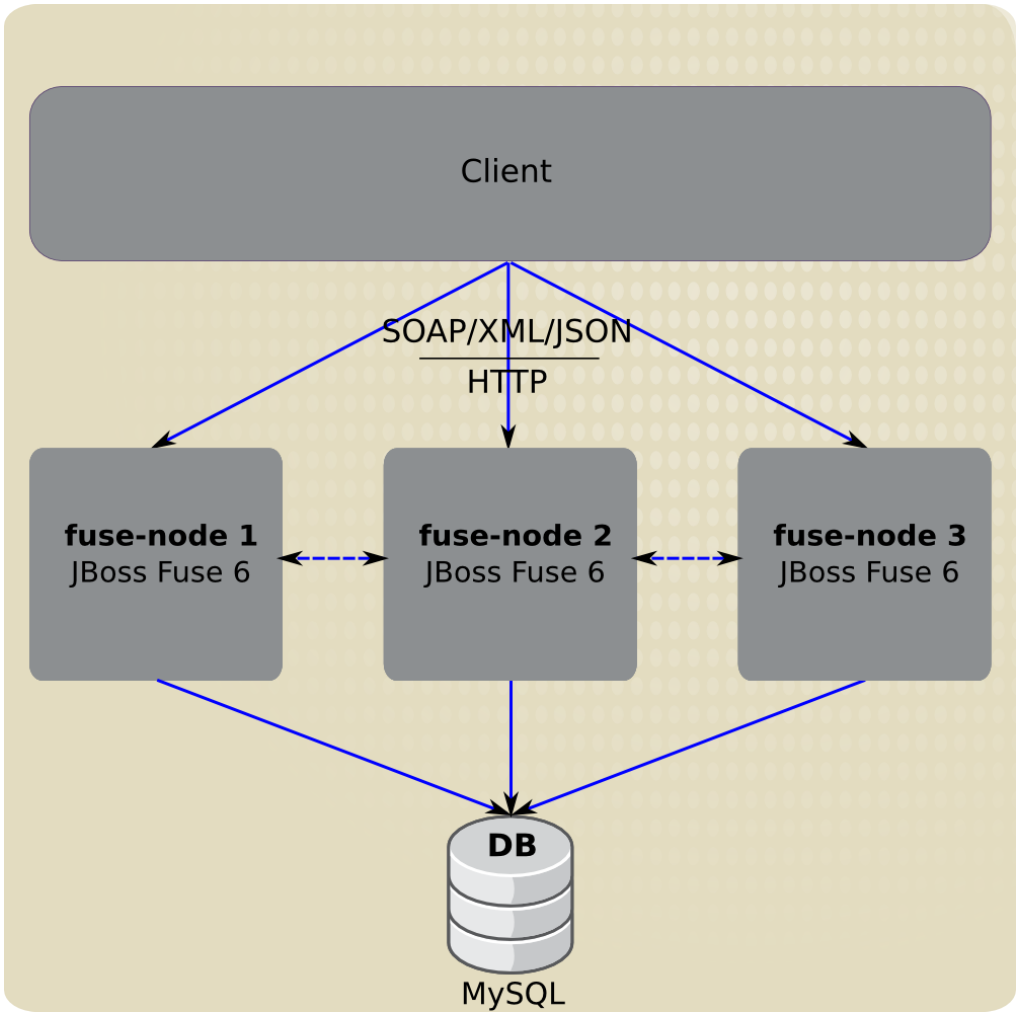


Figure 3.3-1: Fuse Fabric Deployment

4 Creating the Environment

4.1 Prerequisites

Prerequisites for creating this reference architecture include a supported Operating System and JDK. Refer to Red Hat documentation for supported environments.[18]

The reference architecture environment requires Maven to build and install bundles and features into the Maven repository, where they can be accessed by Fuse. This application also relies on a MySQL database to aggregate parts of the same business request that may be picked up by physically separate nodes.

4.2 Downloads

The attachments to this document include the reference application, along with other Maven artifacts to allow building and deploying this application to a Fuse Fabric. These files may be downloaded from:

https://access.redhat.com/node/1274103/40/1

If you do not have access to the Red Hat customer portal, See the Comments and Feedback section to contact us for alternative methods of access to these files.

Download the full installation of JBoss Fuse 6.1.1 from Red Hat's Customer Support Portal:[19]

- Red Hat JBoss Fuse 6.1.1 Full Install

4.3 Installation

4.3.1 MySQL Database

The installation process for MySQL Database Server is beyond the scope of this reference architecture document. On a RHEL system, installing MySQL can be as simple as running:

```
# yum install mysql-server.x86_64
```

4.3.2 JBoss Fuse

Red Hat JBoss Fuse 6.1 is packaged and distributed as a simple archive file. As a pure Java application, there is no installation process and it can simply needs to be extracted:

```
# unzip jboss-fuse-full-6.1.1.redhat-412.zip
```

18 https://access.redhat.com/articles/310603
19 https://access.redhat.com/jbossnetwork/restricted/listSoftware.html?downloadType=distributions&product=jboss.fuse&version=6.1.1

4.4 Configuration

In the reference environment, only a couple of ports are used for intra-node communication. This includes ports 1099 which is used by Fabric to look up and talk to remote instances.

JBoss Fuse uses ZooKeeper to replicate assets and enable the ensemble to synchronize it state. The environment therefore also uses the ZooKeeper client port, set to 2181 by default. The CXF server on child containers would run on port 8182 by default so that is another required access port. This reference architecture uses **`IPTables`**, the default Red Hat Firewall, to block all network packets by default and only allow configured ports and addresses to communicate. Refer to the Red Hat documentation on **`IPTables`**[20] for further details and see the appendix on IPTables configuration for the firewall rules used for the active cluster in this reference environment.

This reference environment has been set up and tested with **`Security-Enhanced Linux (SELinux)`** enabled in *ENFORCING* mode. Once again, refer to the Red Hat documentation on SELinux for further details on using and configuring this feature.[21] For any other operating system, consult the respective documentation for security and firewall solutions to ensure that maximum security is maintained while the ports required by your application are opened.

4.4.1 MySQL Database

For the purpose of this reference architecture, create a MySQL database called *fuse* and a database user with *jboss* and *password* as its username and password.

Execute the following SQL statements through the MySQL administration tool to create the database tables used by the application:

```
# CREATE TABLE order_aggregation
                   (  id varchar(255) NOT NULL,
                      exchange blob NOT NULL,
                      constraint aggregation_pk PRIMARY KEY (id) );

# CREATE TABLE order_aggregation_completed
                   (  id varchar(255) NOT NULL,
                      exchange blob NOT NULL,
                      constraint aggregation_completed_pk PRIMARY KEY (id) );
```

These DDL statements are included in the attachments in a file called *mysql.ddl*.

20 https://access.redhat.com/site/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Security_Guide/sect-Security_Guide-IPTables.html

21 https://access.redhat.com/site/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Security-Enhanced_Linux/

4.4.2 JBoss Fuse

The only required configuration after extracting the archive file and before running JBoss Fuse is setting up an administrative user. By default, JBoss Fuse uses a property file to store user, password and role information: *etc/users.properties*

The last line of this file includes a default administrative user, but is commented out:

```
#All users specified in this file, will be uploaded to the fabric registry
and will
#be available to all containers that join the fabric.
#The password of the first user in the file will also be used as a registry
(zookeeper) password
#unless a password is explicitly specified.
#admin=admin,admin
```

Uncomment this line and configure this user's password as *password*:

```
admin=password,admin
```

Alternatively, if JBoss Fuse is started without a user being configured, you will get prompted to create one before most operations, for example:

```
JBossFuse:karaf@node1> fabric:create --zookeeper-password password --wait-
for-provisioning
No user found in etc/users.properties or specified as an option. Please
specify one ...
New user name: admin
Password for admin:
Verify password for admin:
```

While adding an administrative user is the only required step to use JBoss Fuse, there are multiple other options that may be modified prior to starting the server. The main configuration file for a JBoss Fuse instance is: *etc/system.properties*

Modify the name of each JBoss Fuse instance in the ensemble to a unique name, for example:

```
#
# Name of this Karaf instance.
#
karaf.name=fuse.node1
```

4.5 Deployment

Use Maven to build the reference application before you can deploy it to a Fabric.

The attached artifacts include a code directory with an aggregation POM and three modules:

```
# ls code/
features  fulfillment mysql-fragment   pom.xml
```

Use Maven to build all three modules and install them in the repository by running the install target:

```
# mvn install

[INFO] Scanning for projects...
[INFO]
------------------------------------------------------------------------
[INFO] Reactor Build Order:
[INFO]
[INFO] MySQLFragment
[INFO] Order Fulfillment JBoss Fuse Application
[INFO] Fulfillment feature repository
[INFO] Parent Project
...
...

[INFO] Reactor Summary:
[INFO]
[INFO] MySQLFragment ..................................... SUCCESS [1.135s]
[INFO] Order Fulfillment JBoss Fuse Application .......... SUCCESS [1.437s]
[INFO] Fulfillment feature repository .................... SUCCESS [0.502s]
[INFO] Parent Project .................................... SUCCESS [0.005s]
[INFO]
------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO]
------------------------------------------------------------------------
[INFO] Total time: 3.631s
[INFO] Finished at: ...
[INFO] Final Memory: 26M/310M
[INFO]
------------------------------------------------------------------------
```

You can start JBoss Fuse in the background by running the *bin/start* script and then use the *bin/client* script to enter the Fuse shell environment, or alternatively, run *bin/fuse*:

```
[root@fuse-node1 ~]# bin/client

       _ ____                  ______
      | |  _ \                |  ____|
      | | |_) | ___  ___ ___  | |__ _   _ ___  ___
  _   | |  _ < / _ \/ __/ __| |  __| | | / __|/ _ \
 | |__| | |_) | (_) \__ \__ \ | |  | |_| \__ \  __/
  \____/|____/ \___/|___/___/ |_|   \__,_|___/\___|

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

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

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

Create a new Fabric via 'fabric:create'
```

```
or join an existing Fabric via 'fabric:join [someUrls]'

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

JBossFuse:admin@fuse.node1>
```

Provision a JBoss Fuse Fabric from the first node, specifying a default password and initially including only the current node. Include *--wait-for-provisioning* as an argument so that the command only comes back once the Fabric has been created. Note that the other arguments set default values and are therefore optional:

```
JBossFuse:admin@fuse.node1> fabric:create --zookeeper-password password
--wait-for-provisioning fuse.node1
Waiting for container: fuse.node1
Waiting for container fuse.node1 to provision.
Using specified zookeeper password:password
JBossFuse:admin@fuse.node1>
```

Start the Fuse shell environment from the other two nodes and join the Fabric you created:

```
JBossFuse:admin@fuse.node2> fabric:join --zookeeper-password password fuse-
node1
JBossFuse:admin@fuse.node2>
```

Repeat the same command from node 3.

Once the other two nodes have joined the Fuse Fabric, the remaining commands can be issued from any of the member nodes.

Create child containers for each of the three nodes. You will then proceed to create Fuse profiles with the application's Feature assigned to it and you will set that profile as the only profile of all three child containers.

To create the child containers:

```
JBossFuse:admin@fuse.node1> container-create-child fuse.node1 child1
The following containers have been created successfully:
    Container: child1.
JBossFuse:admin@fuse.node1> container-create-child fuse.node2 child2
The following containers have been created successfully:
    Container: child2.
JBossFuse:admin@fuse.node1> container-create-child fuse.node3 child3
The following containers have been created successfully:
    Container: child3.
```

Create the new profile by inheriting from only the default profile:

```
JBossFuse:admin@fuse.node1> fabric:profile-create --parents default
fulfillment-profile
JBossFuse:admin@fuse.node1>
```

Add the application's feature repository to this profile. Once the repository has been added, the feature itself can be set on the profile:

```
JBossFuse:admin@fuse.node1> profile-edit -r
mvn:com.redhat.refarch.fuse/fulfillment-feature/1.0.0/xml/features
fulfillment-profile
Adding feature repository:mvn:com.redhat.refarch.fuse/fulfillment-
feature/1.0.0/xml/features to profile:fulfillment-profile version:1.0
JBossFuse:admin@fuse.node1> profile-edit --features fulfillment-feature
fulfillment-profile
Adding feature:fulfillment-feature to profile:fulfillment-profile
version:1.0
JBossFuse:admin@fuse.node1>
```

Finally, set this as the main and only profile of all three child containers:

```
JBossFuse:admin@fuse.node1> fabric:container-change-profile child1
fulfillment-profile
JBossFuse:admin@fuse.node1> fabric:container-change-profile child2
fulfillment-profile
JBossFuse:admin@fuse.node1> fabric:container-change-profile child3
fulfillment-profile
```

When a child container is created, a set of directories are created under the *instances* directory of the Fuse installation path. For a container called child1, there would be an equivalent *instances/child1* directory. The log file for this container would be:

- *instances/child1/data/log/karaf.log*

Review this log file to troubleshoot any potential deployment issues. In case of a need to investigate deployment problems, it is often helpful to change the container profile as soon as it is created so any further changes to the profile can be monitored step by step.

4.6 Execution

The child container log files provides multiple clues as to the successful deployment of the application. Given the inclusion of several Camel contexts and routes, the log should includes multiple instances of Camel context and route startups:

```
... Apache Camel 2.12.0.redhat-610379 (CamelContext: camel-5)
started in 0.914 seconds
...
... Total 1 routes, of which 1 is started.
...
```

Other artifacts have their own separate indication of a successful startup. For example, the SOAP web service registration:

```
... registering MBean org.apache.cxf:bus.id=fulfillment-
cxf1902998028,type=Bus.Service.Endpoint,service="{http://ws.fulfillment.fuse
.refarch.redhat.com/}OrderFulfillmentService",port="OrderFulfillmentServiceP
ort",instance.id=1129779289:
javax.management.modelmbean.RequiredModelMBean@1349574f
```

The Apache CXF RESTful service provides the following log message:

```
Setting the server's publish address to be /orderRS
```

The ActiveMQ destination is deployed with routes producing to it and consuming from it, with a confirmation message such as:

```
Route: route3 started and consuming from: Endpoint[amq://queue:order]
```

4.6.1 Legacy File Drop

The application is configured to poll the following directory for comma-separated files. Create this directory, either physically, or using a symbolic link:

/fulfillment/input

Two sample files are provided in the code directory of the attachment that can be dropped to generate an order. Copy these two files to the above directory within no more than a few seconds of one another:

code/fulfillment/requests/file/123-orders.csv
code/fulfillment/requests/file/123-customers.csv

After dropping in the first file, you should notice the following the log file:

```
org.apache.camel.camel-core - 2.12.0.redhat-610379 | Will parse 123-
orders.csv for order items
org.apache.camel.camel-core - 2.12.0.redhat-610379 | Aggregate route
received [Order [itemId=item #1, quantity=5], Order [itemId=item #2,
quantity=7]]
```

Notice that the orders file is parsed and received by the aggregate route, at which time it is stored in the database, waiting for its counterpart customer file before proceeding any further.

Once the customer file is also dropped in:

```
org.apache.camel.camel-core - 2.12.0.redhat-610379 | Will parse 123-
customer.csv for customer
org.apache.camel.camel-core - 2.12.0.redhat-610379 | Aggregate route
received Customer [customerId=333224444, firstName=Babak,
lastName=Mozaffari, telephone=310-555-1234]
org.apache.camel.camel-core - 2.12.0.redhat-610379 | Aggregated: Order
[customer=Customer [customerId=333224444, firstName=Babak,
lastName=Mozaffari, telephone=310-555-1234], orders=[Order [itemId=item #1,
quantity=5], Order [itemId=item #2, quantity=7]]]

Got order with 2 items
    Order for item #1 in the following quantity: 5
    Order for item #2 in the following quantity: 7
org.jboss.amq.mq-fabric - 6.1.0.redhat-379 | OSGi environment detected!
org.apache.activemq.activemq-osgi - 5.9.0.redhat-610379 | Adding new broker
connection URL: tcp://fuse-node1.cloud.lab.eng.bos.redhat.com:61616
org.apache.activemq.activemq-osgi - 5.9.0.redhat-610379 | Successfully
connected to tcp://fuse-node1.cloud.lab.eng.bos.redhat.com:61616
org.apache.camel.camel-core - 2.12.0.redhat-610379 | Receiving order Order
[itemId=item #1, quantity=5]
```

Notice that after parsing and the arrival of the customer data in the aggregating node, aggregation is deemed complete and the order is processed. This prompts a connection to an ActiveMQ destination and each order in the file is fired off separately. In a Fuse ensemble, you are likely to only see one order being received on the first node. The other order is likely to be distributed to another node.

Shortly after, you might see an error and a timeout resulting from a renewed attempt to aggregate the first file:

```
2014-10-24 05:16:47,098 | INFO  | teRecoverChecker | route1
| rg.apache.camel.util.CamelLogger  176 | 121 - org.apache.camel.camel-core
- 2.12.0.redhat-610379 | Aggregated: [Order [itemId=item #1, quantity=5],
Order [itemId=item #2, quantity=7]]
2014-10-24 05:16:47,113 | ERROR | teRecoverChecker | DefaultErrorHandler
| rg.apache.camel.util.CamelLogger  215 | 121 - org.apache.camel.camel-core
- 2.12.0.redhat-610379 | Failed delivery for (MessageId: ID-fuse-node1-
cloud-lab-eng-bos-redhat-com-35358-1414140905761-0-5 on ExchangeId: ID-fuse-
node1-cloud-lab-eng-bos-redhat-com-35358-1414140905761-3-6). Exhausted after
delivery attempt: 2 caught: org.apache.camel.CamelExecutionException:
Exception occurred during execution on the exchange: Exchange[Message:
[Order [itemId=item #1, quantity=5], Order [itemId=item #2, quantity=7]]]

Message History
```

This is a known issue in JBoss Fuse 6.1.[22]

22 https://issues.jboss.org/browse/ENTESB-1956

4.6.2 SOAP-based Web Service

The main Fuse instance runs on port 8181 by default, so a child created in any node would pick the next port by default for its http listening, which is port 8182. This can be confirmed by accessing the CXF page of any of the servers, for example, point your browser to:

http://fuse-node1:8182/cxf/

The expected response page would show one SOAP service as well as on RESTful service deployed:

```
Available SOAP services:
OrderFulfillmentServicePortType
process
Endpoint address: http://fuse-node1:8182/cxf/orderWS/
WSDL :
{http://ws.fulfillment.fuse.refarch.redhat.com/}OrderFulfillmentService
Target namespace: http://ws.fulfillment.fuse.refarch.redhat.com/


Available RESTful services:
Endpoint address: http://fuse-node1:8182/cxf/orderRS
WADL : http://fuse-node1:8182/cxf/orderRS?_wadl
```

Using **`curl`**, you can send a SOAP request to any of the servers. A request has been provided in the attachments for your convenience:

```
# curl -X POST -d @code/fulfillment/requests/ws/request.xml
                                    http://fuse-node1:8182/cxf/orderWS/
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Body><ns1:processResponse
xmlns:ns1="http://ws.fulfillment.fuse.refarch.redhat.com/">
<return>OK</return></ns1:processResponse></soap:Body></soap:Envelope>
```

Upon succesfully reaching and invoking the web service, you will see an OK response returned in a SOAP envelope.

The server log output is also similar, although not identical, to the log generated for the file drops:

```
org.apache.camel.camel-core - 2.12.0.redhat-610379 | Order Web Service
received request
fulfillment - 1.0.0 |

Got order with 2 items
     Order for Chair in the following quantity: 4
     Order for Table in the following quantity: 1
org.apache.camel.camel-core - 2.12.0.redhat-610379 | Receiving order Order
[itemId=Chair, quantity=4]
```

Once again, one of the order items is asynchronously handed off to another node:

```
org.apache.camel.camel-core - 2.12.0.redhat-610379 | Receiving order Order
[itemId=Table, quantity=1]
```

The curl command in this example only specified the request file, the HTTP method as POST and the endpoint address of the service. SOAP-based web services standardized on the request and response content type, therefore not requiring such details to be specified.

4.6.3 XML RESTful Service

Unlike SOAP-based web services, a RESTful service may accept or return either XML or JSON as its message content type. The use of HTTP headers to specify the media type is therefore required.

Once again, use curl with the provided request file:

```
# curl -X POST -H 'Content-Type: application/xml'
-H 'Accept: application/xml' -d @code/fulfillment/requests/rest/request.xml
http://fuse-node1:8182/cxf/orderRS/

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<return><result>OK</result></return>
```

Notice that for a request to the RESTful service, both the request and expected response content type is specified. In this case, they are both set as XML and a succesful XML response is expected to be received and printed.

The server log once again shows the order being processed and the asynchronous messages being distributed among the servers:

```
fulfillment - 1.0.0 |

Got order with 2 items
     Order for Chair in the following quantity: 4
     Order for Table in the following quantity: 1
org.apache.camel.camel-core - 2.12.0.redhat-610379 | Receiving order Order
[itemId=Chair, quantity=4]
```

The other node:

```
org.apache.camel.camel-core - 2.12.0.redhat-610379 | Receiving order Order
[itemId=Table, quantity=1]
```

4.6.4 JSON RESTful Service

Using JSON as the request and response format is a minor variation of a POX service invocation. Point to the JSON sample request file and set the content type and accept headers accordingly:

```
# curl -X POST -H 'Content-Type: application/json'
-H 'Accept: application/json'
-d @code/fulfillment/requests/rest/request.json
http://fuse-node1:8182/cxf/orderRS/

{"result":"OK"}
```

The JSON requests reach the same service as the XML requests and the server logs are identical.

5 Design and Development

5.1 Overview

This section performs a step by step walkthrough of the design and development of the reference architecture application. The first few steps approach the product from the perspective of a beginner, taking small steps to understand and validate how a project is set up and deployed. Gradually and incrementally, the guide tackles more difficult requirements and assumes a more advanced user, providing fewer details and concentrating on specific challenges and use cases.

5.2 Integrated Development Environment

This reference architecture uses JBoss Fuse IDE plugins for JBoss Developer Studio 8.

5.2.1 JBoss Developer Studio

Download the *Stand-alone installer* for **`JBoss Developer Studio (JBDS) 8.0.0`** from the Red Hat Customer Support Portal.[23]

The installer is an executable JAR file. Installing a recent version of the JDK and having the java and associated commands in the execution path is a prerequisite to using JBDS and JBoss Fuse itself.

In most operating systems, it is enough to simply double-click the JBoss Developer Studio installation JAR file to start installing the IDE. You can also trigger the installation from the command line:

```
# java -jar jboss-devstudio-8.0.0.GA-v20141020-1042-B317-installer-standalone.jar
```

Accept the license, choose a location to install the product and proceed with the installation. Select the default or preferred JDK location. Is it not necessary to configure any platform or server location while installing JBoss Developer Studio.

Start JBoss Developer Studio by locating the shortcut created in the designated location. Select a location for the IDE workspace. Once started, an initial welcome screen appears. Close this screen to enter the familiar Eclipse framework environment.

23 https://access.redhat.com/jbossnetwork/restricted/listSoftware.html?downloadType=distributions&product=jbossdeveloperstudio&version=7.1.1

5.2.2 JBoss Fuse IDE plugins

By default, JBoss Developer Studio opens in the *JBoss* perspective which in turn opens the *JBoss Central* view. This view has two tabs in the bottom, where the default selected tab is *Getting Started*. Change it to the *Software/Update* tab and check the box to *Enable Early Access*:

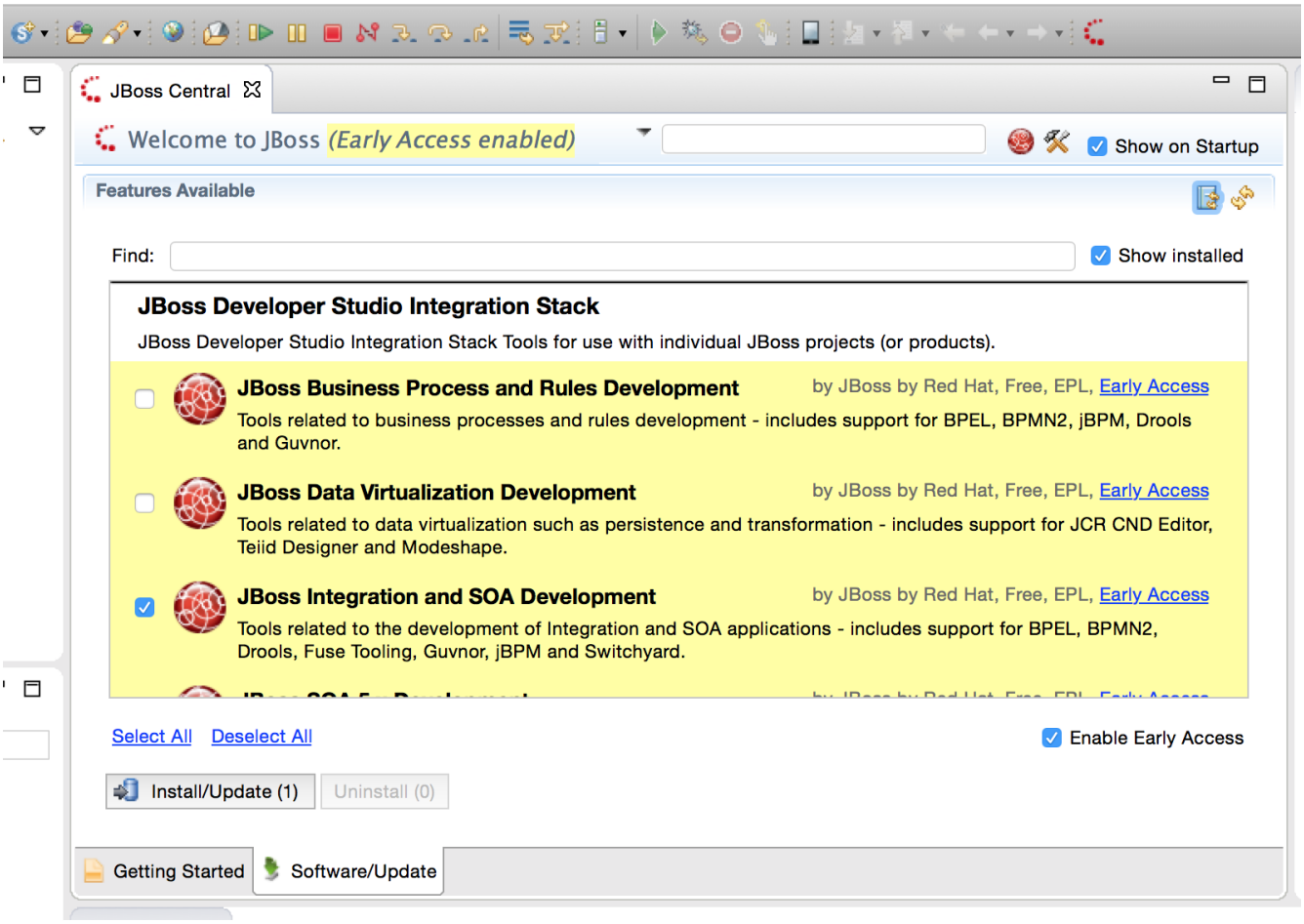


Figure 5.2.2-1: JBDS Software Update

The first category of features and plugins for update is the *JBoss Developer Studio Integration Stack*. The third option is *JBoss Integration and SOA Development*, which also includes JBoss Fuse.

Select only *JBoss Integration and SOA Development* and click the install button at the bottom of the pane.

The selected feature includes a large number of plugins for various JBoss integration products. Only three of these items relate to JBoss Fuse and others may be deselected:

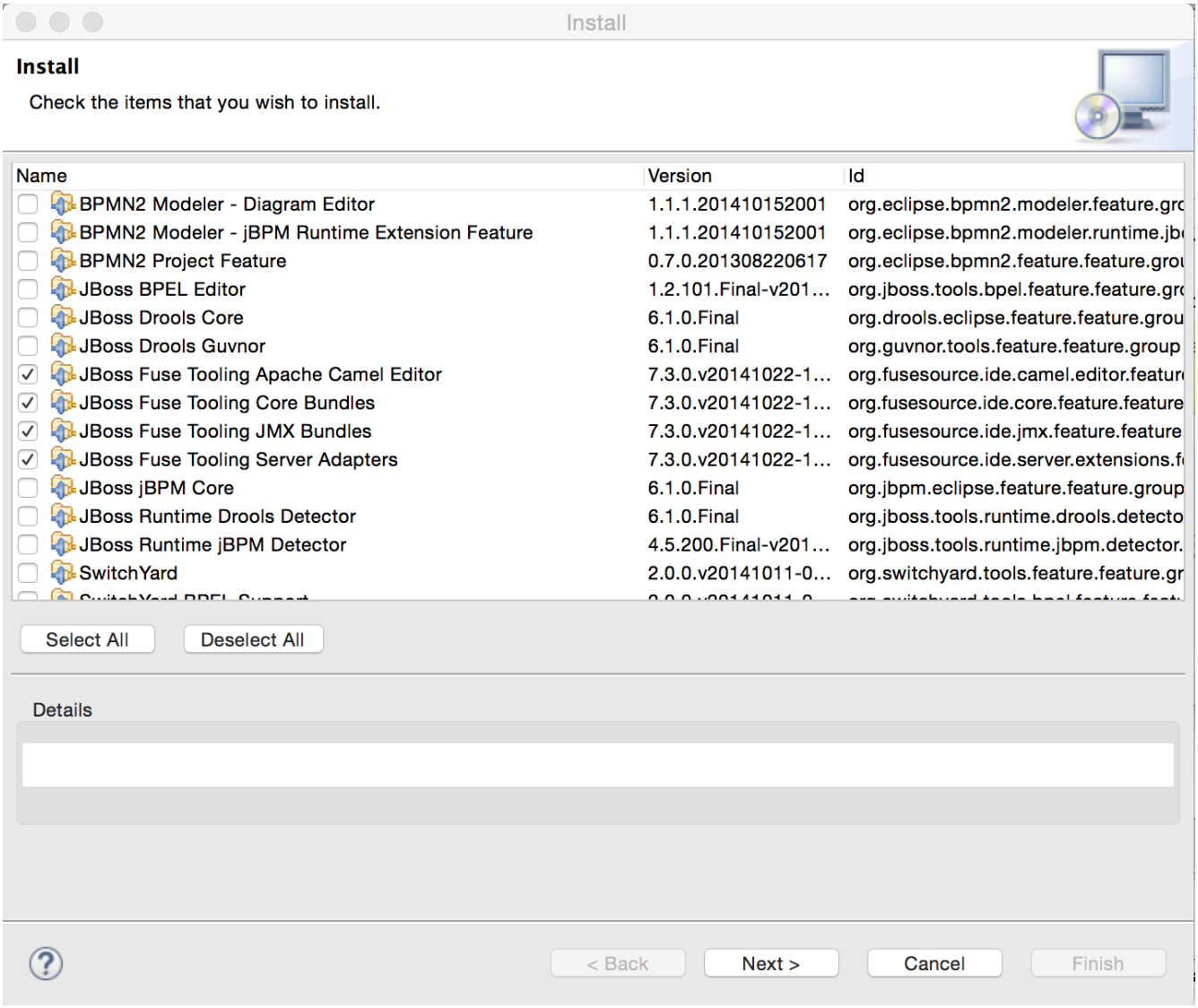


Figure 5.2.2-2: JBDS Fuse Plugins

Proceed forward, accepting the terms and completing the installation of the plugins. Once installed, restart JBoss Developer Studio to ensure all the changes have taken effect.

While not required, you may find it convenient to open the Fuse Integration perspective instead of the default JBoss perspective. The active perspective is displayed at the top-right corner of JBoss Developer Studio and perspectives can be opened and added by pressing the button to its immediate left.

5.2.3 Creating a Fuse Project

Click the drop-down for the *New* toolbar icon at the top left of the JBoss Developer Studio window and select *Fuse Project*. Alternatively, you can click the icon to open the *New* wizard dialog, open the group called *Fuse Tooling* and select *Fuse Project* from there.

The new project wizard prompts you to select a location for the project. For temporary and testing purposes, it is easiest to let JBDS simply create the project in the designated workspace. Carefully pick a location for those projects that you intend to keep and use. Select Next to choose a Maven *archetype*:

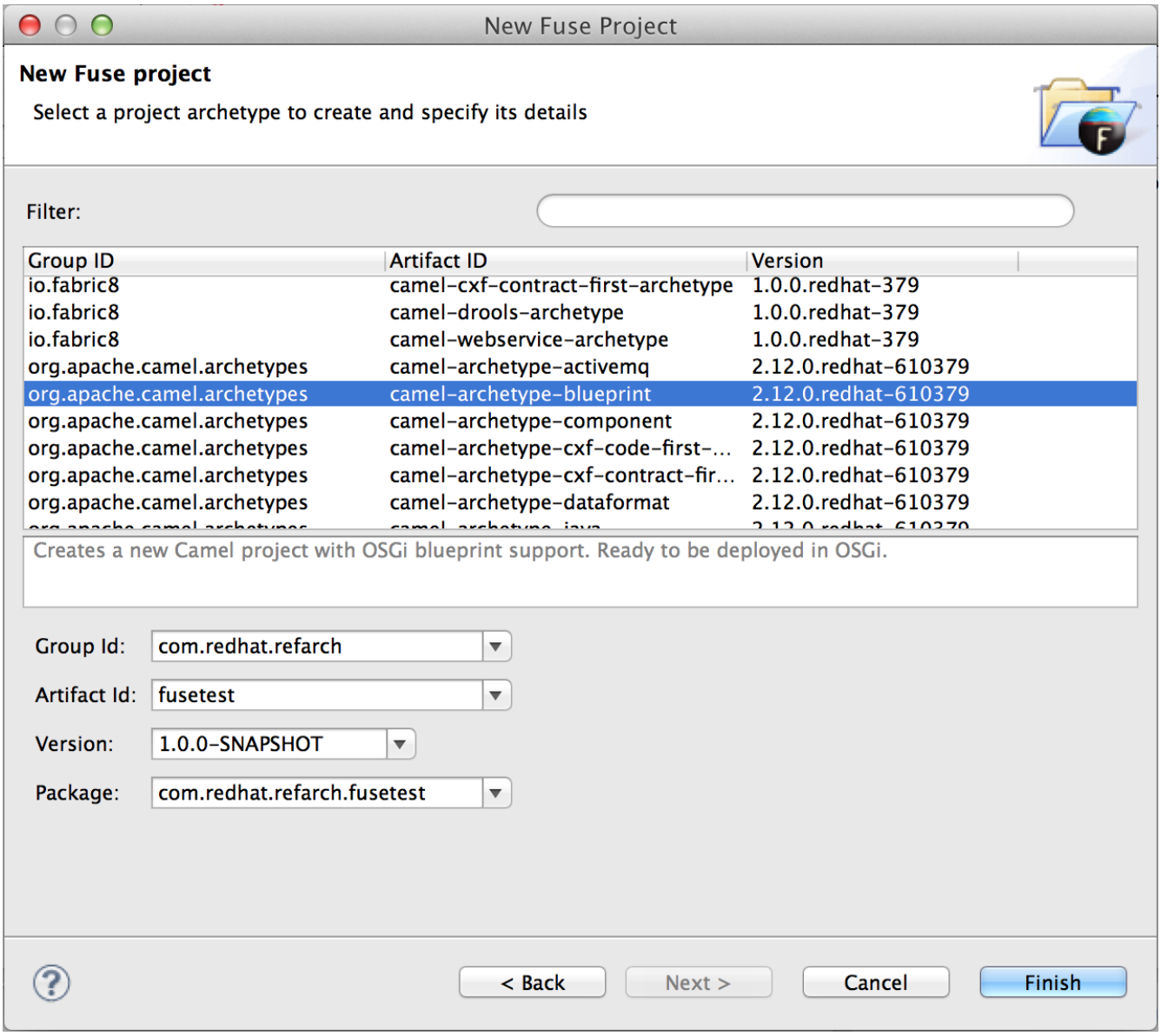


Figure 5.2.3-1: New Fuse Project

5.2.4 Sample Project Review

Once created, the sample project contains a Camel context with a single Camel route. This includes a Java bean in the form of both an interface and an implementation class. The archetype also creates a Maven *Project Object Model (POM)* file to build and run the project.

Note that the Camel blueprint archetype was selected and therefore the blueprint dependency injection model will be used.

The sample project builds a Camel route that is triggered by a timer every 5 seconds, at which time it invokes a simple Java method that returns a message with the current time. The Camel logger is used to print this message. This route gets invoked continuously every 5 seconds unless or until stopped.

In a Maven project using blueprint, any service descriptor must be placed under the *src/main/resources/OSGI-INF/blueprint* directory and have a ".xml" file extension. The generated project has one such file, called 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:camel="http://camel.apache.org/schema/blueprint"
       xsi:schemaLocation="
       http://www.osgi.org/xmlns/blueprint/v1.0.0
http://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd
       http://camel.apache.org/schema/blueprint
http://camel.apache.org/schema/blueprint/camel-blueprint.xsd">

  <bean id="helloBean" class="com.redhat.refarch.fusetest.HelloBean">
      <property name="say" value="Hi from Camel"/>
  </bean>

  <camelContext id="blueprintContext" trace="false"
xmlns="http://camel.apache.org/schema/blueprint">
    <route id="timerToLog">
      <from uri="timer:foo?period=5000"/>
      <setBody>
          <method ref="helloBean" method="hello"/>
      </setBody>
      <log message="The message contains ${body}"/>
      <to uri="mock:result"/>
    </route>
  </camelContext>

</blueprint>
```

Reviewing this file can help you understand some of the basics of JBoss Fuse development.

A blueprint service descriptor is an XML file with a single root element called *blueprint* and defined in the *http://www.osgi.org/xmlns/blueprint/v1.0.0* namespace:

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

The blueprint namespace is defined as the default namespace for any element that does not configure it otherwise. The XML schema instance namespace is defined with a prefix of *xsi* for later use, as is the Camel namespace with a prefix of c*amel*. The schema location for each namespace is provided.

The service describes a simple blueprint service bean for later use:

```
  <bean id="helloBean" class="com.redhat.refarch.fusetest.HelloBean">
      <property name="say" value="Hi from Camel"/>
  </bean>
```

For further information on service beans, refer to the official Red Hat documentation.[24]

The Java class implementing the bean is provided at the following location: *src/main/java/com/redhat/refarch/fusetest/HelloBean.java* and contains the following code:

```
public class HelloBean implements Hello {

    private String say = "Hello World";

    public String hello() {
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        return say + " at " + sdf.format(new Date());
    }

    public String getSay() {
        return say;
    }

    public void setSay(String say) {
        this.say = say;
    }
}
```

The Java bean allows for the *say* variable value to be modified and uses it to return a phrase in its *say* method that includes the current time.

The service descriptor then defines the Camel context:

```
  <camelContext id="blueprintContext" trace="false"
                xmlns="http://camel.apache.org/schema/blueprint">
```

The Camel context changes the default and implicit namespace definition from *blueprint* to *camel blueprint* for all its child elements.

24 https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Fuse/6.1/html/Deploying_into_the_Container/DeploySimple.html#DeploySimple-Blueprint-DefBean

This configuration also turns off Camel route tracing. The *Camel Tracer* prints a log message for every component in the Camel route and can be very useful for understanding and debugging Camel functionality, however it is very verbose and can be counterproductive in routine use.

This Camel context includes a single Camel route. While a single Camel context may include two or more nodes, this reference environment elects to separate the routes and insert each in its own distinct Camel context and file:

```
    <route id="timerToLog">
      <from uri="timer:foo?period=5000"/>
      <setBody>
          <method ref="helloBean" method="hello"/>
      </setBody>
      <log message="The message contains ${body}"/>
      <to uri="mock:result"/>
    </route>
```

The route is given an ID, which is optional since the context only includes a single route.

The Camel route begins with a timer that is arbitrarily named *foo*. This timer is configured on the spot to invoke the route every 5000 milliseconds, or in other words, every 5 seconds. For further information on the Camel Timer component, refer to the official Red Hat documentation.[25]

The first action on the Camel route, once invoked, is to set the message body. The service refers to the previously defined bean, invoking the *hello* method of *helloBean* and setting the response as the Camel route message body.

The next step in the pipeline makes use of the Camel logger. In its simplest form as shown here, the Camel Log component simply logs the provided message. This message can include variables such as body and header. For example to print the value of a header variable called *myVar*, use: *${header.myVar}*

The logger may also be used as a destination with the “log:” URI and include a large number of options to modify its behavior. For example, to view all headers and properties as well as the message body, use the *showAll* option:

```
<to uri="log:com.redhat.refarch.fusetest?showAll=true"/>
```

Multiple options can be added using an ampersand, which has to be escaped in the XML format, for example:

```
<to uri="log:com.redhat.refarch.fusetest?showAll=true&amp;multiline=true"/>
```

For further information on log options, refer to the Red Hat documentation on the subject.[26]

Finally, the message is routed to a mock destination which is a Camel component that retains messages in memory for testing and validation purposes.[27]

25 https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Fuse/6.1/html/Apache_Camel_Component_Reference/IDU-Timer.html
26 https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Fuse/6.1/html/Apache_Camel_Component_Reference/IDU-Log.html
27 https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Fuse/6.1/html/Apache_Camel_Component_Reference/IDU-Mock.html

The *pom* file for this sample JBoss Fuse project is fairly standard. It is configured to build and package an OSGi bundle.

Each Maven project is uniquely identified by the combination of its group ID, artifact ID and version. You entered values for all three while creating the project through the new project wizard dialog. Optionally, edit the pom file to change the name of the OSGi bundle that will be generated and eventually installed in an OSGi container.

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

  <modelVersion>4.0.0</modelVersion>

  <groupId>com.redhat.refarch</groupId>
  <artifactId>fusetest</artifactId>
  <packaging>bundle</packaging>
  <version>1.0.0-SNAPSHOT</version>

  <name>A Camel Blueprint Route</name>
  <url>http://www.myorganization.org</url>
```

To avoid duplication and ease maintenance, pom files allow properties to be defined and used later. These build properties are not used by the automatically generated pom file:

```
<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
</properties>
```

Instead of relying on a user's default Maven settings, a pom file can also define repositories that are used to resolve dependencies:

```
<repositories>
 <repository>
   <id>release.fusesource.org</id>
   <name>FuseSource Release Repository</name>
   <url>http://repo.fusesource.com/nexus/content/repositories/releases</url>
   <snapshots>
     <enabled>false</enabled>
   </snapshots>
   <releases>
     <enabled>true</enabled>
   </releases>
 </repository>
 <repository>
   ...
   ...
  </repository>
</repositories>
```

Plugin repositories can similarly be configured:

```
<pluginRepositories>
  <pluginRepository>
   <id>release.fusesource.org</id>
   <name>FuseSource Release Repository</name>
   <url>http://repo.fusesource.com/nexus/content/repositories/releases</url>
   <snapshots>
     <enabled>false</enabled>
   </snapshots>
   <releases>
     <enabled>true</enabled>
   </releases>
  </pluginRepository>
  <pluginRepository>
   <id>ea.fusesource.org</id>
   <name>FuseSource Community Early Access Release Repository</name>
   <url>http://repo.fusesource.com/nexus/content/groups/ea</url>
   <snapshots>
    <enabled>false</enabled>
   </snapshots>
   <releases>
    <enabled>true</enabled>
   </releases>
  </pluginRepository>
  ...
  ...
</pluginRepositories>
```

This project uses the Camel timer, log, mock and other basic features, all part of the *camel-core* project. Dependency injection is performed through *camel-blueprint*. These dependencies must be declared in the pom file:

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

Logging also introduces a number of dependencies:

```
    <!-- logging -->
    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-api</artifactId>
      <version>1.7.5</version>
    </dependency>
    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-log4j12</artifactId>
      <version>1.7.5</version>
    </dependency>
    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>jcl-over-slf4j</artifactId>
      <version>1.7.5</version>
    </dependency>
    <dependency>
      <groupId>log4j</groupId>
      <artifactId>log4j</artifactId>
      <version>1.2.17</version>
    </dependency>
```

Finally, *camel-test-blueprint* is only required for testing. Accordingly, it is given a scope of test to avoid its inclusion in the production deployment:

```
    <!-- testing -->
    <dependency>
      <groupId>org.apache.camel</groupId>
      <artifactId>camel-test-blueprint</artifactId>
      <version>2.12.0.redhat-610379</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
```

Like most Maven projects, the default Maven goal is to install the project in the Maven repository:

```
  <build>
    <defaultGoal>install</defaultGoal>
```

Build plugins provide Maven with a mechanism to extend its use to various products and features. Installing the project simply requires Java compilation and the inclusion of resources:

```
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>2.5.1</version>
        <configuration>
          <source>1.6</source>
          <target>1.6</target>
        </configuration>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-resources-plugin</artifactId>
        <version>2.6</version>
        <configuration>
          <encoding>UTF-8</encoding>
        </configuration>
      </plugin>
```

Generating an OSGi bundle adds a requirement to insert a manifest file with further information about the project. Maven uses a dedicated plugin for this purpose:

```
      <!-- to generate the MANIFEST-FILE of the bundle -->
      <plugin>
        <groupId>org.apache.felix</groupId>
        <artifactId>maven-bundle-plugin</artifactId>
        <version>2.3.7</version>
        <extensions>true</extensions>
        <configuration>
          <instructions>
            <Bundle-SymbolicName>fusetest</Bundle-SymbolicName>
            <Private-Package>com.redhat.refarch.fusetest.*</Private-Package>
            <Import-Package>*</Import-Package>
          </instructions>
        </configuration>
      </plugin>
```

Note that the manifest file declares a symbolic name for the OSGi bundle, specifies its base working package and declares packages that are imported. In this case, the sample project imports all declared dependency packages.

Finally, another Maven build plugin is required to easily and quickly run the project through the *camel:run* goal, without the benefit of a container or the effort of proper deployment.

```
      <!-- to run the example using mvn camel:run -->
      <plugin>
        <groupId>org.apache.camel</groupId>
        <artifactId>camel-maven-plugin</artifactId>
        <version>2.12.0.redhat-610379</version>
        <configuration>
          <useBlueprint>true</useBlueprint>
        </configuration>
      </plugin>

    </plugins>
  </build>

</project>
```

The *camel:run* goal is only used for testing and development. In larger projects, as development progresses and external dependencies multiply, this approach of local execution often stops being feasible.

5.2.5 Sample Project Execution

JBoss Developer Studio comes pre-installed with a Maven plugin that allows you to build and even run some Maven projects within the IDE environment.

Right-click on the *pom.xml* file and select *Run As* from the context menu. Two instances of *Maven Build* appears for a pom file, where the second option is followed by an ellipsis. Initially, both options behave identically. Once a Maven file is executed through the plugin, the first option acts as a quick action and runs the most recent Maven goal without prompting you for the goal and other configuration options.

Select the first *Maven Build* option from the *Run As* menu that appears for your project pom.xml. In the Goals dialog box, type "*install camel:run*":

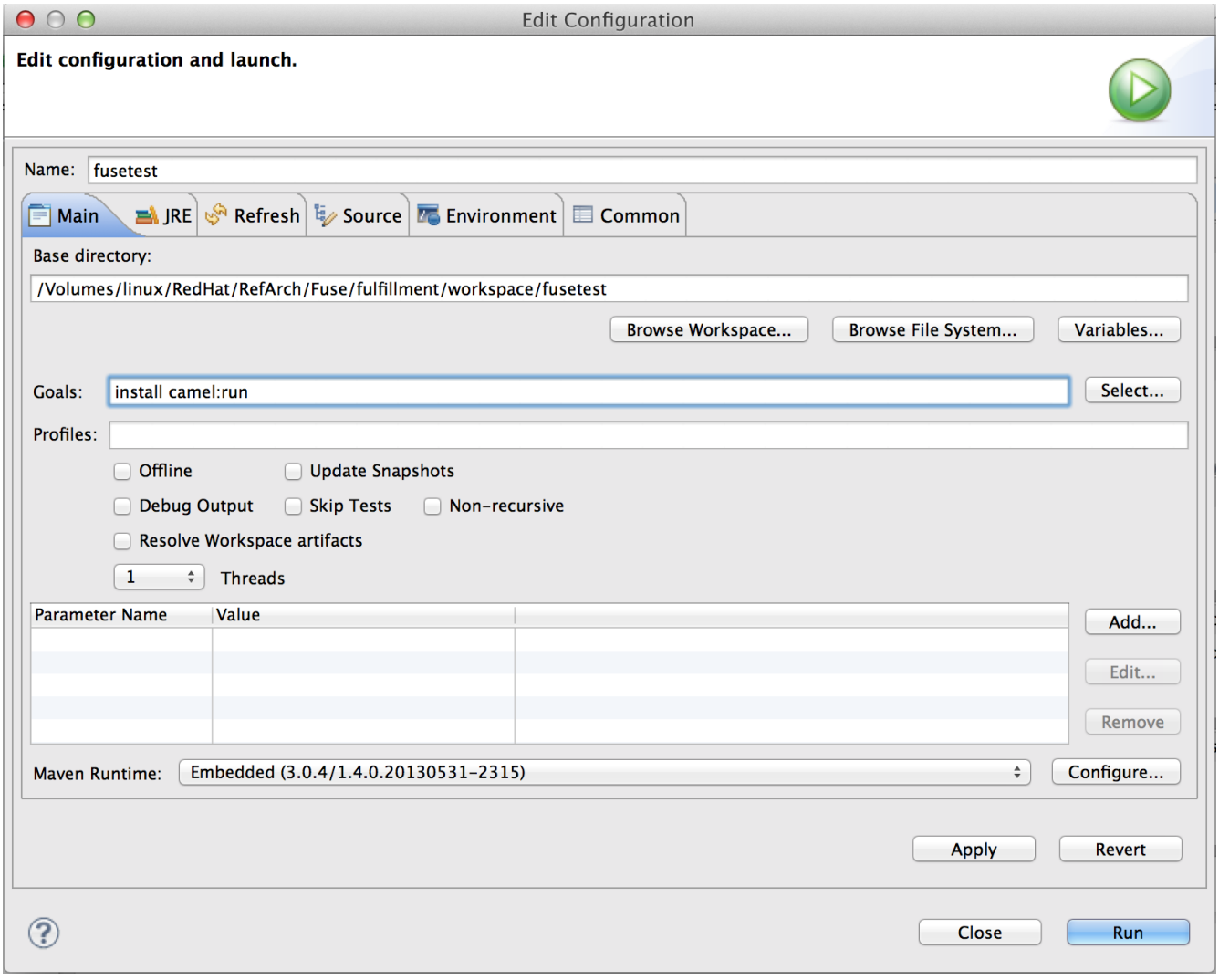


Figure 5.2.5-1: Maven Run Configuration

Run this Maven goal to execute the project and save this configuration under the given name.

Once executed, observe the *Console* view. Adjust the size of the view or maximize it for better monitoring ability.

You should see a message printed every 5 seconds, invoked by the timer:

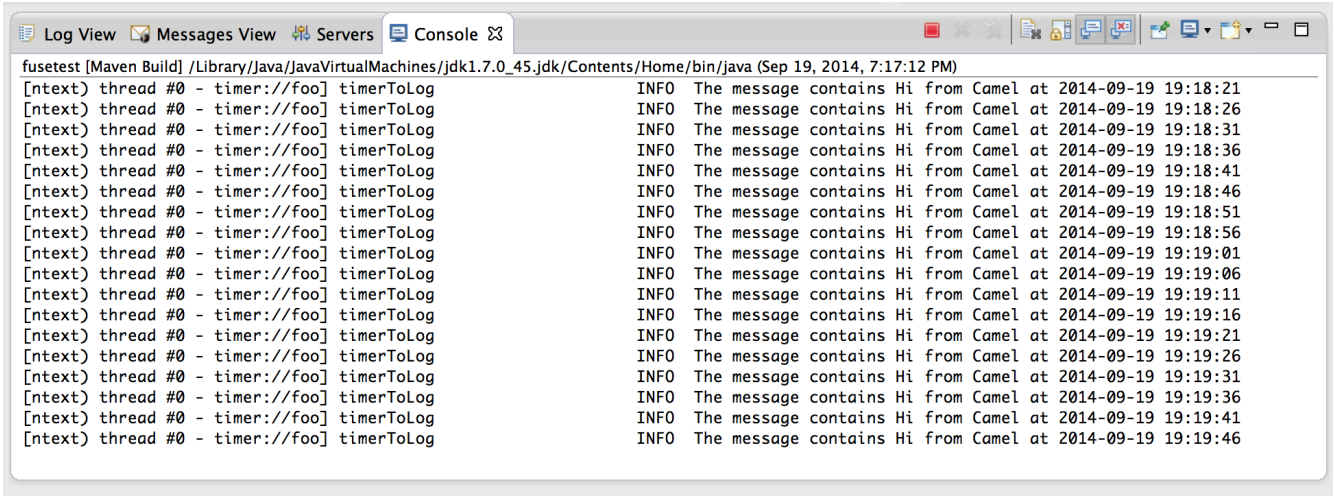


Figure 5.2.5-2: JBDS Output Console

With Maven in your execution path, it is very easy to reproduce the same results in a command prompt. First, to build the project and install it in the Maven repository:

```
# mvn install
```

Look for a success prompt:

```
[INFO]
[INFO] --- maven-bundle-plugin:2.3.7:install (default-install) @ fusetest
---
[INFO] Installing com/redhat/refarch/fusetest/1.0.0-SNAPSHOT/fusetest-1.0.0-
SNAPSHOT.jar
[INFO] Writing OBR metadata
[INFO]
------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO]
------------------------------------------------------------------------
[INFO] Total time: 5.124s
```

Then:

```
# mvn camel:run
```

After initial setup, the timer thread will start and log a line every 5 seconds:

```
[ntext) thread #0 - timer://foo] timerToLog                    INFO  The
message contains Hi from Camel at ...
```

5.3 Initial Project Iteration

5.3.1 Overview

In this iteration, you will design, develop and deploy a very simple application that covers various aspects of using JBoss Fuse without diving deep into technical details, or being concerned with potential challenges. Once able to understand and replicate this process, you will be able to develop a generic Fuse application and deploy it to a distributed and highly-available environment.

5.3.2 Project Structure

To allow the use of multiple Maven projects, creating a directory that will serve as the single parent directory for all the projects. This guide assumes that this parent directory is called *code*.

5.3.3 File Polling Project

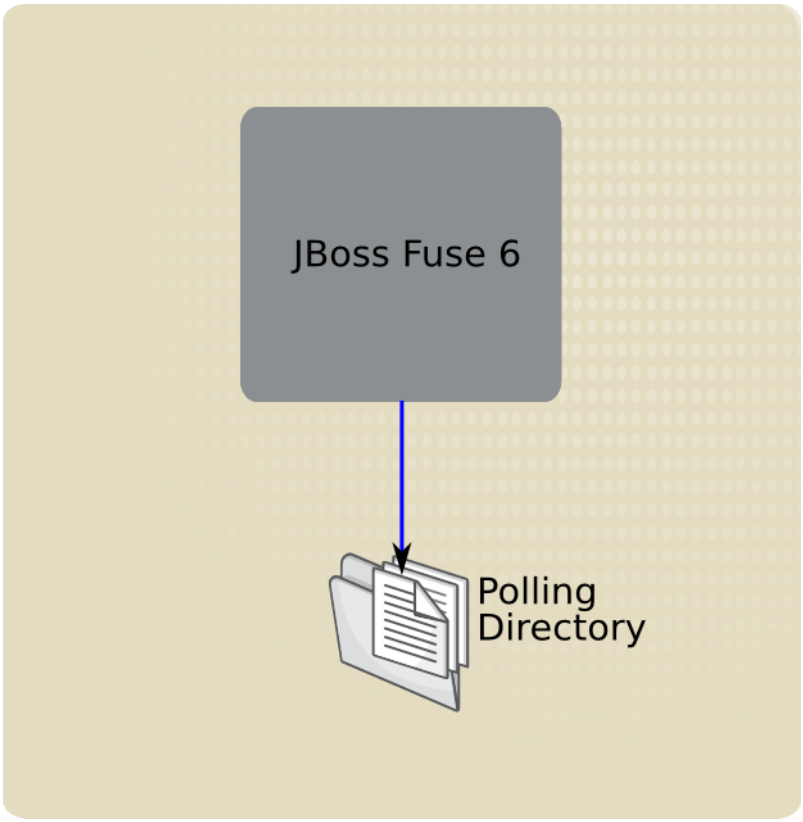


Figure 5.3.3-1: Simple File Polling

5.3.3.1 Development

Create a directory called fulfillment under the previously created code directory. This directory will host the file polling project, which in turn will gradually evolve into the Order Fulfillment application.

Use JBoss Developer Studio to create a new *Fuse Project*. Instead of the default workspace, browse and point to *code/fulfillment* as the project location.

Select the *camel-archetype-blueprint* artifact from the *org.apache.camel.archetypes* group as the project archetype. Enter the following Maven project configuration:

- groupId: com.redhat.refarch.fuse
- artifactId: fulfillment
- version: 1.0.0

Accept the proposed package as it defaults to a combination of the group and artifact: *com.redhat.refarch.fuse.fulfillment.*

The created project uses a few Apache Camel features. You will only use the project structure and build files.

Delete the blueprint service descriptor, the Java Bean and the associated test package:

- *code/fulfillment/src/main/resources/OSGI-INF/blueprint/blueprint.xml*
- *code/fulfillment/src/main/java/com/redhat/refarch/fuse/fulfillment/Hello.java*
- *code/fulfillment/src/main/java/com/redhat/refarch/fuse/fulfillment/HelloBean.java*
- *code/fulfillment/src/test/java/**

Use JBoss Developer Studio to create a new Camel XML file under *src/main/resources/OSGI-INF/blueprint/blueprint.xml*. When JBDS is set to the Fuse Integration perspective, you can use the drop-down from the toolbar and select Camel XML File, while having previously selected the *blueprint* directory so that it becomes the containing folder in the wizard.

Alternatively, right-click on the *src/main/resources/OSGI-INF/blueprint* directory, select *New* and choose *Camel XML File*. Enter the filename as csv.xml, since this Camel route will be used to pick up and process CSV files. Set the framework to OSGi Blueprint:

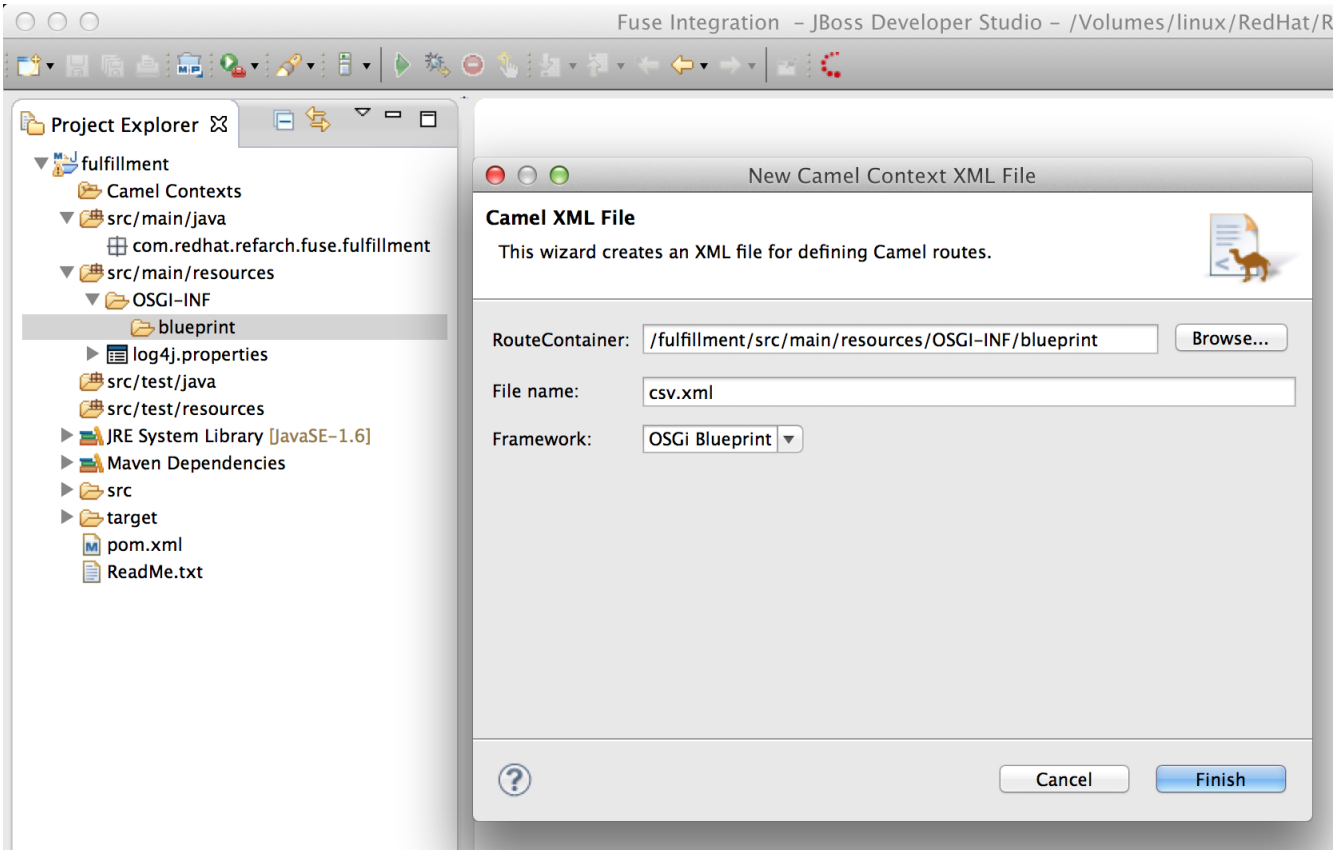


Figure 5.3.3-2: New Camel Context

5.3.3.2 Graphical Development

The Camel context file opens the *Fuse Tooling Routes Editor* by default. This graphical editor allows you to define and configure Camel routes through drag and drop, and property editing. It is particular helpful as a learning tool, although more advanced users will often prefer to directly edit the Camel XML files.

Locate the palette of Camel tools and components on the right-side edge of the canvas. From the *Endpoints* group, drag and drop *Endpoint* onto the canvas

Drag and drop *Log* onto the canvas, placing it or later moving it, to the right of the previously placed *Endpoint*.

Click on the Endpoint node and hover the mouse over the arrow that appears to display its label as “Create Connection”:

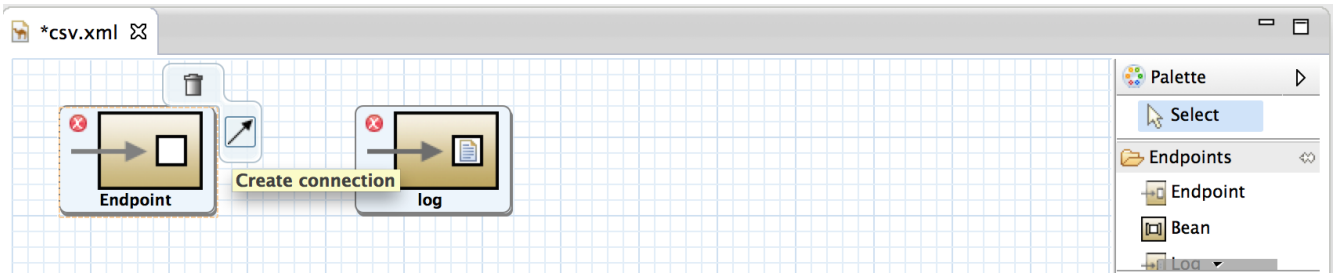


Figure 5.3.3-3: Connecting Nodes

Click the arrow and drag it onto the log node, dropping it there. Click on the first node, the *Endpoint*, and open the *properties* view. Set up the properties for this node as follows:

- Uri: file:///fulfillment/input/
- Id: CSV File Poll

Then click on the log node and set up the log message:

- Message: Received ${file:name} with the following content: ${body}

At this point, you have created a very simple Fuse application consisting of a single Camel context with a single route. The first node uses the Apache Camel File component to poll the specified directory and pick up any files that are dropped in there.[28] Any dropped file is processed and passed down the route to the next node, which is a simple logger.

The log simply prints a message, including the name of the file that's been picked up and its content, respectively identified by the *${file:name}* and *${body}* variables.

28 https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Fuse/6.1/html/Apache_Camel_Component_Reference/IDU-File2.html

5.3.3.3 Service Descriptor

Review the content of Camel *csv.xml* file, either by using a third-party editor or right-clicking on the file in JBDS and selecting *XML Editor* in the *Open With* menu:

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

  <camelContext trace="false"
                      xmlns="http://camel.apache.org/schema/blueprint">
    <route>
        <from uri="file:///fulfillment/input/" customId="true"
                                                    id="CSV File Poll">
            <description/>
        </from>
        <log message="Received ${file:name} with the following content: $
{body}"/>
    </route>
</camelContext>

</blueprint>
```

The route definition is straight forward and easy to understand.

5.3.3.4 Build

Open a terminal and navigate to the *code/fulfillment* directory where the project root is located. Use Maven to build the project:

```
# mvn install
```

5.3.3.5 Execution

Run the project using Maven:

```
# mvn camel:run
```

This starts the Camel route and has it poll for files in the designated directory:

```
Route: route1 started and consuming from:
                            Endpoint[file:///fulfillment/input/]
INFO  Total 1 routes, of which 1 is started.
INFO  Apache Camel 2.12.0.redhat-610379 (CamelContext: camel-1)
                                                started in 0.323 seconds
```

Create a simple file and drop it in this directory. You can use your favorite text editor to type a few words and save the file with in that directory, or in a Linux / Unix environment (including OS X), open another terminal and type:

```
# echo "Testing" > /fulfillment/input/test.csv
```

The file is detected and picked up almost immediately. Observe the other terminal for the log output:

```
[0 - file:///fulfillment/input/] route1
          INFO  Received test.csv with the following content: Testing
```

5.3.3.6 Maven POM File

Review and make sure to understand the generated Maven file that successfully builds and runs this simple application. The Maven application is identified by its group, artifact and version attributes, which you specified while creating the JBDS project:

```
  <groupId>com.redhat.refarch.fuse</groupId>
  <artifactId>fulfillment</artifactId>
  <packaging>bundle</packaging>
  <version>1.0.0</version>

  <name>A Camel Blueprint Route</name>
  <url>http://www.myorganization.org</url>
```

Note that the packaging is set to bundle so that building the project with Maven would create an OSGi bundle and not a simple Java JAR file.

The project file then configures the repositories and plugin repositories to be used for this project. This enables Maven to download any dependencies that are listed, but not available in the local repository. The dependencies are listed next in the pom file:

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

Camel and Camel Blueprint are the two functional dependencies of this project. The application is built on top of the blueprint dependency injection model using a Camel context, so camel-blueprint is an obvious dependency. This module itself depends on camel-core, as do various features such as the file polling component and the log.

Notice that the version of required Camel components is listed as 2.12.0.redhat-610379. This is a release version of Camel that is provided by Red Hat. Look at the *FuseSource Release Repository* and follows its address to find these libraries, for example:

http://repo.fusesource.com/nexus/content/repositories/releases/org/apache/camel/camel-blueprint/2.12.0.redhat-610379/

Another listed dependency is **`slf4j`** and its associated modules. This is used for logging to the console:

```
    <!-- logging -->
    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-api</artifactId>
      <version>1.7.5</version>
    </dependency>
    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-log4j12</artifactId>
      <version>1.7.5</version>
    </dependency>
    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>jcl-over-slf4j</artifactId>
      <version>1.7.5</version>
    </dependency>
    <dependency>
      <groupId>log4j</groupId>
      <artifactId>log4j</artifactId>
      <version>1.2.17</version>
    </dependency>
```

Finally, the Maven file has a testing dependency that is no longer requires, since the generated route and its associated test have been removed:

```
    <!-- testing -->
    <dependency>
      <groupId>org.apache.camel</groupId>
      <artifactId>camel-test-blueprint</artifactId>
      <version>2.12.0.redhat-610379</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
```

Manually remove this dependency, or find it in the Dependencies tab of JBDS while opening the file with the *Maven POM Editor*, and select it and press the remove button.

The pom file also selects install as the default Maven goal for this project:

```
  <build>
    <defaultGoal>install</defaultGoal>
```

The required plugins are configured to compile any Java classes and include any provided resources.

To create an OSGi bundle, the required build plugin extension is declared:

```
      <!-- to generate the MANIFEST-FILE of the bundle -->
      <plugin>
        <groupId>org.apache.felix</groupId>
        <artifactId>maven-bundle-plugin</artifactId>
        <version>2.3.7</version>
        <extensions>true</extensions>
        <configuration>
          <instructions>
            <Bundle-SymbolicName>fulfillment</Bundle-SymbolicName>
            <Private-Package>com.redhat.refarch.fuse.fulfillment.*</Private-
Package>
            <Import-Package>*</Import-Package>
          </instructions>
        </configuration>
      </plugin>
```

Finally, the camel-maven-plugin is configured to allow you to run the project through Maven without having to use an OSGi container.

5.3.3.7 Container Setup

While Camel can be very helpful to quickly test and validate simple projects, deploying to a proper container is almost always a production requirement. Set up a JBoss Fuse environment as previously outlined in the chapter 3, titled Reference Architecture Environment. For the purpose of this application, it is enough to download, unzip and run JBoss Fuse:

```
# unzip jboss-fuse-full-6.1.0.redhat-379.zip

# jboss-fuse-6.1.0.redhat-379/bin/fuse
```

Assuming the JDK dependency is available and found, you will be greeted by a prompt along with a warning that no user has been set up. You will also notice that the fuse instance is simply called *root*:

```
No user found in etc/users.properties. Please use the 'esb:create-admin-
user' command to create one.

JBossFuse:karaf@root>
```

While not necessary for the purpose of this simple application, it is generally required to set up an administrator account for every JBoss Fuse installation. You can configure an account by editing *jboss-fuse-6.1.0.redhat-379/etc/users.properties* and uncommenting the user configuration, while changing the password, or through the fuse shell:

```
JBossFuse:karaf@root> esb:create-admin-user
Please specify a user...
New user name: admin
Password for admin: password
Verify password for admin: password

JBossFuse:karaf@root>
```

It is also good practice to properly name each Fuse installation. Do this by editing *jboss-fuse-6.1.0.redhat-379/etc/system.properties* and changing the karaf.name property to a more meaningful and unique name:

```
#
# Name of this Karaf instance.
#
karaf.name=root
```

Create a simple Fabric Ensemble with a single Fabric Server, appropriate only for testing purposes since it provides no fault tolerance.[29]

Start by creating the first Fabric container. Specify a zookeeper password that will be required for other nodes to join, or in subsequence sessions:

```
JBossFuse:karaf@root> fabric:create --zookeeper-password password --wait-
for-provisioning
Waiting for container: root
Waiting for container root to provision.
Using specified zookeeper password:password

JBossFuse:karaf@root>
```

Extend the fabric by creating a child container. This container will run in its own JVM instance and can be used to deploy and run the application, while being managed from the previously created container. Call this child container *child1*:

```
JBossFuse:karaf@root> fabric:container-create-child root child1
The following containers have been created successfully:
    Container: child1.

JBossFuse:karaf@root>
```

This process creates and runs a JVM for a child container and usually takes a few seconds. Look and tail the log file for this child container at *jboss-fuse-6.1.0.redhat-379/instances/child1/data/log/karaf.log*

Use the container-info command to get the status of the newly created container:

```
JBossFuse:karaf@root> container-info child1
Name:                          child1
Version:                       1.0
Alive:                         false
Resolver:                      localhostname
Network Address:               babaks-mbp.home
SSH Url:                       null
JMX Url:                       null
Process ID:                    null
Profiles:                      default
Provision Status:
```

Notice the provision status has no value when you first query the container.

29 https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Fuse/6.1/html/Fabric_Guide/GetStart.html#Deploy-Fabric-Create

A few seconds later, when the container has been successfully provisioned, the container info looks as follows:

```
JBossFuse:karaf@root> container-info child1
Name:                          child1
Version:                       1.0
Alive:                         true
Resolver:                      localhostname
Network Address:               babaks-mbp.home
SSH Url:                       babaks-mbp.home:8102
JMX Url:                       service:jmx:rmi://babaks-
mbp.home:44445/jndi/rmi://babaks-mbp.home:1100/karaf-child1
Process ID:                    10696
Profiles:                      default
Provision Status:              success
Blueprint Status:              started
```

The blueprint status of started indicates that the container is ready for use.

Connect to the container to install the application:

```
JBossFuse:karaf@root> fabric:container-connect child1
Connected
       _ ____                  ______
      | |  _ \                |  ____|
      | | |_) | ___  ___ ___  | |__ _   _ ___  ___
  _   | |  _ < / _ \/ __/ __| |  __| | | / __|/ _ \
 | |__| | |_) | (_) \__ \__ \ | |  | |_| \__ \  __/
  \____/|____/ \___/|___/___/ |_|   \__,_|___/\___|

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

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

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

Create a new Fabric via 'fabric:create'
or join an existing Fabric via 'fabric:join [someUrls]'

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

JBossFuse:admin@child1>
```

5.3.3.8 Container Deployment

To deploy your application, it is important to fully understand its dependencies.

The new child container is always set up with the *default* Fuse profile, resulting in a number of OSGi bundles being installed, started or otherwise available. This includes the minimum required components for JBoss Fuse to run and start an application.

The fulfillment application depends on *camel-core* and *camel-blueprint* for dependency injection, route execution and file polling. Specifically, its pom file declares dependency on:

```
    <dependency>
      <groupId>org.apache.camel</groupId>
      <artifactId>camel-core</artifactId>
      <version>2.12.0.redhat-610379</version>
    </dependency>
    <dependency>
      <groupId>org.apache.camel</groupId>
      <artifactId>camel-blueprint</artifactId>
      <version>2.12.0.redhat-610379</version>
    </dependency>
```

To install an OSGi bundle, use *osgi:install:*

```
JBossFuse:admin@child1> osgi:install --help
DESCRIPTION
        osgi:install

     Installs one or more bundles.

SYNTAX
        osgi:install [options] urls

ARGUMENTS
        urls
                Bundle URLs separated by whitespaces

OPTIONS
        --help
                Display this help message
        -s, --start
                Starts the bundles after installation
```

In the case of the fulfillment project, you have previously built the project by running *mvn install*. That means the dependencies have been resolved and the project bundle along with the dependency bundles have all been installed in your default local Maven repository. The default local Maven repository is typically located under a directory called “.m2” in the user's home directory. To verify and view the built project bundle, look inside the following directory within the user's home directory:

.m2/repository/com/redhat/refarch/fuse/fulfillment/1.0.0/

Similarly, the camel-core and camel-blueprint bundles will be built and installed in your local repository under the following address relative to your home directory:

- *.m2/repository/org/apache/camel/camel-core/2.12.0.redhat-610379/*
- *.m2/repository/org/apache/camel/camel-blueprint/2.12.0.redhat-610379/*

The fulfillment application, as currently built to only include file polling, depends on both camel-blueprint and camel-core. However camel-blueprint itself depends on camel-core, so the order of installation is first camel-core, followed by camel-blueprint and then fulfillment itself. For a bundle built through Maven and available in the repository, the OSGi URL is in the form of *mvn:groupId/artifactId/version*, resulting in the following install commands:

```
JBossFuse:admin@child1> osgi:install -s mvn:org.apache.camel/camel-
core/2.12.0.redhat-610379
Bundle ID: 102

JBossFuse:admin@child1> osgi:install -s mvn:org.apache.camel/camel-
blueprint/2.12.0.redhat-610379
Bundle ID: 103

JBossFuse:admin@child1> osgi:install -s
mvn:com.redhat.refarch.fuse/fulfillment/1.0.0
Bundle ID: 104
```

Look at the last few lines of the child1 container log and make sure that the container and the Camel route have started successfully. The log file for the child container is located at:

jboss-fuse-6.1.0.redhat-379/instances/child1/data/log/karaf.log

The following is an example of the last few lines of a successful startup log:

```
... BlueprintCamelContext            | e.camel.impl.DefaultCamelContext 1730
| 102 - org.apache.camel.camel-core - 2.12.0.redhat-610379 | StreamCaching
is not in use. If using streams then its recommended to enable stream
caching. See more details at http://camel.apache.org/stream-caching.html
... BlueprintCamelContext            | e.camel.impl.DefaultCamelContext 2224
| 102 - org.apache.camel.camel-core - 2.12.0.redhat-610379 | Route: route1
started and consuming from: Endpoint[file:///fulfillment/input/]
... BlueprintCamelContext            | e.camel.impl.DefaultCamelContext 1568
| 102 - org.apache.camel.camel-core - 2.12.0.redhat-610379 | Total 1 routes,
of which 1 is started.
... BlueprintCamelContext            | e.camel.impl.DefaultCamelContext 1569
| 102 - org.apache.camel.camel-core - 2.12.0.redhat-610379 | Apache Camel
2.12.0.redhat-610379 (CamelContext: camel-1) started in 0.144 seconds
```

Follow the instructions in the Execution section to create a sample file and invoke the file polling project. Once again, the container log file can be used to validate that the application is successfully deployed and running on the container:

```
... | INFO  | lfillment/input/ | route1                                |
rg.apache.camel.util.CamelLogger  176 | 102 - org.apache.camel.camel-core -
2.12.0.redhat-610379 | Received test.csv with the following content: Testing
```

5.3.4 Fuse Feature

Because applications and other tools typically consist of multiple OSGi bundles, it is often convenient to aggregate inter-dependent or related bundles into a larger unit of deployment. Red Hat JBoss Fuse therefore provides a scalable unit of deployment, the Feature, which enables you to deploy multiple bundles (and, optionally, dependencies on other Features) in a single step.[30]

A Fuse Feature needs to be configured in a Feature repository. Start by creating a custom Feature repository called *fulfillment-feature_repository*. Create a directory called *feature* under the previously created *code/* directory and a the following repository XML file:

code/features/src/main/resources/fulfillment.xml

Edit the above Feature repository file and provide the following content:

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

Create a Feature called fulfillment-feature by adding a *<feature>* element in the repository:

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

  <feature name="fulfillment-feature">

  </feature>

</features>
```

The fulfillment Feature includes the application and its dependencies. The application itself is built through Maven as an OSGi bundle. Add it to the Feature with the following bundle declaration:

```
    <bundle>mvn:com.redhat.refarch.fuse/fulfillment/1.0.0</bundle>
```

30 https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Fuse/6.1/html/Deploying_into_the_Container/DeployFeatures.html

JBoss Fuse Features allow building on top of other Features to simplify dependency management. To find out the configured Features, first query the Feature Repositories configured for the target container:

```
JBossFuse:admin@child1> features:listrepositories
Repository
repo-0
karaf-2.3.0.redhat-610379
org.ops4j.pax.web-3.0.6
karaf-enterprise-2.3.0.redhat-610379
camel-2.12.0.redhat-610379
cxf-2.7.0.redhat-610379
jclouds-1.7.1
fabric-camel-cxf-1.0.0.redhat-379
hawtio-1.2-redhat-379
spring-2.3.0.redhat-610379
fabric-activemq-demo-1.0.0.redhat-379
activemq-5.9.0.redhat-610379
activemq-core-5.9.0.redhat-610379
fabric-1.0.0.redhat-379
camel-example-sap-1.0.0.redhat-379
camel-sap-1.0.0.redhat-379
servicemix-4.5.0.redhat-610379
fabric-cxf-demo-1.0.0.redhat-379
repo-0
quickstart-jms-6.1.0.redhat-379
```

The fulfillment application requires camel-core and camel-blueprint. As can be seen from the list of Feature Repositories above, a Feature Repository called *camel-2.12.0.redhat-610379* is configured on the target container. To view all the Features configured in all these repositories:

```
JBossFuse:admin@child1> features:list
State Version  Name                  Repository           Description
...   ...      naming                repo-0
...   ...      document              repo-0
...   ...      ...                   ...
...   ...      xml-specs-api         camel-2.12.0.redhat-610379
...   ...      camel                 camel-2.12.0.redhat-610379
...   ...      camel-core            camel-2.12.0.redhat-610379
...   ...      camel-spring          camel-2.12.0.redhat-610379
...   ...      camel-blueprint       camel-2.12.0.redhat-610379
...   ...      camel-ahc             camel-2.12.0.redhat-610379
...   ...      camel-amqp            camel-2.12.0.redhat-610379
```

Look at the Features configured in the *camel-2.12.0.redhat-610379* repository. Remember that by design, Fuse Features declare all their dependencies. That means that declaring a dependency on *camel-blueprint* is enough to install and import *camel-core*.

Edit the fulfillment-feature repository and add camel-blueprint as a dependency Feature:

```
<?xml version="1.0" encoding="UTF-8"?>
<features name="fulfillment-feature_repository">
  <feature name="fulfillment-feature">
    <bundle>mvn:com.redhat.refarch.fuse/fulfillment/1.0.0</bundle>

    <feature>camel-blueprint</feature>
  </feature>
</features>
```

To build this Feature, create a simple Maven POM file to create the appropriate package:

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

  <modelVersion>4.0.0</modelVersion>
  <groupId>com.redhat.refarch.fuse</groupId>
  <artifactId>fulfillment-feature</artifactId>
  <packaging>jar</packaging>
  <version>1.0.0</version>
  <name>Fulfillment feature repository</name>

  <build>
    <plugins>
      <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/fulfillment.xml</file>
                    <type>xml</type>
                    <classifier>features</classifier>
                  </artifact>
                </artifacts>
              </configuration>
            </execution>
          </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

5.3.5 Aggregation POM

At this point, the fulfillment and fulfillment-feature projects are set up as two distinct modules, each with their own Maven build process. In some cases, although not in this example, one module may depend on another and require it to be built first.

Whether for module inter-dependency or simply convenience, an aggregation POM can be very useful.

Create a file called *pom.xml* under your *code/* directory with the following content:

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

    <modelVersion>4.0.0</modelVersion>
    <groupId>com.redhat.refarch.fuse</groupId>
    <artifactId>order-fulfillment</artifactId>
    <version>1.0.0</version>
    <packaging>pom</packaging>
    <url>http://maven.apache.org</url>
    <name>Parent Project</name>

    <modules>
        <module>fulfillment</module>
        <module>features</module>
    </modules>

</project>
```

The group and artifact ID for your aggregation project has no functional significance. Choose a unique group and artifact ID to avoid clashing with other projects and one that is meaningful for this aggregation.

This aggregation POM allows you to clean both the fulfillment and fulfillment-feature projects by simply running mvn clean from the root code/ directory and install both of them by running mvn install from code/:

```
mvn install
...
[INFO]
------------------------------------------------------------------------
[INFO] Building A Camel Blueprint Route 1.0.0
[INFO]
------------------------------------------------------------------------
...
[INFO]
------------------------------------------------------------------------
[INFO] Building Fulfillment feature repository 1.0.0
[INFO]
------------------------------------------------------------------------
```

5.3.6 Fuse Fabric Deployment

5.3.6.1 Profiles

JBoss Fuse Fabrics use profiles as the basic unit of deployment. Profiles can be created and edited for a Fabric and deployed to any number of containers to simplify deployment across an ensemble and promote consistency. You can also create different versions of a profile, which makes it possible to support rolling upgrades across the containers in your fabric.[31]

A profile is a description of how to provision a logical group of containers. Each profile can have none, one, or more parents, which allows you to have profile hierarchies. A container can be assigned one or more profiles. Profiles are also versioned, which enables you to maintain different versions of each profile, and then upgrade or roll back containers, by changing the version of the profiles they use. A profile can include the following resources:[32]

- OSGi bundle URLs
- Web ARchive (WAR) URLs
- Fuse Application Bundle (FAB) URLs
- OSGi Configuration Admin PIDs
- Apache Karaf feature repository URLs
- Apache Karaf features
- Maven artifact repository URLs
- Blueprint XML files or Spring XML files (for example, for defining broker configurations or Camel routes)
- Any kind of resource that might be needed by an application (for example, Java properties file, JSON file, XML file, YML file)
- System properties that affect the Apache Karaf container (analogous to editing etc/config.properties)
- System properties that affect installed bundles (analogous to editing etc/system.properties)

JBoss Fuse Fabric comes preconfigured with a number of profiles. Most notably, that includes:

- *default*: The default profile defines all of the basic requirements for a Fabric container. For example it specifies the fabric-agent feature, the Fabric registry URL, and the list of Maven repositories from which artifacts can be downloaded.
- *karaf*: Inherits from the *default* profile and defines the **`Karaf`** feature repositories, which makes the **`Apache Karaf`** features accessible.

31 https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Fuse/6.1/html/Fabric_Guide/Profiles.html
32 https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Fuse/6.1/html/Fabric_Guide/Profiles.html#Profiles-Intro

- *feature-camel*: Inherits from *karaf*, defines the Camel feature repositories, and installs some core Camel features: such as *camel-core* and *camel-blueprint*. If you are deploying a Camel application, it is recommended that you inherit from this profile.
- *jboss-fuse-full*: Includes all of the features and bundles required for the JBoss Fuse full container.

For a more comprehensive list, refer to the official Red Hat documentation.[33]

Note that the *feature-camel* profile is a perfect fit for the fulfillment application, as it has developed up to this point. However, this reference architecture elects not to make use of this profile. By starting with a blank slate and inheriting from the simpler *default* profile, you will have more control and awareness over the modules and Features used by your application.

Follow these instructions to create a profile and use it to deploy the fulfillment application to the child container. If you have already created the *child1* container and deployed the OSGi bundles to it, as described in the 5.3.3.8 section, a quick and simple way to revert is to delete the container and recreate it:

```
JBossFuse:karaf@root> fabric:container-delete child1
JBossFuse:karaf@root> fabric:container-create-child root child1
The following containers have been created successfully:
    Container: child1.
JBossFuse:karaf@root>
```

To monitor progress as the child1 container is provisioned, you can also use the *shell:watch* command:

```
shell:watch fabric:container-list

[id]                            [version] [connected] [profiles]
[provision status]
root*                           1.0       true        fabric, fabric-
ensemble-0000-1, jboss-fuse-full    success
  child1                        1.0       true        default

[id]                            [version] [connected] [profiles]
[provision status]
root*                           1.0       true        fabric, fabric-
ensemble-0000-1, jboss-fuse-full    success
  child1                        1.0       true        default
installing

[id]                            [version] [connected] [profiles]
[provision status]
root*                           1.0       true        fabric, fabric-
ensemble-0000-1, jboss-fuse-full    success
  child1                        1.0       true        default
success
```

33 https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Fuse/6.1/html/Fabric_Guide/Profiles.html#Profiles-Intro

5.3.6.2 Creating a profile

Create a basic profile called *fulfillment-profile* that inherits from the default profile:

```
JBossFuse:karaf@root> fabric:profile-create --parents default fulfillment-
profile
JBossFuse:karaf@root>
```

5.3.6.3 Assinging a profile

Now that the new profile has been created, assign it to the *child1* container:

```
JBossFuse:karaf@root> fabric:container-change-profile child1 fulfillment-
profile
JBossFuse:karaf@root>
```

You can check the container log to monitor the deployment of this profile to the container. Previously installed bundles are removed and a new set of bundles are derived from the profile inheritance and its features, then installed and started as the case might be.

5.3.6.4 Feature deployment to profile

Apply the fulfillment-feature you created to this new profile. Make sure you have used *mvn install* to build and install the feature repository to your local Maven repository. Construct the Maven URL of the feature repository using its group ID, artifact ID and version:

mvn:com.redhat.refarch.fuse/fulfillment-feature/1.0.0/xml/features

Use this Maven URL to add the *fulfillment-feature_repository* feature repository to the profile:

```
JBossFuse:karaf@root> profile-edit -r
mvn:com.redhat.refarch.fuse/fulfillment-feature/1.0.0/xml/features
fulfillment-profile
Adding feature repository:mvn:com.redhat.refarch.fuse/fulfillment-
feature/1.0.0/xml/features to profile:fulfillment-profile version:1.0
JBossFuse:karaf@root>
```

Now that *fulfillment-feature_repository* is added to the profile as a repository, the next and final step is to add the feature:

```
JBossFuse:karaf@root> profile-edit --features fulfillment-feature
fulfillment-profile
Adding feature:fulfillment-feature to profile:fulfillment-profile
version:1.0
JBossFuse:karaf@root>
```

Once again, follow the instructions in the Execution section to create a sample file and invoke the file polling project. Once again, the container log file can be used to validate that the application is successfully deployed and running on the container:

```
... | INFO  | lfillment/input/ | route1                                |
rg.apache.camel.util.CamelLogger  176 | 102 - org.apache.camel.camel-core -
2.12.0.redhat-610379 | Received test.csv with the following content: Testing
```

5.3.7 Fuse Fabric Ensemble

Once a Fuse Fabric has been created with a given ZooKeeper password, it is very easy to create a *Fabric Ensemble*.

Install JBoss Fuse on the second node by unzipping the archive and changing the *Karaf* name property to a unique name, for example root2.

Create an administrator account, either by editing the *users.properties* file or through the create-admin-user command as described in the previous Container Setup section.

Once these few initialization steps have been performed and Fuse has been started on another node, join the Fuse Fabric created by the first node:

```
JBossFuse:karaf@root2> fabric:join --zookeeper-password password fuse-node1
JBossFuse:karaf@root2>
```

At this point, your second node has joined the Fuse Fabric. Since a Fuse Ensemble requires an odd number of members[34], repeat these steps on a third node and join the Fabric from there as well:

```
JBossFuse:karaf@root3> fabric:join --zookeeper-password password fuse-node1
JBossFuse:karaf@root3>
```

Now go back to the first node and list the available containers:

```
JBossFuse:admin@root> fabric:container-list
[id]                           [version] [connected] [profiles]
[provision status]
root1*                         1.0       true        fabric,
                     fabric-ensemble-0000-1, jboss-fuse-full       success
  child1                       1.0       true        default
                                                                   success
root2                          1.0       true        fabric
                                                                   success
root3                          1.0       true        fabric
                                                                   success
JBossFuse:admin@fuse.node1>
```

Create child containers for nodes 2 and 3 from the first node:

```
JBossFuse:admin@root> fabric:container-create-child root2 child2
The following containers have been created successfully:
    Container: child2.
JBossFuse:admin@root> fabric:container-create-child root3 child3
The following containers have been created successfully:
    Container: child3.
JBossFuse:admin@root>
```

34 https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Fuse/6.1/html/Getting_Started/Deploy.html#Deploy-Scalable

Using Fuse Profiles provides the advantage of simple and consistent deployment and management of environments. Assign *fulfillment-profile* to the two newly created child containers to deploy the application to the second and third nodes:

```
JBossFuse:karaf@root> fabric:container-change-profile child2 fulfillment-
profile
JBossFuse:karaf@root> fabric:container-change-profile child3 fulfillment-
profile
JBossFuse:karaf@root>
```

Monitor the log files for the child containers on the second and third node to make sure that the application has been successfully deployed. These log files will be located at:

Node2: *jboss-fuse-6.1.0.redhat-379/instances/child2/data/log/karaf.log*

Node3: *jboss-fuse-6.1.0.redhat-379/instances/child3/data/log/karaf.log*

At this point, the application is running on all three nodes. You can test this by following the instructions in the Execution section to create a sample file on each of the 3 nodes and invoking the file polling project. The container log file on the node where the file is dropped can be used to validate that the application is executing and processing the file:

```
... | INFO  | lfillment/input/ | route1                                |
rg.apache.camel.util.CamelLogger  176 | 102 - org.apache.camel.camel-core -
2.12.0.redhat-610379 | Received test.csv with the following content: Testing
```

5.4 Request Aggregation

5.4.1 Requirements

Based on the requirements, the legacy requests are received as files in comma-separated value format. For each order, two separate files are provided:

The first file, containing the customer information, is always called *xxxxxx-customer.csv* where *xxxxxx* is the order number. This file contains a single line with four values:

- Customer's social security number
- Customer's first name
- Customer's last name
- Customer contact telephone number

The second file contains the order information and is called *xxxxxx-orders.csv*. The order number in the names of the customer and order items files links them together. The order items file may include any number of lines, where each line is a single order item. Each line includes two comma-separated values:

- Order Item ID
- Quantity (numeric)

Due to the HA file system that is in use, it is conceivable that two different members of the Fuse Ensemble would pick up the customer and order files of the same order. The Ensemble must therefore synchronize and be able to merge back the pieces of the order (customer info and order items) together.

Once both the orders and the customer information for a particular order are received, a proper request containing the entirety of the information is to be formed and sent to a separate order processing service, implemented as a Java bean.

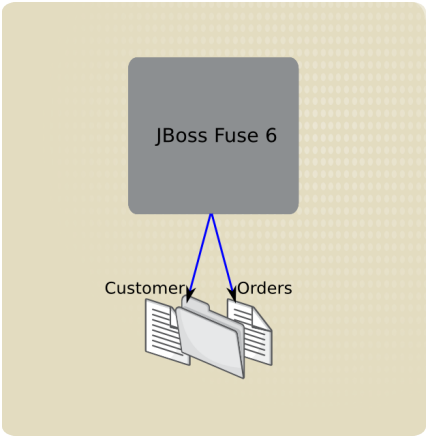


Figure 5.4.1-1:

Aggregating Two Files

5.4.2 Content Based Router

Given the two distinct file types that may dropped in the designated directory, your application needs to first determine the file format and then route it to the appropriate parsing mechanism. This is a very common Enterprise Integration Pattern[35], commonly referred to as the Content-Based Router.

To implement a content-based router with JBoss Fuse, use a *Camel choice* followed by a number of *when* elements and finally an *otherwise* to capture any remaining conditions.[36]

Open the route previously created in section 5.3.3.2. Remove the log element. Instead, locate the choice element in the right-hand palette in the *Routing* group, and drag and drop it onto the canvas. Connect from the *CSV File Poll* to this new *choice* node.

Locate the *When* element in the right-hand palette in the *Routing* group and drag and drop two of them onto the canvas. Connect from your *choice* to one *When*, and then again to the other.

Select the first *when* node and rename it by opening the properties panel and setting its ID to "when orders". Make sure the syntax language for the *when expression* is set to *simple* and enter the expression as:

```
${file:name} regex '.*orders\.csv'
```

This expression uses **`regular expressions`** to compare the name of the dropped file to the given value. In regular expressions, the dot (.) character is a wildcard that refers to any character. It is followed by the asterisk (*) quantifier which means zero or more characters are acceptable. In other words, this expression looks for any number of characters followed by *orders*. The filename should have a *csv* extension. To look for the dot extension separator, escape it with a backslash (\) to distinguish it from the common regular expressions wildcard notation that you used earlier. In summary, this regular expression matches any file name that ends with "*orders.csv*".

Select the second *when* node and rename it by opening the properties panel and setting its ID to "when customer". Make sure the syntax language for the *when expression* is set to *simple* and enter the expression as:

```
${file:name} regex '.*customer\.csv'
```

This expression looks for any file name ending with *customer.csv*.

Find the *Log* element in the *Endpoints* group in the right-hand palette. Drag and drop two *Log* nodes and connect from each of the *When* nodes to a *Log* node. Set the log message and the node Id of each Log node by selecting them and opening the *properties* view. For logging after picking up an orders file:

```
Message: Will parse ${file:name} for order items
```

35 https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Fuse/6.1/html-single/Apache_Camel_Development_Guide/index.html#IntroToEIP

36 https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Fuse/6.1/html-single/Apache_Camel_Development_Guide/index.html#MsgRout-ContentBased

```
Id: Log Orders
```

Set the properties for logging after picking up a customer file as follows:

```
Message: Will parse ${file:name} for customer
Id: Log Customer
```

The design view provides the following service model:

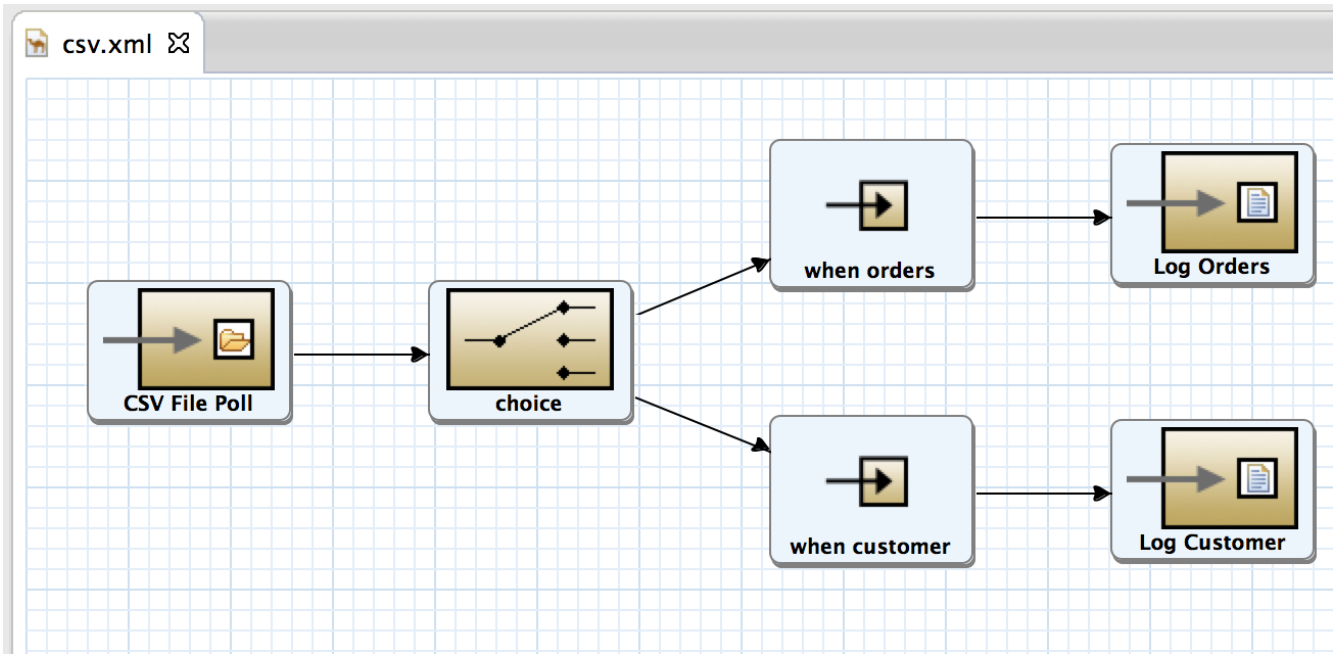


Figure 5.4.2-1: Content-Based Router

The final blueprint service descriptor XML is as follows:

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

  <camelContext trace="false"
xmlns="http://camel.apache.org/schema/blueprint">
    <route>
        <from uri="file:///fulfillment/input/" customId="true" id="CSV File
Poll">
            <description/>
        </from>
        <choice>
            <when customId="true" id="when orders">
                <simple>${file:name} regex '.*orders\.csv'</simple>
                <log message="Will parse ${file:name} for order items"
customId="true" id="Log Orders"/>
            </when>
            <when customId="true" id="when customer">
                <simple>${file:name} regex '.*customer\.csv'</simple>
                <log message="Will parse ${file:name} for customer"
customId="true" id="Log Customer"/>
            </when>
        </choice>
    </route>
</camelContext>

</blueprint>
```

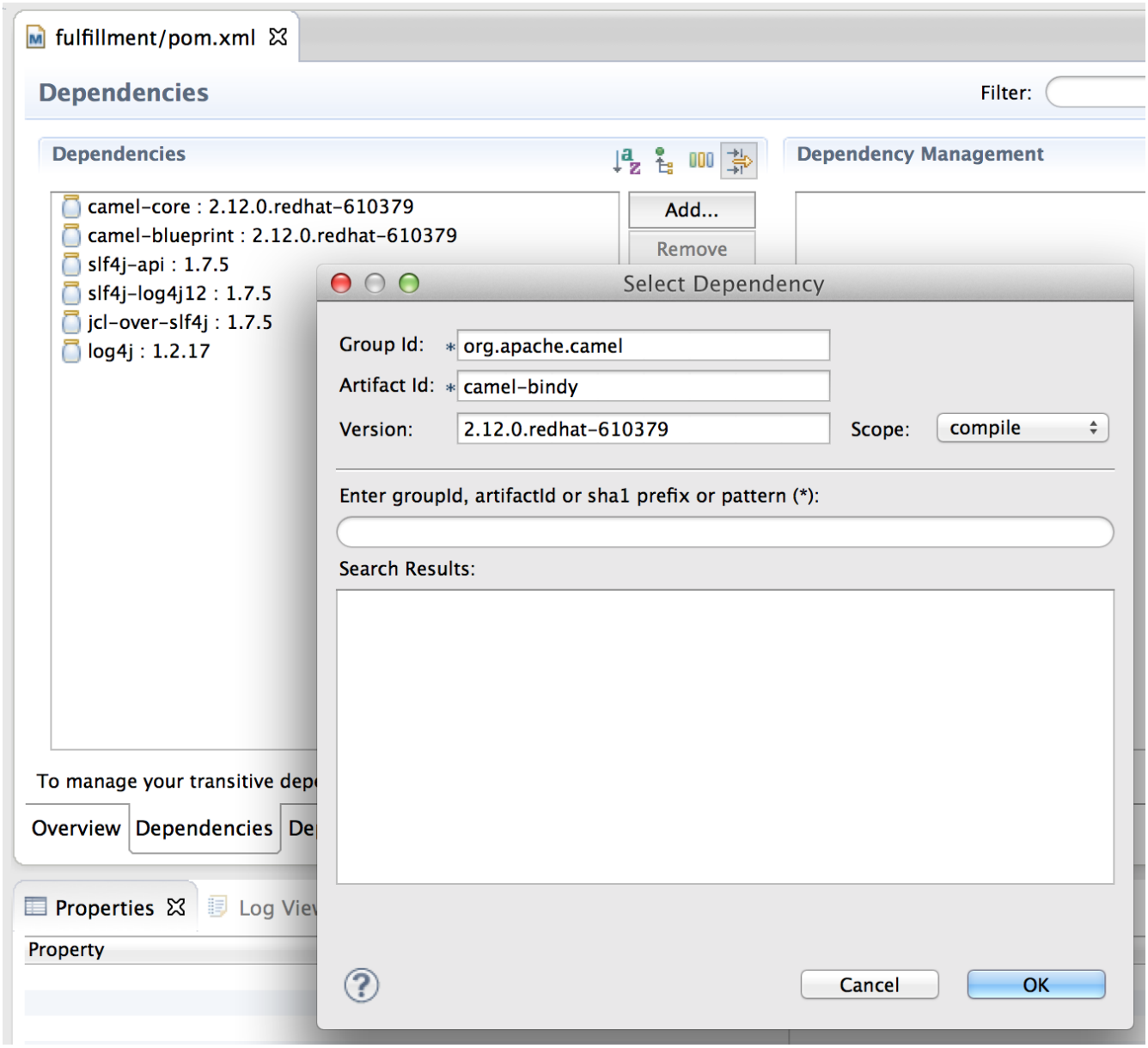
You can test the application at its current stage, either by building the application and feature, applying the feature to a profile and setting the profile on a fuse container, or by simply testing it locally using *mvn camel:run*.

Follow the same steps as outlined in the previous Execution section but create files with names corresponding to the regular expressions, for example, *12345-orders.csv* and *12345-customer.csv*. The log statements show that the content-based router is behaving as expected:

```
[0 - file:///fulfillment/input/] route1
                      INFO  Will parse 12345-orders.csv for order items
[0 - file:///fulfillment/input/] route1
                      INFO  Will parse 12345-customer.csv for customer
```

5.4.3 Unmarshalling CSV

Use the **`Bindy`**[37] component of **`Apache Camel`** to parse flat files and comma-separated values files in particular. The first step is to add Bindy as a Maven dependency of the project. Open the project *pom* file with the (default) **`Maven POM Editor`**. Look at the existing and observe the current version of Camel, used by JBoss Fuse. Add *camel-bindy* with the same version as a dependency:



`Figure 5.4.3-1: Maven Dependency`

37 http://camel.apache.org/bindy.html

According to the (hypothetical) requirements, the CSV file containing the customer includes the following fields:

- The customer's social security number, as a 9 digit number
- The customer's first name
- The customer's last name
- The customer's telephone number as free-form text

An example of a customer comma-separated values file is as follows:

```
333224444,Babak,Mozaffari,310-555-1234
```

Create a Java class to represent this data:

```
package com.redhat.refarch.fuse.fulfillment.customer;

import java.io.Serializable;

import org.apache.camel.dataformat.bindy.annotation.CsvRecord;
import org.apache.camel.dataformat.bindy.annotation.DataField;

@CsvRecord(separator = ",")
public class Customer implements Serializable
{

    private static final long serialVersionUID = 1L;
    @DataField(pos = 1)
    private long customerId;
    @DataField(pos = 2)
    private String firstName;
    @DataField(pos = 3)
    private String lastName;
    @DataField(pos = 4)
    private String telephone;
```

Bindy annotations are used to mark this class as a CSV record and map the fields to the values in the file in the presented order.

Use **`JBoss Developer Studio`** to generate getters and setters for this class, as well as a standard toString method to help log the contents of the created Java objects.

The CSV file containing the orders contains one line per order item. Each order item has only the two following fields:

- The order item ID, as free-form text
- The quantity, which is an integer

An example of an orders comma-separated values file is as follows:

```
item #1,5
item #2,7
```

Create a Java class to represent order items:

```
package com.redhat.refarch.fuse.fulfillment.item;

import java.io.Serializable;

import org.apache.camel.dataformat.bindy.annotation.CsvRecord;
import org.apache.camel.dataformat.bindy.annotation.DataField;

@CsvRecord(separator = ",")
public class OrderItem implements Serializable
{

    private static final long serialVersionUID = 1L;
    @DataField(pos = 1)
    private String itemId;
    @DataField(pos = 2)
    private int quantity;
```

Again, use JBoss Developer Studio to generate getters and setters for this class, as well as a standard toString method to help log the contents of the created Java objects.

Notice that the Customer and OrderItem classes have been created in different packages. This is due to known issues with certain versions of Apache Camel that cause unexpected and incorrect behavior when a given Java package hosts more than a single model class.[38]

Now that the dependency has been declared and Java model classes have been created for the CSV data, proceed to add the required components to parse and convert the files to Java objects.

38 https://issues.apache.org/jira/browse/CAMEL-6234

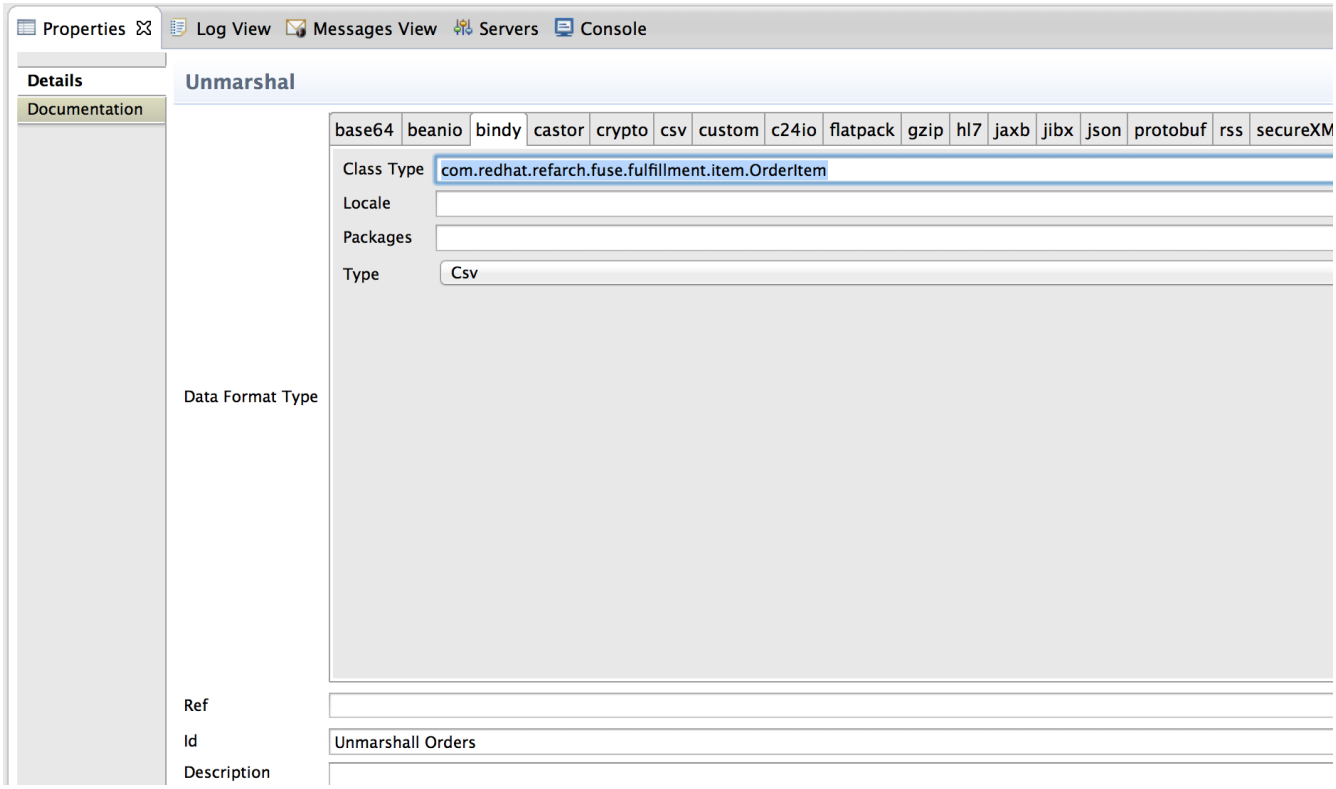


Figure 5.4.3-2: Camel Bindy CSV Unmarshaller

Use the bindy tab and select the type of the file as CSV. Under class type, enter the fully qualified model class name that represents each row of the data:

- *com.redhat.refarch.fuse.fulfillment.item.OrderItem*
- *com.redhat.refarch.fuse.fulfillment.customer.Customer*

Set the node ID of each Unmarshall node to better represent its behavior in the service diagram, for example setting them to *Unmarshall Orders* and *Unmarshall Customer.*

The application now recognizes the type of file dropped in the input folder by its name and sends it to the correct unmarshaller, which in turn converts the text to Java objects. To run the application and see the created Java objects, use a log statement to print the full message after one or both unmarshaller nodes:

```
<to uri="log:com.redhat.refarch.fusetest?showAll=true&amp;multiline=true" />
```

Execute the route by dropping appropriately-named files in the designed directory, for example *12345-orders.csv* and *12345-customer.csv.*

5.4.4 Aggregated Type

While separate files are dropped for the customer and order items, from a business perspective, they are part of the same request. The Customer and OrderItem objects need to be merged into a single Order entity. Create an Order class that represents this aggregate entity and save it in the main package:

```
package com.redhat.refarch.fuse.fulfillment;

import java.io.Serializable;
import java.util.List;

import com.redhat.refarch.fuse.fulfillment.customer.Customer;
import com.redhat.refarch.fuse.fulfillment.item.OrderItem;

public class Order implements Serializable
{

     private static final long serialVersionUID = 1L;
     private Customer customer;
     private List<OrderItem> orders;

     public Customer getCustomer()
     {
          return customer;
     }

     public void setCustomer(Customer customer)
     {
          this.customer = customer;
     }

     public List<OrderItem> getOrders()
     {
          return orders;
     }

     public void setOrders(List<OrderItem> orders)
     {
          this.orders = orders;
     }

     @Override
     public String toString()
     {
          return "Order [customer=" + customer + ", orders=" + orders + "]";
     }
}
```

You can simply declare the two fields and then use JBoss Developer Studio to generate the setters and getters, as well as the toString method.

5.4.5 Setting Camel Message Headers

To create objects of the Order class, you first need to be able to match order item files with their corresponding customer files. Customers and order items are tied together by the order number, which is part of the file name. This means that the order number must be saved before the file is unmarshalled into Java objects.

Given that the two execution paths merge for aggregation to take place, you would also ideally store some sort of flag to indicate whether a customer or an order items file was picked up. Note that there are many simple ways to achieving this objective. You can simply look at the class of the parsed object to determine whether it was a customer or order items. You can also use a static *SetHeader* node with a *constant* expression.

The following solution satisfies both these requirements. It has the downside that it duplicates the logic for distinguishing an order items file from a customer file, but you can optionally change the CBR logic to use this header property:

```
package com.redhat.refarch.fuse.fulfillment.file;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

import org.apache.camel.Exchange;

public class SetOrderHeaderBean
{

    public static final String DATA_TYPE = "dataType";
    public static final String ORDERS = "orders";
    public static final String CUSTOMER = "customer";
    public static final String ORDER_NO = "orderNo";

    public void setHeader(Exchange exchange)
    {
          String filename =
(String)exchange.getIn().getHeader( Exchange.FILE_NAME_ONLY );
          Pattern pattern = Pattern.compile(
                "(\\d*)\\-(" + CUSTOMER + "|" + ORDERS + ")\\.csv" );
          Matcher matcher = pattern.matcher( filename );
          if( matcher.matches() )
          {
                String orderNo = matcher.group( 1 );
                String fileType = matcher.group( 2 );
                exchange.getIn().setHeader( ORDER_NO, orderNo );
                exchange.getIn().setHeader( DATA_TYPE, fileType );
          }
          else
          {
                throw new IllegalArgumentException(
                            "Cannot handle file called " + filename );
          }
    }
}
```

Your Java class sets two header values in the Camel message. Use two groups in the regular expressions:

```
(\\d+)\\-(" + CUSTOMER + "|" + ORDERS + ")\\.csv
```

The first group looks for digits “\d” quantified by a wildcard so that any (non-zero) number of digits are acceptable. These digits form the order number. They are followed by a dash that has been escaped, then followed by the literal value of *customer* or *orders*, which is captured by the second group. In summary, the first group is the order number and the second group is the data type. Store these two values in the header under the respective keys of *orderNo* and *dataType*. Any file name that does not conform to this patterns is invalid and unexpected, so it is best to throw an exception so that the issue can be quickly identified and corrected.

To use this bean, drag and drop a *Bean* node under the canvas from the *Endpoints* group of the right-hand palette. Set the node properties as follows:

- *Bean Type*: com.redhat.refarch.fuse.fulfillment.file.SetOrderHeaderBean
- *Method*: setHeader
- *Id*: Set Headers

After setting these properties, select the connection between the first (CSV File Poll) and second (choice) nodes of the route and press the delete key to remove it. Instead, connect from the first node to this new *Set Headers* bean node and from this node to the *choice* node. Save the *Camel Context* file to have the diagram rearrange itself.

5.4.6 Direct VM Call

Finally, place an endpoint at the end of each branch of the content-based router to redirect execution to a different context and route, with the following endpoint URI:

```
direct-vm:aggregator
```

Set the Id for the nodes to Aggregate Order Items and Aggregate Customer respectively for modeling purposes:

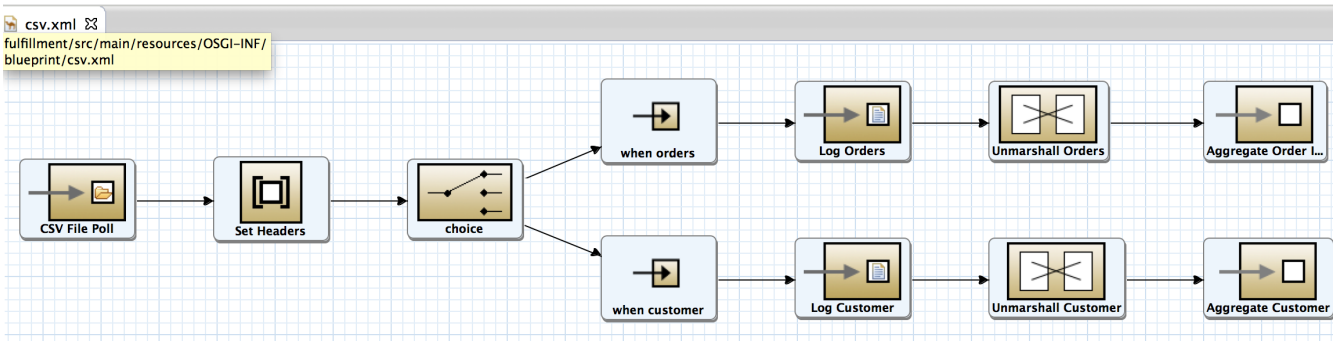


Figure 5.4.6-1: Final CSV Camel Route

Once again, you can use a log statement to print the full message at the end of each branch of the CBR:

```
<to uri="log:com.redhat.refarch.fusetest?showAll=true&amp;multiline=true" />
```

Your final Camel file should look as follows:

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

    <camelContext trace="false"
                    xmlns="http://camel.apache.org/schema/blueprint">
    <route>
        <from uri="file:///fulfillment/input/"
                                customId="true" id="CSV File Poll">
            <description/>
        </from>
        <bean method="setHeader"
    beanType="com.redhat.refarch.fuse.fulfillment.file.SetOrderHeaderBean"
                                customId="true" id="Set Headers"/>
        <choice>
            <when customId="true" id="when orders">
                <simple>${file:name} regex '.*orders\.csv'</simple>
                <log message="Will parse ${file:name} for order items"
                                    customId="true" id="Log Orders"/>
                <unmarshal customId="true" id="Unmarshall Orders">
                    <bindy type="Csv"
          classType="com.redhat.refarch.fuse.fulfillment.item.OrderItem"/>
                </unmarshal>
                <to uri="direct-vm:aggregator" customId="true"
                                        id="Aggregate Order Items"/>
            </when>
            <when customId="true" id="when customer">
                <simple>${file:name} regex '.*customer\.csv'</simple>
                <log message="Will parse ${file:name} for customer"
                                    customId="true" id="Log Customer"/>
                <unmarshal customId="true" id="Unmarshall Customer">
                    <bindy type="Csv"
    classType="com.redhat.refarch.fuse.fulfillment.customer.Customer"/>
                </unmarshal>
                <to uri="direct-vm:aggregator" customId="true"
                                            id="Aggregate Customer"/>
            </when>
        </choice>
    </route>
</camelContext>

</blueprint>
```

To summarize, your application should now consist of a single camel context with a single camel route in it. The route starts with polling for files in a predefined directory and once found, it uses the file name to determine its type as well as the associate order number. Depending on the file type, a content-based router sends the file to be unmarshalled into either a Customer or a number of OrderItem objects. Once this is done, both branches of the execution redirect the message to a new Camel route that exists within the same VM, though not necessarily the same Camel context.

You can create a new Camel route within the existing Camel context but for better IDE modeling compatibility as well as structural practices, create a new context by using JBDS to create a new Camel XML file in the same blueprint directory. Call this new file *aggregator.xml* and select *OSGi Blueprint* as the dependency framework.

Place an *Endpoint* and a *Log* node on the canvas. Set up the Endpoint as follows:

- Uri: direct-vm:aggregator
- Id: Parsed Object

Set up the logger to print the following: *Aggregate route received ${body}*

Save the Camel XML file. The completed context definition looks as follows:

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

  <camelContext trace="false"
xmlns="http://camel.apache.org/schema/blueprint">
    <route>
        <from uri="direct-vm:aggregator" customId="true" id="Parsed Object">
            <description/>
        </from>
        <log message="Aggregate route received ${body}"/>
    </route>
  </camelContext>
</blueprint>
```

Build and run the application by executing Maven with the *install* and *camel:run* targets. Drop files with expected file names in the designated directory. The log will show the file picked up after the content-based router has classified it, as well as the message passed being dispatch through the Direct VM component to the new Camel context and route:

```
[1 - file:///fulfillment/input/] route2
                     INFO  Will parse 123-orders.csv for order items
[1 - file:///fulfillment/input/] route1
                     INFO  Aggregate route received [Order [itemId=item #1,
quantity=5], Order [itemId=item #2, quantity=7]]
```

5.4.7 Aggregator Component

5.4.7.1 Overview

The Apache Camel Aggregator enables you to combine a batch of related messages into a single message.[39]

Use the aggregator to combine the order items and the customer for a given order number into a single order object.

The Aggregator has three important properties, as described in *Enterprise Integration Patterns*:

- *Correlation expression*: Determines which messages should be aggregated together. The correlation expression is evaluated on each incoming message to produce a correlation key. Incoming messages with the same correlation key are then grouped into the same batch. For example, if you want to aggregate all incoming messages into a single message, you can use a constant expression.
- *Completeness condition*: Determines when a batch of messages is complete. You can specify this either as a simple size limit or, more generally, you can specify a predicate condition that flags when the batch is complete.
- *Aggregation algorithm*: Combines the message exchanges for a single correlation key into a single message exchange.

The correlation between a customer and the order items happens through the order number and has largely been previously handled by Setting Camel Message Headers. The order number has been stored under the *orderNo* header key:

```
<correlationExpression>
    <header>orderNo</header>
</correlationExpression>
```

Determining the completion of the aggregation for this use case is very simple. There are always two, and exactly two pieces to an order:

```
<completionSize>
    <constant>2</constant>
</completionSize>
```

You can also set a timeout to stop waiting and expecting a second piece of the order after a given amount of time, specified here in milliseconds:

```
<completionTimeout>
    <constant>20000</constant>
</completionTimeout>
```

39 https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Fuse/6.1/html-single/Apache_Camel_Development_Guide/index.html#MsgRout-Aggregator

5.4.7.2 Aggregation Strategy

The aggregation algorithm is simply a matter of instantiating the *Order* class and setting both the list of order items and the customer on the order object. While simple, the aggregation algorithm requires a custom implementation. Implement the *AggregationStrategy* interface:

```
package com.redhat.refarch.fuse.fulfillment.file;

import static com.redhat.refarch.fuse.fulfillment.file.SetOrderHeaderBean.*;
import java.util.List;
import org.apache.camel.Exchange;
import org.apache.camel.processor.aggregate.AggregationStrategy;
...

public class OrderAggregationStrategy implements AggregationStrategy
{

  @Override
  public Exchange aggregate(Exchange oldExchange, Exchange newExchange)
  {
    if( oldExchange != null )
    {
      Object oldBody = oldExchange.getIn().getBody();
      if( CUSTOMER.equals( oldExchange.getIn().getHeader( DATA_TYPE ) ) )
      {
        Customer customer = (Customer)oldBody;
        @SuppressWarnings("unchecked")
        List<OrderItem> orderList =
                     (List<OrderItem>)newExchange.getIn().getBody();

        Order order = new Order();
        order.setCustomer( customer );
        order.setOrders( orderList );

        newExchange.getIn().setBody( order );
      }
      else if( ORDERS.equals( oldExchange.getIn().getHeader( DATA_TYPE ) ) )
      {
        @SuppressWarnings("unchecked")
        List<OrderItem> orderList = (List<OrderItem>)oldBody;
        Customer customer = (Customer)newExchange.getIn().getBody();

        Order order = new Order();
        order.setCustomer( customer );
        order.setOrders( orderList );

        newExchange.getIn().setBody( order );
      }
    }
    return newExchange;
  }

}
```

Manually edit the blueprint service descriptor and declare *OrderAggregationStrategy* as a bean outside the Camel context:

```
...
    </camelContext>

    <bean id="orderAggregationStrategy"     class=
      "com.redhat.refarch.fuse.fulfillment.file.OrderAggregationStrategy"/>

</blueprint>
```

5.4.7.3 Configuration

Drag and drop *Aggregate* from the *Routing* group onto the canvas. In the *Properties* panel, set the following values:

- Correlation Expression: *orderNo,* language: *header*
- Completion Size Expression: 2, language: *constant*
- Completion Timeout Expression: 20000, language: *constant*
- Strategy Ref: *orderAggregationStrategy* (Select from drop-down)
- Id*: OrderAggregation*

Drag and drop *Log* onto the canvas and set its message to: *Aggregated: ${body}*

Connect from the previous log node to the *Order Aggregation* node and then from there to the new log node. Save the blueprint service descriptor file.

5.4.7.4 Review

The completed camel route should look as follows:

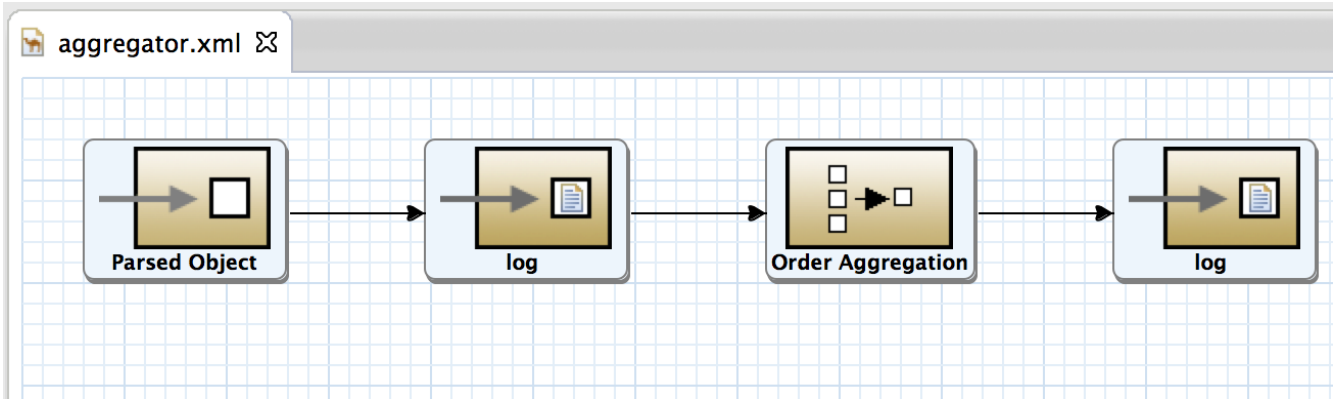


Figure 5.4.7-1: Aggregator Route

The XML representation of the completed aggregator Camel file would look as follows:

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

  <camelContext trace="false"
                xmlns="http://camel.apache.org/schema/blueprint">
    <route>
        <from uri="direct-vm:aggregator" customId="true" id="Parsed Object">
            <description/>
        </from>
        <log message="Aggregate route received ${body}"/>
        <aggregate strategyRef="orderAggregationStrategy"
                     customId="true" id="Order Aggregation">
            <correlationExpression>
                <header>orderNo</header>
            </correlationExpression>
            <completionPredicate>
                <simple></simple>
            </completionPredicate>
            <completionTimeout>
                <constant>20000</constant>
            </completionTimeout>
            <completionSize>
                <constant>2</constant>
            </completionSize>
            <optimisticLockRetryPolicy/>
            <log message="Aggregated: ${body}"/>
        </aggregate>
    </route>
  </camelContext>

    <bean id="orderAggregationStrategy"     class=
       "com.redhat.refarch.fuse.fulfillment.file.OrderAggregationStrategy"/>

</blueprint>
```

5.4.7.5 Test and Execution

At this point, you can build, run and test the application once again. Drop an order CSV and a customer CSV file in the polled directory with valid content, as used in the previous sections. Make sure to copy the two files to the directory only a few seconds apart from each other. The configured timeout will occur if the second file is not detected within 20 seconds of the first file.

The log statements show each file being picked up, correctly recognized, parsed into a Java object, received by the aggregate route and finally, aggregated using the OrderAggregationStrategy implementation:

```
[2 - file:///fulfillment/input/] route2
          INFO  Will parse 123-orders.csv for order items
[2 - file:///fulfillment/input/] route1
          INFO  Aggregate route received [Order [itemId=item #1,
                           quantity=5], Order [itemId=item #2, quantity=7]]

[2 - file:///fulfillment/input/] route2
          INFO  Will parse 123-customer.csv for customer
[2 - file:///fulfillment/input/] route1
          INFO  Aggregate route received Customer [customerId=333224444,
                firstName=Babak, lastName=Mozaffari, telephone=310-555-1234]

[2 - file:///fulfillment/input/] route1
          INFO  Aggregated: Order [customer=Customer [customerId=333224444,
          firstName=Babak, lastName=Mozaffari, telephone=310-555-1234],
          orders=[Order [itemId=item #1, quantity=5], Order [itemId=item #2,
                      quantity=7]]]
```

5.4.8 Distributed Aggregation

Understanding how the Camel aggregator works, is key to determining how it will function in various environments and under different circumstances.

Notice that when the second piece of the order is dropped in the polling directory, the aggregation strategy receives both the first and second objects from the system:

```
public Exchange aggregate(Exchange oldExchange, Exchange newExchange)
```

The Camel aggregator component uses the aggregation repository to store the incomplete parts as they are received, as well as to query and retrieve previous parts when a subsequent part is being processed. By default, Camel uses a memory-based repository implementation.

For an environment that distributes processing, a memory-based aggregation repository is not an acceptable solution. If the HA file system results in the orders being picked up by one node and the customer by another, with a memory-based aggregation repository, these two nodes would be unaware of each other and not able to properly aggregate the request.

As part of its *SQL Component*, Apache Camel provides a JDBC-based aggregation repository.[40] When using this component, aggregation pieces are stored to a configured database instead of remaining in-memory. This could be a single remote database or a clustered enterprise *RDBMS* solution. Avoiding a single point of failure and configuring connection pools and datasources to point to a cluster of databases are standard industry practices and outside the scope of this reference architecture.

40 https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Fuse/6.1/html/Apache_Camel_Component_Reference/IDU-SQLComponent.html#IDU-SQLComponent_HSH_UsingtheJDBCbasedaggregationrepository

The use of the JDBC Aggregation Repository requires setting up a datasource as well as a transaction manager in the Camel route. At this point in the complexity of the application, running it stand-alone becomes more difficult and certain design choices create container dependencies.

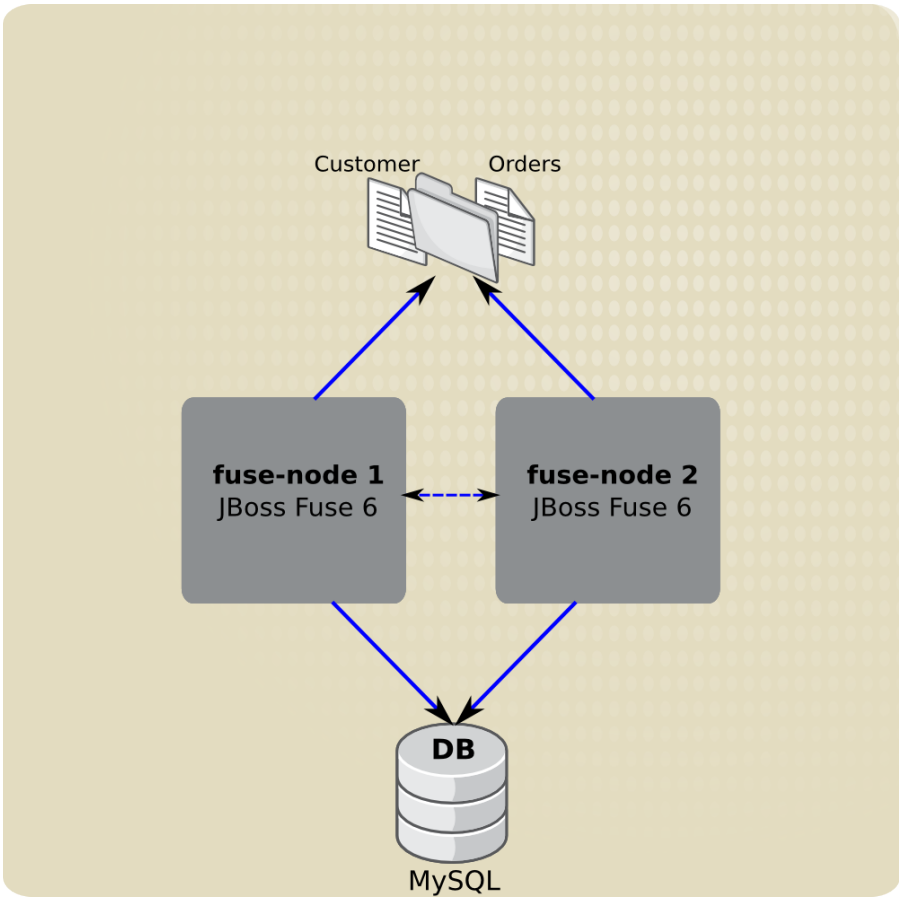


Figure 5.4.8-1: JDBC Aggregator

Manually edit the blueprint service descriptor and declare *JdbcAggregationRepository* as a bean outside the Camel context:

```
...
    </camelContext>


    <bean id="aggreationRepo" class=
      "org.apache.camel.processor.aggregate.jdbc.JdbcAggregationRepository">
          <property name="dataSource" ref="aggregationDS" />
          <property name="repositoryName" value="order_aggregation" />
          <property name="transactionManager" ref="txManager" />
    </bean>

</blueprint>
```

The Camel SQL component itself uses Spring-JDBC when database connectivity is required. Spring also provides a JDBC-based datasource implementation as well as a datasource transaction manager, both the most common options for use with Camel SQL.

Install and configure a MySQL database server. A brief description of the necessary steps is provided in the previous chapter on Creating the Environment but details on how to configure and work with MySQL remain outside the scope of this guide. This reference architecture assumes a database called *fuse*, accessible with the username of *jboss* and password of *password*. The repository name configured for the JDBC Aggregation Repository determines the expected table names. For the given configuration, create tables as follow:

```
CREATE TABLE order_aggregation ( id varchar(255) NOT NULL,
    exchange blob NOT NULL, constraint aggregation_pk PRIMARY KEY (id) );

CREATE TABLE order_aggregation_completed ( id varchar(255) NOT NULL,
    exchange blob NOT NULL,
    constraint aggregation_completed_pk PRIMARY KEY (id) );
```

Add beans for the transaction manager and the datasource outside the Camel context of the *aggregator* blueprint service descriptor:

```
...
    <bean id="txManager" class=
        "org.springframework.jdbc.datasource.DataSourceTransactionManager">
          <property name="dataSource" ref="aggregationDS" />
    </bean>

    <bean id="aggregationDS" class=
        "org.springframework.jdbc.datasource.SimpleDriverDataSource">

          <property name="driverClass" value="com.mysql.jdbc.Driver" />
          <property name="url" value="jdbc:mysql://fuse-web/fuse" />
          <property name="username" value="jboss" />
          <property name="password" value="password" />
    </bean>
</blueprint>
```

5.4.9 JDBC Driver Dependency

The use of the Spring JDBC datasource creates an implicit dependency on the JDBC drivers of the database being used. This reference architecture uses the MySQL database which provides JDBC drivers as OSGi bundles, however this particular dependency is more complicated than the previous ones.

The OSGi model heavily relies on a chain of explicit dependencies, expressed in a clear format to the container at the time of deployment. Dependency injection, whether through the blueprint or sprint model, identifies dependencies and buildling blocks at compile and deploy time. On the other hand, JDBC driver connectivity often relies on Java reflection to remain database-agonistic and refers to all driver classes through their standard interface, which is *java.sql.Driver*. As a result, the OSGi container often remains unaware of the bundle's dependency on the JDBC driver bundle.

5.4.9.1 OSGi Fragment Bundle

An OSGi Bundle Fragment is a bundle that attaches itself to a host bundle, sharing its classloader and making all its content available to the host bundle.[41]

Using an OSGi fragment bundle allows you to create a new and simple bundle that explicitly declares a dependency on the MySQL JDBC driver, while attaching itself to the Spring-JDBC bundle. The ultimate affect of deploying this fragment is force a dependency on MySQL drivers by Spring-JDBC.

To maintain a clean separation between the various project artifacts, create a new directory under *code/* and call it *mysql-fragment*.

Create a new Maven project, with nothing other than a project object model, in the same *com.redhat.refarch.fuse* group with an artifact ID of *mysql-fragment.*

Selective a descriptive name and description for the project and build it as an OSGi bundle:

```
...
  <groupId>com.redhat.refarch.fuse</groupId>
  <artifactId>mysql-fragment</artifactId>
  <version>1.0.0</version>
  <packaging>bundle</packaging>
  <name>MySQLFragment</name>
  <description>OSGi Fragment Bundle</description>
...
```

MySQL drivers are used by the Spring-JDBC bundle declared as the *org.springframework.jdbc* package. Use this package as the fragment host.

To declare a dependency on the bundle containing the MySQL JDBC drivers, import the *com.mysql.jdbc* package.

41 http://wiki.osgi.org/wiki/Fragment

The final Maven file is called *code/mysql-fragment/pom.xml* and looks as follows:

```
<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>com.redhat.refarch.fuse</groupId>
  <artifactId>mysql-fragment</artifactId>
  <version>1.0.0</version>
  <packaging>bundle</packaging>
  <name>MySQLFragment</name>
  <description>OSGi Fragment Bundle</description>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.felix</groupId>
        <artifactId>maven-bundle-plugin</artifactId>
        <version>2.1.0</version>
        <extensions>true</extensions>
        <configuration>
          <instructions>
            <Bundle-SymbolicName>${project.artifactId}</Bundle-SymbolicName>
            <Fragment-Host>org.springframework.jdbc</Fragment-Host>
            <Import-Package>com.mysql.jdbc</Import-Package>
          </instructions>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

To build this bundle, Maven must view it as a non-empty project. This requires a Maven directory structure with at least one file in there. Create the following directory and empty file:

code/mysql-fragment/src/main/resources/META-INF/.ignore

Remember that this new bundle is simply an intermediary and the MySQL bundle itself must still be included. Since you use a feature to deploy the application through a profile, edit the feature and add the MySQL bundle as well as this fragment bundle to the feature. Mark the fragment bundle as a dependency so that it is guaranteed to load the MySQL driver classes into the Spring-JDBC classloader before an attempt is made to initialize the datasource.

5.4.9.2 Feature Dependencies

The previously created Fuse Feature only depended on the camel-blueprint feature to use file polling and logging. The current iteration of the project also requires the following predefined features:

- *camel-bindy*: To unmarshall CSV files into Java objects
- *camel-sql*: To use a JDBC aggregation repository
- *spring-jdbc*: To create a datasource and transaction manager for database connectivity

Your final feature descriptor should resemble the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<features name="fulfillment-feature_repository">
  <feature name="fulfillment-feature">
    <bundle dependency='true'>
                mvn:com.redhat.refarch.fuse/mysql-fragment/1.0.0</bundle>
    <bundle>mvn:mysql/mysql-connector-java/5.1.32</bundle>
    <bundle>mvn:com.redhat.refarch.fuse/fulfillment/1.0.0</bundle>

    <feature>camel-blueprint</feature>
    <feature>camel-bindy</feature>
    <feature>camel-sql</feature>
    <feature>spring-jdbc</feature>
  </feature>
</features>
```

Also remember to edit the Aggregation POM file and add the new fragment bundle to it, so that all 3 artifacts can be built through a single top-level Maven command:

```
<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>com.redhat.refarch.fuse</groupId>
    <artifactId>order-fulfillment</artifactId>
    <version>1.0.0</version>
    <packaging>pom</packaging>
    <url>http://maven.apache.org</url>
    <name>Parent Project</name>

    <modules>
        <module>mysql-fragment</module>
        <module>fulfillment</module>
        <module>features</module>
    </modules>

</project>
```

5.4.9.3 Import Implicit Dependency Packages

There are other instances of the issue of implicit and non-declared dependencies that require special attention. The fulfillment project itself relies on both the *com.mysql.jdbc* and *org.apache.camel.impl* packages. While both bundles are included, due to the implicit usage, they have to be explicitly imported. The manifest file of the fulfillment bundle should therefore include the following import package statement, which it cannot derive at compile time:

```
Import-Package: ...,org.apache.camel.impl,com.mysql.jdbc
```

Include the following in the pom file of your fulfillment project:

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

  <modelVersion>4.0.0</modelVersion>

  <groupId>com.redhat.refarch.fuse</groupId>
  <artifactId>fulfillment</artifactId>
  <packaging>bundle</packaging>
  <version>1.0.0</version>
...
      <!-- to generate the MANIFEST-FILE of the bundle -->
      <plugin>
        <groupId>org.apache.felix</groupId>
        <artifactId>maven-bundle-plugin</artifactId>
        <version>2.3.7</version>
        <extensions>true</extensions>
        <configuration>
          <instructions>
            <Bundle-SymbolicName>fulfillment</Bundle-SymbolicName>
    <Private-Package>com.redhat.refarch.fuse.fulfillment.*</Private-Package>
    <Import-Package>*,org.apache.camel.impl,com.mysql.jdbc</Import-Package>
          </instructions>
        </configuration>
      </plugin>
...
```

With these updates, running *mvn install* from the top *code/* directory builds the fragment bundle as well as the main application bundle, before creating a feature repository with a single feature that lists all the project dependencies.

Deploy the updated application in a fabric as previously described in the Fuse Fabric Deployment section. If your fuse environment is already configured and the application deployed, a quicker way to refresh the container to take advantage of the updated code is to restart the *child1* container.

Test the deployed application in the same manner as the previous Test and Execution. The expected results are the same. You might however notice a second execution of the aggregate node, as if it was never completed and had timed out. This is a known issue in JBoss Fuse 6.1.[42]

42 https://issues.jboss.org/browse/ENTESB-1956

5.4.9.4 Container-Only Build

It is fairly common for enterprise applications to require container services that prevent them from running directly through Maven's *camel:run* target. After this iteration, the fulfillment application will only be deployed to a Fuse Fabric and executed within the container.

Update the Maven pom file at this stage to remove unnecessary dependencies. The only listed items should be compile-time dependencies. For the fulfillment application in its current iteration, this only includes *camel-core* and *camel-bindy*. Both dependencies are provided to the application in the container through a predefined Fuse Feature and should be included with a *scope* of *provided* in the pom file:

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

The camel-maven-plugin build plugin was included in the pom file to enable the use of *camel:run*. Remove this snippet from your pom file:

```
                <!-- to run the example using mvn camel:run -->
                <plugin>
                      <groupId>org.apache.camel</groupId>
                      <artifactId>camel-maven-plugin</artifactId>
                      <version>2.10.0.fuse-71-047</version>
                      <configuration>
                            <useBlueprint>true</useBlueprint>
                      </configuration>
                </plugin>

          </plugins>
```

Update the pom file and set a unique and identifiable name that describes the application. This will be the exported name of the OSGi bundle in the container:

```
    <groupId>com.redhat.refarch.fuse</groupId>
    <artifactId>fulfillment</artifactId>
    <packaging>bundle</packaging>
    <version>1.0.0</version>

    <name>Order Fulfillment JBoss Fuse Application</name>
```

5.5 Order Processing Service

Create a new blueprint service to receive aggregated orders and process them. For the purpose of this application and in the current iteration, simply use a Java bean to print the content of the order object. Create a new Java class and use the *@org.apache.camel.Body* annotation to accept the Camel message body as an argument:[43]

```
package com.redhat.refarch.fuse.fulfillment.process;

import java.io.StringWriter;

import org.apache.camel.Body;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import com.redhat.refarch.fuse.fulfillment.Order;
import com.redhat.refarch.fuse.fulfillment.item.OrderItem;

public class OrderFulfillmentBean
{

  public void process(@Body Order order)
  {
    Logger logger = LoggerFactory.getLogger( getClass() );
    StringWriter stringWriter = new StringWriter();
    stringWriter.append( "\n\nGot order with " );
    stringWriter.append( String.valueOf( order.getOrders().size() ) );
    stringWriter.append( " items" );
    for( OrderItem orderItem : order.getOrders() )
    {
      stringWriter.append( "\n\tOrder for " );
      stringWriter.append( orderItem.getItemId() );
      stringWriter.append( " in the following quantity: " );
      stringWriter.append( String.valueOf( orderItem.getQuantity() ) );
    }
    logger.info( stringWriter.toString() );
  }
}
```

Create a new context by using JBDS to create a new Camel XML file in the same blueprint directory. Call this new file process-order.xml and select OSGi Blueprint as the dependency framework.

Place an *Endpoint* and a *Bean* node on the canvas.

Set up the *Endpoint* node as follows:

- Uri: direct-vm:process-order
- Id: Process Order

43 https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Fuse/6.1/html/Apache_Camel_Development_Guide/BasicPrinciples-BeanIntegration.html#BasicPrinciples-BeanIntegration-BasicAnnotations

Configure the *Bean* node to use the new Java class:

- Bean Type: com.redhat.refarch.fuse.fulfillment.process.OrderFulfillmentBean
- Method: process
- Id: Process Order

Connect the receiving endpoint node to the bean and save the service. This simple service now receives an order object in the message and print its details.

Add an endpoint node to the end of the aggregate service and configure it:

- Uri: direct-vm:process-order
- Id: Process Order

This new final node of the aggregate route redirects execution to the process-order route in a different context. The message body is picked up by this route and the order object is traversed and printed by the Java bean.

Test the application with the latest updates and make sure it is correctly configured:

```
INFO  | lfillment/input/ | route2                                |
rg.apache.camel.util.CamelLogger  176 | 107 - org.apache.camel.camel-core -
2.12.0.redhat-610379 | Will parse 123-customer.csv for customer
INFO  | lfillment/input/ | route1                                |
rg.apache.camel.util.CamelLogger  176 | 107 - org.apache.camel.camel-core -
2.12.0.redhat-610379 | Aggregate route received Customer
[customerId=333224444, firstName=Babak, lastName=Mozaffari, telephone=310-
555-1234]
INFO  | lfillment/input/ | route2                                |
rg.apache.camel.util.CamelLogger  176 | 107 - org.apache.camel.camel-core -
2.12.0.redhat-610379 | Will parse 123-orders.csv for order items
INFO  | lfillment/input/ | route1                                |
rg.apache.camel.util.CamelLogger  176 | 107 - org.apache.camel.camel-core -
2.12.0.redhat-610379 | Aggregate route received [Order [itemId=item #1,
quantity=5], Order [itemId=item #2, quantity=7]]
INFO  | lfillment/input/ | route1                                |
rg.apache.camel.util.CamelLogger  176 | 107 - org.apache.camel.camel-core -
2.12.0.redhat-610379 | Aggregated: Order [customer=Customer
[customerId=333224444, firstName=Babak, lastName=Mozaffari, telephone=310-
555-1234], orders=[Order [itemId=item #1, quantity=5], Order [itemId=item
#2, quantity=7]]]
INFO  | lfillment/input/ | OrderFulfillmentBean                  |
ent.process.OrderFulfillmentBean   29 | 102 - fulfillment - 1.0.0 |

Got order with 2 items
    Order for item #1 in the following quantity: 5
    Order for item #2 in the following quantity: 7
```

5.6 Web Service Interface

Create a new Camel service to accept order requests in SOAP over HTTP format. JBoss Fuse uses Apache CXF to provide extensive Web Service support.[44]

The request type for the order web service is defined by the Order class and the response, as per the requirements, is just a confirmation string. Furthermore, the SOAP-based web service merely acts as a facade, providing an interface to calling clients and placing the message on the Camel route to be processed by subsequent nodes. Based on these requirements, the CXF *Service Endpoint Interface (SEI)* approach is a perfect fit.[45]

Create a simple Java interface describing the web service. Include a single method to represent the required web service operation with *Order* as the input argument and simple text as the response:

```
package com.redhat.refarch.fuse.fulfillment.ws;

import com.redhat.refarch.fuse.fulfillment.Order;

public interface OrderFulfillmentService
{

    public String process(Order order);
}
```

Create a new context by using JBDS to create a new Camel XML file in the blueprint directory. Call this new file ws.xml and select OSGi Blueprint as the dependency framework.

Manually edit the blueprint service descriptor to declare a CXF endpoint. First add an XML schema namespace for CXF:

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
    xmlns:cxf="http://camel.apache.org/schema/blueprint/cxf"
...
```

Declare the CXF endpoint to point to the Java interface to describe its interface and specify its relative binding address as */orderWS/*. The CXF endpoint ID, set to orderWS, is used to reference it from the Camel route.

```
...
    </camelContext>

    <cxf:cxfEndpoint id="orderWS" address="/orderWS/" serviceClass=
        "com.redhat.refarch.fuse.fulfillment.ws.OrderFulfillmentService" />

</blueprint>
```

44 https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Fuse/6.1/html/Apache_Camel_Development_Guide/CamelCxf.html

45 https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Fuse/6.1/html/Apache_Camel_Development_Guide/ImplWs-JavaFirst-SEI.html

Save the Camel file and use the design view to create the Camel route with IDE assistance.

Place an *Endpoint* and a *Log* node on the canvas.

Set up the *Endpoint* node as follows:

- Uri: cxf:bean:orderWS
- Id: Order Web Service

Configure the log message as: *Order Web Service received request*

Connect the endpoint node to the log and save the service. At this point, the service is able to receive SOAP requests and log them. Drop another Endpoint node onto the canvas as the third node and connect from the log node to it. Set up its properties:

- Uri: direct-vm:process-order
- Id: Process Order

Next, as a final and fourth node, drop a *Transformation* node from the *Transformation* group onto the canvas. Set the expression language to *constant* and the expression itself to *OK*. Name the node by setting its *Id* to: *OK Response*

Given the purely declarative references to CXF by the service, no compile-time dependency is introduced and as a result, there is no need to update the project pom file. On the other hand, update the fulfillment.xml feature file to add deploy-time dependency on CXF:

```
    <feature>camel-blueprint</feature>
    <feature>camel-bindy</feature>
    <feature>camel-sql</feature>
    <feature>spring-jdbc</feature>
    <feature>camel-cxf</feature>
  </feature>
</features>
```

After the update, redeploy the application to the OSGi container. CXF services deployed to the child container listen on port 8182 by default. To verify the port, connect to the container to look it up:

```
JBossFuse:karaf@root> container-connect child1
Connected
       _ ____                   ______
     | |  _ \                 |  ____|
     | | |_) | ___  ___ ___  | |__ _   _ ___  ___
 _   | |  _ < / _ \/ __/ __| |  __| | | / __|/ _ \
| |__| | |_) | (_) \__ \__ \ | |  | |_| \__ \  __/
 \____/|____/ \___/|___/___/ |_|   \__,_|___/\___|


...

JBossFuse:admin@child1> config:list | grep org.osgi.service.http.port
   org.osgi.service.http.port = 8182
JBossFuse:admin@child1>
```

Once you know the address and port of CXF services, you can list available services by issuing a GET request to the following URL: http://localhost:8182/cxf/

```
Available SOAP services:
OrderFulfillmentServicePortType
process
Endpoint address: http://localhost:8182/cxf/orderWS/
WSDL :
{http://ws.fulfillment.fuse.refarch.redhat.com/}OrderFulfillmentService
Target namespace: http://ws.fulfillment.fuse.refarch.redhat.com/


Available RESTful services:
```

The web service is listed above and its WSDL URL is provided. Use an HTTP GET request to retrieve the WSDL at http://localhost:8182/cxf/orderWS/?wsdl:

```
<?xml version='1.0' encoding='UTF-8'?><wsdl:definitions
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:tns="http://ws.fulfillment.fuse.refarch.redhat.com/"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:ns1="http://schemas.xmlsoap.org/soap/http"
name="OrderFulfillmentService"
targetNamespace="http://ws.fulfillment.fuse.refarch.redhat.com/">
  <wsdl:types>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:tns="http://ws.fulfillment.fuse.refarch.redhat.com/"
attributeFormDefault="unqualified" elementFormDefault="unqualified"
targetNamespace="http://ws.fulfillment.fuse.refarch.redhat.com/">
  <xs:complexType name="order">
    <xs:sequence>
      <xs:element minOccurs="0" name="customer" type="tns:customer"/>
      <xs:element maxOccurs="unbounded" minOccurs="0" name="orders"
nillable="true" type="tns:orderItem"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="customer">
    <xs:sequence>
      <xs:element name="customerId" type="xs:long"/>
      <xs:element minOccurs="0" name="firstName" type="xs:string"/>
      <xs:element minOccurs="0" name="lastName" type="xs:string"/>
      <xs:element minOccurs="0" name="telephone" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="orderItem">
    <xs:sequence>
      <xs:element minOccurs="0" name="itemId" type="xs:string"/>
      <xs:element name="quantity" type="xs:int"/>
    </xs:sequence>
  </xs:complexType>
  <xs:element name="process" type="tns:process"/>
  <xs:complexType name="process">
```

```
    <xs:sequence>
      <xs:element minOccurs="0" name="arg0" type="tns:order"/>
    </xs:sequence>
  </xs:complexType>
  <xs:element name="processResponse" type="tns:processResponse"/>
  <xs:complexType name="processResponse">
    <xs:sequence>
      <xs:element minOccurs="0" name="return" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
  </wsdl:types>
  <wsdl:message name="processResponse">
    <wsdl:part element="tns:processResponse" name="parameters">
    </wsdl:part>
  </wsdl:message>
  <wsdl:message name="process">
    <wsdl:part element="tns:process" name="parameters">
    </wsdl:part>
  </wsdl:message>
  <wsdl:portType name="OrderFulfillmentServicePortType">
    <wsdl:operation name="process">
      <wsdl:input message="tns:process" name="process">
    </wsdl:input>
      <wsdl:output message="tns:processResponse" name="processResponse">
    </wsdl:output>
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="OrderFulfillmentServiceSoapBinding"
type="tns:OrderFulfillmentServicePortType">
    <soap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>
    <wsdl:operation name="process">
      <soap:operation soapAction="" style="document"/>
      <wsdl:input name="process">
        <soap:body use="literal"/>
      </wsdl:input>
      <wsdl:output name="processResponse">
        <soap:body use="literal"/>
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="OrderFulfillmentService">
    <wsdl:port binding="tns:OrderFulfillmentServiceSoapBinding"
name="OrderFulfillmentServicePort">
      <soap:address location="http://localhost:8182/cxf/orderWS/"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

As shown above, default values have been generated for the Web Service name, namespace, operation name, request schema and other factors.

To invoke the web service, send a sample request through HTTP POST to the service URL, which in this example is http://localhost:8182/cxf/orderWS/:

```
<soapenv:Envelope
     xmlns:fulfill="http://ws.fulfillment.fuse.refarch.redhat.com/"
     xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
     <soapenv:Header />
     <soapenv:Body>
          <fulfill:process>
               <arg0>
                    <customer>
                         <customerId>333224444</customerId>
                         <firstName>Babak</firstName>
                         <lastName>Mozaffari</lastName>
                         <telephone>3105551234</telephone>
                    </customer>
                    <orders>
                         <itemId>Chair</itemId>
                         <quantity>4</quantity>
                    </orders>
                    <orders>
                         <itemId>Table</itemId>
                         <quantity>1</quantity>
                    </orders>
               </arg0>
          </fulfill:process>
     </soapenv:Body>
</soapenv:Envelope>
```

The expected response:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
     <soap:Body>
          <ns1:processResponse
xmlns:ns1="http://ws.fulfillment.fuse.refarch.redhat.com/">
               <return>OK</return>
          </ns1:processResponse>
     </soap:Body>
</soap:Envelope>
```

The return value is the hardcoded *OK* value, which is returned by the transformation node. Look at the container log to see the processing results:

```
INFO  | tp1951416716-183 | route4                           |
rg.apache.camel.util.CamelLogger  176 | 108 - org.apache.camel.camel-core -
2.12.0.redhat-610379 | Order Web Service received request
INFO  | tp1951416716-183 | OrderFulfillmentBean             |
ent.process.OrderFulfillmentBean   29 | 102 - fulfillment - 1.0.0 |

Got order with 2 items
     Order for Chair in the following quantity: 4
     Order for Table in the following quantity: 1
```

5.7 RESTful Service Interface

To create a RESTful service that accepts XML and JSON requests for the *Order* service, take a slightly different approach. Instead of a Camel XML file, create a JAX-RS blueprint service.[46] You can use the *rest quickstart* under your Fuse installation as a baseline for the new artifacts.

To use the *Order* class as an argument to the REST service, annotate it as a JAXB root element. In cases where the data model classes are not under your control and annotating the class is not practical, you can also use advanced configuration to map a regular Java class to XML and JSON format.

Add the XmlRootElement annotation to the class:

```
@XmlRootElement(name = "Order")
public class Order implements Serializable
```

In this approach, the service receives the request and is responsible for redirecting the message to the appropriate Camel route. The service is implemented using standard JAX-RS API. To use JAX-RS annotations in your Java classes, declare a compile-time dependency on the JSR 311 API module:

```
<dependency>
    <groupId>org.apache.servicemix.specs</groupId>
    <artifactId>org.apache.servicemix.specs.jsr311-api-1.1.1</artifactId>
    <scope>provided</scope>
    <version>2.3.0.redhat-610394</version>
</dependency>
```

Your application declares a REST server with a single service, so the relative path of the service may be left as blank:

```
@Path("/")
public class OrderFulfillmentService
{
```

Again, you only need a single operation for the service, so a relative context for the operation is not required. Set this operation to accept both XML and JSON as the request format and POST as the HTTP method:

```
    @POST
    @Path("/")
    @Consumes({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
    public Response addOrder(Order order)
```

46 https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Fuse/6.1/html/Apache_CXF_Development_Guide/RESTGuide.html

Given that the REST service is only a facade to accept requests in XML and JSON format, the only responsibility of the service implementation is to redirect the request to the Order Processing Service and return a confirmation response. Use a Camel producer to send the request to a Camel route. Use dependency injection to create the producer:

```
    @Produce(uri = "direct-vm:process-order")
    private ProducerTemplate template;
```

Implement the service method to redirect the request:

```
          template.sendBody( order );
```

Your JAX-RS service should return the response in the expected and requested format, which is normally a JSON response when the request is JSON and an XML response when the request is XML. To take advantage of the automatic mapping provided by the framework, the method should return a JAXB object.

Create an inner class that can serve as the response type and be annotated as a JAXB root element:

```
    @XmlRootElement
    private static class Return
    {

          @SuppressWarnings("unused")
          public String result;
    }
```

Instantiate this class in the service operation and set the return value to a constant value of *OK*, similar to the Web Service implementation:

```
          Return result = new Return();
          result.result = "OK";
```

Finally, use the JAX-RS API to create a response based on *javax.ws.rs.core.Response* and have it implicitly build the correct media type:

```
return Response.ok( result ).build();
```

The completed service class is as follows:

```
package com.redhat.refarch.fuse.fulfillment.rest;

import javax.ws.rs.Consumes;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import javax.xml.bind.annotation.XmlRootElement;

import org.apache.camel.Produce;
import org.apache.camel.ProducerTemplate;

import com.redhat.refarch.fuse.fulfillment.Order;

@Path("/")
public class OrderFulfillmentService
{

    @Produce(uri = "direct-vm:process-order")
    private ProducerTemplate template;

    @POST
    @Path("/")
    @Consumes({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
    public Response addOrder(Order order)
    {
          template.sendBody( order );
          Return result = new Return();
          result.result = "OK";
          return Response.ok( result ).build();
    }

    @XmlRootElement
    private static class Return
    {

          @SuppressWarnings("unused")
          public String result;
    }
}
```

The blueprint service descriptor is largely standard and unchanged from the quickstart:

```
<?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:jaxrs="http://cxf.apache.org/blueprint/jaxrs"
    xmlns:cxf="http://cxf.apache.org/blueprint/core"
    xsi:schemaLocation="
               http://www.osgi.org/xmlns/blueprint/v1.0.0
               http://www.osgi.org/xmlns/blueprint/v1.0.0/blueprint.xsd
               http://cxf.apache.org/blueprint/jaxrs
               http://cxf.apache.org/schemas/blueprint/jaxrs.xsd
               http://cxf.apache.org/blueprint/core
               http://cxf.apache.org/schemas/blueprint/core.xsd"
    >

  <jaxrs:server id="orderServer" address="/orderRS">
    <jaxrs:serviceBeans>
      <ref component-id="orderService" />
    </jaxrs:serviceBeans>
    <jaxrs:providers>
      <bean class="com.fasterxml.jackson.jaxrs.json.JacksonJsonProvider" />
    </jaxrs:providers>
  </jaxrs:server>

  <bean id="orderService" class=
    "com.redhat.refarch.fuse.fulfillment.rest.OrderFulfillmentService" />

</blueprint>
```

The JAX-RS server runs on the same CXF server as the Web Service Interface and therefore shares the same URL context. Once again, you can list available services by issuing a GET request to the following URL: http://localhost:8182/cxf/

```
Available SOAP services:
OrderFulfillmentServicePortType
process
Endpoint address: http://localhost:8182/cxf/orderWS/
WSDL :
{http://ws.fulfillment.fuse.refarch.redhat.com/}OrderFulfillmentService
Target namespace: http://ws.fulfillment.fuse.refarch.redhat.com/


Available RESTful services:
Endpoint address: http://localhost:8182/cxf/orderRS
WADL : http://localhost:8182/cxf/orderRS?_wadl
```

Notice that the service listens on the same relative URL as is set for the *jaxrs:server*, relative to the CXF server.

Add the following two features to the project feature file:
code/features/src/main/resources/fulfillment.xml

```
...
    <feature>camel-cxf</feature>
    <feature>swagger</feature>
  </feature>
</features>
```

Redeploy the application to the container and test that the new *REST* service has been successfully deployed by posting JSON and XML requests to the service address. For example, using the *curl* tool:

```
# curl -X POST  -H 'Content-Type: application/xml'
                -H 'Accept: application/xml'
                -d @request.xml
                http://localhost:8182/cxf/orderRS/

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<return><result>OK</result></return>
```

Notice that the content type and the *Accept* header both indicate XML as the desired media type. The request should therefore also be in XML format and the response will come back as XML. The *request.xml* file for this request looks as follows:

```
<Order>
    <customer>
          <customerId>333224444</customerId>
          <firstName>Babak</firstName>
          <lastName>Mozaffari</lastName>
          <telephone>3105551234</telephone>
    </customer>
    <orders>
          <itemId>Chair</itemId>
          <quantity>4</quantity>
    </orders>
    <orders>
          <itemId>Table</itemId>
          <quantity>1</quantity>
    </orders>
</Order>
```

To send a similar request in JSON and receive a JSON response:

```
# curl -X POST  -H 'Content-Type: application/json'
                -H 'Accept: application/json'
                -d @request.json
                http://localhost:8182/cxf/orderRS/

{"result":"OK"}
```

The *request.json* file:

```
{
    "customer":
    {
          "customerId": "333224444",
          "firstName": "Babak",
          "lastName": "Mozaffari",
          "telephone": "3105551234"
    },

    "orders":
    [
          {
                "itemId": "Chair",
                "quantity": "4"
          },

          {
                "itemId": "Table",
                "quantity": "1"
          }
    ]
}
```

5.8 Asynchronous Messaging

5.8.1 Overview

Once orders are received by the application, regardless of the protocol and interface used by the client, each order item needs to be processed individually. It is assumed that the processing of each single order is an expensive and lengthy process, best handled by using a fire and forget pattern to allow processing to happen asynchronously. JBoss Fuse includes ActiveMQ as a reliable messaging platform with high performance. Use a JMS queue to process the order items.

5.8.2 Producer

Order items are enclosed in a Java Order object and processed by a Java bean, so the easiest and most direct way to send each item to a queue for processing is to a use a Camel producer in the Java bean. Use dependency injection to create a producer that points to an ActiveMQ JMS Queue, called *order*:

```
    @Produce(uri = "amq:queue:order")
    private ProducerTemplate template;
```

Add a loop at the end of the Java bean method to send each order, individually, to this queue. The final bean class looks as follows:

```
public class OrderFulfillmentBean
{

    @Produce(uri = "amq:queue:order")
    private ProducerTemplate template;

    public void process(@Body Order order)
    {
          Logger logger = LoggerFactory.getLogger( getClass() );
          StringWriter stringWriter = new StringWriter();
          stringWriter.append( "\n\nGot order with " );
          stringWriter.append( String.valueOf( order.getOrders().size() ) );
          stringWriter.append( " items" );
          for( OrderItem orderItem : order.getOrders() )
          {
                stringWriter.append( "\n\tOrder for " );
                stringWriter.append( orderItem.getItemId() );
                stringWriter.append( " in the following quantity: " );
                stringWriter.append( String.valueOf(
                                          orderItem.getQuantity() ) );
          }
          logger.info( stringWriter.toString() );
          for( OrderItem orderItem : order.getOrders() )
          {
                template.sendBody( orderItem );
          }
    }
}
```

When an ActiveMQ destination is specified as the target Camel route, the Camel producer effectively acts as a JMS producer.

5.8.3 Consumer

Create a new Camel context by using JBDS to create a new Camel XML file in the OSGI-INF blueprint directory. Call this file *message-consumer* and select OSGi Blueprint as the dependency injection framework.

Place an *Endpoint* and a *Log* node on the canvas.

Set up the *Endpoint* node as follows:

- Uri: amq:queue:order
- Id: Receive Order Message

Configure the log message as: *Receiving order ${body}*

Connect the endpoint node to the log and save the service. By specifying an endpoint address with the *amq:queue* context, you have created a Camel ActiveMQ consumer. Messages produced by the order fulfillment bean Camel producer are sent to the queue and consumed by this Camel route:

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

  <camelContext trace="false"
                    xmlns="http://camel.apache.org/schema/blueprint">
    <route>
        <from uri="amq:queue:order" customId="true"
                                            id="Receive Order Message">
            <description/>
        </from>
        <log message="Receiving order ${body}"/>
    </route>
  </camelContext>

</blueprint>
```

5.8.4 Dependencies

The provided *mq-fabric-camel* feature includes all required Camel/ActiveMQ dependencies and configures a default broker so that messages can be both produced and consumed. The default broker is started on port 61616 of the server and configured for discover through the OSGi registry with the default credentials of admin and password. This requires that the fabric ensemble zookeeper password would also be set to the default value of *password*.

Edit the project feature declaration and add *mq-fabric-camel* as a feature dependency:

```
<?xml version="1.0" encoding="UTF-8"?>
<features name="fulfillment-feature_repository">
  <feature name="fulfillment-feature">
    <bundle dependency='true'>mvn:com.redhat.refarch.fuse/mysql-
fragment/1.0.0</bundle>
    <bundle>mvn:mysql/mysql-connector-java/5.1.32</bundle>
    <bundle>mvn:com.redhat.refarch.fuse/fulfillment/1.0.0</bundle>

    <feature>camel-blueprint</feature>
    <feature>camel-bindy</feature>
    <feature>camel-sql</feature>
    <feature>spring-jdbc</feature>
    <feature>camel-cxf</feature>
    <feature>swagger</feature>
    <feature>mq-fabric-camel</feature>
  </feature>
</features>
```

5.8.5 Testing

Use Maven to build the project, include the new Camel context, modified Java bean and updated feature repository. Redeploy the application to the container and invoke the order services through one of the previously defined interfaces. For example, to use the RESTful service as the gateway to the Order Fulfillment Service, send a request in JSON:

```
# curl -X POST  -H 'Content-Type: application/json'
                -H 'Accept: application/json'
                -d @request.json
                http://localhost:8182/cxf/orderRS/

{"result":"OK"}
```

Look at the log file for the application container, normally called karaf.log under the created instance. In addition to the previous log messages where the order is received and printed, you will see log statements pertaining to the ActiveMQ broker being set up and connections being established:

```
INFO  | t-1.0.0-thread-1 | FabricDiscoveryAgentFactory      |
bric.FabricDiscoveryAgentFactory   44 | 199 - org.jboss.amq.mq-fabric -
6.1.0.redhat-379 | OSGi environment detected!
INFO  | ZooKeeperGroup-0 | DiscoveryTransport               |
ort.discovery.DiscoveryTransport   78 | 117 - org.apache.activemq.activemq-
osgi - 5.9.0.redhat-610379 | Adding new broker connection URL:
tcp://....cloud.lab.eng.bos.redhat.com:61616
INFO  | ActiveMQ Task-1  | FailoverTransport                |
sport.failover.FailoverTransport 1055 | 117 - org.apache.activemq.activemq-
osgi - 5.9.0.redhat-610379 | Successfully connected to
tcp://....cloud.lab.eng.bos.redhat.com:61616
INFO  | t-1.0.0-thread-1 | BlueprintCamelContext            |
e.camel.impl.DefaultCamelContext 2224 | 121 - org.apache.camel.camel-core -
2.12.0.redhat-610379 | Route: route3 started and consuming from:
Endpoint[amq://queue:order]
```

Subsequently, you can see each of the JMS messages being consumed:

```
INFO  | qtp369188086-297 | OrderFulfillmentBean             |
ent.process.OrderFulfillmentBean   34 | 108 - fulfillment - 1.0.0 |

Got order with 2 items
     Order for Chair in the following quantity: 4
     Order for Table in the following quantity: 1

INFO  | qtp369188086-297 | FabricDiscoveryAgentFactory      |
bric.FabricDiscoveryAgentFactory   44 | 199 - org.jboss.amq.mq-fabric -
6.1.0.redhat-379 | OSGi environment detected!

INFO  | ZooKeeperGroup-0 | DiscoveryTransport               |
ort.discovery.DiscoveryTransport   78 | 117 - org.apache.activemq.activemq-
osgi - 5.9.0.redhat-610379 | Adding new broker connection URL:
tcp://....cloud.lab.eng.bos.redhat.com:61616

INFO  | ActiveMQ Task-1  | FailoverTransport                |
sport.failover.FailoverTransport 1055 | 117 - org.apache.activemq.activemq-
osgi - 5.9.0.redhat-610379 | Successfully connected to
tcp://....cloud.lab.eng.bos.redhat.com:61616

INFO  | sConsumer[order] | route3                           |
rg.apache.camel.util.CamelLogger  176 | 121 - org.apache.camel.camel-core -
2.12.0.redhat-610379 | Receiving order Order [itemId=Chair, quantity=4]

INFO  | sConsumer[order] | route3                           |
rg.apache.camel.util.CamelLogger  176 | 121 - org.apache.camel.camel-core -
2.12.0.redhat-610379 | Receiving order Order [itemId=Table, quantity=1]
```

5.8.6 Broker Configuration

With the fulfillment profile including the *mq-fabric-camel* feature, each container that is assigned the profile will start a default broker. This broker is set up as part of a group called *default*, In a fabric ensemble where such a profile is assigned to multiple containers, multiple brokers will be created in the same group, resulting in a distributed network of message brokers.

ActiveMQ brokers may also be created manually using the *fabric:mq-create* command. Use the *group* option to specify the group name and indicate the containers to which the generated profile should be added. The final argument to the command is the name of the booker that will be created:

```
JBossFuse:admin@child1> mq-create --assign-container child1 --group broker-
group order-broker
```

Refer to the official Red Hat documentation for JBoss ActiveMQ for further details.[47]

47 https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_A-MQ/index.html

6 Conclusion

Red Hat JBoss Fuse leverages Apache Camel and Apache CXF to provide a large number of features and integration capabilities. ActiveMQ can be used as a fast reliable messaging server and with Fuse Fabric as the deployment platform, high-availability can be combined with horizontal scaling to provide robust enterprise solutions.

This reference architecture sets up a Fuse Fabric ensemble with three nodes to eliminate any given node from being a single point of failure. By exposing the same service through multiple interfaces, this reference architecture demonstrates the abilities of JBoss Fuse to both integrate legacy systems and accelerate new development.

While there are simply too many features and capabilities in JBoss Fuse 6.1 for any such effort to cover, every attempt is made to touch upon the various technologies and provide a framework that can easily be expanded based on individual requirements.

Appendix A: Revision History

Revision 1.1	Babak Mozaffari
Updated documentation links for Fuse GA 6.1	
Revision 1.0	Babak Mozaffari
Initial Release	

Appendix B: Contributors

We would like to thank the following individuals for their time and patience as we collaborated on this process. This document would not have been possible without their many contributions.

Contributor	Title	Contribution
Gary Lamperillo	Principal Solution Architect	Technical Review
John Osborne	Solution Architect	Technical Review
Vishal Ghariwala	Principal Product Manager – Technical	Technical Review

Appendix C: IPTables configuration

An ideal firewall configuration constraints open ports to the required services based on respective clients. This reference environment includes a set of ports for the active cluster along with another set used by the passive cluster, which has an offset of 100 over the original set. Other than the TCP ports accessed by callers, there are also a number of UDP ports that are used within the cluster itself for replication, failure detection and other HA functions. The following iptables configuration opens the ports for known JBoss services within the set of IP addresses used in the reference environment, while also allowing UDP communication between them on any port. The required multicast addresses and ports are also accepted. This table shows the ports for the active domain. The passive domain would include an offset of 100 over many of these ports and different usage and configuration of components may lead to alternate firewall requirements.

```
# Rules for the JBoss Fuse 6.1 reference architecture environment
*filter
:INPUT ACCEPT [0:0]
:FORWARD ACCEPT [0:0]
:OUTPUT ACCEPT [75:8324]
-A INPUT -m state --state RELATED,ESTABLISHED -j ACCEPT
-A INPUT -p icmp -j ACCEPT
-A INPUT -i lo -j ACCEPT
-A INPUT -p tcp -m state --state NEW -m tcp --dport 22 -j ACCEPT
-A INPUT -s 10.19.137.0/24 -p tcp -m tcp --dport 1099 -j ACCEPT
-A INPUT -s 10.19.137.0/24 -p tcp -m tcp --dport 2181 -j ACCEPT
-A INPUT -s 10.19.137.0/24 -p tcp -m tcp --dport 8182 -j ACCEPT
-A INPUT -j REJECT --reject-with icmp-host-prohibited
-A FORWARD -j REJECT --reject-with icmp-host-prohibited
COMMIT
```