



Camel Integration and Development with Red Hat Fuse on OpenShift



Red Hat Fuse 7.1 AD421
Camel Integration and Development with Red Hat Fuse on
OpenShift
Edition 120200701
Publication date 20181129

Authors: Richard Allred, Douglas Silva, Ricardo Jun, Zach Guterman,
Fernando Lozano, Ravi Srinivasan
Editor: Seth Kenlon

Copyright © 2018 Red Hat, Inc.

The contents of this course and all its modules and related materials, including handouts to audience members, are
Copyright © 2018 Red Hat, Inc.

No part of this publication may be stored in a retrieval system, transmitted or reproduced in any way, including, but
not limited to, photocopy, photograph, magnetic, electronic or other record, without the prior written permission of
Red Hat, Inc.

This instructional program, including all material provided herein, is supplied without any guarantees from Red Hat,
Inc. Red Hat, Inc. assumes no liability for damages or legal action arising from the use or misuse of contents or details
contained herein.

If you believe Red Hat training materials are being used, copied, or otherwise improperly distributed, please send
email to training@redhat.com or phone toll-free (USA) +1 (866) 626-2994 or +1 (919) 754-3700.

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

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

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

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

The OpenStack® word mark and the Square O Design, together or apart, are trademarks or registered trademarks
of OpenStack Foundation in the United States and other countries, and are used with the OpenStack Foundation's
permission. Red Hat, Inc. is not affiliated with, endorsed by, or sponsored by the OpenStack Foundation or the
OpenStack community.

All other trademarks are the property of their respective owners.

Contributors: Sajith Sugathan, Heather Charles, Dave Sacco, Rob Locke, Achyut Madhusudhan,
Rudolf Kastl, George Hacker

Document Conventions	ix
Introduction	xi
AD421 xi
Orientation to the Classroom Environment xiii
Internationalization xvi
1. Introducing Fuse and Camel	1
Developing Integration Solutions with Red Hat Fuse and Camel	2
Quiz: Describing Red Hat Fuse and Camel Quiz	7
Describing Enterprise Integration Patterns	11
Quiz: Enterprise Integration Patterns Quiz	17
Describing Camel Concepts	21
Quiz: Core Camel Concepts Quiz	24
Summary	26
2. Creating Routes	27
Reading and Writing Files	28
Guided Exercise: Processing Orders with the File Component	32
Reading Messages from an FTP Server	36
Guided Exercise: Creating a route with the FTP component	43
Filtering Messages	49
Guided Exercise: Developing Routes Filtering Messages	55
Routing Messages from JMS	58
Guided Exercise: Developing Camel routes with JBoss Developer Studio	66
Configuring Exchange Headers	80
Guided Exercise: Developing a Camel Processor	87
Lab: Creating Routes	91
Summary	99
3. Transforming Data	101
Transforming Data Automatically and Explicitly	102
Guided Exercise: Transforming and Filtering Messages	110
Transforming Data with the Message Translator Pattern	115
Guided Exercise: Transforming Message with Custom Type Converters	124
Combining Messages Through Aggregation	129
Guided Exercise: Aggregating Messages	138
Accessing Databases with Camel Routes	144
Guided Exercise: Scheduling Routes that Access a Database	152
Lab: Transforming Data	157
Summary	167
4. Creating Tests for Routes and Error Handling with Camel	169
Testing Routes with Camel Test Kit	170
Guided Exercise: Testing Routes with Camel Test Kit	174
Developing Routes with Mock Components	180
Guided Exercise: Verifying Route Processing with Mocks Components	189
Handling Errors in Camel	201
Guided Exercise: Handling Errors in Camel	209
Lab: Testing and Error Handling with Camel	217
Summary	228
5. Routing with Java Beans	229
Developing Routes with Java Beans and Bean Registries	230
Guided Exercise: Invoking Beans in Camel Routes	239
Dynamic Routing with CDI	243
Guided Exercise: Implementing Dynamic Routing with CDI	249
Executing Bean Methods in Predicates	254

Guided Exercise: Using Beans in Predicates	256
Lab: Routing with Java Beans	259
Summary	269
6. Implementing REST Services	271
Implementing a REST Service with the REST DSL	272
Guided Exercise: Guided Exercise: Implementing a REST Service with the REST DSL	280
Consuming REST Services with the HTTP Component	287
Guided Exercise: Consuming a REST Service with the HTTP Component	292
Implementing REST Service Documentation with Swagger	299
Guided Exercise: Implementing Swagger Documentation with Camel	304
Lab: Implementing REST Services	310
Summary	317
7. Deploying Camel Routes	319
Deploying Routes with Karaf	320
Guided Exercise: Deploying Camel Projects with Red Hat Fuse	330
Deploying Routes with JBoss EAP	336
Guided Exercise: Deploying Camel projects with JBoss EAP	342
Deploying Routes with Spring Boot	348
Guided Exercise: Guided Exercise: Deploying a Camel project with Spring Boot	354
Lab: Deploying Camel Routes	360
Summary	365
8. Implementing Transactions	367
Developing Transactional Routes with Fuse on EAP	368
Guided Exercise: Enhancing a Route with Transactions	374
Testing Transactional Routes	380
Guided Exercise: Testing a Transactional Route	386
Lab: Implementing Transactions	390
Summary	394
9. Implementing Parallel Processing	395
Increasing Throughput with Parallel Processing	396
Guided Exercise: Testing the Performance of Memory Queues with SEDA	405
Implementing Parallel Processing with Camel Enterprise Integration Patterns	412
Guided Exercise: Increasing Throughput When Reading Files	419
Lab: Implementing Parallel Processing	423
Summary	429
10. Creating Microservices with Red Hat Fuse	431
Describing Red Hat Fuse Microservices	432
Quiz: Quiz: Describing Microservices	437
Developing Microservices with Camel Routes	441
Guided Exercise: Developing Microservices with Camel Routes	446
Designing Microservices for Failures	459
Guided Exercise: Designing Microservices for Failures	464
Lab: Deploying Camel Routes	477
Summary	491
11. Deploying Microservices with Fuse on OpenShift	493
Describing Red Hat Fuse on OpenShift	494
Quiz: Quiz: Describing Red Hat Fuse on OpenShift	499
Deploying a Microservice to an OpenShift Cluster	503
Guided Exercise: Deploying a Microservice to an OpenShift Cluster	517
Lab: Deploying Camel Routes	524
Summary	534

12. Comprehensive Review	535
Comprehensive Review	536
Lab: Implementing Routes with Camel	539
Lab: Implementing REST-based Routes	552
Lab: Managing Transacted Routes with Camel	563
Lab: Deploying Microservices With Fuse on OpenShift	569

Document Conventions



References

"References" describe where to find external documentation relevant to a subject.



Note

"Notes" are tips, shortcuts or alternative approaches to the task at hand. Ignoring a note should have no negative consequences, but you might miss out on a trick that makes your life easier.



Important

"Important" boxes detail things that are easily missed: configuration changes that only apply to the current session, or services that need restarting before an update will apply. Ignoring a box labeled "Important" will not cause data loss, but may cause irritation and frustration.



Warning

"Warnings" should not be ignored. Ignoring warnings will most likely cause data loss.

Introduction

AD421

Camel Integration and Development with Red Hat Fuse is a hands-on, lab-based course that gives Java developers and architects an understanding of Apache Camel and the enhancements and tools Red Hat offers in support of Camel development. Camel and Red Hat Fuse enable developers to create complex integrations in a simple and maintainable format. Camel development is organized around:

- Routes that define a sequence or flow of processing.
- Processors that transform, interpret, and modify messages within a Camel route.
- Components that enable creating endpoints that interact with the outside world for acquiring and transmitting data.

Attendees will learn the skills required to develop, implement, test, and deploy applications utilizing enterprise integration patterns (EIP) based on Apache Camel in Red Hat Fuse running on Apache Karaf and Red Hat JBoss Enterprise Application Platform. This course can assist in the preparation for the Red Hat Certificate of Expertise in Camel Development exam (EX421).

Course Objectives

- Students will work with a number of use cases that utilize the major features and capabilities of Camel to develop realistic Camel integration applications.

Audience

- Java Developers who need to learn how to implement Enterprise Integration Patterns (EIPs) using Camel features and capabilities.
- Architects who need to learn how Camel can be used as a component of their software architectures.

Prerequisites

- Experience developing Java SE applications with Java 1.7+, including use of editors such as JBoss Developer Studio (Eclipse) (required)
- Experience developing Java EE applications with JPA, JTA, and CDI (recommended)
- Experience developing with the Spring Framework (recommended)
- Experience building and packaging applications using Apache Maven (required)
- Experience developing applications that use relational databases (required)
- Experience with SQL (required)

Companion Book

Red Hat Training is pleased to recommend a companion book to this course, *Camel in Action, Second Edition*, by Claus Ibsen and Jonathan Anstey. The book is published by Manning Publications, 2017. More information can be found here: <https://www.manning.com/books/camel-in-action-second-edition>. We have included relevant references to the book throughout this course and hope this provides additional information as you continue to develop your Camel skills beyond this course.

Orientation to the Classroom Environment

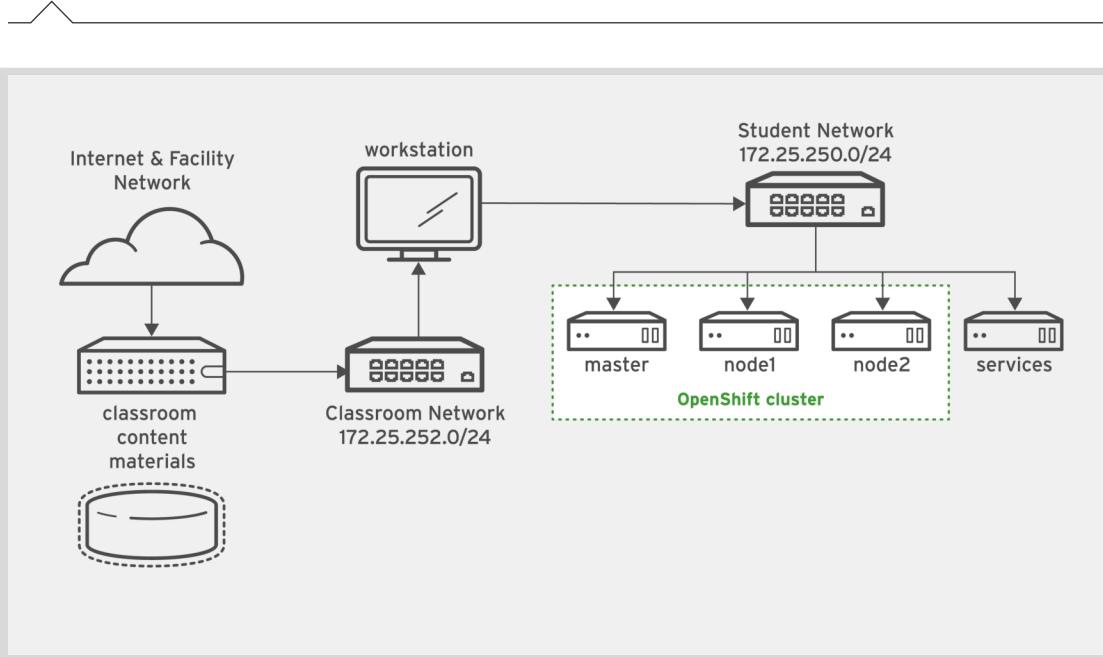


Figure 0.1: Classroom Environment

In this course, the main computer system used for hands-on learning activities is **workstation**. Four other machines will also be used by students for these activities. These are **master**, **node1**, **node2**, and **services**. All four of these systems are in the **lab.example.com** DNS domain.

All student computer systems have a standard user account, **student**, which has the password **student**. The **root** password on all student systems is **redhat**.

Classroom Machines

Machine name	IP addresses	Role
workstation.lab.example.com	172.25.250.254	Graphical workstation used for system administration
master.lab.example.com	172.25.250.10	Master of the OpenShift cluster
node1.lab.example.com	172.25.250.11	Node in the OpenShift cluster
node2.lab.example.com	172.25.250.12	Node in the OpenShift cluster
services.lab.example.com	172.25.250.13	Provides supporting services such as: container image registry, Nexus repository, and Git server

Introduction

One additional function of **workstation** is that it acts as a router between the network that connects student machines and the classroom network. If **workstation** is down, other student machines will only be able to access systems on the student network.

There are several systems in the classroom that provide supporting services. Two servers, **content.example.com** and **materials.example.com** are sources for software and lab materials used in hands-on activities. Information on how to use these servers will be provided in the instructions for those activities.

Lab Environment Actions

Machine States

Action	Description
PROVISION LAB	If you do not have a lab created, PROVISION LAB creates an environment.
START LAB	Starts all components of the lab environment. Lab machines are meant to be run as an ecosystem. There may be interdependencies between machines, so starting the entire lab (as opposed to individually starting machines) is recommended. In some cases, lab designers keep a machine powered off by default, but the START LAB button accounts for those design specifics.
DELETE LAB	Deletes the entire lab environment, and opens the option to provision a new one.

Controlling Your Station

The top of the console describes the state of your machine.

Machine States

State	Description
STARTING	Your machine is in the process of booting.
STARTED	Your machine is running and available (or, when booting, soon will be.)
STOPPING	Your machine is in the process of shutting down.
STOPPED	Your machine is completely shut down. Upon starting, your machine will boot into the same state as when it was shut down (the disk will have been preserved).

Depending on the state of your machine, a selection of the following actions will be available to you.

Machine Actions

Action	Description
Start	Start ("power on") the machine.
Shutdown	Shutdown the machine, preserving the contents of its disk.
Power off	Force the machine to power off. Would be similar to focusing off or unplugging a physical machine.
Reset Station	Stop ("power off") the machine, resetting the disk to its initial state. Caution: Any work generated on the disk will be lost.

The Station Timer

The Red Hat Online Learning enrollment entitles students to a certain amount of computer time. To help conserve time, the ROL classroom has an associated countdown timer, which will shut down the classroom environment when the timer expires. To adjust the timer, click **MODIFY**. A New Autostop Time dialog opens. Set the autostop time in hours. Click **ADJUST TIME** to adjust the time accordingly.

The timer operates as a "dead man's switch," which decrements as your machine is running. If the timer is winding down to 0, you may choose to increase the timer.

Internationalization

Language Support

Red Hat Enterprise Linux 7 officially supports 22 languages: English, Assamese, Bengali, Chinese (Simplified), Chinese (Traditional), French, German, Gujarati, Hindi, Italian, Japanese, Kannada, Korean, Malayalam, Marathi, Odia, Portuguese (Brazilian), Punjabi, Russian, Spanish, Tamil, and Telugu.

Per-user Language Selection

Users may prefer to use a different language for their desktop environment than the system-wide default. They may also want to set their account to use a different keyboard layout or input method.

Language Settings

In the GNOME desktop environment, the user may be prompted to set their preferred language and input method on first login. If not, then the easiest way for an individual user to adjust their preferred language and input method settings is to use the Region & Language application.

Run the command **gnome-control-center region**, or from the top bar, select **(User) → Settings**. In the window that opens, select Region & Language. Click the **Language** box and select the preferred language from the list that appears. This also updates the **Formats** setting to the default for that language. The next time the user logs in, these changes will take full effect.

These settings affect the GNOME desktop environment and any applications, including **gnome-terminal**, started inside it. However, they do not apply to that account if accessed through an **ssh** login from a remote system or a local text console (such as **tty2**).



Note

A user can make their shell environment use the same **LANG** setting as their graphical environment, even when they log in through a text console or over **ssh**. One way to do this is to place code similar to the following in the user's `~/ .bashrc` file. This example code will set the language used on a text login to match the one currently set for the user's GNOME desktop environment:

```
i=$(grep 'Language=' /var/lib/AccountService/users/${USER} \
| sed 's/Language=//')
if [ "$i" != "" ]; then
    export LANG=$i
fi
```

Japanese, Korean, Chinese, and other languages with a non-Latin character set may not display properly on local text consoles.

Individual commands can be made to use another language by setting the **LANG** variable on the command line:

```
[user@host ~]$ LANG=fr_FR.utf8 date  
jeu. avril 24 17:55:01 CDT 2014
```

Subsequent commands will revert to using the system's default language for output. The **locale** command can be used to determine the current value of **LANG** and other related environment variables.

Input Method Settings

GNOME 3 in Red Hat Enterprise Linux 7 automatically uses the IBus input method selection system, which makes it easy to change keyboard layouts and input methods quickly.

The Region & Language application can also be used to enable alternative input methods. In the Region & Language application window, the **Input Sources** box shows what input methods are currently available. By default, **English (US)** may be the only available method. Highlight **English (US)** and click the **Keyboard** icon to see the current keyboard layout.

To add another input method, click the **+** button at the bottom left of the **Input Sources** window. An **Add an Input Source** window will open. Select your language, and then your preferred input method or keyboard layout.

When more than one input method is configured, the user can switch between them quickly by typing **Super+Space** (sometimes called **Windows+Space**). A *status indicator* will also appear in the GNOME top bar, which has two functions: It indicates which input method is active, and acts as a menu that can be used to switch between input methods or select advanced features of more complex input methods.

Some of the methods are marked with gears, which indicate that those methods have advanced configuration options and capabilities. For example, the Japanese **Japanese (Kana Kanji)** input method allows the user to pre-edit text in Latin and use **Down Arrow** and **Up Arrow** keys to select the correct characters to use.

US English speakers may also find this useful. For example, under **English (United States)** is the keyboard layout **English (international AltGr dead keys)**, which treats **AltGr** (or the right **Alt**) on a PC 104/105-key keyboard as a "secondary shift" modifier key and dead key activation key for typing additional characters. There are also Dvorak and other alternative layouts available.



Note

Any Unicode character can be entered in the GNOME desktop environment if the user knows the character's Unicode code point, by typing **Ctrl+Shift+U**, followed by the code point. After **Ctrl+Shift+U** has been typed, an underlined **u** will be displayed to indicate that the system is waiting for Unicode code point entry.

For example, the lowercase Greek letter lambda has the code point U+03BB, and can be entered by typing **Ctrl+Shift+U**, then **03bb**, then **Enter**.

System-wide Default Language Settings

The system's default language is set to US English, using the UTF-8 encoding of Unicode as its character set (**en_US.utf8**), but this can be changed during or after installation.

From the command line, the **root** user can change the system-wide locale settings with the **localectl** command. If **localectl** is run with no arguments, it displays the current system-wide locale settings.

To set the system-wide language, run the command **localectl set-locale LANG=locale**, where *locale* is the appropriate **\$LANG** from the "Language Codes Reference" table in this chapter. The change will take effect for users on their next login, and is stored in **/etc/locale.conf**.

```
[root@host ~]# localectl set-locale LANG=fr_FR.utf8
```

In GNOME, an administrative user can change this setting from Region & Language and clicking the **Login Screen** button at the upper-right corner of the window. Changing the **Language** of the login screen will also adjust the system-wide default language setting stored in the **/etc/locale.conf** configuration file.



Important

Local text consoles such as **tty2** are more limited in the fonts that they can display than **gnome-terminal** and **ssh** sessions. For example, Japanese, Korean, and Chinese characters may not display as expected on a local text console. For this reason, it may make sense to use English or another language with a Latin character set for the system's text console.

Likewise, local text consoles are more limited in the input methods they support, and this is managed separately from the graphical desktop environment. The available global input settings can be configured through **localectl** for both local text virtual consoles and the X11 graphical environment. See the **localectl(1)**, **kbd(4)**, and **vconsole.conf(5)** man pages for more information.

Language Packs

When using non-English languages, you may want to install additional "language packs" to provide additional translations, dictionaries, and so forth. To view the list of available language packs, run **yum langavailable**. To view the list of language packs currently installed on the system, run **yum langlist**. To add an additional language pack to the system, run **yum langinstall code**, where *code* is the code in square brackets after the language name in the output of **yum langavailable**.



References

locale(7), **localectl(1)**, **kbd(4)**, **locale.conf(5)**, **vconsole.conf(5)**,
unicode(7), **utf-8(7)**, and **yum-langpacks(8)** man pages

Conversions between the names of the graphical desktop environment's X11 layouts and their names in **localectl** can be found in the file **/usr/share/X11/xkb/rules/base.lst**.

Language Codes Reference

Language Codes

Language	\$LANG value
English (US)	en_US.utf8
Assamese	as_IN.utf8

Language	\$LANG value
Bengali	bn_IN.utf8
Chinese (Simplified)	zh_CN.utf8
Chinese (Traditional)	zh_TW.utf8
French	fr_FR.utf8
German	de_DE.utf8
Gujarati	gu_IN.utf8
Hindi	hi_IN.utf8
Italian	it_IT.utf8
Japanese	ja_JP.utf8
Kannada	kn_IN.utf8
Korean	ko_KR.utf8
Malayalam	ml_IN.utf8
Marathi	mr_IN.utf8
Odia	or_IN.utf8
Portuguese (Brazilian)	pt_BR.utf8
Punjabi	pa_IN.utf8
Russian	ru_RU.utf8
Spanish	es_ES.utf8
Tamil	ta_IN.utf8
Telugu	te_IN.utf8

Chapter 1

Introducing Fuse and Camel

Goal

Describe how Fuse and Camel are used to integrate applications.

Objectives

- Discuss integration concepts with Red Hat Fuse and Camel.
- Describe Enterprise Integration Patterns with Camel.
- Describe the basics of Camel.

Sections

- Developing Integration Solutions with Red Hat Fuse and Camel (and Quiz)
- Describing Enterprise Integration Patterns (and Quiz)
- Describing Camel Concepts (and Quiz)

Developing Integration Solutions with Red Hat Fuse and Camel

Objectives

After completing this section, students should be able to:

- Describe integration concepts with Red Hat Fuse and Camel.
- Describe software requirements to justify the use of Camel and Fuse.
- Describe what the various Fuse runtimes are between Standalone, OpenShift, and Online.

Introducing Red Hat Fuse and Agile Integration

Red Hat Fuse 7, or simply Fuse, is a distributed integration platform that enables integration experts, application developers, and business users to collaborate and independently develop connected solutions.

Fuse is a truly universal platform that connects everything from legacy systems to APIs to partner networks to Internet of things (IoT) devices. Fuse services IT departments, developers, and non-technical users alike creating a unified solution that brings the benefits of agile integration to the edges of the enterprise.

Fuse embraces the concept of *agile integration*, which means developing integration applications using agile development techniques. Agile integration enables business to break integration applications along feature or functionality lines, that are developed incrementally by independent teams. Agile integration is a core capability in digital transformation.

Agile integration bridges *legacy* to the *new*, such as existing apps with new channels, Software-as-a-Service (SaaS) applications with legacy applications, monolithic applications with microservices.

The pillars of agile integration are:

- Distributed integration: multiple integration applications are spread throughout the enterprise, instead of being centralized in a dedicated middleware server.
- Reusable, pattern based integration: integration applications are composed of standard building blocks, enabling quicker time-to-market and development by business experts.
- API-first approach: integration applications consume APIs provided by traditional applications and microservices, and they provide new APIs for these applications to consume.
- Containers: integration applications are deployed as containers to improve the software delivery pipeline and ease scaling as application complexity and workloads increase.

Agile integration contrasts with the traditional approach of using an *enterprise service bus (ESB)*: in this model, integration is a service owned by the IT department and shared in the same way as telecommunication and network services. Each project interfaces with an ESB to minimize dependency on other departments as well as to maximize the reuse of integration skills, which have historically been expensive.

The traditional approach worked well in a waterfall era, when project duration was typically many months and the integration needs could be planned out. It does not allow an organization to evolve

quickly enough to meet current market challenges of the digital economy. Centralized integration was difficult to scale from either a personal perspective or an infrastructure perspective.

In a traditional approach, integration is often viewed as part of the IT infrastructure. With an agile approach, integration is a core framework of the application architecture.

Introducing Apache Camel

The core component that enables agile integration in Red Hat Fuse is *Apache Camel*, an open source library that implements enterprise integration patterns (EIP).

Enterprise Integration Patterns (EIP) are proven solutions to recurring integration problems. EIPs describe design considerations and common issues, and are brought to life using messaging technologies. EIPs establish a technology-independent vocabulary and a visual notation to design and document integration solutions.

Besides ready-to-use implementations of most EIPs, the Apache Camel community provides connectors to:

- Common middleware services, such as databases, message-oriented middleware, and web servers
- Popular enterprise information systems, such as SAP/R3 and SalesForce
- Popular web sites and services, such as Amazon and Twitter

Apache Camel also provides facilities for data transformation, supporting popular data formats such as XML, JSON, CSV, and HL7.

While Camel provides a lot of the heavy lifting with respect to integration development, Red Hat Fuse provides the reliability, performance, scalability, security, and high availability for integration solutions. Further, Red Hat Fuse subscriptions provides a certified set of Camel libraries and components, together with enterprise-level support with strict service-level agreements.

Introducing Red Hat Fuse 7 Distributions and Runtimes

Red Hat Fuse 7 is provided in the form of three different distributions:

Fuse Standalone

This is the traditional distribution of Fuse that supports any operating system with a certified Java Virtual Machine. This distribution is supported on the following application containers or runtimes:

- Apache Karaf, an OSGi container
- JBoss Enterprise Application Platform (EAP), a certified Enterprise Java (Java EE) application server
- Spring Boot, a popular framework for developing microservices

Fuse on OpenShift

This distribution supports deploying integration applications as containerized applications on OpenShift, which is Red Hat's enterprise distribution of Kubernetes.

All three runtimes supported for Fuse Standalone are supported by Fuse on OpenShift, providing a migration path from legacy deployments to containerized deployments.

Fuse Online

Fuse Online is provided as a Software-as-a-Service on the OpenShift Online Professional tier, and is also available for installing into an on-premises OpenShift cluster.

Fuse Online is a web-based user interface for developing integration applications in a "low code" environment that does not require Java development skills.

Red Hat Fuse 7 provides a unified technology stack for all supported runtimes, which is built upon the following components:

- Apache Camel: Enterprise Integration Patterns (EIP)
- Apache CXF: SOAP-based Web Services
- AMQ clients: connect with an external AMQ broker
- Naranaya: distributed transaction manager
- Undertow: lightweight and high-performance web container based on asynchronous I/O

A Red Hat Fuse 7 subscription includes entitlements to two other Red Hat Middleware products:

- JBoss Enterprise Application Platform (EAP)
- Red Hat AMQ, a messaging server

To use Red Hat Fuse on OpenShift, you must have a subscription to Red Hat OpenShift Container Platform or any other edition of OpenShift, such as OpenShift.io.

Solving Integration Problems using Fuse

To illustrate how Fuse and Camel helps solve integration problems, consider a typical order management system:

The system includes a number of endpoints and some business and integration logic. Vendors from traditional stores use legacy, scheduled *extract transform load* (ETL) processes to place orders.

These ETL processes drop files containing many orders per file on the file system to be processed later and saved into a *relational database management system* (RDBMS). Data from the RDBMS is in turn used to process order fulfillment and other internal business processes.

Different vendors generate orders using different formats, which require specific ETL processes for each, so they comply to the RDBMS database schema. For example, one vendor sends orders in a CVS format, produced by a mainframe application, while other vendor sends orders using a XML format produced by an *Enterprise Resource Planning* (ERP) package.

Vendors that use newer web-based storefronts place orders in real time using a REST API supported by an off-the-shelf online shopping application.

Further, these orders trigger events within your system, which starts back-end processes, such as payment confirmation and shipping of products, over a messaging middleware using *Java Message Service* (JMS).

Each new vendor requires a different set of data transformations, and may use a different networking protocol. In the past, updating an integration application to support a new vendor was a time-consuming process that impacted the business ability to make new partnerships and lead to lost opportunities. Red Hat Fuse allows implementing integration services that are stateless,

fault-tolerant, and largely reusable. A new integration application can be quickly added to match a new vendor requirements, without impacting the integration applications already deployed to serve current vendors.

The following figure shows a high-level overview of the set of integration applications used to address the order management system:

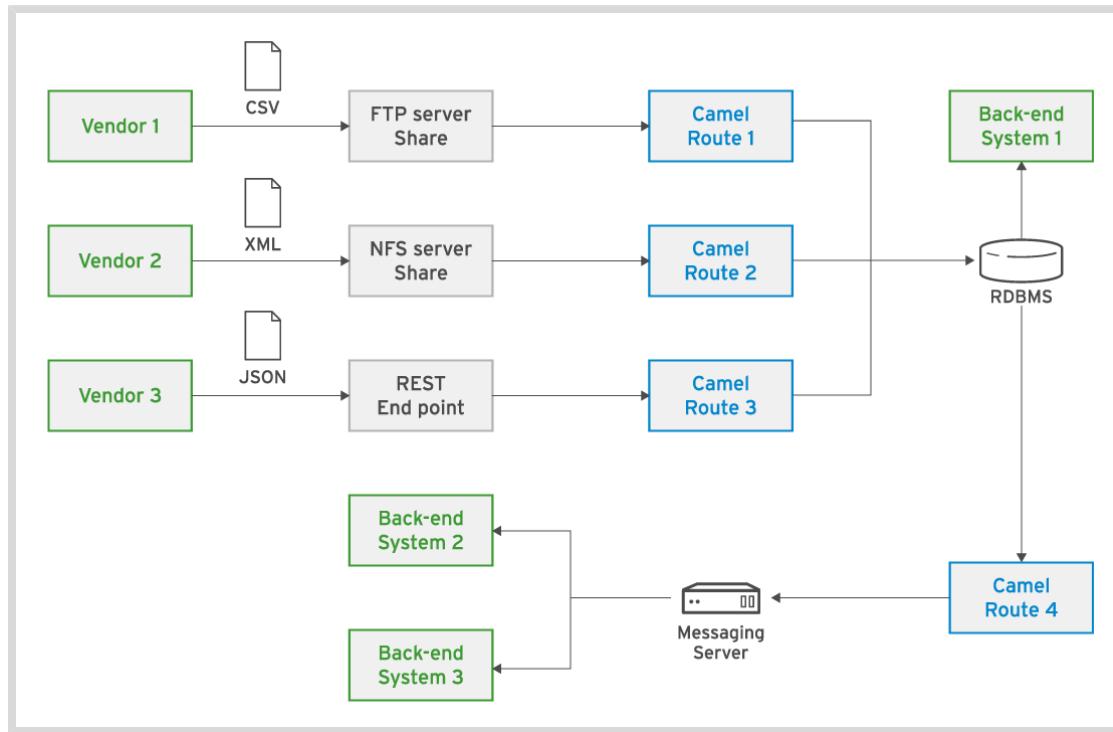


Figure 1.1: Service architecture

A developer designs and implements an integration application as a set of Camel *routes* that:

- Receive data from FTP servers and file systems, transform, and save to the RDBMS
- Receive data from REST endpoints, transform the data, and then saves it to the RDBMS
- Watches for new records on the RDBMS and dispatch notification to the back-end systems

A developer builds Camel routes by connecting components that receive, marshal, transform, and send data. Different components deal with different data sources (file systems, RDBMS), data formats (CSV, XML, JSON), and data sinks (RDBMS, JMS).

These routes could all be deployed as a single application or multiple applications on the same Fuse runtime, or they could be deployed as independent services over different Fuse runtimes. Depending on the deployment architecture, you can view the set of Camel routes as a single integration application or as a loosely coupled set of integration applications.



References

Red Hat Fuse

<https://www.redhat.com/en/technologies/jboss-middleware/fuse>

Red Hat OpenShift

<https://www.openshift.com/>

What Is Agile Integration? from Red Hat Developers Blog

<https://middlewareblog.redhat.com/2017/09/13/what-is-agile-integration/>

Enterprise Integration Patterns book by Gregor Hohpe and Bobby Woolf

<http://www.enterpriseintegrationpatterns.com/>

Apache Camel project in the Apache Software Foundation

<http://camel.apache.org>

Red Hat JBoss Enterprise Application Platform

<https://www.redhat.com/en/technologies/jboss-middleware/application-platform>

Apache Karaf project in the Apache Software Foundation

<https://karaf.apache.org/>

Spring Boot open source project

<https://spring.io/projects/spring-boot>

► Quiz

Describing Red Hat Fuse and Camel Quiz

Choose the correct answers to the following questions:

► 1. **One of the pillars of agile integration is:**

- a. Centralized Enterprise Service Bus (ESB)
- b. Message-oriented middleware such as Red Hat AMQ
- c. Standard document formats described by a machine-readable specification such as XML Schema or Swagger
- d. Pattern-based integration

► 2. _____ is an upstream open source Apache project and Java framework implementing the EIPs. _____ incorporates it into an Enterprise product.

- a. Red Hat Fuse; Red Hat JBoss EAP
- b. Apache Camel; Red Hat Fuse
- c. Spring Integration; Red Hat OpenShift
- d. Apache Karaf; Spring Boot

► 3. **Which three of the following features are part of a Red Hat Fuse 7 subscription? (Choose three)**

- a. Certified Camel libraries built by Red Hat
- b. Certified production runtimes for integration applications based on OSGi and Java EE specifications
- c. Certified Apache Tomcat Web Container for deployment of stand-alone integration applications
- d. Production support with strict service level agreements (SLA)
- e. Container images for deploying integration applications into any compatible container engine

► 4. **Which four of the following tasks are usually performed by Camel routes (Choose four)**

- a. Retrieve data stored as files in a networked file system
- b. Save data to a relational database
- c. Transform between data formats, such as from XML to JSON
- d. Submit data to a messaging server
- e. Display a web form for data entry
- f. Implement business logic such as price calculation

► **5. Which three of the following are valid deployment architectures for a set of integration applications based on Red Hat Fuse 7 and Apache Camel? (Choose three)**

- a. A client-based application that runs on the user web browser
- b. A central stand-alone Apache Karaf application container that hosts multiple integration applications
- c. Multiple stand-alone or clustered JBoss EAP application containers, each hosting one integration application from different business units
- d. Multiple containerized integration microservices orchestrated by an OpenShift cluster
- e. Stand-alone or clustered certified Java EE application servers from any vendor

► Solution

Describing Red Hat Fuse and Camel Quiz

Choose the correct answers to the following questions:

► 1. **One of the pillars of agile integration is:**

- a. Centralized Enterprise Service Bus (ESB)
- b. Message-oriented middleware such as Red Hat AMQ
- c. Standard document formats described by a machine-readable specification such as XML Schema or Swagger
- d. Pattern-based integration

► 2. _____ is an upstream open source Apache project and Java framework implementing the EIPs. _____ incorporates it into an Enterprise product.

- a. Red Hat Fuse; Red Hat JBoss EAP
- b. Apache Camel; Red Hat Fuse
- c. Spring Integration; Red Hat OpenShift
- d. Apache Karaf; Spring Boot

► 3. **Which three of the following features are part of a Red Hat Fuse 7 subscription? (Choose three)**

- a. Certified Camel libraries built by Red Hat
- b. Certified production runtimes for integration applications based on OSGi and Java EE specifications
- c. Certified Apache Tomcat Web Container for deployment of stand-alone integration applications
- d. Production support with strict service level agreements (SLA)
- e. Container images for deploying integration applications into any compatible container engine

► 4. **Which four of the following tasks are usually performed by Camel routes (Choose four)**

- a. Retrieve data stored as files in a networked file system
- b. Save data to a relational database
- c. Transform between data formats, such as from XML to JSON
- d. Submit data to a messaging server
- e. Display a web form for data entry
- f. Implement business logic such as price calculation

► **5. Which three of the following are valid deployment architectures for a set of integration applications based on Red Hat Fuse 7 and Apache Camel? (Choose three)**

- a. A client-based application that runs on the user web browser
- b. A central stand-alone Apache Karaf application container that hosts multiple integration applications
- c. Multiple stand-alone or clustered JBoss EAP application containers, each hosting one integration application from different business units
- d. Multiple containerized integration microservices orchestrated by an OpenShift cluster
- e. Stand-alone or clustered certified Java EE application servers from any vendor

Describing Enterprise Integration Patterns

Objectives

After completing this section, students should be able to:

- Describe enterprise integration patterns with Camel.
- Describe software requirements at a design level using EIPs.

Problems Solved by Enterprise Integration Patterns and Camel

While every software application is distinct and unique, many technical problems are similar or common among all applications. To avoid such pitfalls and to approach application design in a reliable way, software developers use *design patterns*. Java developers are used to design patterns, such as Singleton and Factory, that solve technical problems and provide a common vocabulary to describe solutions and their applicability.

Organizations deal with integration problems involving enterprise software by leveraging *Enterprise Integration Patterns*. In 2003, Hohpe and Woolf published the book *Enterprise Integration Patterns* (EIP) describing 65 separate patterns that represent common approaches for designing integration solutions. These EIPs provide the framework and inspiration for Apache Camel's approach to integration.

In their book, Hohpe and Woolf describe EIPs using the concept of *messaging*. Messaging in the context of EIPs and Camel is distinct from *message-oriented middleware* (MOM), such as Red Hat AMQ and Java Message Service (JMS) from *Java Enterprise Edition* (Java EE). Messaging in EIPs is a high-level concept that describes information flow from one system to another managed by EIPs.

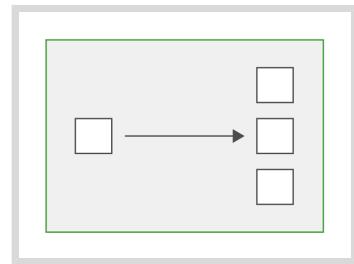
Traditional integration applications approach integration by implementing XML-based *Simple Object Access Protocol* (SOAP) web services. The contemporary approach to application integration leverages messaging and EIPs using JSON-based *representation state transfer* (REST) HTTP APIs. Camel is able to use JMS, REST, and other messaging technologies with its implementation of EIPs.

Describing Enterprise Integration Patterns

The following is a review of some of the Enterprise Integration Patterns (EIPs) used in this course. This is not an exhaustive list of all EIPs that Camel provides, rather this list illustrates some of the most common EIPs.

Splitter

This pattern separates a list or collection of items in a message payload into individual messages. This can be used when you need to process smaller, rather than larger, bulk messages.

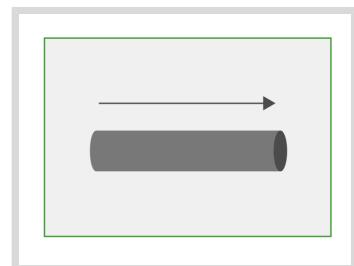
**Figure 1.2: Splitter EIP**

A benefit of the Splitter EIP is that when a message is split into chunks that can be processed independently of each other, an error within the split message does not cause all message processing to stop. A single split message can be retried or an error response can be sent back to originating system while remaining messages are processed despite the error. Further, a Splitter is useful for consuming messages in parallel, as discussed in detail in a later chapter about Concurrency.

An example of a Splitter EIP in action is an online order system that sends orders in batches. The Splitter EIP is used to split individual orders from a single large batch message. With the messages split, each order is checked out from inventory. With an additional Splitter, each item in an order is individually validated to ensure that the item is in stock. Because of the Splitter, all of this processing occurs in parallel and simplifies the business logic by allowing each system to work with smaller messages.

Message Channel

The Message Channel pattern represents point-to-point communication between two applications or systems, with the additional benefits of providing asynchronous communication and decoupled endpoints using a messaging system.

**Figure 1.3: Message Channel EIP**

Asynchronous and decoupled communication is typically implemented with the help of a message-oriented middleware, such as Red Hat AMQ, allowing an application to provide quick responses and continue servicing requests when there is a failure in some dependent system.

A message channel is a common pattern to use in e-commerce applications, such as an online store. When a user checks out the shopping cart, there is no need to wait for payment confirmation, checking inventory levels, and shipping. All these processes can work independently of each other, using message channels. Checking out the shopping cart puts a message in a channel, where it waits for the payment process. Then the payment process puts a message in another channel, where it waits for the inventory process.

If a payment processor is unavailable, this does not prevent processing orders that require a different payment processors. Orders that require the unavailable payment processor keep waiting in a queue until that payment processor is available again.

Normalizer

This pattern processes messages from systems using different formats and converts the message into a common format.

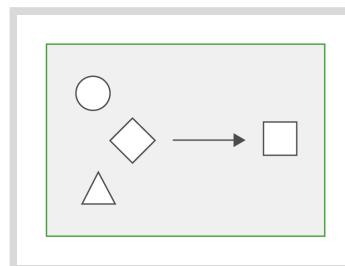


Figure 1.4: Normalizer EIP

For example, normalizer is useful for a company consumes orders from multiple vendors. Some vendors might use legacy mainframe systems and submit orders as CSV files. Other vendors use *enterprise resource planning (ERP)* systems that submit orders as XML files. And many vendors are switching to newer REST-based systems that submit orders as JSON files.

A normalizer converts all of these orders to the data format expected by the company's internal order processing system. A benefit of this approach is that the enterprise does not need to change their business logic as they expand to include new vendors that send orders in a specific format. Rather, they only need to update their integration solution to support additional formats.

Transactional Client

A Transactional Client calls to another system to ensure delivery and ensure data integrity during the transaction.

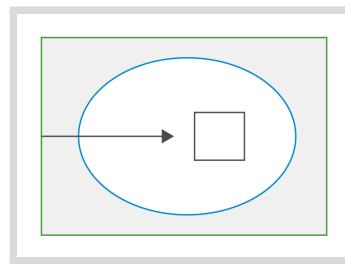


Figure 1.5: Transactional Client EIP

Many developers are familiar with the concept of a *transaction*, or *single unit of work*, from relational databases. However, transactions also apply to non-relational databases and messaging systems.

Consider a process that reads data from a message queue and saves the data to a database. One concern in this approach is that if the process reads the message from the queue, but the database write fails, the message is lost because it is no longer available in the queue. To solve this, you want the message to remain in the queue in the event of a database write failure, so it can be read again later. Further, you want to ensure that if the write is successful to the database, that the message is removed from the queue.

Wrapping these three operations (reading a message from a queue, writing to a database, and removing a message from a queue) into a single transaction guarantees that either all operations succeed or the system rolls back the entire operation to maintain data integrity.

Messaging Gateway

The Messaging Gateway pattern delegates the responsibility of messaging to a dedicated messaging layer so that business applications do not need to contain additional logic specific to messaging. The dedicated messaging layer becomes the source or destination of messages from EIPs.

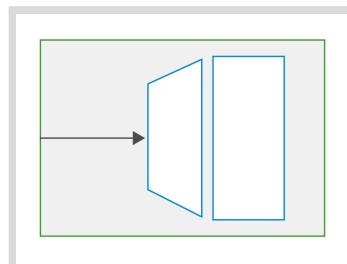


Figure 1.6: Messaging Gateway EIP

For example, a web server that throttles traffic and authenticates clients as they submit HTTP requests destined to business applications without regard for the request contents is an example of a messaging gateway. The gateway is able to process each request before passing the message to the business application without the business application needing to contain the added complexity of messaging protocols and logic.

This kind of functionality is the motivation for many *API Gateway* products in the market, such as Red Hat 3Scale. Red Hat Fuse provides an easy way to implement simple internal API gateways by combining EIPs, while Red Hat 3Scale provides built-in features, such as metering and billing, that simplifies implementing an API gateway for external users. These two products can be combined for more complex gateway scenarios.

Content Based Router

A Content Based Router pattern routes messages to an endpoint dynamically based on the content of the message's body or header.

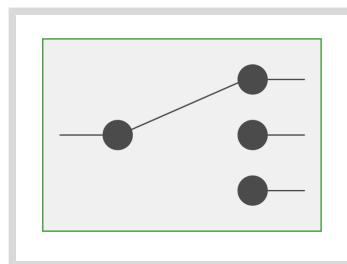


Figure 1.7: Content Based Router EIP

This pattern is useful in the following scenario: an online store, which used to sell only products from its own inventory, starts a partnership with local business to sell their items. Some orders are to be processed by the internal order processing system, but others need to be sent to external systems for fulfillment.

A content-based router would look at the products included in each order. If these products come from the store's own inventory, they are sent to the internal order fulfillment system. If the products come from the external vendor, they are sent to the external vendor system.

Aggregator

The Aggregator pattern collapses multiple messages into a single message. This is helpful to aggregate many business processes into a single event message to be delivered to another client or to rejoin a list of messages which were previously split using the Split pattern.

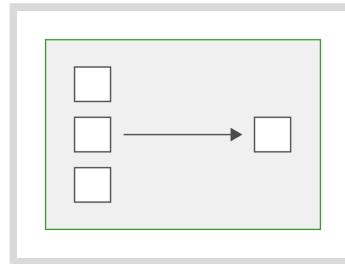


Figure 1.8: Aggregator EIP

The aggregator pattern makes efficient use of information systems that perform better when they process data in batches instead of as individual messages.

Consider a checking account system that processes transfers from other banks to customers' checking accounts. Because communication happens over slow WAN links, and there is no requirement that the funds transfer occur in real time, the checking accounts systems use an aggregator to batch transfer messages that send funds to the same bank. If a transfer operation is made to wait for too long, it is sent alone in a batch containing a single message.

Composing Multiple EIPs Into a Camel Integration Application

A typical integration application relies on more than just a single EIP. One of the challenges of being a good Camel developer is understanding which patterns to use to provide the most reliable and efficient integration system.

Consider the example of an online web store that sells goods from external vendors. The same order could contain items from different vendors. A Splitter EIP generates multiple smaller orders, each one destined to a single vendor, and a Content Based Router EIP submits the individual smaller orders to the correct vendor.

A Camel route composes multiple EIPs into a pipeline that has a message source and one or more message destinations.

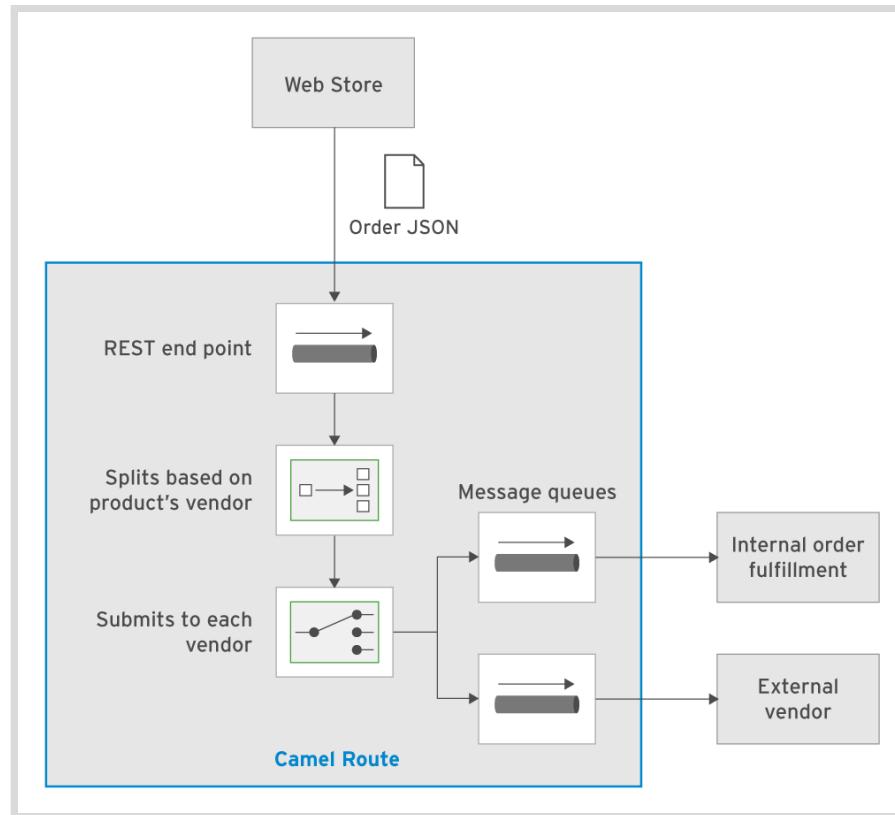


Figure 1.9: Composing multiple EIPs Into a Camel Route

A Camel route in the online web store example receives messages from a REST API end point, passes each message through a Splitter, then to a Content Router, which in turn submits each message to a Message Channel specific to each external vendor.



References

Enterprise Integration Patterns book by Gregor Hohpe and Bobby Woolf

<http://www.enterpriseintegrationpatterns.com/>

Apache Camel EIP icon library

<http://camel.apache.org/eip.html>

► Quiz

Enterprise Integration Patterns Quiz

Choose the correct answers to the following questions:

- ▶ 1. **Enterprise Integration Patterns (EIPs) are based around the concept of asynchronous messages. Which of the following can be sources or destinations of messages for EIPs implemented using Apache Camel? (Choose four)**
 - a. JMS Queues
 - b. SOAP Web Services
 - c. REST APIs
 - d. FTP server files
 - e. Remote Procedure Call (RPC) servers

- ▶ 2. **Which EIP is useful when a large message can be decomposed into atomic units and processed individually?**
 - a. Splitter
 - b. Message Channel
 - c. Normalizer
 - d. Transactional Client
 - e. Content Based Router

- ▶ 3. **Which EIP is useful when multiple message exchanges need to be processed as a single unit of work?**
 - a. Splitter
 - b. Message Channel
 - c. Normalizer
 - d. Transactional Client
 - e. Content Based Router

- ▶ 4. **Which EIP is useful when the data contained in the input messages should be used to determine its destination?**
 - a. Splitter
 - b. Message Channel
 - c. Normalizer
 - d. Transactional Client
 - e. Content Based Router

► **5. Which EIP is useful when the input messages use a data format that is not compatible with the intended destination?**

- a. Splitter
- b. Message Channel
- c. Normalizer
- d. Transactional Client
- e. Content Based Router

► Solution

Enterprise Integration Patterns Quiz

Choose the correct answers to the following questions:

- ▶ 1. **Enterprise Integration Patterns (EIPs) are based around the concept of asynchronous messages. Which of the following can be sources or destinations of messages for EIPs implemented using Apache Camel? (Choose four)**
 - a. JMS Queues
 - b. SOAP Web Services
 - c. REST APIs
 - d. FTP server files
 - e. Remote Procedure Call (RPC) servers

- ▶ 2. **Which EIP is useful when a large message can be decomposed into atomic units and processed individually?**
 - a. Splitter
 - b. Message Channel
 - c. Normalizer
 - d. Transactional Client
 - e. Content Based Router

- ▶ 3. **Which EIP is useful when multiple message exchanges need to be processed as a single unit of work?**
 - a. Splitter
 - b. Message Channel
 - c. Normalizer
 - d. Transactional Client
 - e. Content Based Router

- ▶ 4. **Which EIP is useful when the data contained in the input messages should be used to determine its destination?**
 - a. Splitter
 - b. Message Channel
 - c. Normalizer
 - d. Transactional Client
 - e. Content Based Router

► **5. Which EIP is useful when the input messages use a data format that is not compatible with the intended destination?**

- a. Splitter
- b. Message Channel
- c. Normalizer
- d. Transactional Client
- e. Content Based Router

Describing Camel Concepts

Objectives

After completing this section, students should be able to:

- Describe the basic concepts of Camel.
- Describe the different DSL options offered by Camel.

Describing the Camel Architecture

Apache Camel is a Java library that implements Enterprise Integration Patterns (EIPs) and provides a solid foundation to develop integration systems for both legacy and new applications.

Apache Camel works with different messaging systems, ranging from older mainframe-based applications, passing through the WS-* standards world, to the new HTTP API-based style of development. Camel also works with message-oriented middleware (MOM) that can be accessed using the Java Message Service (JMS) API, such as Red Hat AMQ.

Not all inputs and outputs to a Camel route need to come from an actual messaging system. Apache Camel can also process data from file systems, FTP servers, databases, and other data stores. Camel encapsulates data originating from these data stores as messages, so these data stores can act as source or destination of messages in an integration application.

Camel's architecture is based on a small number of Java classes and interfaces: **CamelContext**, **Route**, **Endpoint**, **Component**, **Exchange**, **Message Processor** and **Predicate**.

The following section discusses concepts related to these classes and interfaces at a very high level, not the syntax around them. All these concepts are revisited during this course, along with coding examples.

Context, Routes, and Components

Integration applications based on Camel define a *context* that aggregates one or more *routes* and zero or more *components*.

A context provides the environment where running routes and components run. Apache Camel provides integration with many application frameworks and runtimes so most developers do not need to know how to initialize and start a Camel context.

A route describes an integration scenario, where a message flows from a source to one or more destination *endpoints*, and may be transformed along the way. An endpoint is a name, usually using URI syntax, to a specific source or destination for messages.

A component provides configuration for endpoints and encapsulates the logic required to work with different sources or destinations of messages, such as FTP servers, JMS Queues, or relational databases. Multiple endpoints can reuse the same component, and multiple instances of the same component can be defined in the same context.

For example, many endpoints can use the **File** component, which accesses files from the local file system. Each endpoint can specify a different file system folder. Another example is the **JDBC**

component, which can access a relational database. Multiple instances of the **JDBC** component can connect to different database servers using different datasources.

A context only needs to describe components that require additional, explicit configuration, such as a database server, which requires at least a server address and login credentials. Many components take all the configuration they need from an endpoint URI, such as the path for a file system folder.

Camel routes are defined using *Domain Specific Language* (DSL). The Apache Camel project provides a number of DSLs optimized for programming languages, such as Scala and Groovy. Red Hat Fuse provides production support for the following two DSLs:

Java DSL

Uses a fluent style of concatenated method calls to define routes and components. A route contains no procedural code, therefore if you need to embed complex conditional or transformation logic inside a route, you can invoke a Java Bean.

XML DSL

Uses XML elements inside a Spring beans or an OSGi Blueprint configuration file to define routes and components. Though the XML DSL is not as complete or as flexible as the Java DSL, the XML DSL does allow non-developers to create routes using a graphical tool if desired.

These two DSLs can be mixed: the same integration application can define some routes using the Java DSL, and others using the XML DSL. Components configured using the XML DSL can be used in routes defined using the Java DSL, and vice versa.

Exchanges, Messages, and Endpoints

A route connects at least two endpoints: a *consumer* and one or more *producers*. A consumer defines the origin of messages the route receives, and a producer defines a destination for messages that the route sends.

These terms can be counter-intuitive for users accustom to messaging systems. To avoid confusion, remember they follow the point of view of the route: a route consumes messages that it receives from an endpoint, and produces messages that it sends to another endpoint.

A consumer provides a route with an *exchange*, which flows through the route until it reaches a producer. An exchange contains one *inbound* and optionally one *outbound* message. Notice that these are not input and output messages. They are request and response messages. The inbound and outbound denotation is relative to the consumer endpoint's point of view.

Depending of the component type of an endpoint, the Camel route may not produce an outbound message, that is, a response to the consumer. The message a route sends to a producer is always the inbound message of the exchange.

Processors, Predicates and Expressions

Between the consumer and producer endpoints, a number of *processors* can manipulate the exchange contents. Processors implement most of the EIPs provided by Camel. Most processors manipulate only the inbound message, as this is the message the route sends to a producer. An example processor takes a message in XML format and transforms it into a different XML format using a XSLT transformation.

Many processors make use of *predicates* and *expressions*. Predicates allow you to create conditions based on a message's body, headers, and properties. Expressions extract data from

a message. Camel supports a number of expression languages, such as XPath, JSONpath, and MVEL.

MVEL is very similar to the Java EE *expression language* (EL) used by Java Server Pages (JSP) and the Java Server Faces (JSF) framework, and provides a simple way to access properties of Java Beans.

Predicates allow a developer to define conditions, such as a message that does not have an empty body, or a message that contains a given header. Expressions are more flexible, but the expression language must match the data format of a message. For example, MVEL only works for messages that contain Java Beans, and XPath only works for XML data.

Extending and Supporting Camel

You can implement custom components, processors, and predicates to encapsulate complex logic and use them inside a route. With this functionality you can extend Camel's feature set to meet almost any requirement, adding new integration patterns and connectors to new message sources or destinations.

These custom components, processors, and predicates can be glued to routes using many approaches, for example a dependency injection framework, such as the Spring framework or Context and Dependency Injection (CDI) from Java EE.

Camel itself is agnostic to any application framework or runtime. Be aware that the Apache Camel community may provide a number of integrations with programming languages, frameworks, and information systems that are not mature enough for production usage. This is one reason why Red Hat Fuse does not include or support all of Apache Camel, but only a large subset of the components.

The parts of Camel that are included in the Red Hat Fuse product receive full production support from Red Hat. Subscription users can open support tickets and rely on Red Hat to provide a solution according to the user's service-level agreement.

Be careful to not abuse Camel's power: Camel's features are sufficient to implement many application scenarios, and you may end up implementing business logic inside a Camel route. Remember that Camel's goal is not to replace existing applications, nor to implement new applications. Camel's goal is to integrate existing and new applications, moving data between them in an easy and flexible way.



References

Apache Camel Architecture

<http://camel.apache.org/architecture.html>

► Quiz

Core Camel Concepts Quiz

Choose the correct answers to the following questions:

- ▶ 1. Which two of the following Camel DSLs does the Red Hat Fuse subscription support?
 - a. Groovy DSL
 - b. Java DSL
 - c. Scala DSL
 - d. XML DSL

- ▶ 2. Each producer and consumer can be considered as _____ that Camel provides as integration hooks. These can be plugged into your routes as configurable _____.
 - a. endpoints; components
 - b. building blocks; libraries
 - c. Java code; endpoints

- ▶ 3. Camel architecture consists of _____, which flow message exchanges from _____ to _____ with integration logic in between.
 - a. DSLs; client; server
 - b. contexts; endpoints; components
 - c. routes; consumer ; producer

- ▶ 4. Which of the following two allow a route to access data inside an inbound message to make transformations or perform conditional operations?
 - a. endpoint
 - b. component
 - c. predicate
 - d. expression

► Solution

Core Camel Concepts Quiz

Choose the correct answers to the following questions:

- ▶ 1. Which two of the following Camel DSLs does the Red Hat Fuse subscription support?
 - a. Groovy DSL
 - b. Java DSL
 - c. Scala DSL
 - d. XML DSL

- ▶ 2. Each producer and consumer can be considered as _____ that Camel provides as integration hooks. These can be plugged into your routes as configurable _____.
 - a. endpoints; components
 - b. building blocks; libraries
 - c. Java code; endpoints

- ▶ 3. Camel architecture consists of _____, which flow message exchanges from _____ to _____ with integration logic in between.
 - a. DSLs; client; server
 - b. contexts; endpoints; components
 - c. routes; consumer ; producer

- ▶ 4. Which of the following two allow a route to access data inside an inbound message to make transformations or perform conditional operations?
 - a. endpoint
 - b. component
 - c. predicate
 - d. expression

Summary

In this chapter, you learned:

- Red Hat Fuse is an integration platform, based on Apache Camel, that embraces the concept of agile integration.
- Agile integration is based on four pillars: distributed platform, pattern-based integration, API first, and containerization.
- Apache Camel is a library that implements Enterprise Integration Patterns (EIPs): reusable and proven solutions to recurring integration problems, based on messaging concepts.
- Apache Camel provides built-in connectors to both new and legacy systems, ranging from Enterprise Resource Planning (ERP) systems to cloud services.
- Red Hat Fuse supports three runtimes: JBoss EAP, Apache Karaf, and Spring Boot. These three runtimes can be deployed as either stand-alone application containers over a certified Java Virtual Machine (JVM) or as containerized applications on OpenShift.
- A Red Hat Fuse subscription includes entitlements and full support for Red Hat JBoss EAP and Red Hat AMQ, but does not include a subscription to Red Hat OpenShift Container Platform.
- A Fuse developer solves integration problems by designing Camel routes, which connect two endpoints. A route transforms and moves data from a source to one or more destination endpoints.
- Camel endpoints can be file servers, message-oriented middleware, ERP systems, SOAP web services, cloud applications, and more. Camel connectors provide configuration for these endpoints.
- A Fuse developer solves complex integration problems by composing multiple EIPs into one or more Camel routes, which can be deployed on a single runtime, on multiple runtimes, and as multiple containers.
- A Fuse developer describes a Camel route using either the Java or the XML Domain Specific Language (DSL).
- A Fuse developer can choose among the Context and Dependency Injection (CDI), the Spring framework, and the OSGi Blueprint frameworks to wire routes, connectors, and custom Java Beans inside an application.

Chapter 2

Creating Routes

Goal

Develop simple Camel routes.

Objectives

- Create a route that reads and writes data to the file system.
- Create a route that reads from an FTP server.
- Develop a route that filters messages.
- Implement a Camel route with a Content Based Router and provide basic error handling.
- Manipulate exchange headers in routes.

Sections

- Reading and Writing Files (and Guided Exercise)
- Reading Messages from an FTP Server (and Guided Exercise)
- Filtering Messages (and Guided Exercise)
- Routing Messages from JMS (and Guided Exercise)
- Configuring Exchange Headers (and Guided Exercise)

Lab

Creating Routes

Reading and Writing Files

Objectives

After completing this section, students should be able to create a route that reads and writes data to the file system.

Camel Routes

In Camel, a *route* describes the path of a message from one *endpoint* (the origin) to another endpoint (the destination). Routes are a critical aspect of Camel because they define integration between endpoints. With the help of components, Routes are able to move, transform, and split messages. Traditionally, integration implementation requires lots of complicated and unnecessary coding. With Camel, routes are defined in a few simple, human-readable lines of code in either Java DSL or Java XML.

A route starts with a *consumer*, which receives the data from a point of origin. The term "consumer" is slightly counter-intuitive because it is often used to refer to the destination of a message. With Camel, consider that the consumer is referring to where and how the initial message is being picked up. The origin endpoint is any Camel component, such as a location on the file system, a JMS queue, or even a tweet from Twitter. The route then directs the message to the *producer*, which sends data to a similarly configured destination. By abstracting the integration code, developers are able to implement EIPs that manipulate or transform the data within the Camel route without requiring changes to either the origin or the destination.

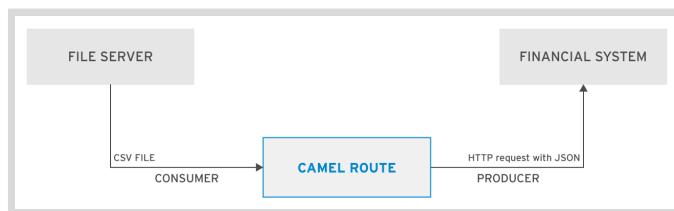


Figure 2.1: Route Example

In this example, a consumer receives a CSV file from a file server. The file server is the origin while the CSV file is the message. The route transforms the data into the JSON format, and the producer routes the JSON to a web system through an HTTP request.

CamelContext

To implement a Camel route, the route must be attached to a **CamelContext** instance. The **CamelContext** loads all resources like components, endpoints, and converters in the runtime system to execute the routes.

A common way to instantiate a **CamelContext** and attach routes to it is by using the Spring Framework. Camel provides integration with this framework, including an XML configuration dialect to use inside Spring beans configuration files.

Alternatively, you can instantiate a new **CamelContext** using the Java language:

```
DefaultCamelContext camelContext = new DefaultCamelContext(registry); ①  
camelContext.setName("camelContext"); ②  
camelContext.addRoutes(new FileRouter()); ③  
camelContext.start(); ④
```

- ① Instantiates a new context.
- ② Sets the name of the context.
- ③ Adds a new route to the context. You can invoke this method multiple times to add multiple routes.
- ④ Starts the context service.

The Camel-specific configuration inside a Spring beans configuration uses the XML DSL. A Spring beans configuration can also refer to routes defined using the Java DSL by declaring **RouteBuilder** objects as Spring beans.

Java DSL

In Java DSL routes in Camel are created by extending the **org.apache.camel.builder.RouteBuilder** class and overriding the **configure** method. A route is composed of two endpoints: a consumer and a producer. In Java DSL, this is represented by the **from** method for the consumer and the **to** method for the producer. Inside the overridden **configure** method, use the **from** and **to** methods to define the route.

```
public class FileRouteBuilder extends RouteBuilder {  
  
    @Override  
    public void configure() throws Exception {  
  
        from("file:orders/incoming")  
            .to("file:orders/outgoing");  
  
    }  
}
```

In this example, the consumer uses the **file** component to read all files from the **orders/incoming** path and the producer uses the **file** component to move to the **orders/outgoing** path.

Inside a **RouteBuilder** class, each route can be uniquely identified using a **routeId** method. Naming a route makes it easy to verify route execution in the logs, and also simplifies the process of creating unit tests. Multiple routes can be created by calling the **from** method many times.

```
public void configure() throws Exception {  
  
    from("file:orders/incoming")  
        .routeId("route1")  
        .to("file:orders/outgoing");  
  
    from("file:orders/new")  
        .routeId("routefinancial")  
        .to("file:orders/financial");  
}
```

Each component can specify configuration attributes. You can add attributes by appending a `?` after the origin or destination. Refer to the Camel documentation for component-specific attributes.

```
public void configure() throws Exception {  
  
    from("file:orders/incoming?include=order.*xml") ①  
        .to("file:orders/outgoing/?fileExist=Fail"); ②  
}
```

- ①** The route consumes only files with a name starting with "order", and it is an XML file.
- ②** Throws an exception if a given file already exists.

To enable a route, you need to register it as a Spring bean inside a Spring beans configuration file:

```
<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
       xsi:schemaLocation="http://www.springframework.org/schema/beans  
                           http://www.springframework.org/schema/beans.xsd  
                           http://camel.apache.org/schema/spring  
                           http://camel.apache.org/schema/spring/camel-spring.xsd">  
    <bean class="com.redhat.training.jb421.FileRouteBuilder" id="fileRouteBuilder"/>  
①  
    <camelContext id="jb421Context" xmlns="http://camel.apache.org/schema/spring">  
        <routeBuilder ref="fileRouteBuilder"/> ②  
    </camelContext>  
</beans>
```

- ①** Creates a new Spring bean.
- ②** Enables the route in the **CamelContext**.

XML DSL

Method names from the Java DSL map directly to XML elements in the Spring DSL in most cases. However, due to syntax differences between Java and XML, sometimes the name and the structure of elements are different in Spring DSL. Refer to Camel documentation for the correct structure of the route methods.

To use the Spring DSL, declare a **camelContext** element, using the custom Camel Spring namespace, inside a Spring beans configuration file. Inside the **camelContext** element, declare one or more **route** elements starting with a **from** element and usually ending with a **to** element. These **from** and **to** elements are similar to the Java DSL **from** and **to** methods:

```
<camelContext id="jb421Context" xmlns="http://camel.apache.org/schema/spring">  
    <route>  
        <from uri="file:orders/incoming"/>  
        <to uri="file:orders/outgoing"/>  
    </route>  
</camelContext>
```



References

For more information, refer to the *Building Blocks for Route Definitions* chapter in the *Apache Camel Development Guide* at

https://access.redhat.com/documentation/en-us/red_hat_fuse/7.0/html/apache_camel_development_guide/fusemrstartedblocks

► Guided Exercise

Processing Orders with the File Component

In this exercise, you will receive multiple files containing orders and save them to a different folder avoiding duplicated data.

Outcomes

You should be able to complete the project POM file to have the correct Camel dependencies and complete the Camel route using the Java DSL.

Before You Begin

Run the following command to download the starter project and data files used by this exercise. Click **Applications** → **Favorites** → **Terminal** in the desktop menu and type:

```
[student@workstation ~]$ lab processing-orders setup
```

- ▶ 1. Import the processing-orders project into JBoss Developer Studio.
 - 1.1. Open JBoss Developer Studio (**Applications** → **Programming** → **JBoss Developer Studio**). JBoss Developer Studio requests a default workspace; keep the default configuration (**/home/student/workspace**) and click **Launch**.
 - 1.2. Import the Maven project by selecting from the JBoss Developer Studio main menu **File** → **Import**.
 - 1.3. From the **Import** dialog box, select **Maven** → **Existing Maven Projects** and click **Next >**.
 - 1.4. Click **Browse** and choose **/home/student/JB421/labs/processing-orders** and click **Ok** to choose the Maven project. Click **Finish** to complete the import process in JBoss Developer Studio.
A new project named **processing-orders** will be listed in the **Project Explorer** view.



Important

The project imports with errors, this is expected. You will resolve these errors in the subsequent steps of this exercise.

- ▶ 2. Complete the Maven project dependencies.
 - 2.1. Expand the **processing-orders** item in the **Project Explorer** pane on the left, and then double-click the **pom.xml** file.
 - 2.2. Click the **pom.xml** tab at the bottom of the file to view the contents of the **pom.xml** file.

2.3. Add the **camel-core** and **camel-spring** dependencies to the project.

```
<!-- TODO add camel dependencies -->
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-core</artifactId>
</dependency>

<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-spring</artifactId>
</dependency>
```

Notice that there is no **version** element specified. This is because the version is inherited from the **jboss-fuse-parent** bill of materials (BOM) included the parent project **pom.xml** file.

2.4. Press **Ctrl+S** to save the changes.

► 3. Create the route by updating the **FileRouteBuilder** class.

- 3.1. Open the **FileRouteBuilder** class by expanding the **processing-orders** item in the **Project Explorer** tab in the left pane of JBoss Developer Studio, then click **processing-orders** → **src/main/java** → **com.redhat.training.jb421** to expand it. Double-click the **FileRouteBuilder.java** file.
- 3.2. Update this class to extend the **org.apache.camel.builder.RouteBuilder** superclass. Camel provides the **RouteBuilder** class to enable a new route.

```
//TODO: Enable the route by extending the RouteBuilder superclass
public class FileRouteBuilder extends RouteBuilder {
    ...
}
```

- 3.3. The superclass requires the implementation of the **configure** method. Add an empty method:

```
//TODO Implement the configure method
@Override
public void configure() throws Exception {
}
```

- 3.4. Add a file consumer to the route using the **file:** component.

Configure the endpoint to consume from the **orders/incoming** directory and use the **include** option to configure the endpoint to consume only XML files where the name starts with **order**.

```
public void configure() throws Exception {
    from("file:orders/incoming?include=order.*xml")
}
```

- 3.5. Add a file producer to the route using the **file:** component.

Configure the endpoint to create the outgoing files to the **orders/outgoing** folder. The route must throws a **GenericFileOperationException** exception if a duplicate file is provided in the **orders/incoming** folder.

```
public void configure() throws Exception {  
    from("file:orders/incoming?include=order.*xml")  
        .to("file:orders/outgoing?fileExist=Fail");  
}
```

3.6. Save your changes to the file using **Ctrl+S**.

► 4. Populate the **orders/incoming** folder.

4.1. Go back to the terminal window and run the provided shell script to populate the **orders/incoming** folder:

```
[student@workstation ~]$ cd ~/JB421/labs/processing-orders  
[student@workstation processing-orders]$ ./setup-data.sh  
...  
'Preparation complete!'
```

4.2. Inspect the incoming files:

```
[student@workstation processing-orders]$ ls orders/incoming  
noop-1.xml  order-2.xml  order-3.xml  order-4.xml  order-5.xml  order-6.xml
```

There should be five **order-?.xml** files and one **noop-?.xml** file.

► 5. Test the route.

5.1. Run the route by using the **camel:run** Maven goal:

```
[student@workstation processing-orders]$ mvn clean camel:run
```

The expected output of a successful execution:

```
...  
[INFO] Building GE: Processing Orders with the File Component 1.0  
...  
[INFO] Using org.apache.camel.spring.Main to initiate a CamelContext  
[INFO] Starting Camel ...  
...  
INFO Route: route1 started and consuming from: file://orders/incoming?  
include=order.*xml  
INFO Total 1 routes, of which 1 are started  
INFO Apache Camel 2.21.0.fuse-000077-redhat-1 (CamelContext: jb421Context)  
started in 0.254 seconds  
...
```

5.2. Open a new terminal window and inspect the **orders/outgoing** folder to verify that only order files are available:

```
[student@workstation ~]$ cd ~/JB421/labs/processing-orders
[student@workstation processing-orders]$ ls orders/outgoing
order-2.xml order-3.xml order-4.xml order-5.xml order-6.xml
```

The folder should not include the **noop-1.xml** file.

- 5.3. Run the shell script to recreate the files:

```
[student@workstation processing-orders]$ ./duplicate-files.sh
...
'Duplication complete!'
```

- 5.4. Go back to the terminal window where the Camel route is running and verify that the **GenericFileOperationException** exception was thrown due to the duplicate files.

```
Message History
-----
RouteId          ProcessorId      ...
[route1]          [route1]          ...
[route1]          [to1]            ...

Stacktrace
-----
org.apache.camel.component.file.GenericFileOperationFailedException: ...
```

- 5.5. Terminate the route using **Ctrl+C**.

The following is the expected output when terminating the route:

```
INFO Received hang up - stopping the main instance.
...
INFO Route: route1 shutdown complete, was consuming from: file://orders/incoming?
include=order.*xml
INFO Graceful shutdown of 1 routes completed in 0 seconds
...
INFO Apache Camel 2.21.0.fuse-000077-redhat-1 (CamelContext: jb421Context) uptime
2 minutes
INFO Apache Camel 2.21.0.fuse-000077-redhat-1 (CamelContext: jb421Context) is
shutdown in 0.044 seconds
```

► 6. Clean up.

Close the project in JBoss Developer Studio to conserve memory.

In the **Project Explorer** view, right-click **processing-orders** and click **Close Project**.

This concludes the guided exercise.

Reading Messages from an FTP Server

Objective

After completing this section, students should be able to create a route that reads from an FTP server.

Message Exchange Patterns

The message exchange pattern describes the behavior of the communication between the producer and the consumer. Two of the primary message exchange patterns are:

Synchronous (**InOut**)

Consumer sends the request and consumer waits for a response.

Asynchronous (**InOnly**)

Consumer sends the request and does not wait for a response.

To support these patterns, Camel uses an *exchange* object. The exchange object encapsulates a received message, known as **inMessage**, and stores its metadata. When using the **InOut** pattern, the exchange also encapsulates a reply message, also known as the **outMessage**.

Camel supports several message exchange patterns, including **InOnly** and **InOut**. These patterns serve different purposes, and one may be more appropriate than the other, depending on the functionality of your route.

By default, many Camel components, such as jms, file, or ftp, use **InOnly** exchanges unless otherwise specified. However, this can vary from one component to another, so you should check the documentation for the components you are using in your route for their default exchange pattern.

The **InOnly** Exchange Pattern

Messages in routes that use the **InOnly** pattern are sometimes referred to as event messages. These messages are a one-way communication and, as such, only the **inMessage** will ever have a value on the message exchange. The **InOnly** pattern is appropriate to use when the producer in your route does not need any reply information from the route consumer.

For example, a route consumes an order file from an FTP server and produces a message to the financial system. The consumer does not need to wait for a reply from the producer.

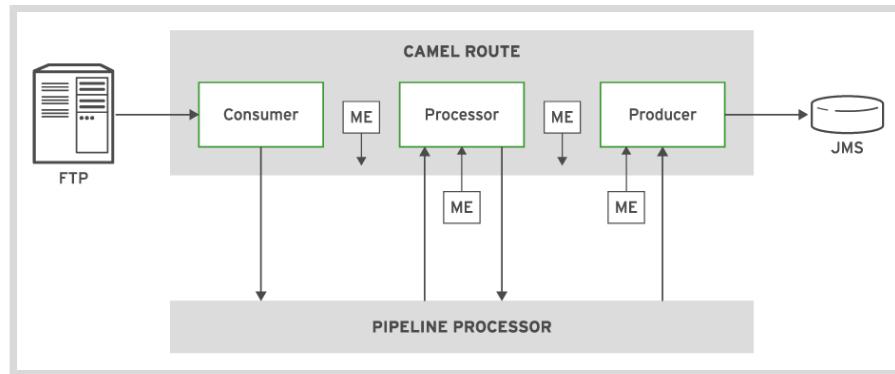


Figure 2.2: Camel route and pipeline processor

Note

Processors do not affect the out message, because they are **InOnly** and the exchange does not contain an out message.

To access the **inMessage** object on the Camel exchange at runtime, use the **getIn** method provided by the **Exchange** interface as shown below:

```
Message inMessage = exchange.getIn();
```

You should access the **inMessage** object of the exchange object when you want to manipulate the original message, such as adding new headers or changing the original content.

The InOut Exchange Pattern

Messages in routes using the **InOut** pattern are referred to as *request-reply* messages. The **InOut** pattern is used when a route's consumer requires a response from the producer before proceeding. This can be useful to report the results of some action that the route has taken, or to make a callback when the route is complete.

For example, the route consumes an order request from a client. The producer sends a message to the bank system to approve the transaction, and needs to wait for a reply message with the transaction status. If the transaction is approved, the route needs to produce a new message to the shipment route. If not, the route needs to produce a new message that sends an email to the client asking for a new payment transaction.

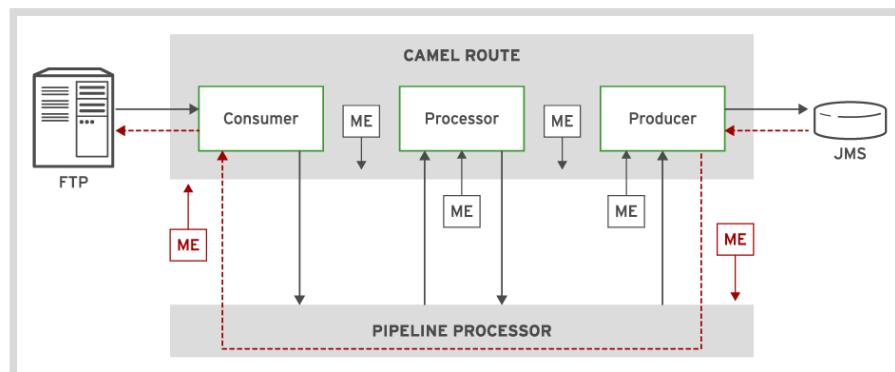


Figure 2.3: Camel route with request and reply messages

To access the **outMessage** object on the Camel exchange at runtime, use the **getOut** method provided by the **Exchange** interface as listed:

```
Message outMessage = exchange.getOut();
```

You should access the **outMessage** of the exchange object when you want to manipulate the reply message.

Messages in Camel

A message is a uniquely identified data structure for storing data from a producer to a consumer. The following are the primary components of messages:

Headers

The message header contains metadata such as information about the sender of the message and security details in a **java.util.Map** collection. Use the **setHeader** method to add a new header.

Placing custom metadata in a message header is a useful approach for dynamically routing messages or providing additional ways to categorize messages.

Attachments

Contain file-based contents to be transmitted to a receiver. For example, you can attach an image in the email component.

Body

The message body contains the data to be processed by the receiver or manipulated by Camel processors. This is usually JSON or XML data, but can be an **Object** of any type.

Fault flag

The fault flag is a property that developers can use to identify whether the message contained an error during processing.

Accessing Component and EIP Documentation

Extensive documentation for all Camel components and EIPs are available directly from camel.apache.org. Red Hat also maintains documentation for all supported Camel components and EIPs, which can be found in the Red Hat Fuse documentation at https://access.redhat.com/documentation/en-us/red_hat_fuse/7.1/.

While some of information available from these two sources is mostly identical, be aware that:

- The Camel community documentation may contain changes that have not yet been incorporated by the Red Hat Fuse product.
- Some Camel components may not be included as part of the Red Hat Fuse product. They can be added by the developer, but its use is not supported by Red Hat. This might be an important consideration for production environments.
- The Camel community documentation does not contain details about packaging and deploying to the supported Apache Karaf, JBoss EAP, and Spring Boot containers. These containers are discussed in more detail in a later chapter in this course.

The component documentation provides useful information about component syntax and attributes. When reviewing the documentation for components, carefully observe the following information:

URI format

A component URI format specifies the format required when defining an endpoint. This typically involves a specific scheme to the URI such as `ftp:` or `file:` as well as other required and optional values to be included in the URI to control component behavior. The URI always begins with the name of the Camel component. For example, the `ftp` component URI format:

```
ftp://[username@]hostname[:port]/foldername[?options]
```

For the FTP component, **hostname** is a required options, but the **username** and **port** are optional.

URI options

The URI options are where most of the specific configuration parameters for a component are defined. These often have default values, so be sure to review the expected component behavior.

Message Headers

Camel components frequently provide special headers that are automatically populated when that component is used. For example, in the `ftp` component, you have a message header containing the original file name.

Required Dependencies

Most components are not included in **camel-core**. This section of the documentation informs you about required dependencies for those components that need to be added to the `pom.xml` file.

Logging in Camel

Camel provides two logging features: the *log component*, and the *log EIP*.

The log component is used as a producer endpoint to output log events using SLF4J logging. Logging is an important feature to debug and collect information about routes. Output is delegated to the first logging API available in the class path. For example, the Apache log4J or `java.util.logging` API is used, based on which API is available first in the class path.

The log component URI format is:

```
log:loggingCategory[?options]
```

For the log component, the **loggingCategory** option is the name of the logging category to use. Usually, this is the Java package name.

The following are examples of using the **log** component:

- Set the logging name and level explicitly:

```
from("activemq:input")
.to("log:com.foo?level=DEBUG")
.to("activemq:output");
```

- Format the output of the information from the exchange:

```
from("activemq:input")
.to("log:com.foo?showAll=true&multiline=true&level=DEBUG")
.to("activemq:output");
```

The log EIP, provided by the **log** DSL method, is another way to add logging to a Camel route. You should use the log EIP when you want to log custom messages. It also uses the SLF4J logging facade and messages are formatted using the *Simple expression language* (Simple EL), covered in detail later in this section.

Some examples of using the **log** DSL method are:

- Set the logging name and level explicitly using the Java DSL

```
from("activemq:input")
.log(LoggingLevel.DEBUG, "com.foo", "Got a message")
.to("activemq:output");
```

- Set the logging name using the XML DSL:

```
<from uri="activemq:input"/>
<log loggingLevel="DEBUG" loggerRef="com.foo" message="Got a message"/>
<to uri="activemq:output"/>
```

Using Simple Expression Language

Log, filter, and other EIPs can use the Camel Simple EL to get the current exchange attributes (body, headers, properties) from a message. For example:

```
from("file:in")
.log("message body:${body}")
.to("file:out")
```

Simple expressions are delimited by a dollar sign followed by braces: `${...}` . It is similar to the Java Server Pages (JSP) EL. Objects such as the route parent Camel context, the current exchange, the input and output messages, among other convenience objects, have names predefined by Camel. Refer to Camel Simple EL documentation for a list of all available variables.

You can use Simple EL to specify dynamic URI options for your endpoints, which refer to specific data from the exchange. For example, using the **CamelFileHost** header in the file name option of the file component in the following XML DSL:

```
<to uri="file:orders?fileExist=Append&fileName=${header.CamelFileHost}" />
```

Running Camel Using Maven

Camel provides a plug-in for Maven to initialize the **CamelContext** object using the Camel Spring integration libraries. This is beneficial because you can easily test your routes without having to initialize the **CamelContext** in a **main** method.

Using the Camel Maven plug-in, the application can be started using the Maven goal **camel:run** that starts an instance of **CamelContext** inside Maven in a separate thread.

Any Camel project using Maven can use this feature by adding the following to the **pom.xml** file:

- The Spring Framework integration dependency.
- The plug-in reference.

The Spring Framework integration dependency can be referenced using the following statement:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-spring</artifactId>
</dependency>
```

The Camel Maven plug-in is activated by adding the following to the POM:

```
<plugin>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-maven-plugin</artifactId>
</plugin>
```

Message Exchanges in Camel

Camel uses exchange objects to store messages processed through Camel routes. Camel exchanges also store routing information and properties. Exchange objects are composed of the following:

- *Exchange ID*: A unique identifier generated by Camel.
- *Message Exchange Pattern (MEP)*: Describes how messages are exchanged from a sender to a receiver. The pattern can be an **InOnly** request (contains only one message) or an **InOut** request (containing two messages, one from the sender and another from the receiver).
- *Exception*: Stores any errors generated during the exchange processing.
- *Properties*: Stores metadata from the exchange processing, such as processing status. The route code can use properties to safely store information not meant to be processed by endpoints.
- *In message*: The request sent by the sender.
- *Out message*: The answer sent by the receiver.



Note

See the **org.apache.camel.Exchange** class interface documentation for the full API.

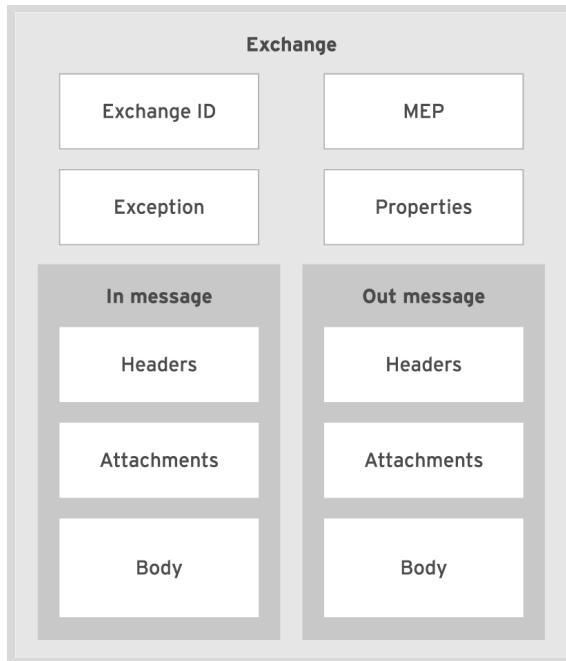


Figure 2.4: Exchange data structure



References

For more information, refer to the *Log EIP* chapter in the *Apache Camel Development Guide* at

https://access.redhat.com/documentation/en-us/red_hat_fuse/7.1/html/apache_camel_development_guide/sysman#LogEIP

For more information, refer to *The Simple Expression Language* chapter in the *Apache Camel Development Guide* at

https://access.redhat.com/documentation/en-us/red_hat_fuse/7.1/html-single/apache_camel_development_guide/sysman#Simple

► Guided Exercise

Creating a route with the FTP component

In this exercise, you will create a route that receives incoming transactions from multiple files on a remote FTP server, reads the body and headers of the exchange message, and saves the transactions to a local directory.

Outcomes

You should be able to complete the project POM file to have the correct Camel dependencies and complete the Camel route source using the Java DSL.

Before You Begin

Run the following command to download the starter project and data files used by this exercise. Click **Applications** → **Favorites** → **Terminal** in the desktop menu and type:

```
[student@workstation ~]$ lab route-ftp setup
```

- ▶ 1. Import the route-ftp project into JBoss Developer Studio.
 - 1.1. Open JBoss Developer Studio (**Applications** → **Programming** → **JBoss Developer Studio**). JBoss Developer Studio requests a default workspace; keep the default configuration (**/home/student/workspace**) and click **Launch**.
 - 1.2. Import the Maven project by selecting the JBoss Developer Studio main menu **File** → **Import**.
 - 1.3. From the **Import** dialog box, select **Maven** → **Existing Maven Projects** and click **Next >**.
 - 1.4. Click **Browse** and choose **/home/student/JB421/labs/route-ftp** and click **OK** to choose the Maven project. Click **Finish** to complete the import process in JBoss Developer Studio.
A new project named **route-ftp** is listed in the **Project Explorer** view.
- ▶ 2. Complete the Maven project dependencies.
 - 2.1. Expand the **route-ftp** item in the **Project Explorer** pane on the left, and then double-click the **pom.xml** file.
 - 2.2. Click the **pom.xml** tab at the bottom of the file to view the contents of the **pom.xml** file.
 - 2.3. Add the **camel-ftp** dependency to the project.

```
<!-- TODO add camel-ftp dependency -->
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-ftp</artifactId>
</dependency>
```

Notice that there is no **version** element specified. This is because the version is inherited from the **jboss-fuse-parent** bill of materials (BOM) included the parent project **pom.xml** file.

- 2.4. Press **Ctrl+S** to save the changes.

- 3. Create the route by updating the **FTPRouteBuilder** class.
- 3.1. Open the **FTPRouteBuilder** class by expanding the **route-ftp** item in the **Project Explorer** tab in the left pane of JBoss Developer Studio, then click **route-ftp** → **src/main/java** → **com.redhat.training.jb421** to expand it. Double-click the **FTPRouteBuilder.java** file.

- 3.2. Add a file consumer to the route using the ftp component.

Configure the endpoint to consume from the FTP server available from the **services** VM and use the **include** option to configure the endpoint to consume only XML files where the name starts with **order**.

```
public void configure() throws Exception {
    from("ftp://services.lab.example.com?"
        + "username=student&password=student"
        + "&delete=true&include=order.*xml")
}
```

- 3.3. Add the log DSL method to the route definition to print the server name of the FTP server and file name to the console:

```
public void configure() throws Exception {
    from("ftp://services.lab.example.com?"
        + "username=student&password=student"
        + "&delete=true&include=order.*xml")
        .log("New file ${header.CamelFileName} picked up"
            +" from ${header.CamelFileHost}")
}
```

- 3.4. Add the process DSL method to the route definition to add custom code to a Camel route. Create a new instance of the **ExchangePrinter** java class.

```
public void configure() throws Exception {
    from("ftp://services.lab.example.com?"
        + "username=student&password=student"
        + "&delete=true&include=order.*xml")
        .log("New file ${header.CamelFileName} picked up"
            +" from ${header.CamelFileHost}")
        .process(new ExchangePrinter())
}
```

The compilation error in the **process** method is expected because the **ExchangePrinter** class is not fully implemented.

The **process** DSL method is covered in detail later in the course. For this exercise, you only need to understand that the **ExchangePrinter** class is a Camel processor that is a generic way to add custom code to a Camel route.

- 3.5. Add a file producer to the route using the file component.

Configure the endpoint to create the outgoing files to the **orders/outgoing** folder. The route must throw a **GenericFileOperationException** exception if a duplicate file is provided in the **orders/incoming** folder.

```
public void configure() throws Exception {  
    from("ftp://services.lab.example.com?"  
        + "username=student&password=student"  
        + "&delete=true&include=order.*xml")  
    .log("New file ${header.CamelFileName} picked up"  
        +" from ${header.CamelFileHost}")  
    .process(new ExchangePrinter())  
    .to("file:orders/outgoing?fileExist=Fail");  
}
```

- 3.6. Save your changes to the file using **Ctrl+S**.

► 4. Implement the **ExchangePrinter** class to output the exchange content to the log.

- 4.1. Open the **ExchangePrinter** class by expanding the **route-ftp** item in the **Project Explorer** tab in the left pane of JBoss Developer Studio, then click **route-ftp** → **src/main/java** → **com.redhat.training.jb421** to expand it. Double-click the **ExchangePrinter.java** file.
- 4.2. Update this class to implement the **org.apache.camel.Processor** interface. Camel provides the **Processor** interface to create a new processor. This processor provides detailed logging information about the Camel **Exchange**, dumping the input message body and all its headers.

```
//TODO: Implements the Processor interface  
public class ExchangePrinter implements Processor {  
}
```

- 4.3. The interface requires the implementation of the **process** method. Add an empty method:

```
//TODO: Implement the process method  
@Override  
public void process(Exchange exchange) throws Exception {  
}
```

- 4.4. Recover the text of the body of the exchange message and log it:

```
@Override  
public void process(Exchange exchange) throws Exception {  
    String body = exchange.getIn().getBody(String.class);  
    log.info("Body: " + body);  
}
```

The **getIn** method retrieves the **Message** object from the exchange, and the **getBody** method retrieves the message body. The desired data type is passed as an argument (in this case **String.class**) and Camel uses its data conversion features if needed.



Note

Using a cast to a **String** may not print a readable information every time because the message body can be an **Object** of any type, and might not override the **toString** method.

4.5. Recover the headers of the exchange message and log them:

```
@Override  
public void process(Exchange exchange) throws Exception {  
    ...  
    log.info("Headers:");  
    Map<String, Object> headers = exchange.getIn().getHeaders();  
    for(String key: headers.keySet()){  
        log.info("Key: " + key + " | Value: " + headers.get(key));  
    }  
}
```

As with the body, it is necessary to retrieve the **in** message from the exchange, then the **getHeaders** method is used to pull the set of headers.

Notice the **keySet** method provided by **Set** to iterate over this list of keys and access every header name available in the message.

4.6. Save your changes to the file using **Ctrl+S**.

▶ 5. Populate the FTP server.

5.1. Go back to the terminal window and run the provided shell script to populate the FTP server:

```
[student@workstation ~]$ cd ~/JB421/labs/route-ftp  
[student@workstation route-ftp]$ ./setup-data.sh  
...  
'Upload complete!'
```

5.2. Inspect the incoming files:

```
[student@workstation route-ftp]$ ls orders/incoming  
noop-1.xml order-2.xml order-3.xml order-4.xml order-5.xml order-6.xml
```

There should be five **order-?.xml** files and one **noop-?.xml** file.

► 6. Test the route.

- 6.1. Run the route by using the **camel:run** Maven goal:

```
[student@workstation route-ftp]$ mvn clean camel:run
```

Inspect that the route picked up the data files, and both the **log** DSL method and the **ExchangePrinter** class wrote details of the files to the console, as shown:

```
...
INFO New file order-1.xml picked up from services.lab.example.com ①
INFO Body: RemoteFile[order-1.xml] ②
INFO Headers: ③
INFO Key: breadcrumbId | Value: ID-workstation-lab-example-
com-51477-1477509329708-0-1
INFO Key: CamelFileAbsolute | Value: false
INFO Key: CamelFileAbsolutePath | Value: order-1.xml
INFO Key: CamelFileHost | Value: infrastructure.lab.example.com
INFO Key: CamelFileLastModified | Value: 1477523520000
INFO Key: CamelFileLength | Value: 271
INFO Key: CamelFileName | Value: order-1.xml
INFO Key: CamelFileNameConsumed | Value: order-1.xml
INFO Key: CamelFileNameOnly | Value: order-1.xml
INFO Key: CamelFileParent | Value: /
INFO Key: CamelFilePath | Value: /order-1.xml
INFO Key: CamelFileRelativePath | Value: order-1.xml
INFO Key: CamelFtpReplyCode | Value: 226
INFO Key: CamelFtpReplyString | Value: 226 Transfer complete.
...
```

- ① The **log** DSL method output.
- ② The **ExchangePrinter** class output of the String representation of the message body.
- ③ The **ExchangePrinter** class output of all headers attached to the message.

- 6.2. Open a new terminal window and inspect the **orders/outgoing** folder to verify that only order files are available:

```
[student@workstation ~]$ cd ~/JB421/labs/route-ftp
[student@workstation route-ftp]$ ls orders/outgoing
order-2.xml order-3.xml order-4.xml order-5.xml order-6.xml
```

The folder should not include the **noop-1.xml** file.

- 6.3. Terminate the route using **Ctrl+C**.

The following is the expected output when terminating the route:

```
INFO Received hang up - stopping the main instance.  
...  
...  
INFO Apache Camel 2.21.0.fuse-000077-redhat-1 (CamelContext: jb421Context) uptime  
2 minutes  
INFO Apache Camel 2.21.0.fuse-000077-redhat-1 (CamelContext: jb421Context) is  
shutdown in 0.044 seconds
```

► 7. Clean up.

Close the project in JBoss Developer Studio to conserve memory.

In the **Project Explorer** view, right-click the route-ftp project and click **Close Project**.

This concludes the guided exercise.

Filtering Messages

Objectives

After completing this section, students should be able to:

- Develop a route that filters messages.
- List automatic conversion mechanisms, endpoints, languages, and some components in Camel that are used to manage routes.
- Identify use cases where message filter enterprise integration pattern (EIP) can be used.
- Implement XPath filters.

Introducing CamelContext

The **CamelContext** class is the Camel runtime environment that provides important features, such as:

- *Simple extension support:* Camel works with components that can be injected by using popular *inversion of control* (IoC) frameworks based on either XML configuration or annotations such as Spring and CDI. These frameworks allow developers to integrate various components in a loosely coupled type-safe manner. Additionally, by using these industry standard techniques, developers can leverage existing solutions in their Camel routes.
- *Life cycle:* A **CamelContext** instance must be started to make its routes available for execution. Sometimes the routes must be stopped for maintenance purposes, and they can be suspended and resumed. The **CamelContext** instances can also be stopped to shut down each route cleanly and avoid any processing problems.
- *Services:* Services can be added to the **CamelContext** that provide important features such as persistence, configuration, or integration with other frameworks.

When using the **camel:run** Maven goal or a runtime environment with Camel support, such as Red Hat Fuse, the **camelContext** is automatically started.

The **org.apache.camel.impl.DefaultCamelContext** class is used to instantiate a **CamelContext** instance. However, the **CamelContext** instance needs to be started explicitly to enable processing the routes. You can start the **CamelContext** in the **main** method.

```
CamelContext context = new DefaultCamelContext();
context.start();
```

To shut down the context gracefully, invoke the **stop** method when the route execution is completed in the **main** method.

```
context.stop();
```

Creating a RouteBuilder to Build Routes

Use the **RouteBuilder** abstract class provided by Camel API to develop routes. Because of the dynamic nature of routes, it has many methods that are often chained together. This results in a route that is easily read and understood by a human. In Java programming, this technique is known as a *fluent interface*; essentially creating a domain-specific language.

The fluent interface from the **RouteBuilder** class provides a clean and direct code to define routes and create helper objects used in your Camel routes, such as enterprise integration pattern implementations.

Normally, the starting point to create a route is to extend a **RouteBuilder** class and override the **configure** method, for example:

```
public class FileRouteBuilder extends RouteBuilder {

    @Override
    public void configure() throws Exception {
        ...
    }
}
```

Alternatively, a Spring beans configuration file can declare a route. To achieve this goal, a **route** element is declared, nested in the **camelContext** element. For example:

```
<camelContext id="jb421Context" xmlns="http://camel.apache.org/schema/spring">
    <route>
        ...
    </route>
</camelContext>
```

Routes defined this way use the Camel XML domain specific language (DSL), also known as Spring DSL.



Important

Multiple routes can be declared in the **configure** method of a **RouteBuilder** implementation.

A **camelContext** element can also contain references to routes defined using the Java DSL, by declaring the Java class as a Spring bean and referencing it inside the **camelContext** element:

```
<bean class="com.example.MyRouteBuilder" id="myRouteBuilder"/>
<camelContext id="jb421Context" xmlns="http://camel.apache.org/schema/spring">
    <routeBuilder ref="myRouteBuilder"/>
</camelContext>
```

You can use both XML DSL and Java DSL routes inside the same **camelContext** element.

Declaring Endpoints Using Components

Data flows from an consumer to one or more producers in Camel. Each consumer or producer is an endpoint, which abstracts the end of a channel through which messages can be sent or received.

Declaring an endpoint is done using URI syntax:

```
<scheme>://<context/path>?<options>
```

For example, to connect to a file system, you must use the **file** component, which is associated with the **file:** URI prefix:

```
file://folder?fileExist=Append
```

Camel parses this URI string to instantiate an **Endpoint** and configure it according to the URI options. In the previous example, if the destination file already exists in the file system, it is not overwritten but appended. If multiple options are needed, the each one is separated by an ampersand (&).

A component is an endpoint factory and the main abstraction in Camel. A **Component** class is responsible for the following tasks:

- Connecting to the integration system
- Marshaling and unmarshaling data to Camel Message standards

Components may be explicitly configured and added to the **CamelContext**, injected by an IoC framework, or auto-discovered using URLs.

Implementing the Filter EIP

Camel implements most of the enterprise integration patterns and provides them as methods in the **RouteBuilder** class or elements in the XML DSL.

Camel defines the message filter pattern to remove some messages during route execution based on the content of the Camel message.

To filter messages sent to a destination, use the following Java DSL:

```
from("<Endpoint URI>")
.filter(<filter>)
.to("<Endpoint URI>");
```

The filter must be a Camel *predicate*, which evaluates to either **true** or **false** based on the message content. The filter drops any messages that evaluate to **false** and the remainder of the route is not processed. Predicates can be created using expressions, which Camel evaluates at runtime.

Because of the number of data formats supported by integration systems, a set of technologies can be used to filter information. For example, in an XML-based message, XPath can be used to identify fields in an XML file.

To evaluate if a certain value is available at a specific XML element, use the following syntax:

```
filter(xpath("/body/title/text() = 'Hello World'"))
```

Likewise, the **Simple** expression language can be used to filter Java objects.

```
from("direct:a")
    .filter(simple("${header.foo} == 'bar')")
        .to("direct:b")
    .filter().xpath("/person[@name='James']")
        .to("mock:result");
```

Introducing Expressions

Camel uses *expressions* to look for information inside messages. They support a large number of data formats, including Java-based data and common data format exchanges (XML, JSON, SQL, and so on).

An expression language used to identify Java objects is the Simple EL. It uses a syntax that resembles the Java Server Pages expression languages to search for attributes in an object. For example, in a class named **Order**, with an **address** attribute that contains a ZIP code, the Simple expression language to search for the zip code 33212 is:

```
 ${order.address.zipCode = '33212'}
```

Implementing Predicates

Predicates in Camel are essentially expressions that must return a boolean value. This is often used to look for a certain value in an **Exchange** instance. Predicates can be leveraged by Camel in conjunction with expression languages to customize routes and filter data in a route. For example, to use the Simple expression language in a filter, the Simple expression must be called inside a route calling the **simple** method. Similarly, to use an XPath expression, there is a method named **xpath**.

The following XML is used as an example to explain the XPath syntax.

```
<order>
    <orderId>100</orderId>
    <shippingAddress>
        <zipCode>22322</zipCode>
    </shippingAddress>
</order>
```

To navigate in an XML file, XPath separates each element with a forward slash (/). Therefore, to get the text within the **<orderId>** element, use the following XPath expression:

```
/order/orderId/text()
```

To get the **zipCode** from the previous XML, use the following expression:

```
/order/orderId/shippingAddress/zipCode/text()
```

To get all XML contents where **zipCode** is *not* 23221, use the following expression:

```
/order/orderId/shippingAddress/[not(contains(zipCode,'23221'))]
```

To use the expression as a predicate in a Camel route, the **xpath** method parses the XPath expression and returns a boolean:

```
xpath("/order/orderId/shippingAddress/[not(contains(zipCode,'23221'))]")
```

Demonstration: Implementing the Message Filter Pattern

1. Import the filter-demo project into JBoss Developer Studio.
2. Update the **FileRouteBuilder** class to create the filter.

```
@Override  
public void configure() throws Exception {  
    from("file:orders/incoming?include=order.*xml")  
        .filter(xpath("/order/orderItems/orderItem/orderItemQty > 1"))  
        .to("file:orders/outgoing/?fileExist=Fail");  
}
```

3. Populate the **orders/incoming** folder by running the **setup-data.sh** shell script:

```
[student@workstation ~]$ cd ~/JB421/labs/filter-demo  
[student@workstation filter-demo]$ ./setup-data.sh  
...  
'Preparation complete!'
```

4. Inspect the order quantity from the incoming files:

```
[student@workstation filter-demo]$ grep orderItemQty orders/incoming/*
```

The route should discard all of the orders expect the **order-4.xml** order file.

5. Run the route by using the **camel:run** maven goal:

```
[student@workstation filter-demo]$ mvn clean camel:run
```

6. Open a new terminal window and inspect the **orders/outgoing** folder to verify that only **order-4.xml** order file is available:

```
[student@workstation ~]$ cd ~/JB421/labs/filter-demo  
[student@workstation filter-demo]$ ls orders/outgoing  
order-4.xml
```

7. Terminate the route using **Ctrl+C**.
8. Clean up. Close the project in JBoss Developer Studio to conserve memory.



References

A comprehensive list of expression languages supported by Camel

<http://camel.apache.org/languages.html>

A comprehensive list of patterns supported by Camel

<https://camel.apache.org/enterprise-integration-patterns.html>

Message filter pattern

<https://camel.apache.org/message-filter.html>

XPath

<https://www.w3.org/TR/xpath-3/>

W3Schools XPath tutorial

http://www.w3schools.com/xml/xpath_intro.asp

► Guided Exercise

Developing Routes Filtering Messages

In this exercise, you will receive multiple incoming files with XML orders, and discard those for ABC company using XPath and **filter** methods.

Outcomes

You should be able to complete a Camel route source using the Java DSL to read the incoming files and produce the output files filtering messages from the ABC Company.

Before You Begin

Run the following command to download the starter project and data files used by this exercise. Click **Applications** → **Favorites** → **Terminal** in the desktop menu and type:

```
[student@workstation ~]$ lab filter-messages setup
```

- 1. Import the filter-messages project into JBoss Developer Studio.
 - 1.1. Open JBoss Developer Studio (**Applications** → **Programming** → **JBoss Developer Studio**). JBoss Developer Studio requests a default workspace; keep the default configuration (**/home/student/workspace**) and click **Launch**.
 - 1.2. Import the Maven project by selecting the JBoss Developer Studio main menu **File** → **Import**.
 - 1.3. From the **Import** dialog box, select **Maven** → **Existing Maven Projects** and click **Next >**.
 - 1.4. Click **Browse** and choose **/home/student/JB421/labs/filter-messages** and click **Ok** to choose the Maven project. Click **Finish** to complete the import process in JBoss Developer Studio.
A new project named **filter-messages** will be listed in the **Project Explorer** view.
- 2. Create the route by updating the **FileRouteBuilder** class.
 - 2.1. Open the **FileRouteBuilder** class by expanding the **filter-messages** item in the **Project Explorer** tab in the left pane of JBoss Developer Studio, then click **filter-messages** → **src/main/java** → **com.redhat.training.jb421** to expand it. Double-click the **FileRouteBuilder.java** file.
 - 2.2. Update the route to process only messages that are not from the ABC Company.
Use the **xpath** method in the filter because every file to be processed is an XML file.
Use the following XPath expression to match the files whose **orderItemPublisherName** is ABC Company:
`/order/orderItems/
orderItem[not(contains(orderItemPublisherName, 'ABC Company'))]`

```
from("file:orders/incoming?include=order.*xml")
.filter(xpath("/order/orderItems/")
+"orderItem[not(contains(orderItemPublisherName, 'ABC Company'))]")
.to("file:orders/outgoing/?fileExist=Fail");
```

2.3. Save your changes to the file using **Ctrl+S**.

► 3. Populate the **orders/incoming** folder.

3.1. Go back to the terminal window and run the provided shell script to populate the **orders/incoming** folder:

```
[student@workstation ~]$ cd ~/JB421/labs/filter-messages
[student@workstation filter-messages]$ ./setup-data.sh
...
'Preparation complete!'
```

3.2. Inspect the incoming files:

```
[student@workstation filter-messages]$ ls orders/incoming
order-1.xml order-2.xml order-3.xml order-4.xml order-5.xml order-6.xml
```

There should be six **order-?.xml** files.

3.3. Inspect the incoming files from the ABC Company. They must be filtered by the route:

```
[student@workstation filter-messages]$ grep ABC orders/incoming/*
```

The **order-2.xml**, **order-4.xml**, and **order-6.xml** files are from ABC Company.

► 4. Test the route.

4.1. Run the route by using the **camel:run** Maven goal:

```
[student@workstation filter-messages]$ mvn clean camel:run
```

The following is the expected output of a successful execution:

```
...
[INFO] Building GE: Developoing routes filtering messages 1.0
...
[INFO] Using org.apache.camel.spring.Main to initiate a CamelContext
[INFO] Starting Camel ...
...
INFO Route: route1 started and consuming from: file://orders/incoming?
include=order.*xml
INFO Total 1 routes, of which 1 are started
INFO Apache Camel 2.21.0.fuse-000077-redhat-1 (CamelContext: jb421Context)
started in 0.254 seconds
...
```

- 4.2. Open a new terminal window and inspect the **orders/outgoing** folder to verify that order files from ABC Company are not available:

```
[student@workstation ~]$ cd ~/JB421/labs/filter-messages  
[student@workstation filter-messages]$ ls orders/outgoing  
order-1.xml order-3.xml order-5.xml
```

- 4.3. Terminate the route using **Ctrl+C**.

The following is the expected output when terminating the route:

```
INFO Received hang up - stopping the main instance.  
...  
INFO Route: route1 shutdown complete, was consuming from: file://orders/incoming?  
include=order.*xml  
INFO Graceful shutdown of 1 routes completed in 0 seconds  
...  
INFO Apache Camel 2.21.0.fuse-000077-redhat-1 (CamelContext: jb421Context) uptime  
2 minutes  
INFO Apache Camel 2.21.0.fuse-000077-redhat-1 (CamelContext: jb421Context) is  
shutdown in 0.044 seconds
```

- 5. Clean up: close the project in JBoss Developer Studio to conserve memory.

In the **Project Explorer** view, right-click **filter-messages** and click **Close Project**.

This concludes the guided exercise.

Routing Messages from JMS

Objectives

After completing this section, students should be able to implement a Camel route with a Content Based Router and provide basic error handling.

XML DSL Review

The Camel XML DSL (or Spring DSL) provides a way to define routes without using a programming language. The XML DSL is not as flexible as the Java DSL, but has the advantage of being supported by JBoss Developer Studio with a GUI editor.

To use the Spring DSL, declare a **camelContext** element, using the custom Camel Spring name space, inside a Spring Beans configuration file. Inside the **camelContext** element, declare one or more **route** elements starting with a **from** element and usually ending with a **to** element:

```
<camelContext id="myContext" xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="..."/>
        <!-- other DSL elements -->
        <to uri="..."/>
    </route>
</camelContext>
```

Most XML elements from the XML DSL do not require an **id** attribute, but this attribute is required by the JBoss Developer Studio route editor and will be added using generated values, if omitted.

The Content Based Router EIP in Camel

The *Content Based Router (CBR)* EIP allows routing messages to the correct destination based on the message contents.

The Camel CBR implementation is similar to a programming language construct: the **choice** DSL element contains multiple **when** DSL elements and optionally an **otherwise** DSL element:

```
<route>
    <from uri="schema:origin"/>
    <choice>
        <when>
            <!-- predicate1 -->
            <to uri="schema:destination1"/>
        </when>
        <when>
            <!-- predicate2 -->
            <to uri="schema:destination2"/>
        </when>
        <!-- other when elements -->
        <otherwise>
```

```
<!-- exception flow or destination -->
</otherwise>
</choice>
</route>
```

Each **when** DSL element requires a predicate that, when true, triggers sending the message to its destination. If the predicate is false, the route flow moves to the next **when** element. The predicate can use Simple EL, XPath expressions, or any other expression language supported by Camel.

The **otherwise** DSL element allows having a destination for messages that fail to match any of the **when** predicates. The **otherwise** DSL element usually starts an exception flow.

CBR Using the Java DSL

The general Content Based Router (CBR) template using the Java DSL is:

```
from("schema:origin")
    .choice()
        .when(predicate1)
            .to("schema:destination1")
        .when(predicate2)
            .to("schema:destination2")
        ...
        .otherwise()
            .to("schema:destinationN");
```

Using the XML DSL, predicates are nested elements, at the same level of the destination, for example:

```
<when>
    <xpath>xpath expression</xpath>
    <to uri="schema:destination"/>
<when>
```

When using the Java DSL, predicates are arguments to the **when** method, for example:

```
when(xpath("xpath expression"))
    .to("schema:destination")
```

This is an example of how the host language (Java or XML) affects the DSL syntax.

The **otherwise** call is optional, as its XML counterpart. But in most cases the **otherwise** call should exist at least to log the message, otherwise the message will be lost and an important business transaction might never be processed.

Introduction to Red Hat AMQ 7

Red Hat AMQ 7 is a lightweight, flexible, and reliable messaging platform that simplifies both the integration of application components as well as workload scaling. Based on the upstream projects Apache ActiveMQ and Apache Qpid, Red Hat AMQ integrates enterprise applications by providing a standards-based platform for application components and subsystems to communicate in a loosely coupled manner regardless of language or platform.

The AMQ Broker is a Message Oriented Middleware (MOM) component based on the ActiveMQ Artemis messaging implementation that provides queuing, message persistence, and manageability. The broker supports a variety of messaging styles such as point-to-point, publish-subscribe, store, and forward as well as multiple languages and platforms.

The main benefits of using a messaging system such as Red Hat AMQ include the fact that you can easily decouple the integrations between your systems such that there is little to no direct communication between programs. Additionally, messaging enables communications between systems to be asynchronous and event-driven. Messaging also supports additional features such as message priorities, message security, as well as data integrity and persistence.

Despite the similar characteristics this messaging system has with Camel, Camel provides more features, such as allowing integration with other components and using multiple enterprise integration patterns.

AMQ Queues with Camel

Using MOMs for integration purposes has been very common since the mainframe days because MOMs decouples applications from each other. The main MOM abstraction is the *queue*, which stores messages sent by a publisher until they are received by a consumer. The publisher and the consumer do not know about each other, and the publisher does not wait for a reply from the consumer. Guaranteeing message delivery is the MOM's responsibility.

The Java Message System API allows applications to connect to any MOM that provides a **JMSConnectionFactory**, in a similar way to connecting to relational databases using the JDBC API and a **DataSource**.

Camel provides a generic jms component and a specialized activemq component that share most implementation code and configuration options. Both use the JMS API to access the MOM.

The general URI syntax for the activemq component is as follows:

```
activemq:queue:queue_name
```

Using the Camel activemq component requires registering a **JmsConfiguration** with the Camel context. To do that using a Spring beans configuration file, declare an **activemq** bean with a **configuration** property.

```
<bean class="org.apache.activemq.camel.component.ActiveMQComponent" id="activemq">
    <property name="configuration" ref="jmsConfig"/>
</bean>
```

To create a **JmsConfiguration**, you need to register a JMS **JMSConnectionFactory** with the Camel context. To do that using a Spring beans configuration file, declare a **jmsConfig** bean with a **connectionFactory** property.

```
<bean class="org.apache.camel.component.jms.JmsConfiguration" id="jmsConfig">
    <property name="connectionFactory" ref="connectionFactory"/>
</bean>
```

To create a **JMSConnectionFactory**, you need to register a **ActiveMQConnectionFactory** with the Camel context. To do that using a Spring beans configuration file, declare a **connectionFactory** bean with a **brokerURL** property and a **trustAllPackages** property to trust all packages.

```
<bean class="org.apache.activemq.spring.ActiveMQConnectionFactory"
  id="connectionFactory">
  <property name="brokerURL" value="activemq_server_uri"/>
  <property name="trustAllPackages" value="true"/>
</bean>
```

The `activemq_server_uri` provides connection parameters and credentials to access an ActiveMQ broker.

Camel Message Redelivery

Connecting to network resources such as MOMs, FTP servers, and other middleware, involves the risk of experiencing failures. Camel deals with this by using a *redelivery policy* component that stores exchanges in memory and retries sending messages a number of times, usually with a delay between attempts.

A redelivery policy can be attached to the entire Camel context or to a specific route, by means of an exception handling configuration, or a more powerful error handing configuration.

The Camel redelivery policy is independent of the route endpoints and can also handle transient errors from EIPs in the middle of a route.

Configuring a *redelivery policy* is covered in detail later in the course.



Note

Some middleware products, especially MOMs, might also provide redelivery features and those are not tied to Camel's redelivery policy.

Demonstration: Routing Data with Content Based Routing

1. Import the **route-content-jms** project into JBoss Developer Studio.
2. Inspect the ActiveMQ configuration from the starter project.
In JBDS **Project Explorer** view, expand **src/main/resources**, **META-INF**, and **spring**. Double-click **bundle-context.xml**.
Click the **Source** tab to see that the Spring beans file contains a **bean** element for an embedded ActiveMQ broker and an empty **camelContext** element.
3. Create a route that consumes from XML files and routes messages to multiple JMS queues using the CBR pattern.
 - a. Back to the **bundle-context.xml** file, click the **Design** tab. In the **Palette**, expand the **Routing** category and drag a Route icon to anywhere in the canvas.
 - b. Add a File component as a consumer.
Expand the **Components** category in the **Palette** and drag a File icon to the **Route** box.
In **Properties** view, change the **File** box **Uri** property to **file:orders/incoming**.
 - c. Expand the **Transformation** category in the **Palette** and drag a Set Header icon over the **File** box.

Change the **Set Header** box properties as follows:

- **Expression Language:** xpath
 - **Expression:** /order/orderId/text()
 - **Header Name:** orderId
- d. Extract the vendor name to an exchange header.

Drag another Set Header icon, this time over the first **SetHeader** box.

Change the second **SetHeader** box properties as follows:

- **Expression Language:** xpath
 - **Expression:** /order/orderItems/orderItem/orderItemPublisherName/text()
 - **Header Name:** vendorName
- e. Add a CBR EIP to the route, based on the vendor string.

Expand the **Routing** category in the **Palette** and drag a Choice icon to the second **SetHeader** box.

Drag a When icon to the **Choice** three times, and also drag an Otherwise icon.

- f. Configure the CBR predicates.

Click the **Design** tab to continue, if you are still in the **Source** tab.

Click the first of the **When** boxes and change their properties as follows:

- **Expression Language:** simple
- **Expression:** \${header.vendorName} == 'ABC Company'

Do the same for the other two **When** boxes, but use the following Simple expressions:

- \${header.vendorName} == 'ORly'
- \${header.vendorName} == 'Namming'

- g. Configure logging for messages.

In the **Palette**, expand the **Components** category and drag a Log icon to the first **When** box. Do the same for the other two **When** boxes.

Change the **Log** box **Message** property of each **Log** box as follows:

- Delivering order \${header.orderId} to vendor ABC
- Delivering order \${header.orderId} to vendor ORly
- Delivering order \${header.orderId} to vendor Namming

- h. Configure logging for messages not handled.

In the **Palette**, expand the **Components** category and drag a Log icon to the **Otherwise** box.

Change the **Log** box **Message** property to:

**Failed to deliver order: \${header.orderId} to vendor:
\${header.vendorName}**

- i. Add the ActiveMQ endpoints.

Drag an ActiveMQ icon to the first **When** box. Do the same for the other two **When** boxes.

Change the **Uri** property of each **ActiveMQ** box as follows:

- **activemq:queue:abc**
- **activemq:queue:orly**
- **activemq:queue:namming**

Save your work. In JBoss Developer Studio main menu, click **File → Save**.

- j. Inspect the first route final source code.

```
<route id="_route1">
    <from id="_from1" uri="file:orders/incoming"/>
    <setHeader headerName="orderId" id="_setHeader1">
        <xpath>/order/orderId/text()</xpath>
    </setHeader>
    <setHeader headerName="vendorName" id="_setHeader2">
        <xpath>/order/orderItems/orderItem/orderItemPublisherName/text()</xpath>
    </setHeader>
    <choice id="_choice1">
        <when id="_when1">
            <simple>${header.vendorName} == 'ABC Company'</simple>
            <log id="_log2" message="Delivering order ${header.orderId} to vendor ABC."/>
        >
            <to id="_to1" uri="activemq:queue:abc"/>
        </when>
        <when id="_when2">
            <simple>${header.vendorName} == 'ORly'</simple>
            <log id="_log2" message="Delivering order ${header.orderId} to vendor ORly."/>
            <to id="_to2" uri="activemq:queue:orly"/>
        </when>
        <when id="_when3">
            <simple>${header.vendorName} == 'Namming'</simple>
            <log id="_log2" message="Delivering order ${header.orderId} to vendor Namming."/>
            <to id="_to3" uri="activemq:queue:namming"/>
        </when>
    <otherwise id="_otherwise1">
```

Chapter 2 | Creating Routes

```
<log id="_log1" message="Failed to deliver order: ${header.orderId} to  
vendor: ${header.vendorName}" />  
</otherwise>  
</choice>  
</route>
```

4. Populate the **orders/incoming** folder by running the **setup-data.sh** shell script:

```
[student@workstation ~]$ cd ~/JB421/labs/filter-demo  
[student@workstation filter-demo]$ ./setup-data.sh  
...  
'Preparation complete!'
```

5. Start the context using JBoss Developer Studio.

In the **Project Explorer** view, right-click **/src/main/resources/META-INF/spring/bundle-context.xml** and click **Run As → Local Camel Context (without tests)**.

The **Console** view shows log messages generated by **mvn camel:run** which is invoked by JBoss Developer Studio:

```
...  
[INFO] Building Demo: Routing data with content based routing 1.0  
...  
INFO Route: _route1 started and consuming from: file://orders/incoming  
...  
INFO Delivering order 3 to vendor Namming.  
...  
INFO Delivering order 1 to vendor ORly.  
...  
INFO Delivering order 5 to vendor ORly.  
...  
INFO Delivering order 2 to vendor ABC.  
...  
INFO Delivering order 6 to vendor ABC.  
...  
INFO Delivering order 4 to vendor ABC.
```

The order of these log messages might be different because each route is processed in parallel.

6. Terminate the context.

Click the Stop icon in the **Console** view.

7. Clean up: close the project in JBoss Developer Studio to conserve memory.



References

Camel and Spring integration

<http://camel.apache.org/spring.html>

Content Based Router (CBR) pattern

<http://camel.apache.org/content-based-router.html>

Camel ActiveMQ component

<http://camel.apache.org/activemq.html>

For more information, refer to JBoss Developer Studio Integration Stack product documentation at

<https://access.redhat.com/documentation/en/red-hat-jboss-developer-studio-integration-stack/>

► Guided Exercise

Developing Camel routes with JBoss Developer Studio

In this exercise, you will use JBoss Developer Studio to graphically create routes that consume orders from an XML file and route them to different JMS queues based on the vendor string.

Outcomes

You should be able to use the JBoss Developer Studio graphical Camel route editor to design a set of routes with XML DSL and with an embedded ActiveMQ instance for the producer endpoints.

Before You Begin

Run the following command to download the starter project and data files used by this exercise. Click **Applications** → **Favorites** → **Terminal** in the desktop menu and type:

```
[student@workstation ~]$ lab route-jbds setup
```

- ▶ 1. Import the route-jbds project into JBoss Developer Studio.
 - 1.1. Open JBoss Developer Studio (**Applications** → **Programming** → **JBoss Developer Studio**). JBoss Developer Studio requests a default workspace; keep the default configuration (**/home/student/workspace**) and click **Launch**.
 - 1.2. Import the Maven project by selecting the JBoss Developer Studio main menu **File** → **Import**.
 - 1.3. From the **Import** dialog box, select **Maven** → **Existing Maven Projects** and click **Next >**.
 - 1.4. Click **Browse** and choose **/home/student/JB421/labs/route-jbds** and click **Ok** to choose the Maven project. Click **Finish** to complete the import process in JBoss Developer Studio.

A new project named **route-jbds** is listed in the **Project Explorer** view.
- ▶ 2. Inspect the ActiveMQ configuration from the starter project.

In JBoss Developer Studio **Project Explorer** view, expand **src/main/resources** → **META-INF/spring**. Double-click **bundle-context.xml** file.

Click the **Source** tab to see that the Spring beans file contains **bean** elements for an embedded ActiveMQ broker and an empty **camelContext** element:

```
<bean class="org.apache.activemq.spring.ActiveMQConnectionFactory"
  id="connectionFactory">
  <property name="brokerURL" value="tcp://localhost:61616"/>
  <property name="trustAllPackages" value="true"/>
```

```
</bean>
<bean class="org.apache.camel.component.jms.JmsConfiguration" id="jmsConfig">
    <property name="connectionFactory" ref="ConnectionFactory"/>
</bean>
<bean class="org.apache.activemq.camel.component.ActiveMQComponent" id="activemq">
    <property name="configuration" ref="jmsConfig"/>
</bean>
<camelContext id="_camelContext1" xmlns="http://camel.apache.org/schema/spring"/>
```

- ▶ 3. Create a route that consumes from XML files and routes messages to multiple JMS queues using the content-based router (CBR) pattern.

- 3.1. Create a new route using the JBoss Developer Studio graphical Camel route editor.

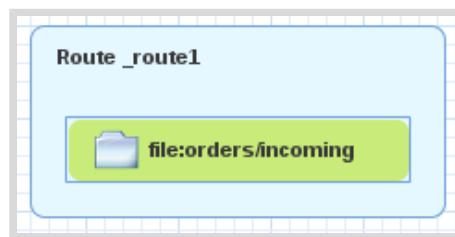
Back to the **bundle-context.xml** file, click the **Design** tab. In the **Palette** tab, expand the **Routing** category and drag a Route icon to anywhere in the canvas.

- 3.2. Add a file component as a consumer.

Expand the **Components** category in the **Palette** and drag a File icon to the **Route** box.

In **Properties** view, change the **File** box **Uri** property to **file:orders/incoming**.

At this point your diagram should look like:



- 3.3. Extract the order ID to an exchange header.

Expand the **Transformation** category in the **Palette** and drag a Set Header icon over the **File** box.

Change the **Set Header** box properties as follows:

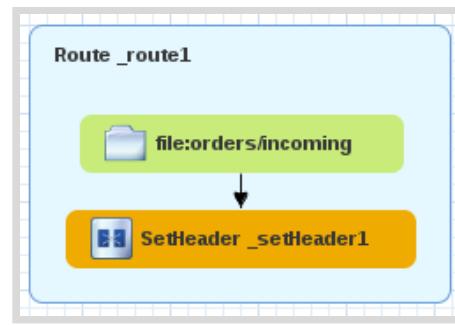
- **Expression Language:** xpath
- **Expression:** /order/orderId/text()
- **Header Name:** orderId



Important

There are two **Header Name** fields in the **Properties** view after you select the XPath language. You can use the first **Header Name** to retrieve your XPath expression from an exchange header. For this exercise, change only the last one, which controls the name of the header that the Camel route sets, and is required.

At this point, your diagram should look like:



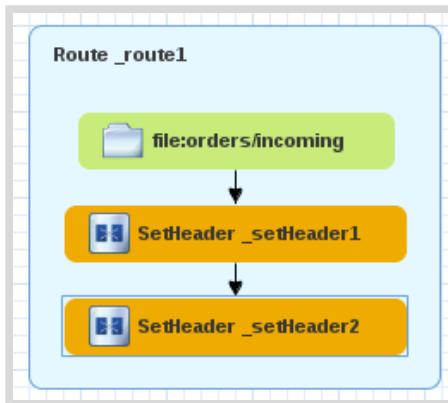
- 3.4. Extract the vendor name to an exchange header.

Drag another Set Header icon, this time over the first **SetHeader** box.

Change the second **SetHeader** box properties as follows:

- **Expression Language:** xpath
- **Expression:** /order/orderItems/orderItem/orderItemPublisherName/text()
- **Header Name:** vendorName

At this point, your diagram should look like:



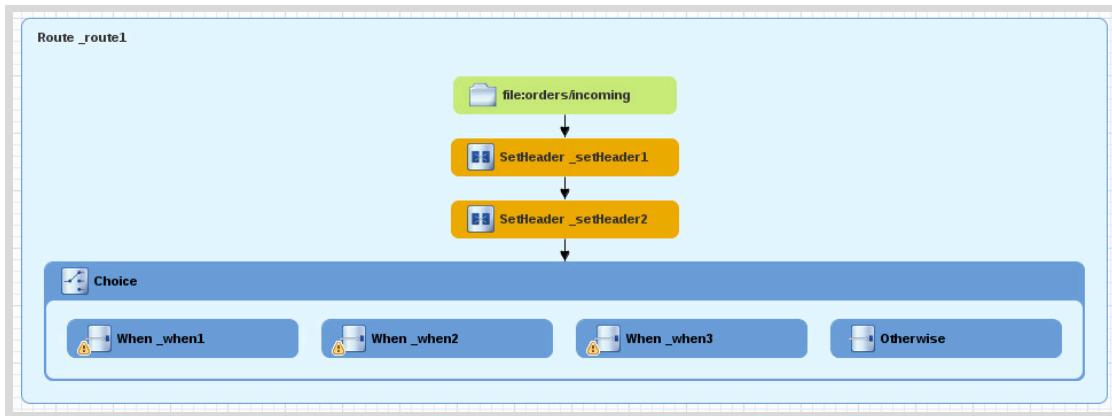
Save your work. In the JBoss Developer Studio main menu, click **File → Save**.

- 3.5. Add a CBR EIP to the route, based on the vendor string.

Expand the **Routing** category in the **Palette** and drag a Choice icon to the second **SetHeader** box.

Drag a When icon to the **Choice** box three times, and also drag an Otherwise icon.

Your diagram should now look like:



Click the **Source** tab to review the route XML source so far. It should look like:

```

<route id="_route1">
    <from id="_from1" uri="file:orders/incoming"/>
    <setHeader headerName="orderId" id="_setHeader1">
        <xpath>/order/orderId/text()</xpath>
    </setHeader>
    <setHeader headerName="vendorName" id="_setHeader2">
        <xpath>/order/orderItems/orderItem/orderItemPublisherName/text()</xpath>
    </setHeader>
    <choice id="_choice1">
        <when id="_when1"/>
        <when id="_when2"/>
        <when id="_when3"/>
        <otherwise id="_otherwise1"/>
    </choice>
</route>
  
```

If your boxes are not connected as in the figure, drag the arrows to change their order, and use the **Source** tab to double check the results. You can also make edits directly in the **Source** tab instead of using drag and drop in the **Design** tab.

The order of the **setHeader** elements makes no difference to this route as long as they come after the consumer (**from** element) and before the CBR EIP (**choice** element).

Save your work. In the JBoss Developer Studio main menu, Click **File → Save**.

3.6. Configure the CBR predicates.

If you are still in the **Source** tab, click the **Design** tab to continue.

Click the first of the **When** boxes and change their properties as follows:

- **Expression Language:** simple
- **Expression:** \${header.vendorName} == 'ABC Company'



Important

Type the vendor names exactly as presented.

Make sure you choose the **simple** language and *not xpath*, as in the previous steps.

Do the same for the other two **When** boxes, but use the following Simple expressions:

- `${header.vendorName} == 'ORly'`
- `${header.vendorName} == 'Namming'`



Important

You can copy and paste properties from one box and change the vendor name to the other to save yourself from typing.

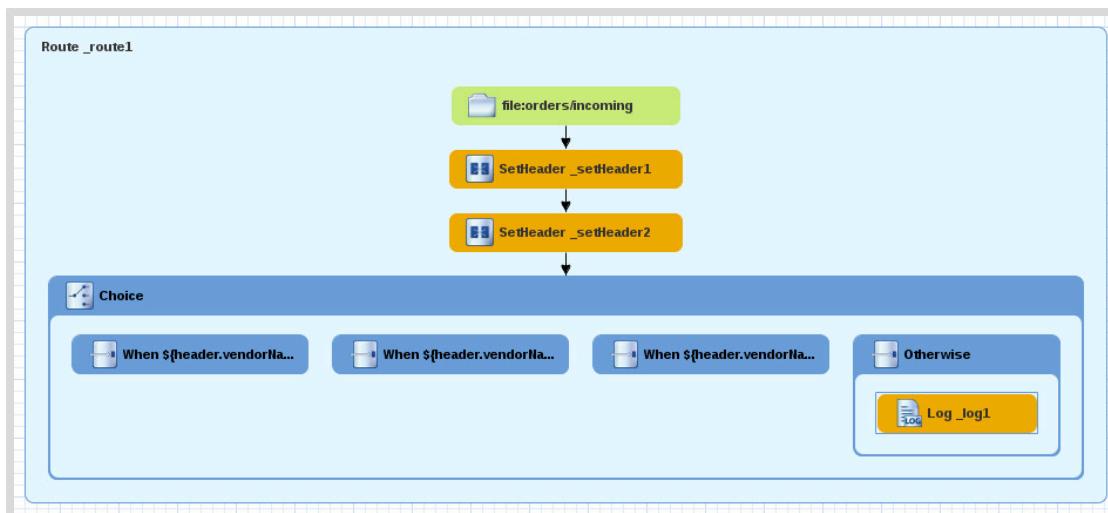
Another way to ease typing is, after changing the first **When** boxes properties, change to the **Source** tab and copy and paste the XML element changes to the other two **When** boxes.

Save your work. In the JBoss Developer Studio main menu, Click **File → Save**.

- 3.7. Configure logging for messages not handled.

In the **Palette**, expand the **Components** category and drag a Log icon to the **Otherwise** box.

Your diagram should now look like:



Change the **Log** box **Message** property to:

Failed to deliver order: \${header.orderId} to vendor: \${header.vendorName}

Save your work. In the JBoss Developer Studio main menu, click **File → Save**.

- 3.8. Add the ActiveMQ endpoints.

Drag an ActiveMQ icon to the first **When** box. Do the same for the other two **When** boxes.

Change the **Uri** property of each **ActiveMQ** box as follows:

- `activemq:queue:abc`
- `activemq:queue:orly`

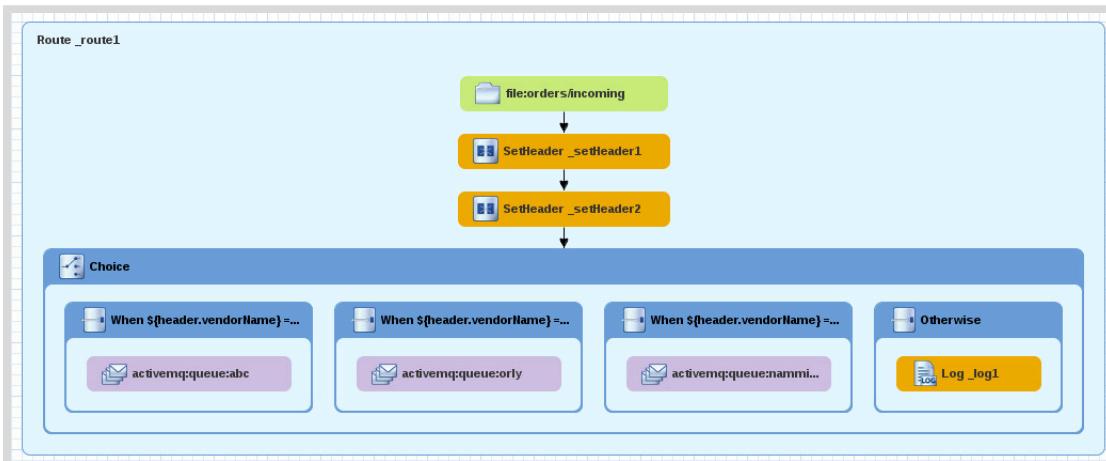
- **activemq:queue:namming**

Make sure the predicate of each **When** box and the queue name for its child **ActiveMQ** box match each other.

Save your work. In the JBoss Developer Studio main menu, click **File → Save**.

3.9. Inspect the first route final source code.

The first route is now completed and look like:



Click the **Source** tab. Inside the first route, the code for the **choice** element is now:

```
<choice id="_choice1">
    <when id="_when1">
        <simple>${header.vendorName} == 'ABC Company'</simple>
        <to id="_to1" uri="activemq:queue:abc"/>
    </when>
    <when id="_when2">
        <simple>${header.vendorName} == 'ORLy'</simple>
        <to id="_to2" uri="activemq:queue:orly"/>
    </when>
    <when id="_when3">
        <simple>${header.vendorName} == 'Namming'</simple>
        <to id="_to3" uri="activemq:queue:namming"/>
    </when>
    <otherwise id="_otherwise1">
        <log id="_log1"
            message="Failed to deliver order: ${header.orderId} to vendor:
            ${header.vendorName}"/>
    </otherwise>
</choice>
```

The order of **when** elements for each publisher makes no difference, so it is alright having first the ORLy company and ABC as the last one, for example. The **id** attribute values might also be different.

Make sure the vendor names in each predicate (the **simple** expressions) match the queue name in the corresponding consumer (the **to** element) inside the same **when** element. If not, make edits directly there in the XML DSL source.

Each XML element has to be in a single line. Do NOT add line breaks to make your source match the previous listing.

- 4. Create a second route that consumes messages from the **abc** queue and logs the order IDs and vendor names.

This route and the similar ones that will be created for each of the vendor queues are used to visually verify the route works as expected.

- 4.1. Add a second route to the same Camel Context.

Back to the **Design** tab, expand the **Routing** category in the **Palette** and drag a Route icon to any empty area in the canvas.

The new **Route** box is to the right of the first one. You might have to scroll horizontally to see it.

- 4.2. Add an ActiveMQ consumer to the second route.

Expand the **Components** category in the **Palette** and drag an **ActiveMQ** icon to the second route.

Change the **ActiveMQ** box **Uri** property to **activemq:queue:abc**.

- 4.3. Add a **SetHeader** DSL to the second route.

Expand the **Transformation** category in the **Palette** and drag a Set Header icon to the **ActiveMQ** box in second route.

Change the **SetHeader** box properties as follows:

- **Expression Language:** xpath
- **Expression:** /order/orderId/text()
- **Header Name:** orderId

- 4.4. Add a **Log** EIP to the second route.

Expand the **Components** category in the **Palette** and drag a Log icon to the SetHeader box in the second route.

Change the **Log** box **Message** property to: **Delivered order: \${header.orderId} to ABC Company**.

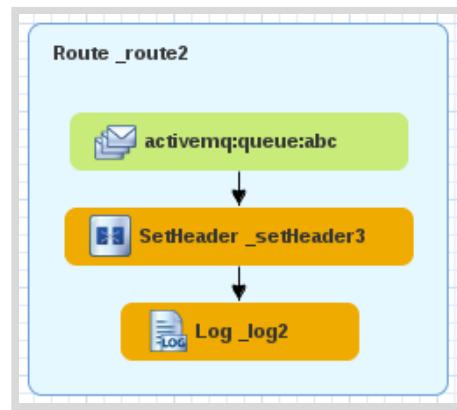
Save your work. In the JBoss Developer Studio main menu, Click **File → Save**.

- 4.5. Inspect the second route final source code.

Click the **Source** tab. The code for the second route is:

```
<route id="_route2">
    <from id="_from2" uri="activemq:queue:abc"/>
    <setHeader headerName="orderId" id="_setHeader3">
        <xpath>/order/orderId/text()</xpath>
    </setHeader>
    <log id="_log2" message="Delivered order: ${header.orderId} to ABC Company"/>
</route>
```

Click the **Design** tab. The second route now is:



- ▶ 5. Create two more routes that consume messages from queues **orly** and **namming**, and logs the order ids.
- 5.1. Follow steps Step 4.1 to Step 4.4 replacing the queue names in the consumer and the vendor name in the log message.



Note

To go a little faster, you can copy and paste properties from the boxes inside the second **Route** box to their corresponding ones in the other **Route** boxes.

An even faster approach is to cut and paste and edit the XML DSL source code in the **Source** tab for the second route to create the third and fourth routes, instead of doing multiple clicks and drags in the **Design** tab. If you do so, remember to change the **id** attributes to avoid collisions.

- 5.2. Inspect the source code for the third and fourth routes.

Click the **Source** tab. The code for the second and third routes is:

```

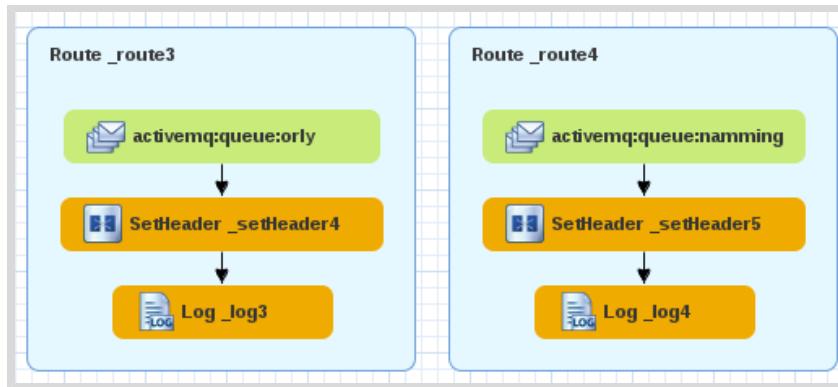
<route id="_route3">
    <from id="_from3" uri="activemq:queue:orly"/>
    <setHeader headerName="orderId" id="_setHeader4">
        <xpath>/order/orderId/text()</xpath>
    </setHeader>
    <log id="_log3" message="Delivered order: ${header.orderId} to ORly"/>
</route>
<route id="_route4">
    <from id="_from4" uri="activemq:queue:namming"/>
    <setHeader headerName="orderId" id="_setHeader5">
        <xpath>/order/orderId/text()</xpath>
    </setHeader>
    <log id="_log4" message="Delivered order: ${header.orderId} to Namming"/>
</route>

```

Remember the sequence numbers in each element **id** attribute are not important. They reflect the order in which you dragged the icons. You should observe the relative position and nesting of XML elements.

Verify that, for the second, third, and fourth routes, the queue name and the vendor name in the logging messages matches each other.

Click the **Design** tab. The second and third routes should look like:



You might have to scroll horizontally to see the third and fourth routes in the JBoss Developer Studio **Route Editor** canvas.

► 6. Start the Red Hat AMQ 7.0 broker.

Open a terminal window (**Applications** → **System Tools** → **Terminal**) and execute the following commands:

```
[student@workstation ~]$ cd ~/JB421/labs/jaxb-filter/broker1/bin
[student@workstation bin]$ ./artemis run
```

When the broker finishes booting up, the terminal contains the following message:

```
[org.apache.activemq.artemis] AMQ241001: HTTP Server started at http://
localhost:8161
[org.apache.activemq.artemis] AMQ241002: Artemis Jolokia REST API available at
http://localhost:8161/console/jolokia
[org.apache.activemq.artemis] AMQ241004: Artemis Console available at http://
localhost:8161/console
```

► 7. Manually test your routes using the provided sample data.

7.1. Run a script to prepare the input folder.

Make sure all changes to your routes are saved. In the JBoss Developer Studio main menu, Click **File** → **Save**.

Click **Applications** → **Favorites** → **Terminal** in the desktop menu and type:

```
[student@workstation ~]$ cd ~/JB421/labs/route-jbds
[student@workstation route-jbds]$ ./setup-data.sh
...
Preparation complete!
```

7.2. Start the context using JBoss Developer Studio.

In the **Project Explorer** view, right-click **/src/main/resources/META-INF/spring/bundle-context.xml** and click **Run As** → **Local Camel Context (without tests)**.

**Important**

Make sure you click the correct menu item. If you mistakenly click **Run As → Local Camel Context** the JUnit test in the starter project will consume all files in the input folder and you will have to run **setup-data.sh** again to see the routes working.

**Important**

If the build fails due to a missing dependency of **camel-groovy**, this dependency was added automatically by JBDS when you scrolled by Groovy in the list of possible expression languages. This dependency is not necessary for this project and you must remove it from your **pom.xml**, save the changes, and re-build to correct the build errors.

The **Console** view shows log messages generated by **mvn camel:run** command which is invoked by JBoss Developer Studio:

```
...
[INFO] Building GE: Developing Camel routes with JBDS 1.0
...
[INFO] Tests are skipped.
...
[INFO] Starting Camel ...
...
INFO Route: _route1 started and consuming from: Endpoint[file://orders/incoming]
...
INFO Connector vm://localhost started
INFO Route: _route2 started and consuming from: activemq://queue:abc
INFO Route: _route3 started and consuming from: activemq://queue:orly
INFO Route: _route4 started and consuming from: activemq://queue:namming
INFO Total 4 routes, of which 4 are started
INFO Apache Camel 2.21.0.fuse-000077-redhat-1 (CamelContext: _camelContext1)
started in 0.679 seconds
```

The first route should start consuming the files dropped by the **setup-data.sh** script, and the remaining routes show their logging messages:

```
...
INFO Delivered order: 1 to ORly
INFO Delivered order: 2 to ABC Company
INFO Delivered order: 3 to Namming
INFO Delivered order: 4 to ABC Company
INFO Delivered order: 5 to ORly
INFO Delivered order: 6 to ABC Company
```

The order of these log messages might be different because each route is processed in parallel.

7.3. Terminate the context.

Click the Terminate icon in the **Console** view.

If you made a mistake and need to go back to fix it, remember to run **setup-data.sh** before starting the routes again.

► 8. Debug your routes.

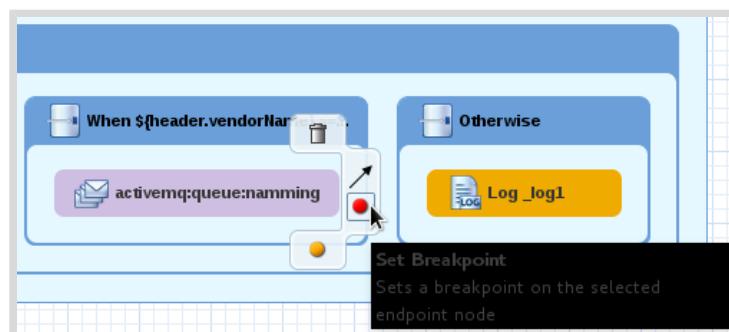
- 8.1. Add a breakpoint to the producer for the Namming vendor in the first route.

In the JBoss Developer Studio **Route** editor, click the **Design** tab.

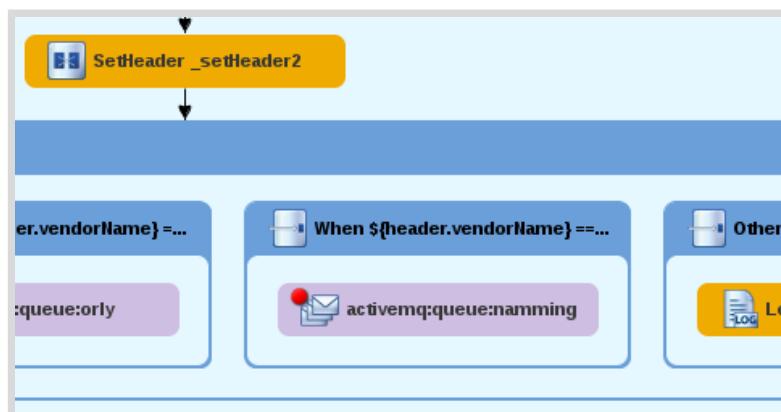
The first route is the one that has a **Choice** box. If you followed these steps without error, it is the route with ID **_route1**.

You might have to scroll horizontally the canvas to see the first route.

In the **bundle-context.xml** file **Design** tab, hover the mouse over the box labeled **activemq:camel:namming**, and click the red circle icon:



A red circle now marks the **ActiveMQ** box as having an active breakpoint.

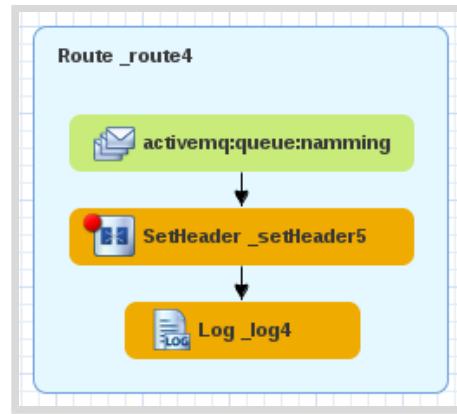


- 8.2. Add a breakpoint to the logger for the Namming vendor.

Use the previous step as a guide to add a breakpoint to the fourth route **SetHeader** box.

You might have to scroll horizontally the canvas to see the fourth route.

A red circle now marks the **SetHeader** box as having an active breakpoint.



- 8.3. Run the script to prepare the input folder:

Go back to the terminal and run `./setup-data.sh`.

- 8.4. Start a Camel Debugging session using JBoss Developer Studio.

In the **Project Explorer** view, right-click `/src/main/resources/META-INF/spring/bundle-context.xml` and click **Debug As → Local Camel Context (without tests)**.



Important

Make sure you click the correct menu item. If you mistakenly click **Debug As → Local Camel Context** the JUnit test in the starter project consumes all files in the input folder and you will have to run `setup-data.sh` again to have the routes stopping at the breakpoints.

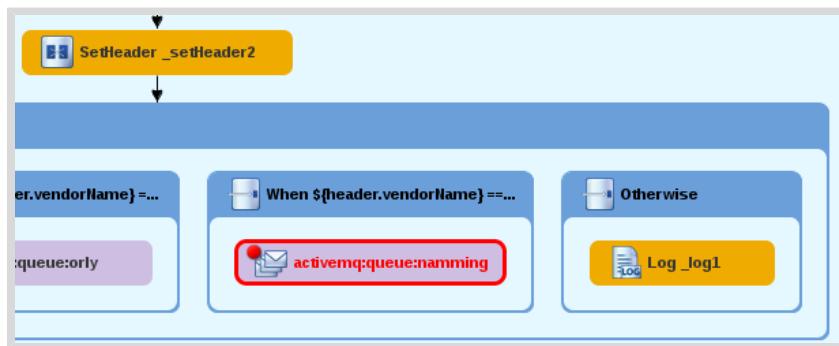
JBoss Developer Studio should ask to switch to the **Debug** perspective. Accept it.



Warning

The first time you try a Camel debugging session, JBoss Developer Studio might timeout and terminate the context. If this happens, just try again.

A red border should now highlight the producer. You might need to scroll the Route Editor canvas both horizontally and vertically to see the producer for the namming queue:



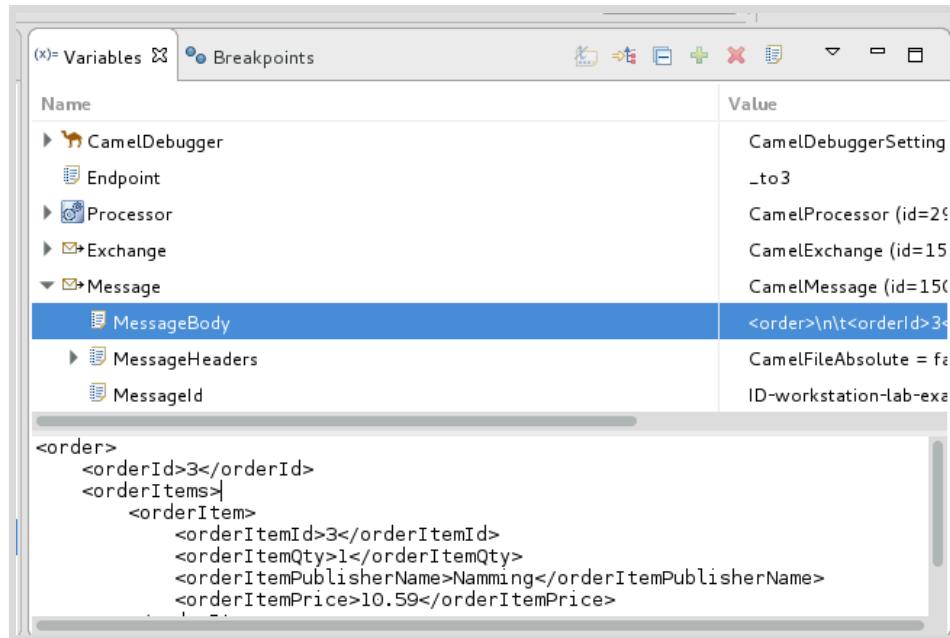
Chapter 2 | Creating Routes

The **Console** view should show logging messages about the breakpoints being injected into the routes, and that the first two messages were consumed without interruption from JBoss Developer Studio:

```
...
INFO | Apache Camel 2.21.0.fuse-000077-redhat-1 (CamelContext: _camelContext1) is
starting
...
INFO | Adding breakpoint _to3
INFO | Adding breakpoint _setHeader5
...
INFO | Enabling debugger
...
INFO | Total 4 routes, of which 4 are started.
INFO | Apache Camel 2.21.0.fuse-000077-redhat-1 (CamelContext: _camelContext1)
started in 1.978 seconds
...
INFO | Delivered order: 1 to ORLY
INFO | Delivered order: 2 to ABC Company
INFO | NodeBreakpoint at node _to3 is waiting to continue for exchangeId: ID-
workstation-lab-example-com-45077-1479997087468-0-8
INFO | Dump trace message from breakpoint _to3
INFO | Dump trace message from breakpoint _to3
```

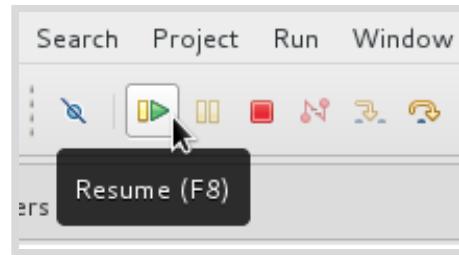
8.5. Inspect the current exchange.

In the **Variables** view, expand the **Message** variable and click **MessageBody**. Verify that the current message has the `<orderItemPublisherName>` element with the text **Namming**:



8.6. Let the route processing continue.

In the JBoss Developer Studio main menu, click the Resume icon.



The fourth route **SetHeader** box now has a red border.

You might need to scroll the **Route** editor canvas both horizontally and vertically to see the **SetHeader** box from the fourth route.



Warning

The **Console** view may show other messages being processed between each breakpoint.

Routes are processed in parallel by Camel, so there is no guarantee about the order in which the messages will be consumed and their log messages shown in the **Console** view.

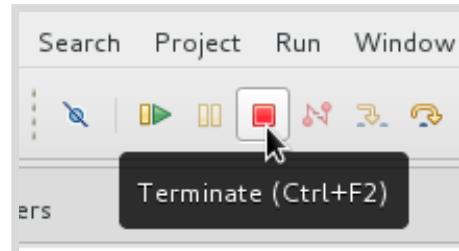
8.7. Finish processing files.

In JBoss Developer Studio main toolbar, click the Resume icon again.

The **Console** view now shows that all messages were processed.

8.8. Terminate the context.

In the main toolbar of JBoss Developer Studio, click the Terminate icon.



You can also find the same icon in the **Console** view toolbar.

Switch back to the **JBoss** perspective by clicking the icon in the top right toolbar.

► 9. Clean up.

Close the project in JBoss Developer Studio to conserve memory.

In the **Project Explorer** view, right-click the route-jbds project and click **Close Project**.

This concludes the guided exercise.

Configuring Exchange Headers

Objectives

After completing this section, students should be able to manipulate exchange headers in routes.

The Direct Component

The direct component connects a producer endpoint directly to a consumer endpoint in the same Camel context. This component is less of an integration feature and more of a way to break a route into small pieces.. The direct component is intended to makes it easier to handle large work flows in smaller chunks as well as reuse common routing logic for different consumers.

For example, you have multiple routes to consume data from many sources. All of them needs to convert the content in the JSON format and send to the database. Instead of replicating the code that converts and sends the content to the database for each route, you can create a route with the direct component that performs the conversion and sends the results to the database. Then, each route can produce a message to the new route using the direct component.

The direct URI syntax is:

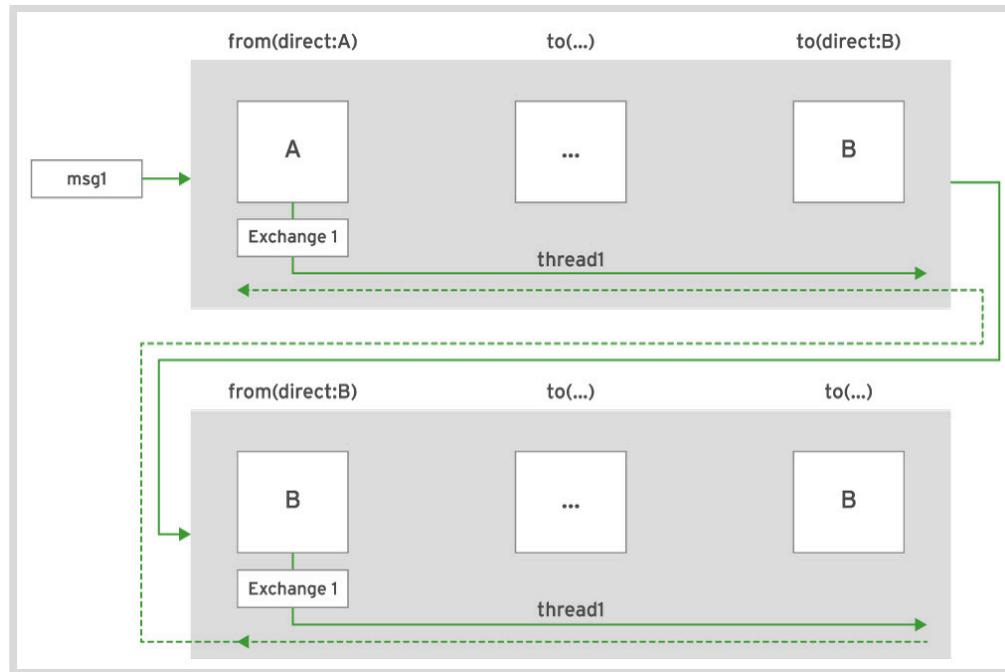
```
direct:name[?options]
```

The **name** is a handle defined by the developer to connect producers and consumers in the same Camel context, for example:

```
<route id="route1">
    <from uri="file:input"/>
    <to uri="direct:mylogic"/> ①
</route>
<route id="route2">
    <from uri=""direct:mylogic""/> ②
    <!-- do something here -->
    <to uri=""file:output"/>
</route>
```

- ① Produces a message exchange to a direct component called **myLogic**.
- ② Receives a message exchange from a direct component called **myLogic**.

The direct component provides synchronous invocation of any consumers when a producer sends a message exchange. A synchronous invocation means that the producer waits for the execution of the consumer. The following diagram depicts one Camel route sending a message to another using a direct endpoint named **B**. Notice that the same thread is used to execute both routes, this is why direct components are considered synchronous.



The following example demonstrates using the direct component with Java DSL:

```

from("activemq:queue:order.in")
    .to("bean:orderService?method=validate")
    .to("direct:processOrder");

from("direct:processOrder")
    .to("bean:orderService?method=process")
    .to("activemq:queue:order.out");

```

In this example, the Camel route receives messages on an ActiveMQ queue named **order.in**. The first route validates those messages using a bean, and then sends the messages to a second Camel route using a direct endpoint named **processOrder**. Camel then uses the same thread to execute the second route, using a bean to process the message before sending the result to a second ActiveMQ queue named **order.out**.

The direct component has a few limitations:

- Message passing is synchronous.
- There can be only a single consumer for each direct URI.
- Producer and consumer have to be part of the same Camel context.



Note

To overcome some of those limitations, Camel provides the seda and vm components which are presented later in this course.

The Processor Interface

The **org.apache.camel.Processor** interface provides a generic way to write custom logic to manipulate an exchange. This can be useful for creating new headers or modifying a message

body; for example, adding new content such as the date that the message was processed. You can use processors to implement the **Message Translator** and **Event Driven Consumer** EIPs.

Implementing the **Processor** interface requires implementing a single method named **process**:

```
void process(Exchange exchange) throws Exception
```

The exchange argument allows access to both the input and output messages and the parent Camel context. **Processor** implementation classes usually take advantage of other Camel features, such as data type converters and fluent expression builders.

To use a processor inside a route, insert the **process** element. For example:

```
<bean class="com.example.MyProcessor" id="myProcessor"/>
...
<route id="route1">
    <from uri="file:inputFolder"/>
    <process beanRef="myProcessor"/>
    <to uri="activemq:outputQueue"/>
</route>
```

The same example using Java DSL:

```
.from("file:inputFolder")
.process(new com.example.MyProcessor())
.to("activemq:outputQueue");
```

Before writing your own **Processor** implementation, check first whether there are components, EIP, or other Camel features that might provide the same result with less custom code, for example:

- The **transform** DSL method allows changing a message body using any expression language supported by Camel.
- The **setHeader** DSL method allows changing header values.
- The **bean** DSL method allows calling any Java bean method from inside a route.



Note

Camel is so powerful that there is a risk of embedding business logic inside a route, as a processor or by other means. Avoid doing that: keep your routes just about integration, and leave business logic to application components that are interconnected by Camel routes.

Compared to Java Beans, a Camel processor is preferred when there is a need to call Camel APIs from the custom Java code. A Java Bean is preferred when a transformation can reuse code that has no knowledge of Camel APIs.

Dynamic Endpoints

An endpoint URI may include a few dynamic components, such as simple expressions to provide an optional value. Sometimes this is not sufficient if an application requires the ability to route to destinations using different components based on message contents or other runtime data.

Chapter 2 | Creating Routes

In some application containers, Camel might cache and reuse endpoint instances in a way that the dynamic part is not re-evaluated. This could lead to messages being delivered to the wrong destination.

For those scenarios, Camel provides the **toD** DSL method, which stands for *dynamic to*, or *dynamic producer*. The URI is re-evaluated for each message sent and might even have different schema and components for each message.

By default, the **toD** method uses the Simple EL to resolve the dynamic endpoint URI. For example, to send a message to an endpoint defined by a header, you can:

```
<route>
  <from uri="direct:start"/>
  <toD uri="${header.foo}" />
</route>
```

In Java DSL:

```
from("direct:start")
  .toD("${header.foo}");
```

You can also prefix the URI with a value because by default the URI is evaluated using the Simple EL, which supports string concatenation.

```
<route>
  <from uri="direct:start"/>
  <toD uri="mock:${header.foo}" />
</route>
```

In Java DSL:

```
from("direct:start")
  .toD("mock:${header.foo}");
```

In the example above, an endpoint that has the `mock` scheme is computed, and then the value of header `foo` is appended.



Important

Use **toD** DSL method instead of **to** every time the consumer URI includes some part that needs to be calculated at runtime.

Using Java and XML DSLs in the Same Camel Context

The XML DSL syntax makes it very clear when a single Camel context contains multiple routes, for example:

```
<camelContext id="myContext">
  <route id="route1">
    <from uri="origin1"/>
    ...
    <to uri="destination1"/>
```

```

</route>
<route id="route2">
    <from uri="origin2"/>
    ...
    <to uri="destination2"/>
</route>
...
<camelContext>
```

The Java DSL also allows adding multiple **RouteBuilder** instances to the same context and allows defining multiple routes inside a single **RouteBuilder** instance:

```

public class MyRouteBuilder extends RouteBuilder {
    @Override
    public void configure() throws Exception {
        from("origin1")
        ...
        .to("destionation1");
        from("origin2")
        ...
        .to("destionation2");
    }
}
```

Each statement beginning with a **from** method defines another route. Each route can optionally be named using the **routeId** DSL method. For example:

```

public class MyRouteBuilder extends RouteBuilder {
    @Override
    public void configure() throws Exception {
        from("origin1")
            .routeId("route1")
        ...
        .to("destionation1");
        from("origin2")
            .routeId("route12")
        ...
        .to("destionation2");
    }
}
```

Camel DSL is flexible and allows many ways to pipe, branch, and aggregate messages inside a routing flow. Even multicasts are supported.

The syntax for using Java DSL routes with a Camel context defined inside a Spring beans configuration file was presented earlier in this course:

```

<bean class="com.example.MyRouteBuilder" id="myRouteBuilder"/>
...
<camelContext id="myContext">
    <routeBuilder ref="myRouteBuilder"/>
    ...
</camelContext>
```

Multiple routes defined using either the XML DSL or Java DSL can be part of the same Camel context defined in the same Spring beans configuration:

```
<bean class="com.example.MyRouteBuilder" id="myRouteBuilder"/>
<bean class="com.example.OtherRouteBuilder" id="otherRouteBuilder"/>
...
<camelContext id="myContext">
    <routeBuilder ref="myRouteBuilder"/>
    <routeBuilder ref="otherRouteBuilder"/>
    ...
    <route id="route1">
        <from uri="origin1"/>
        ...
        <to uri="destination1"/>
    </route>
    <route id="route2">
        <from uri="origin2"/>
        ...
        <to uri="destination2"/>
    </route>
    ...
</camelContext>
```

Each **RouteBuilder** class may define multiple routes. The order the routes are defined, inside either the **camelContext** element or the **RouteBuilder** classes. Each route gets its messages based on its consumer endpoint, the **from** DSL method, and the routes are run in parallel.

Custom and Pre-defined Message Headers

Most Camel components, when used as endpoints, provide metadata as message headers, and also behave differently based on received headers. Header names and their meaning depends on the component used. Carefully check the component and middleware documentation before writing routing logic that relies on a header value because they can change.

The following example uses a simple expression to access a header provided by the file component:

```
.from("file:incoming")
.log("${header.CamelFileName}")
...;
```

Camel provides the **setHeader** DSL method to set header values. For example:

```
...
.setHeader("CamelFileName", "destinationFile.txt")
.to("file:outgoing");
```

Headers defined by Camel components usually have the **Camel** prefix.

It is a common practice to create custom headers to store values for later use inside a route, or for debugging purposes. For example, you can create a header that contains an attribute from the message exchange and then route the message based on this attribute. Any header whose meaning is not known by a Camel component is ignored.



References

For more information, refer to the *Direct endpoints* chapter in the *Apache Camel Development Guide* at

https://access.redhat.com/documentation/en-us/red_hat_fuse/7.0/html-single/apache_camel_development_guide/#BasicPrinciples-MultipleInputs-DirectEndpoints

For more information, refer to the *Implementing a Processor* chapter in the *Apache Camel Development Guide* at

https://access.redhat.com/documentation/en-us/red_hat_fuse/7.0/html-single/apache_camel_development_guide/#Processors

► Guided Exercise

Developing a Camel Processor

In this exercise, you will receive multiple files containing orders and save them organized by the date of the order.

Outcomes

You should be able to complete the Camel route using the Java DSL.

Before You Begin

Run the following command to download the starter project and data files used by this exercise. Click **Applications** → **Favorites** → **Terminal** in the desktop menu and type:

```
[student@workstation ~]$ lab camel-processor setup
```

► 1. Import the camel-processor project into JBoss Developer Studio.

- 1.1. Open JBoss Developer Studio (**Applications** → **Programming** → **JBoss Developer Studio**). JBoss Developer Studio requests a default workspace; keep the default configuration (**/home/student/workspace**) and click **Launch**.
- 1.2. Import the Maven project by selecting the JBoss Developer Studio main menu **File** → **Import**.
- 1.3. From the **Import** dialog box, select **Maven** → **Existing Maven Projects** and click **Next >**.
- 1.4. Click **Browse** and choose **/home/student/JB421/labs/camel-processor** and click **OK** to choose the Maven project. Click **Finish** to complete the import process in JBoss Developer Studio.

A new project named **camel-processor** is listed in the **Project Explorer** view.

► 2. Update the **HeaderProcessor** class.

- 2.1. Open the **HeaderProcessor** class by expanding the **camel-processor** item in the **Project Explorer** tab in the left pane of JBoss Developer Studio, then click **camel-processor** → **src/main/java** → **com.redhat.training.jb421** to expand it. Double-click the **HeaderProcessor.java** file.
- 2.2. Update this class to implement the **org.apache.camel.Processor** interface. Camel provides the **Processor** interface to create a new processor.

This processor must add a new header containing the order date.

```
//TODO: Implements the Processor interface
public class HeaderProcessor implements Processor {
}
```

- 2.3. The interface requires the implementation of the **process** method. Add an empty method:

```
//TODO: Implement the process method
@Override
public void process(Exchange exchange) throws Exception {
}
```

- 2.4. Recover the text of the body of the exchange message:

```
@Override
public void process(Exchange exchange) throws Exception {
    String orderXml = exchange.getIn().getBody(String.class);
}
```

The **getIn** method retrieves the **Message** object from the exchange, and the **getBody** method retrieves the message body. The desired data type is passed as an argument (in this case **String.class**) and Camel uses its data conversion features if needed.

- 2.5. Extract the order date from the body:

```
@Override
public void process(Exchange exchange) throws Exception {
    String orderXml = exchange.getIn().getBody(String.class);
    String orderDateTime =
        XPathBuilder.xpath(XPATH_DATE).evaluate(exchange.getContext(), orderXml);
}
```

This step uses the **XPathBuilder** which is a Camel API that leverages the XPath XML parsing language to extract data from an XML document. XPath is covered in more detail later in the course.

- 2.6. Format the date in the **YYYYMMDD** syntax.

```
@Override
public void process(Exchange exchange) throws Exception {
    String orderXml = exchange.getIn().getBody(String.class);
    String orderDateTime =
        XPathBuilder.xpath(XPATH_DATE).evaluate(exchange.getContext(), orderXml);
    String formattedOrderDate = getFormattedDate(orderDateTime);
}
```

- 2.7. Create a new header called **orderDate** containing the formatted order date:

```
@Override  
public void process(Exchange exchange) throws Exception {  
    String orderXml = exchange.getIn().getBody(String.class);  
    String orderDateTime =  
        XPathBuilder.xpath(XPATH_DATE).evaluate(exchange.getContext(), orderXml);  
    String formattedOrderDate = getFormattedDate(orderDateTime);  
    exchange.getIn().setHeader("orderDate", formattedOrderDate);  
}
```

2.8. Save your changes to the file using **Ctrl+S**.

► 3. Create the route by updating the **FileRouteBuilder** class.

3.1. Open the **FileRouteBuilder** class by expanding the **camel-processor** item in the **Project Explorer** tab in the left pane of JBoss Developer Studio, then click **camel-processor** → **src/main/java** → **com.redhat.training.jb421** to expand it. Double-click the **FileRouteBuilder.java** file.

3.2. Add the **process** DSL method to the route definition to add a custom processor to the Camel route.

Create a new instance of the **HeaderProcessor** java class and use it as an argument to the **process** method:

```
public void configure() throws Exception {  
    from("file:orders/incoming?include=order.*xml")  
        .process(new HeaderProcessor())  
        .to("file:orders/outgoing");  
}
```

3.3. Update the file producer to separate the files according to the order date:

```
public void configure() throws Exception {  
    from("file:orders/incoming?include=order.*xml")  
        .process(new HeaderProcessor())  
        .to("file:orders/outgoing?fileName=${header.orderDate}/  
            ${header.CamelFileName}");  
}
```

3.4. Save your changes to the file using **Ctrl+S**.

► 4. Populate the **orders/incoming** folder.

4.1. Go back to the terminal window and run the provided shell script to populate the **orders/incoming** folder:

```
[student@workstation ~]$ cd ~/JB421/labs/camel-processor  
[student@workstation camel-processor]$ ./setup-data.sh  
...  
Preparation complete!
```

4.2. Inspect the incoming files:

```
[student@workstation camel-processor]$ ls orders/incoming  
order-1.xml order-2.xml order-3.xml order-4.xml order-5.xml order-6.xml
```

There should be six **order-?.xml** files.

► 5. Test the route.

- 5.1. Run the route by using the **camel:run** Maven goal:

```
[student@workstation camel-processor]$ mvn clean camel:run
```

The following is the expected output of a successful execution:

```
...  
[INFO] Building GE: Developing a Camel processor 1.0  
...  
[INFO] Using org.apache.camel.spring.Main to initiate a CamelContext  
[INFO] Starting Camel ...  
...  
INFO Route: route1 started and consuming from: file://orders/incoming?  
include=order.*xml  
INFO Total 1 routes, of which 1 are started  
INFO Apache Camel 2.21.0.fuse-000077-redhat-1 (CamelContext: jb421Context)  
started in 0.254 seconds  
...
```

- 5.2. Give the route a few moments to process the incoming files, and then terminate it using **Ctrl+C**.
- 5.3. Inspect the **orders/outgoing** folder to verify that Camel created two folders containing the order date:

```
[student@workstation camel-processor]$ ls orders/outgoing  
20161210 20161211
```

- 5.4. Inspect the **orders/outgoing/20161210** folder to verify orders from this date:

```
[student@workstation camel-processor]$ ls orders/outgoing/20161210  
order-1.xml order-2.xml order-3.xml order-4.xml
```

- 5.5. Inspect the **orders/outgoing/20161211** folder to verify orders from this date:

```
[student@workstation camel-processor]$ ls orders/outgoing/20161211  
order-5.xml order-6.xml
```

► 6. Clean up.

Close the project in JBoss Developer Studio to conserve memory.

In the **Project Explorer** view, right-click **camel-processor** and click **Close Project**.

This concludes the guided exercise.

► Lab

Creating Routes

Performance Checklist

In this lab, you will create a route that sends orders to different folders based on vendor name. Orders containing a test element are skipped.

Outcomes

You should be able to create a route that filters XML orders and sends the non-filtered orders to queues based on the vendor name, using the CBR pattern.

Before You Begin

Run the following command to download the starter project and data files used by this exercise. Click **Applications** → **Favorites** → **Terminal** in the desktop menu and type:

```
[student@workstation ~]$ lab create-routes setup
```

1. Import the create-routes project as an existing Maven project into JBoss Developer Studio. The project is available in the **/home/student/JB421/labs/create-routes** directory.



Important

The project imports with errors, but this is expected. You will resolve these errors in the subsequent steps of this exercise.

2. Add the **camel-core** and the **camel-spring** dependencies in the **pom.xml** file.
3. Orders that contain a **test** element cannot be processed by the route. Update the **com.redhat.training.jb421.TestProcessor** class to implement a Camel processor that checks for this element and sets a header called **skipOrder** with value **Y** when the **test** element is present in the order XML.

- For use by Antonio Martinez amartinezfasa amartinez@ahumada.cl Copyright © 2020 Red Hat, Inc.
4. Create the route by updating the `com.redhat.training.jb421.CBRRouteBuilder` class according to the following requirements:
 - Configure the endpoint to consume only XML files where the name starts with `order`. The files are available in the `orders/incoming` folder.
 - Create a header called `orderId`. The header must have the value provided in the `XPATH_ORDERID` constant.
 - Create a header called `vendorName`. The header must have the value provided in the `XPATH_VENDOR_NAME` constant.
 - Use the filter EIP to skip orders based on the `skipOrder` header existence.
 - Use the `TestProcessor` Camel processor in the route to add the `skipOrder` header if required.
 - Use the CBR pattern to send orders to the correct vendor folder. Complete the CBR EIP in the route with conditions for each vendor name. The vendor name was stored previously in the header. The vendor names are ABC Company, ORly, and Namming, and they should be stored in the `orders/outgoing/abc`, `orders/outgoing/orly`, and `orders/outgoing/namming` folders respectively.
 5. Test that the route works as expected. Open a terminal window and run the shell script available at `/home/student/JB421/labs/create-routes/setup-data.sh` to populate the `orders/incoming` folder. Run the route by invoking the Camel Maven plug-in. Shut down Camel after testing by pressing **Ctrl+C**.
 6. Grade the lab.

```
[student@workstation create-routes]$ lab create-routes grade
```

7. Clean up: close the project in JBoss Developer Studio to conserve memory.
In the **Project Explorer** view, right-click `create-routes` and click **Close Project**.
This concludes the lab.

► Solution

Creating Routes

Performance Checklist

In this lab, you will create a route that sends orders to different folders based on vendor name. Orders containing a test element are skipped.

Outcomes

You should be able to create a route that filters XML orders and sends the non-filtered orders to queues based on the vendor name, using the CBR pattern.

Before You Begin

Run the following command to download the starter project and data files used by this exercise. Click **Applications** → **Favorites** → **Terminal** in the desktop menu and type:

```
[student@workstation ~]$ lab create-routes setup
```

- Import the create-routes project as an existing Maven project into JBoss Developer Studio. The project is available in the **/home/student/JB421/labs/create-routes** directory.



Important

The project imports with errors, but this is expected. You will resolve these errors in the subsequent steps of this exercise.

1. Open JBoss Developer Studio (**Applications** → **Programming** → **JBoss Developer Studio**). JBoss Developer Studio requests a default workspace; keep the default configuration (**/home/student/workspace**) and click **Launch**.
 2. Import the Maven project by selecting the JBoss Developer Studio main menu **File** → **Import**.
 3. From the **Import** dialog box, select **Maven** → **Existing Maven Projects** and click **Next >**.
 4. Click **Browse** and choose **/home/student/JB421/labs/create-routes** and click **OK** to choose the Maven project. Click **Finish** to complete the import process in JBoss Developer Studio.
A new project named **create-routes** is listed in the **Project Explorer** view.
2. Add the **camel-core** and the **camel-spring** dependencies in the **pom.xml** file.
 - 2.1. Expand the **create-routes** item in the **Project Explorer** pane on the left, and then double-click the **pom.xml** file.
 - 2.2. Click the **pom.xml** tab at the bottom of the file to view the contents of the **pom.xml** file.

- 2.3. Add the **camel-core**, and **camel-spring** dependencies to the project.

```
<!-- TODO add camel dependencies -->
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-core</artifactId>
</dependency>

<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-spring</artifactId>
</dependency>
```

3. Orders that contain a **test** element cannot be processed by the route. Update the **com.redhat.training.jb421.TestProcessor** class to implement a Camel processor that checks for this element and sets a header called **skipOrder** with value **Y** when the **test** element is present in the order XML.

- 3.1. Open the **TestProcessor** class by expanding the **create-routes** item in the **Project Explorer** tab in the left pane of JBoss Developer Studio, then click **create-routes** → **src/main/java** → **com.redhat.training.jb421** to expand it. Double-click the **TestProcessor.java** file.

- 3.2. Update this class to implement the **org.apache.camel.Processor** interface.

```
//TODO: Implements the Processor interface
public class TestProcessor implements Processor {
}
```

- 3.3. Set the **skipOrder** header:

```
if (test.getLength() != 0) {
    log.info("Adding skipOrder header");
    //TODO set the skipOrder header
    exchange.getIn().setHeader("skipOrder", "Y");
}
```

4. Create the route by updating the **com.redhat.training.jb421.CBRRouteBuilder** class according to the following requirements:

- Configure the endpoint to consume only XML files where the name starts with **order**. The files are available in the **orders/incoming** folder.
- Create a header called **orderId**. The header must have the value provided in the **XPATH_ORDERID** constant.
- Create a header called **vendorName**. The header must have the value provided in the **XPATH_VENDOR_NAME** constant.
- Use the filter EIP to skip orders based on the **skipOrder** header existence.
- Use the **TestProcessor** Camel processor in the route to add the **skipOrder** header if required.

- Use the CBR pattern to send orders to the correct vendor folder. Complete the CBR EIP in the route with conditions for each vendor name. The vendor name was stored previously in the header. The vendor names are ABC Company, ORly, and Namming, and they should be stored in the **orders/outgoing/abc**, **orders/outgoing/orly**, and **orders/outgoing/namming** folders respectively.
- 4.1. Open the **CBRRouteBuilder** class by expanding the **create-routes** item in the **Project Explorer** tab in the left pane of JBoss Developer Studio, then click **create-routes** → **src/main/java** → **com.redhat.training.jb421** to expand it. Double-click the **CBRRouteBuilder.java** file.
 - 4.2. Update this class to extend the **org.apache.camel.builder.RouteBuilder** superclass. Camel provides the **RouteBuilder** class to enable a new route.

```
//TODO: Enable the route by extending the RouteBuilder superclass
public class CBRRouteBuilder extends RouteBuilder {  
}
```

- 4.3. The superclass requires the implementation of the **configure** method. Add an empty method:

```
//TODO Implement the configure method
@Override
public void configure() throws Exception {  
}
```

- 4.4. Add a file consumer to the route using the file component.

Configure the endpoint to consume from the **orders/incoming** directory and use the **include** option to configure the endpoint to consume only XML files with the name that starts with **order**.

```
public void configure() throws Exception {
    from("file:orders/incoming?include=order.*xml")
}
```

- 4.5. Create the **orderId** header with the value obtained from the XML using the **XPATH_ORDERID** constant.

```
public void configure() throws Exception {
    from("file:orders/incoming?include=order.*xml")
        .setHeader("orderId", xpath(XPATH_ORDERID))
}
```

- 4.6. Create the **vendorName** header with the value obtained from the XML using the **XPATH_VENDOR_NAME** constant.

```
public void configure() throws Exception {  
    from("file:orders/incoming?include=order.*xml")  
    .setHeader("orderId", xpath(XPATH_ORDERID))  
    .setHeader("vendorName", xpath(XPATH_VENDOR_NAME))  
}
```

4.7. Add a call to the **TestProcessor** Camel processor in the route.

```
public void configure() throws Exception {  
    from("file:orders/incoming?include=order.*xml")  
    .setHeader("orderId", xpath(XPATH_ORDERID))  
    .setHeader("vendorName", xpath(XPATH_VENDOR_NAME))  
    .process(new TestProcessor())  
}
```

4.8. Add the filter EIP to the route, to filter based on the **skipOrder** header.

```
public void configure() throws Exception {  
    from("file:orders/incoming?include=order.*xml")  
    .setHeader("orderId", xpath(XPATH_ORDERID))  
    .setHeader("vendorName", xpath(XPATH_VENDOR_NAME))  
    .process(new TestProcessor())  
    .filter(simple("${header.skipOrder} == null"))  
}
```

4.9. Use the CBR pattern to send orders to the correct folder.

There are three different vendor names and corresponding folders. Keep the code to log orders whose vendor does not match any of the known vendors.

Complete the CBR EIP in the route with conditions for each vendor name. The vendor name was stored previously in the header.

```
public void configure() throws Exception {  
    from("file:orders/incoming?include=order.*xml")  
    .setHeader("orderId", xpath(XPATH_ORDERID))  
    .setHeader("vendorName", xpath(XPATH_VENDOR_NAME))  
    .process(new TestProcessor())  
    .filter(simple("${header.skipOrder} == null"))  
    .choice()  
        .when(simple("${header.vendorName} == 'ABC Company'"))  
            .log("sending order ${header.orderId} to folder abc")  
        .to("file:orders/outgoing/abc")  
        .when(simple("${header.vendorName} == 'ORly'"))  
            .log("sending order ${header.orderId} to folder orly")  
            .to("file:orders/outgoing/orly")  
        .when(simple("${header.vendorName} == 'Namming'"))  
            .log("sending order ${header.orderId} to folder namming")  
            .to("file:orders/outgoing/namming")  
        .otherwise()  
            .log("Unknown vendor");  
}
```

5. Test that the route works as expected. Open a terminal window and run the shell script available at `/home/student/JB421/labs/create-routes/setup-data.sh` to populate the **orders/incoming** folder. Run the route by invoking the Camel Maven plug-in. Shut down Camel after testing by pressing **Ctrl+C**.

- 5.1. Run the **setup-data.sh** script to populate the **orders/incoming** folder.

```
[student@workstation ~]$ cd ~/JB421/labs/create-routes  
[student@workstation create-routes]$ ./setup-data.sh  
...  
'Preparation complete!'
```

- 5.2. Inspect the incoming files:

```
[student@workstation create-routes]$ ls orders/incoming  
noop-1.xml order-1.xml order-2.xml order-3.xml order-4.xml order-5.xml  
order-6.xml
```

- 5.3. Run the route by using the **camel:run** Maven goal:

```
[student@workstation create-routes]$ mvn clean camel:run  
...  
[INFO] Building Lab: Creating Routes 1.0  
...  
INFO sending order 5 to folder orly  
...  
INFO Adding skipOrder header  
...  
INFO sending order 6 to folder abc  
...  
INFO sending order 3 to folder namming  
...  
INFO sending order 4 to folder abc  
...  
INFO sending order 1 to folder orly
```

- 5.4. Terminate the route using **Ctrl+C**.

- 5.5. Inspect the **orders/outgoing** folder to verify that order files are in the correct vendor folder. The **order-2.xml** file must not be available since it contains the **test** node.

```
[student@workstation create-routes]$ tree orders/outgoing  
orders/outgoing/  
└── abc  
    ├── order-4.xml  
    └── order-6.xml  
└── namming  
    └── order-3.xml  
└── orly  
    ├── order-1.xml  
    └── order-5.xml
```

6. Grade the lab.

```
[student@workstation create-routes]$ lab create-routes grade
```

7. Clean up: close the project in JBoss Developer Studio to conserve memory.
In the **Project Explorer** view, right-click **create-routes** and click **Close Project**.
This concludes the lab.

Summary

In this chapter, you learned:

- A route describes the path of a message from one endpoint (the origin) to another endpoint (the destination).
- To implement a Camel route, the route must be attached to a **CamelContext** instance.
- Messages in routes that use the **InOnly** pattern are a one-way communication.
- Messages in routes using the **InOut** pattern are referred to as request-reply messages.
- Camel provides two logging features: the log component, and the log EIP.
- Simple expressions are delimited by a dollar sign followed by braces (\${}).

Chapter 3

Transforming Data

Goal

Convert messages between data formats using implicit and explicit transformation.

Objectives

- Invoke data transformation automatically and explicitly using a variety of different techniques.
- Transform messages using additional data formats, custom type converters, and the Message Translator pattern.
- Merge multiple messages using the Aggregator pattern and customizing the output to a traditional data format.
- Access a database with JDBC and JPA.

Sections

- Transforming Data Automatically and Explicitly (and Guided Exercise)
- Transforming Data with the Message Translator Pattern (and Guided Exercise)
- Combining Messages Through Aggregation (and Guided Exercise)
- Accessing Databases with Camel Routes (and Guided Exercise)

Lab

Transforming Data

Transforming Data Automatically and Explicitly

Objective

After completing this section, students should be able to invoke data transformation automatically and explicitly using a variety of different techniques.

Transforming Data in a Camel Route

One of the most common problems when integrating disparate systems is that those systems do not use a consistent data format. For example, you might receive new product information from a vendor in CSV format, but your production information system only accepts JSON data. This difference in data format means you need to transform the data from one system before another system can correctly process it. For this reason, Camel is designed to handle these differences by providing support for dozens of different data formats, including the ability to transform data from one format to another. The data formats that Camel supports include JSON, XML, CSV, flat files, EDI, and many others.

Marshaling Data

Marshaling is the process where Camel converts the message payload from a memory-based format (for example, a Java object) to a data format suitable for storage or transmission (XML or JSON, for example). To perform marshaling, Camel uses the `marshal` method, which requires a `DataFormat` object as a parameter.

Data Formats

Data Formats in Camel are the various forms that your data can be represented, either in binary or text. The data formats that Camel supports include standard JVM serialization, XML, JSON, CSV data, and others. Each data format has a class you need to instantiate and optionally configure before you can use it in a route. For example, the `JaxbDataFormat` class allows you to specify the package where your XML model classes are present. Once you instantiate the `JaxbDataFormat` class, it uses your model classes, complete with JAXB annotations as the blueprint for how to marshal and unmarshal your XML data to your Java model classes. Additionally, data format classes support a number of custom options, such as "pretty print", or other configurations that affect the behavior of marshaling or unmarshaling. These options allow you to have more control over the marshal and unmarshal behavior such as how to handle missing elements, what type of encoding to use, namespace prefixing, and more.

Unmarshaling Data

Unmarshaling is the opposite process of marshaling, where Camel converts the message payload from a data format suitable for transmission such as XML or JSON to a memory-based format, typically a Java object. To perform unmarshaling explicitly in a Camel route, use the `unmarshal` method in the Java DSL. Similar to the marshaling, the method requires a `DataFormat` object as a parameter.

Transforming XML Data using JAXB

There are multiple options for working with XML data in Camel. JAXB is a very popular XML framework and is the XML marshaling library focused on throughout this course. To use JAXB with Camel, include the **camel-jaxb** library as a dependency of your project:

```
<dependency>
<groupId>org.apache.camel</groupId>
<artifactId>camel-jaxb</artifactId>
</dependency>
```

The **camel-jaxb** library also provides the JAXB annotations that you need to annotate your model class. These annotations are used to tell JAXB how to unmarshal the XML data into the Java objects correctly. JAXB matches fields on the model class to elements and attributes contained in the XML data using the information provided by the JAXB annotations.

Given the following XML content:

```
<order id="10" description="N2PENCIL" value="1.5" tax="0.15"/>
```

The following JAXB model class contains the necessary annotations to marshal the XML to a Java object:

```
package com.redhat.training;

import java.io.Serializable;

import javax.xml.bind.annotation.*;

@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class Order implements Serializable{❶

    private static final long serialVersionUID = 5851038813219503043L;
    @XmlAttribute
    private String id;
    @XmlAttribute
    private String description;
    @XmlAttribute
    private double value;
    @XmlAttribute
    private double tax;
}
```

- ❶ Note that this model class implements the **java.io.Serializable** interface which is required by JAXB to execute the marshaling and unmarshaling process.

In conjunction with the **camel-jaxb** library, you must also include JAXB annotations to instruct Camel which XML fields map to which properties in your Java model classes. In the previous example, each field in the Java class represents an attribute on the root **Order** element. By default, if no alternate name is specified in the JAXB annotation parameters, the marshaller uses the name of the field directly to map the XML data.

**Note**

JAXB annotations are beyond the scope of this course. You can find more documentation in the JAXB user guide.

In the following sample route, an external system places XML data in the body of a message and then sends the message to an ActiveMQ queue named **itemInput**. The Camel route consumes the messages, and JAXB immediately unmarshals the XML data and replaces the contents of the exchange body with the corresponding Java object. The route then sends the Java object data in a message to an ActiveMQ queue named **itemOutput**.

For an XML-based route, declare the JAXB **dataFormat** and then use it to unmarshal XML data as shown in the following example:

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
    <dataFormats>
        <jaxb id="jaxb" contextPath="com.redhat.training"/> ①
    </dataFormats>
    <route>
        <from uri="activemq:queue:itemInput"/>
        <unmarshal ref="jaxb"/> ②
        <to uri="activemq:queue:itemOutput"/>
    </route>
</camelContext>
```

- ① The **contextPath** parameter refers to which package JAXB should scan for model classes that are annotated with JAXB annotations.
- ② Convert the XML data in the body of the exchange into an instance of the corresponding model class and replace the exchange body with that model class instance.

Similarly, the following example demonstrates marshaling Java objects into XML using the Java DSL:

```
from("activemq:queue:itemInput")
    .marshal().jaxb()
    .to("activemq:queue:itemOutput");
```

Notice that in the previous example Java DSL route, no JAXB data format is instantiated. This is possible because **camel-jaxb** automatically finds all classes that contain JAXB annotations if no context path is specified, and uses those annotations to unmarshal the XML data.

Marshaling and Unmarshaling JSON Data in a Camel Route with Jackson

Similar to XML, Camel offers multiple libraries for working with JSON data. Like JAXB, Jackson is a very popular framework for working with JSON data in Java and this course focuses on this approach for working with JSON data. To use Jackson with Camel, you need to include the **camel-jackson** library as a dependency of your project.

```
<dependency>
<groupId>org.apache.camel</groupId>
<artifactId>camel-jackson</artifactId>
</dependency>
```

Similar to JAXB, Jackson also provides a set of annotations you use to control the mapping of JSON data into your model classes.

Given the following JSON data:

```
{
    "_id": "1",
    "value": 5.00,
    "tax": 0.50,
    "description": "Sample text",
    ...}
```

The following example is a Jackson-annotated model class which you can use to marshal the JSON data:

```
package com.redhat.training;
...
public class Order implements Serializable{

    private static final long serialVersionUID = 5851038813219503043L;
    @JsonProperty("_id") ①
    private String id;
    @JsonIgnore ②
    private String description;
    @JsonProperty
    private double value;
    @JsonProperty
    private double tax;
}
```

- ① Use the **@JsonProperty** annotation to explicitly declare a field in the model class, and optionally include a name for Jackson to use when marshaling and unmarshaling JSON data for that property.
- ② Use the **@JsonIgnore** annotation to make Jackson ignore a field entirely when marshaling an instance of the model class into JSON data.

For an XML-based Camel route configuration, declare a Jackson **dataFormat** and then unmarshal JSON data, as in this example:

```
<camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
    <dataFormats>
        <json id="jackson" prettyPrint="true" library="Jackson"/>
    </dataFormats>
    <route>
        <from uri="file:inbox"/>
        <unmarshal ref="jackson"/>
```

```

<to uri="activemq:queue:itemInput"/>
</route>
</camelContext>

```

In the previous example route, JSON files are consumed from the **inbox** directory, and are unmarshaled into Java objects. Those Java objects are then sent to the **itemInput** queue.

The following is a Java DSL example of using Jackson to marshal JSON data before writing the JSON data to a file in the **outbox** directory:

```

from("queue:activemq:queue:itemInput")
    .marshal().json(JsonLibrary.Jackson)
    .to("file:outbox")

```

Transforming Directly Between XML and JSON Data Using the camel-xmljson Module

As discussed previously, Camel supports data formats to perform both XML and JSON-related conversions. However, both of these transformations require a Java model class object either as an input (marshaling) or they produce a Java object as output (unmarshaling). The **camel-xmljson** data format provides the capability to convert data from XML to JSON directly and vice versa without needing to use intermediate Java objects. Directly transforming XML to JSON is preferred as there is significant performance overhead involved in doing extra transformations.

When you use the **camel-xmljson** module, the terminology "marshaling" and "unmarshaling" are not as obvious since there is no Java objects involved. To resolve this, the module defines XML as the high-level format or the equivalent of what your Java model classes typically represent, and JSON as the low-level format more suitable for transmission or storage. This designation is mostly arbitrary for the purpose of defining the marshal and unmarshal terms.

This implies that the terms marshal and unmarshal are defined as follows:

marshaling

Converting from XML to JSON

unmarshaling

Converting from JSON to XML

To use the **XmlJsonDataFormat** class in your Camel routes you need to add the following dependencies to your POM file:

```

<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-xmljson</artifactId>
</dependency>
<dependency>
    <groupId>xom</groupId>
    <artifactId>xom</artifactId>
    <version>1.2.5</version>
</dependency>

```

**Note**

The XOM library cannot be included by default due to an incompatible license with Apache Software Foundation, so you need to add this dependency manually for the **camel-xmljson** module to function.

The following example Camel route includes the use of the **XmlJsonDataFormat**:

```
XmlJsonDataFormat xmlJsonFormat = new XmlJsonDataFormat();

// From XML to JSON
from("direct:marshal")
    .marshal(xmlJsonFormat)
    .to("direct:json");

// From JSON to XML
from("direct:unmarshal")
    .unmarshal(xmlJsonFormat)
    .to("direct:xml");
```

Implementing the Wire Tap Pattern in a Camel Route

The wire tap EIP is an integration pattern that creates a duplicate copy of each message that the route processes and then forwards the duplicate message to a second destination without interrupting the flow from the source to the original destination. The following diagram illustrates this pattern:

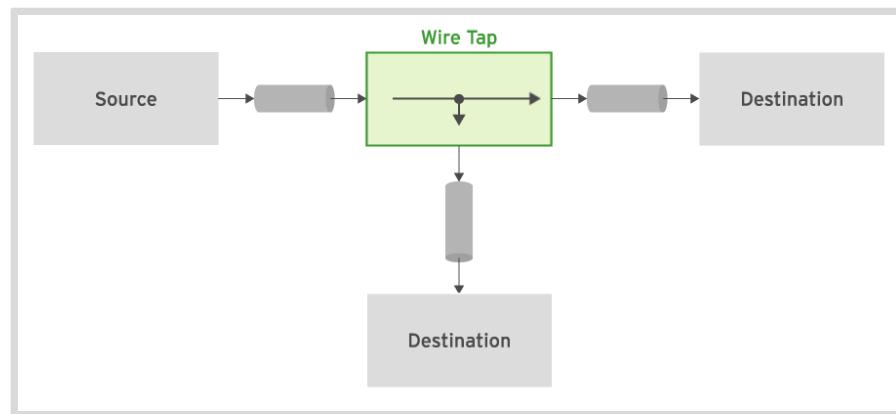


Figure 3.1: The Wire Tap EIP

This functionality is most useful for inspecting messages as they travel along a Camel route without impacting the route execution itself. Often Camel developers use wire taps to debug routes by checking the message content matches expectations mid-route. For instance, you could use a wire tap to write message body content to a file mid-route, or notify an external system with real-time updates asynchronously.

To implement this pattern, Camel provides the **wireTap** component, which you can use in your routes to duplicate message exchanges and send a copy of the exchange to a second destination. The following Camel route demonstrates the use of this component:

```
from("activemq:queue:orders.in")
    .wireTap("file:backup")
    .to("direct:start");
```

In this example, a copy of every exchange received from the **orders.in** queue is written to a file in the **backup** folder using a **wireTap**.



Note

The **wireTap** component has numerous advanced options, which you can use to modify the copy of the exchange that Camel sends to the tapped route. For more information on these and other advanced options, see the wire tap documentation [<http://camel.apache.org/wire-tap.html>].

Introducing Mock Endpoints in Camel Routes

During the development of a Camel route, it is often necessary to create a mock destination to represent and behave like an external system with which you have not yet integrated. You can use a mock as a place holder any time you are integrating with other development teams in parallel, and real integration points are not ready.

To use a mock endpoint in a route definition, use the URI prefix **mock:**, as the following Spring DSL example demonstrates:

```
<route>
    <from uri="direct:inventoryUpdate"/>
    <to uri="log:com.mycompany.inventory?level=DEBUG"/>
    <to uri="mock:fulfillmentSystem"/>
</route>
```

In this example, the mock component is providing a placeholder for an integration with an external fulfillment system that has not been implemented yet.

In addition, mocks can be used when testing route functionality. The mock component provides a powerful declarative testing mechanism, which is similar to jMock or Mockito testing frameworks. Mock endpoints allow you to define expectations for any endpoint before running your unit tests. When the test executes, the Camel route processes messages that are then sent to one or more endpoints. By using a mock you can then assert the expectations defined by your test case were met to ensure the system worked as expected.

This allows you to test various things, for example:

- The correct number of messages are received by each endpoint.
- The correct payloads are received, in the right order.
- The order that the messages arrived at an endpoint using some **Expression** to create an order testing function.
- The messages match some kind of **Predicate** such as containing specific headers with certain values, or that parts of the messages like the body match some predicate, for example an XPath or XQuery expression.



Note

The next chapter in the course covers how write tests for your Camel routes including how to set these types of expectations on your mock endpoints.



References

JAXB data format

<http://camel.apache.org/jaxb.html>

JAXB User Guide

<https://javaee.github.io/jaxb-v2/doc/user-guide/ch03.html>

JSON data format

<http://camel.apache.org/json.html>

XML JSON data format

<http://camel.apache.org/xmljson.html>

Wire tap pattern

<http://camel.apache.org/wire-tap.html>

Mock component

<http://camel.apache.org/mock.html>

Camel in Action, Second Edition - Chapter Transforming Data with Camel

Camel in Action, Second Edition - Chapter Enterprise Integration Patterns

► Guided Exercise

Transforming and Filtering Messages

In this exercise, you will transform Java objects to XML and JSON formats using Camel data transformation.

Outcomes

You should be able to:

- Read the incoming **Order** objects from an ActiveMQ queue.
- Marshal the order into XML.
- Convert that XML to JSON.
- Filter out orders that have already been delivered using the filter pattern and JSON path.
- Use a wire tap to send all undelivered orders to a mock logging system.

Before You Begin

A starter project is provided for you to start that already includes the DSL required to marshal data to XML. This starter project also includes a unit test, which you must update to check whether JSON marshaling is done correctly and the filter is functioning properly.

Run the following command to download the starter project and data files used by this exercise:

```
[student@workstation ~]$ lab jaxb-filter setup
```

Steps

► 1. Start the Red Hat AMQ 7.0 broker.

Open a terminal window (**Applications** → **System Tools** → **Terminal**) and execute the following commands:

```
[student@workstation ~]$ cd ~/JB421/labs/jaxb-filter/broker1/bin  
[student@workstation bin]$ ./artemis run
```

When the broker finishes booting up, the terminal contains the following message:

```
[org.apache.activemq.artemis] AMQ241001: HTTP Server started at http://  
localhost:8161  
[org.apache.activemq.artemis] AMQ241002: Artemis Jolokia REST API available at  
http://localhost:8161/console/jolokia  
[org.apache.activemq.artemis] AMQ241004: Artemis Console available at http://  
localhost:8161/console
```

► 2. Import the starter Maven project.

- 2.1. Open JBoss Developer Studio (**Applications** → **Programming** → **JBoss Developer Studio**). JBoss Developer Studio requests a default workspace; keep the default configuration (**/home/student/workspace**) and click **Launch**.
- 2.2. Import the Maven project by selecting from the JBoss Developer Studio main menu **File** → **Import**.
- 2.3. From the **Import** dialog box, select **Maven** → **Existing Maven Projects** and click **Next >**.
- 2.4. Click **Browse** and choose **/home/student/JB421/labs/jaxb-filter** and click **Ok** to choose the Maven project. Click **Finish** to complete the import process in JBoss Developer Studio.
A new project named **jaxb-filter** will be listed in the **Project Explorer** view.

► 3. Complete the project Maven dependencies.

- 3.1. Open the project POM file.

In the **Project Explorer** view, expand **jaxb-filter** and double-click **pom.xml**.

A new JBoss Developer Studio editor tab opens with the **jaxb-filter/pom.xml** file in a Maven POM editor.

Click the **pom.xml** tab at the bottom of the editor to see the POM raw source code.

- 3.2. Beneath the following comment:

```
<!-- TODO Add camel-xmljson and xom dependencies here -->
```

Add the following dependency definitions:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-xmljson</artifactId>
</dependency>
<dependency>
  <groupId>xom</groupId>
  <artifactId>xom</artifactId>
  <version>1.2.5</version>
</dependency>
```

The **camel-xmljson** and **xom** library dependencies are required to convert XML directly to JSON with Camel.

- 3.3. Look for this comment:

```
<!-- TODO Add camel-jsonpath dependency here -->
```

Replace it with the following dependency definition:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-jsonpath</artifactId>
</dependency>
```

The **camel-jsonpath** library allows you to use the **jsonpath** DSL method for creating expressions or predicates that parse JSON messages similar to using the **xpath** DSL method with XML data.

- 3.4. Press **Ctrl+S** to save your updates to the **pom.xml** file.
- ▶ 4. Review the existing route definition.
 - 4.1. Open the route builder class for editing.

In the **Project Explorer** view, expand **src/main/java** and then expand the **com.redhat.training.jb421** package and double-click on **TransformRouteBuilder.java** file to open the route builder.
 - 4.2. Review the route definition:

```
from("activemq:queue:orderInput?username=admin&password=admin") ①
    .marshal().jaxb() ②
    .log("XML Body: ${body}")
    .to("mock:fufillmentSystem"); ③
```

 - ① Using the **activemq** component, read messages from the queue named **orderInput**.
 - ② Marshal the incoming messages using JAXB into XML.
 - ③ Mock endpoint representing our fulfillment system. Mock endpoints are covered in more detail in later sections.
- ▶ 5. Update the included route to convert the XML message to JSON format and log the output to the console.
 - 5.1. Create the **XmlJsonDataFormat** instance that will be used in the Java DSL.

Look for the comment **//TODO add the XmlJsonDataFormat** and below it add the following code:

```
//TODO add the XmlJsonDataFormat
XmlJsonDataFormat xmlJson = new XmlJsonDataFormat();
```

This class is used by the **marshal** method of the Java DSL to convert the XML data into JSON data.
 - 5.2. Marshal the XML data into JSON.

In the route definition, after the **.log("XML Body: \${body}")** method invocation, add the following code snippet to convert the XML to JSON and log the result to the console:

```
//TODO Marshal JSON
.marshal(xmlJson)
.log("JSON Body: ${body}")
```
- ▶ 6. Using the **delivered** field on the **Order** object, filter out any orders that were already delivered.

Continue editing the route definition in **TransformRouteBuilder.java**. After the **.log("JSON Body: \${body}")** method invocation add the following code snippet to filter out **Order** objects that have a property of **delivered** set to **true**.

```
//TODO Filter JSON  
.filter().jsonpath("$.#[?(@.delivered !='true')])")
```

Since the data is now in JSON format, you can use a predicate that uses the **jsonpath** method to only allow messages where the value of the **delivered** field is not equal to **true**.

- ▶ 7. Send all undelivered orders to a mock endpoint representing an order logging system.
- 7.1. Add the **wireTap** DSL method to the route definition.
Continue editing the route definition in **TransformRouteBuilder.java**. After the filter line added in the previous step, add the following code snippet to use the wire tap pattern to send a copy of all undelivered orders to a separate direct endpoint:

```
//TODO wire tap  
.wireTap("direct:jsonOrderLog")
```

- 7.2. Establish a new direct route to a mock order logging system.

Continue editing the route definition in **TransformRouteBuilder.java** file. Find the comment **//TODO add direct route to mock order log end point** and add the following code snippet to provide the order logging route:

```
from("direct:jsonOrderLog")  
.log("Order received: ${body}")  
.to("mock:orderLog");
```

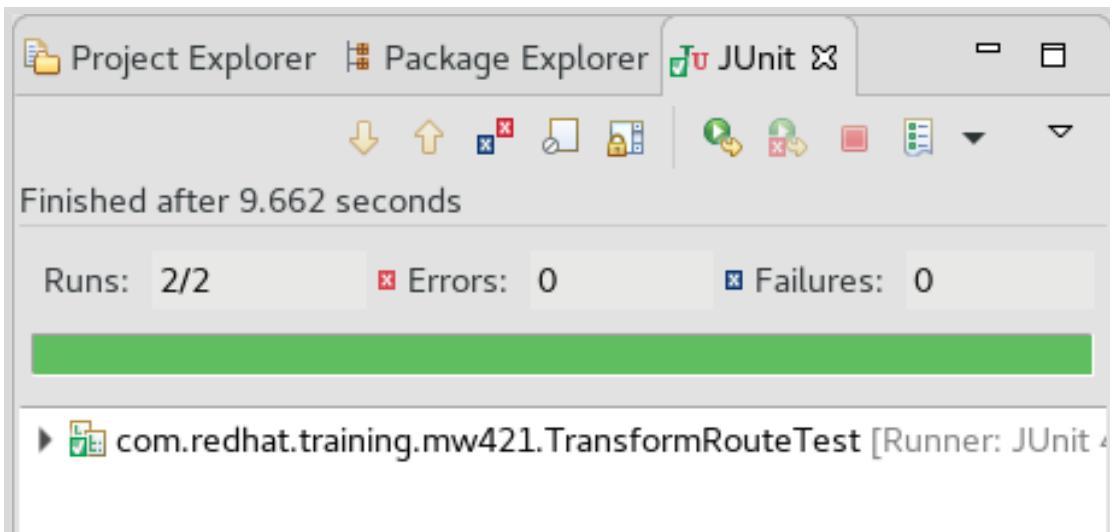
Your route definitions should match closely to the following:

```
from("activemq:queue:orderInput?username=admin&password=admin")  
.marshal().jaxb()  
.log("XML Body: ${body}")  
.marshal(xmlJson)  
.log("JSON Body: ${body}")  
.filter().jsonpath("$.#[?(@.delivered !='true')])")  
.wireTap("direct:jsonOrderLog")  
.to("mock:fulfillmentSystem");  
  
from("direct:jsonOrderLog")  
.log("Order received: ${body}")  
.to("mock:orderLog");
```

- 7.3. Press **Ctrl+S** to save your updates to the **TransformRouteBuilder** class.

▶ 8. Run the provided JUnit test case to verify your route.

 - 8.1. Run the unit test.
Right-click the **com.redhat.training.jb421.TransformRouteTest** class and select **Run As → JUnit Test** to run the test.
 - 8.2. Check the JUnit test outcome.
Open the **JUnit** tab, if both test passed, it resembles the following screen shot:



8.3. Review the console log.

Open the **Console** tab in the bottom pane of JBoss Developer Studio, if the route is functioning properly it contains the following text during the first test:

```
INFO XML Body: <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<order>
    <orderItems>
        <orderItem>
            <extPrice>15.99</extPrice>
            <item>
                <author>Yann Martel</author>
            ...
        ...
    ...
INFO Using default type string
INFO JSON Body: {"orderItems":[{"extPrice":"15.99","item":{"author":...
INFO Asserting: mock://orderLog is satisfied
INFO Order received: {"orderItems":[{"extPrice":"15.99","item":{"author":...
```

▶ 9. Close the project and stop the Red Hat AMQ Broker.

- 9.1. In the **Project Explorer** view, right-click **jaxb-filter** and click **Close Project**.
 - 9.2. Return to the terminal window where Red Hat AMQ is running, and press **Ctrl+C** to stop it.

This concludes the guided exercise.

Transforming Data with the Message Translator Pattern

Objective

After completing this section, students should be able to use additional data formats, custom type converters, and the message translator pattern to transform messages.

Introducing Camel Custom Type Converters

Camel provides a built-in type-converter system that automatically converts between well-known types. This system allows Camel components to easily work together without having type mismatches. From the Camel developer's perspective, type conversions are built into the API in many places without being invasive. When routing messages from one endpoint to another, it is often necessary for Camel to convert the body payload from one Java type to another. Common Java types that are frequently converted between include:

- **File**
- **String**
- **byte[]** or **ByteBuffer**
- **InputStream** or **OutputStream**

The **Message** interface defines **getBody** helper method to allow such automatic conversion. For example:

```
Message message = exchange.getIn();
byte[] image = message.getBody(byte[]);
```

In this example, Camel converts the body payload during the routing execution from a data format such as **File** to a Java **byte[]** array. Suppose you need to route files to a JMS queue using **javax.jmx.TextMessage** objects. To do so, you can convert each file to a **String**, which forces the JMS component to use the **TextMessage** class. This is easy to do in Camel—you use the **convertBodyTo** method, as shown here:

```
from("file://orders/inbox")
    .convertBodyTo(String.class)
    .to("activemq:queue:inbox");
```

In some use cases, however, it is necessary to transform a known format that is not supported by Camel by default (such as a proprietary file format) to a Java class. For instance, you might work with a vendor that uses proprietary encryption technology, which requires a custom API to decrypt. This custom transformation is possible using a *custom type converter*.

To develop a type converter, use the Camel annotation **@Converter** in any class that implements custom conversion logic. Additionally, to allow Camel to find your type converter classes, you must include a file named **TypeConverter** in the **META-INF/services/org/apache/camel/** directory. The **TypeConverter** file must include any packages where you keep your custom converter implementation classes, each package on a new line, as shown in the following example:

```
com.redhat.training.jb421.converters
```

In addition to the class-level annotation, a custom converter class must also have a static method (also annotated with `@Converter`) whose signature must meet the following requirements:

- The return value must be a type that is compatible with the resulting object.
- The first parameter must be a type that is compatible with the format you wish to convert.
- Optionally, an `Exchange` object can be used as a second parameter.

The following class demonstrates an implementation of a custom type converter that converts a CSV file to an instance of the Java class `Order`.

```
//imports removed for brevity
@Converter
public class OrderTypeConverter {

    @Converter
    public static <T>Order toOrder(GenericFile<T> data ①, Exchange exchange ②) {
        TypeConverter converter = exchange.getContext().getTypeConverter(); ③
        String s = converter.convertTo(String.class, data);
        if (s == null) {
            throw new IllegalArgumentException("data is invalid");
        }
        String[] split = s.split(",");
        String idAsString = split[0];
        String description = split[1];
        String priceAsString = split[2];
        String taxAsString = split[3];
        Integer id = converter.convertTo(Integer.class, idAsString);
        Double price = converter.convertTo(Double.class, priceAsString);
        Double tax = converter.convertTo(Double.class, taxAsString);
        return new Order(id,description, price, tax);
    }
}
```

- ① Transforms a `GenericFile` object (from Camel's file component) to an `Order` object.
- ② Obtains an `Exchange` object to use default type converters provided by Camel.
- ③ The `Exchange` object provides a `getContext().getTypeConverter()` which returns any type converter in the registry that can support the desired conversion, if one exists.

The file to be read is:

```
10,N2PENCIL,0.15,1.50
```

The content order is:

```
id,description,price,tax
```

The order is transformed to the following Java class:

```
//imports removed for brevity
public class Order implements Serializable{

    private int id;
    private String description;
    private double price;
    private double tax;

    public Order(){}
    public Order(int id, String description, double price, double tax) {
        this.id = id;
        this.description = description;
        this.price = price;
        this.tax = tax;
    }
    // getters and setters
}
```

This converter can be used in a route with the following code:

```
public class JavaRouteBuilder extends RouteBuilder {

    @Override
    public void configure() throws Exception {
        from("file:in")
            .convertBodyTo(Order.class)
            .to("file:out");
    }
    ...
}
```

Implementing the Message Translator Pattern using **transform**

The message translator pattern provides a mechanism to make updates or transformations to your Camel message exchanges to comply with requirements of the various integration endpoints used by your system. This pattern is usually used to adapt the message payload from one system to another, whether it is an entirely different data format, or just the addition of an extra field, or a tweak to the file delimiter. Message translation is often necessary for integration systems that interface between many different applications, including legacy applications, proprietary software solutions, and external vendor systems. Typically each of these systems use their own data model. This means, for instance, that each system may have a slightly different notion of the *order* entity, including fields that define an *order* and which other entities an *Order* has relationships with. For example, the accounting system may be more interested in the order's sales tax numbers while the fulfillment system stores order item SKUs and quantity.

In Camel, one method of implementing the message translator pattern is using the **transform** DSL method. The **transform** method makes inline changes to the **Message** instance to update its contents or format. In the following code, the route changes all of the commas (,) from a file to semicolons (;) to create a semicolon-separated file:

```

package com.redhat.training;

import org.apache.camel.builder.RouteBuilder;

public class JavaRouteBuilder extends RouteBuilder {

    @Override
    public void configure() throws Exception {
        from("file:in")
            .transform(❶
                body()❷
                    .regexReplaceAll(", ", ";")❸
            )
            to("file:out");
    }
}

```

- ❶ **transform** method call to start the update of file content to a different format
- ❷ **body** method call to update the contents from the message's body (the file content). A change to the header is also possible, calling the **header** method instead
- ❸ Looks for all commas (,) in the file and changes each occurrence to a semicolon (;)

The **transform** method supports using Simple EL, as shown in the previous example, as well as using the **constant** method to overwrite the entire message body, as demonstrated in the following route:

```

from("direct:start")
    .log("Article name is ${body}")
    .choice()
        .when().simple("${body} contains 'Camel'")
            .transform(constant("Yes"))
        .otherwise()
            .transform(constant("No"))
    .end();
    .to("stream:out")

```

In this example, the body of the message is set to either **Yes** or **No** based on whether the message body content contains the string **Camel**.

Introducing camel-bindy for Translating CSV files to Java Objects

Camel allows marshaling and unmarshaling to and from traditional text formats, such as a CSV or flat file, to Java objects, using the **camel-bindy** library. Bindy uses a similar approach as JAXB to map Java objects to text format and vice versa.

The following route transforms the data from a CSV format to a Java object:

```
DataFormat bindy = new BindyCsvDataFormat("com.redhat.training");

from("file:in")
    .unmarshal(bindy) ①
    .to("jms:queue:orders");
```

- ① Scan for classes available at **com.redhat.training** using camel-bindy annotations and unmarshal from CSV files to Java objects. This class also includes a constructor that takes a single class name instead of a package, as shown below.

```
DataFormat bindy = new BindyCsvDataFormat(Order.class)
```

The **camel-bindy** module processes the contents from a CSV file and transforms them into a **List<Map<String, Object>>**. For each position of this **List**, there will be a record in the CSV file, which is represented as a **Map**.

For each record in the **Map**, there will be a Java object retrieved from the CSV file. This allows the mapping of multiple objects from a record in the CSV file.



Note

The key for each object stored in this map is the fully qualified name of the class.

Mapping Java object fields to variables using **camel-bindy** is done using annotations similarly to JPA. The following code shows some important annotations.

```
package com.redhat.training;
@CsvRecord(separator=",",crlf="UNIX") ①
public class Order implements Serializable{ ②

    @DataField(pos=1) ③
    private int id;
    @DataField(pos=2)
    private String description;
    @DataField(pos=3)
    private double price;
    @DataField(pos=4)
    private double tax;
}
```

- ① The **@CsvRecord** annotation marks the actual class as managed by Bindy. In this case, the annotation specifies that it will parse a CSV file separated by a comma (**separator** parameter) where the line breaks (**crlf** parameter) are UNIX-compliant. The separator parameter accepts one of **WINDOWS**, **MAC**, or **UNIX** as its parameter, allowing you to ensure you are parsing your files correctly based on the operating system platform where the CSV files were created.
- ② The class implements the **java.io.Serializable** interface to allow marshaling processing.
- ③ The **DataField** annotation requires a **pos** parameter to map the position where an attribute will be read in a CSV file.

Transforming XML Data Using the XSLT Component

Another common mechanism for transforming XML data is XSLT (Extensible Stylesheet Language Transformations). XSLT can be used to transform XML, altering elements, values and attributes to match a different schema, or even to transform XML data to another format such as HTML.

In Camel, the `xslt` component provides the ability to process a message using an XSLT template. The basic URI format of the `xslt` component is as follows:

```
xslt:templateName[?options]
```

Here, **templateName** is the URI to an XSLT template that is available on the class path, or a complete URL to a remote XSLT template. Two examples are listed, as follows:

```
xslt:file:///tmp/transform.xsl # will use the file located at /tmp/transform.xsl
```

```
xslt:http://example.com/xslt/foo.xsl # will use the remote resource at the specified URL
```

An XSLT endpoint in a route would appear similar to the following:

```
from("activemq:myQueue").
    to("xslt:file:///tmp/transform.xsl");
```

Implementing the Splitter Pattern

The splitter pattern provides a way to break the contents of a message, into smaller chunks. This is especially useful when you are dealing with lists of objects, and you want to process each item in the list separately. An example where this is useful is splitting the individual items from a customer's order.

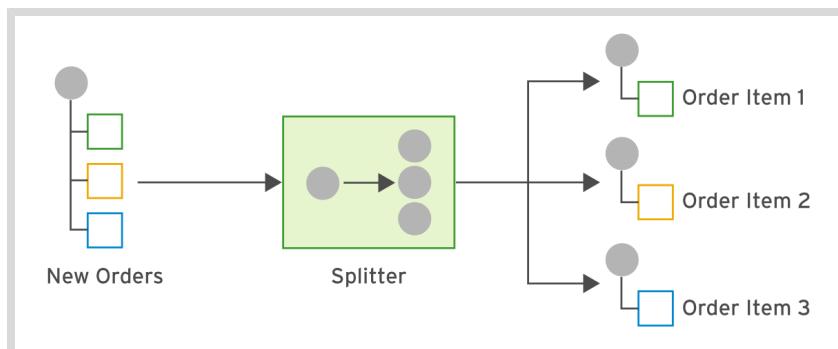


Figure 3.3: Splitter pattern

To implement this pattern, Camel provides the **split** method, which you can use when creating the route. The following example demonstrates using an XPath predicate to split XML data stored in the exchange body:

```
from("activemq:queue>NewOrders")
    .split(body(xpath("/order/product")))
    .to("activemq:queue>Orders.Items");
```

In Spring, the same pattern can be implemented using the following code:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
  <route>
    <from uri="activemq>NewOrders"/>
    <split>
      <xpath>/order/product</xpath>
      <to uri="activemq:Orders.Items"/>
    </split>
  </route>
</camelContext>
```

The **split** function also supports splitting of certain Java types by default without any predicate specified. A common use case is to split a **Collection**, **Iterator**, or array from the exchange body.

The **split** method creates separate exchanges for each part of the data that is split and then sends those exchanges along the route separately.

For example, consider a route that has unmarshaled CSV data into a **List** object. Once the data is unmarshaled, the exchange body contains a **List** of model objects, and a call to **split** creates a new exchange for each record from the CSV file.

The following examples demonstrate splitting the exchange body as well as splitting an exchange header:

```
from("direct:splitUsingBody").split(body()).to("mock:result");

from("direct:splitUsingHeader").split(header("foo")).to("mock:result");
```

The equivalent route definitions in Spring DSL appear as follows:

```
<split>
  <simple>${body}</simple> ❶
  <to uri="mock:result"/>
</split>

<split>
  <simple>${header.foo}</simple> ❷
  <to uri="mock:result"/>
</split>
```

- ❶ This example splits the body containing an **Iterable** Java object (**List**, **Set**, **Map**, etc) into separate exchanges.
- ❷ This example splits a header containing an **Iterable** Java object into separate exchanges with the same body. For example, if the exchange head a header containing a list of users, split could transform it into multiple exchanges, one per user in the list, all containing the same exchange body.

Using the Tokenizer with the Splitter Pattern

The tokenizer language is intended to *tokenize* or break up text documents, such as CSV or XML, using a specified delimiter pattern. You can use the tokenizer expression with **split** to split the

exchange body using a token. This allows you to split your text content without the need to first unmarshal it into Java objects.

Because this is a common use-case, a **tokenize** XML element is provided for this in the Spring DSL, and a **tokenize** method is provided in the Java DSL. In the XML sample below, the body is split using an "at" (@) symbol as a separator. You can also use a comma, space, or even a regular expression pattern using the option **regex=true**.

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
        <from uri="direct:start"/>
        <split>
            <tokenize token="@"/>
            <to uri="mock:result"/>
        </split>
    </route>
</camelContext>
```

Additionally, if you are splitting XML data, an optimized version of the tokenizer is provided using the DSL method **tokenizeXML**. The Spring DSL tokenizer has an attribute named **xml**, which can be set to true or false to enable the XML-optimized tokenization.

Java DSL example:

```
from("file:inbox")
    .split().tokenizeXML("order")
    .to("activemq:queue:order");
```

Spring DSL example:

```
<route>
    <from uri="file:inbox"/>
    <split>
        <tokenize token="order" inheritNamespaceTagName="orders" xml="true"/>
        <to uri="activemq:queue:order"/>
    </split>
</route>
```

Addressing Memory Usage Issues in the Splitter Pattern with Streaming

When consuming large pieces of data with Camel, the splitter pattern is often used to divide up this data into small, more manageable pieces. In addition to using the **split** method split the data, Camel offers an option called streaming, which can alleviate memory issues by not loading the entire piece of data into memory. If streaming is enabled, then Camel splits in a streaming fashion, which means it splits the input message in chunks, instead of attempting to read the entire body into memory at once, and then splitting it. This reduces the memory required for each invocation of the route. For this reason, if you split messages with extremely large payloads, it is recommended that you enable streaming. However, if your data is small enough to hold in memory, streaming is probably unnecessary overhead.

You can split streams by enabling the streaming mode using the streaming builder method. The following route demonstrates streaming in Java DSL:

```
from("direct:streaming")
    .split(body().tokenizeXML("order")).streaming()
    .to("activemq:queue.order");
```

Create the same route In XML DSL as follows:

```
<route>
    <from uri="direct:streaming"/>
    <split streaming="true">
        <tokenize token="order" xml="true"/>
        <to uri="activemq:queue:order"/>
    </split>
</route>
```

If the data you are splitting is in XML format, be sure to use the **tokenizeXML** instead of an XPath expression. This is because the XPath engine in Java loads the entire XML content into memory, negating the effects of streaming for very big XML payloads.



References

Bindy documentation

<http://camel.apache.org/bindy.html>

Splitter pattern

<http://camel.apache.org/splitter.html>

XSLT component

<http://camel.apache.org/xslt.html>

Camel in Action, Second Edition - Chapter Transforming Data with Camel

Camel in Action, Second Edition - Chapter Enterprise Integration Patterns

► Guided Exercise

Transforming Message with Custom Type Converters

In this exercise, you will receive incoming orders from a CSV file, transform the file contents to update the CSV separator character, and unmarshal the CSV data to Java objects. Then you will use a custom type converter to convert those objects to JSON, and finally log the JSON data for the orders to the console.

Outcomes

You should be able to implement Camel routes to:

- Read the incoming CSV data from a local file.
- Update the CSV separator character.
- Unmarshal the orders into Java objects.
- Convert those objects to JSON using a custom type converter.

Before You Begin

A starter project is provided for you to start, which already includes the route definition. This starter project also includes a unit test to verify whether the CSV unmarshaling is done correctly and the type converter is functioning properly.

Run the following command to download the starter project and data files used by this exercise:

```
[student@workstation ~]$ lab custom-converter setup
```

Steps

- ▶ 1. Open JBoss Developer Studio and import the starter Maven project.
 - 1.1. Open JBoss Developer Studio (**Applications** → **Programming** → **JBoss Developer Studio**). JBoss Developer Studio requests a default workspace; leave the default configuration (**/home/student/workspace**) and click **Launch**.
 - 1.2. Import the Maven project by selecting from the JBoss Developer Studio main menu **File** → **Import**.
 - 1.3. From the **Import** dialog box, select **Maven** → **Existing Maven Projects** and click **Next >**.
 - 1.4. Click **Browse** and choose **/home/student/JB421/labs/custom-converter** and click **Ok** to choose the Maven project. Click **Finish** to complete the import process in JBoss Developer Studio.
- A new project named **custom-converter** will be listed in the **Project Explorer** view.

► 2. Inspect the starter project.

2.1. Inspect the `TransformRouteBuilder` class.

In the **Project Explorer** view, expand `custom-converter` → `src/main/java` → `com.redhat.training.jb421`. Double-click `TransformRouteBuilder.java` file.

The file has a single route that reads from a file endpoint, and then forward the contents to a mock endpoint representing a JMS queue used to send orders to a downstream system.

2.2. Review the Bindy annotations used in the model classes.

In the **Project Explorer** view, expand `custom-converter` → `src/main/java` → `com.redhat.training.jb421.model`. Double-click `Order.java` file.

```
@CsvRecord(separator="`") ①
public class Order implements Serializable {

    private static final long serialVersionUID = 1L;

    @DataField(pos=1)
    private Integer id;
    @DataField(pos=2, pattern="MM-dd-yyyy") ②
    private Date orderDate = new Date();
    @DataField(pos=3, pattern="##.##") ③
    private BigDecimal discount;
    @DataField(pos=4)
    private Boolean delivered=false;
    @Link ④
    private Customer customer;
    @DataField(pos=19)
    private String itemSku;
```

- ① Set the `separator` used in the CSV data to the backquote (`) character.
- ② `pattern` defines the date format to parse the string from the CSV file into an instance of `java.util.Date` class.
- ③ `pattern` defines the decimal format in order to parse the string from the CSV file into an instance of `java.math.BigDecimal` class.
- ④ `@Link` annotation is used to tell Bindy to create an instance of `Customer` for every instance of `Order` created. Notice the position of the data fields on the `Order` object skip from 4 - 19. Fields 5 - 18 are all contained inside the `Customer` class.

In the **Project Explorer** view, expand `custom-converter` → `src/main/java` → `com.redhat.training.jb421.model`. Double-click `Customer.java`.

Notice the `@Link` annotation present at the class level. This tells camel to instantiate this class for every `Order` created from a CSV record. Also, all the member variables are annotated with `@DataField` to set their positions in the CSV record.

► 3. Update the route definition to change the CSV separator from a comma (,) to backquote (`).

3.1. Update the route definition in `TransformRouteBuilder.java` file to replace the comment `//TODO add transform to update the CSV separator` with the following DSL snippet:

```
.transform(body().regexReplaceAll(", ", ``))
```

- 3.2. Save your updates to the route definition using **Ctrl+S**.

► 4. Update the route builder to unmarshal the CSV format to a Java object.

- 4.1. Create the **BindyCsvDataFormat** instance to be used by **unmarshal** method.

Look for the comment **//TODO add bindy data format** and replace it with the following code snippet:

```
BindyCsvDataFormat bindy = new BindyCsvDataFormat(Order.class);
```

- 4.2. Add the call to the **unmarshal** method to convert the data into Java objects, using the data format created in the previous step.

Look for the comment **//TODO unmarshal with bindy** and replace it with the following DSL snippet:

```
.unmarshal(bindy)
```

- 4.3. Save your updates to the route definition using **Ctrl+S**.

► 5. Create a new custom type converter class to transform **Order** into a JSON string.

- 5.1. Create a new Java class for the custom type converter.

In the **Project Explorer** view, right-click the `com.redhat.training.jb421` package and then click **New → Class** to launch the new class creation wizard. Enter **OrderJSONTypeConverter** as the class name and click **Finish** to create the class.

When the class is open for editing, update its contents to match the following, which is provided for you to copy in paste in the **OrderJSONTypeConverter.txt** file.

```
package com.redhat.training.jb421;

import org.apache.camel.Converter;

import com.google.gson.Gson;
import com.redhat.training.jb421.model.Order;

@Converter
public class OrderJSONTypeConverter {

    @Converter
    public String convertToJson(Order order){
        Gson gson = new Gson();
        return gson.toJson(order);
    }
}
```

This converter does a simple conversion from an **Order** object to a JSON string using the `gson` library. Because it is annotated with **@Converter** at the class and method levels, once this class is registered to the **CamelContext**, it is automatically used to convert instances of **Order** to **String**.

- 5.2. Create the necessary directory structure for the configuration file.

In the **Project Explorer** view, right-click the **src/main/resources/META-INF** folder and select **New → Folder**. In the **New Folder** wizard, set the folder name to **services/org/apache/camel/**.

- 5.3. Add the **TypeConverter** file to the directory.

Right-click the **src/main/resources/META-INF/services/org/apache/camel** directory that you just created, and select **New → File**. In the **New File** wizard, set the file name to **TypeConverter**.

Once the file is created and it has opened for editing, add the following contents:

```
com.redhat.training.jb421
```

Now Camel will scan the *com.redhat.training.jb421* package for classes that are annotated with **@Converter** and register the type converter you created in the previous step.

- 5.4. Press **Ctrl+S** to save the **TypeConverter** file.

► **6.** Output the order information using the JSON format.

- 6.1. Add a second destination to the first route.

Open **TransformRouteBuilder.java** again and update the following DSL:

```
.to("mock:orderQueue");
```

to:

```
.to("mock:orderQueue", "direct:orderLog");
```

- 6.2. Add the second route definition.

Look for the comment **//TODO add second direct route** and add the following DSL:

```
from("direct:orderLog")
    .split(body())
    .log("${body}")
    .to("mock:orderLoggingSystem");
```

This route splits the list of **Order** objects you unmarshaled with Bindy into separate exchanges and then logs the body of each of those to the console. Since **log** method transforms the body content to a **String** to print, Camel uses the custom type converter defined earlier in the exercise to generate a **String** for each **Order** record.

► **7.** Test the routes.

- 7.1. Start the routes.

Open a new terminal window and run the following commands:

```
[student@workstation ~]$ cd JB421/labs/custom-converter
[student@workstation custom-converter]$ mvn camel:run
```

The route is started with the following message:

```
INFO Apache Camel 2.21.0.fuse-000077-redhat-1 (CamelContext: jb421Context)
started in 0.283 seconds
```

- 7.2. Copy the CSV file to the input directory and check the results in the log.

In a new terminal window, run the following command to execute the provided bash script to copy the CSV file into the **items/incoming** folder.

```
[student@workstation ~]$ cd JB421/labs/custom-converter
[student@workstation custom-converter]$ ./setup-data.sh
```

- 7.3. Verify the output of the plug-in matches what is expected.

Return to the terminal where you started the Camel Maven plug-in. You should see the JSON output for each order in the CSV file, similar to sample output below (sample has been truncated):

```
INFO {"id":1,"orderDate":"Oct 11, 2016 12:00:00
AM","discount":6.08,"delivered":false,...}
INFO {"id":2,"orderDate":"Jan 22, 2014 12:00:00
AM","discount":75.70,"delivered":false,...}
INFO {"id":3,"orderDate":"May 1, 2013 12:00:00
AM","discount":6.88,"delivered":false,...}
INFO {"id":4,"orderDate":"Aug 15, 2012 12:00:00
AM","discount":5.85,"delivered":false,...}
INFO {"id":5,"orderDate":"Sep 1, 2016 12:00:00
AM","discount":61.02,"delivered":false,...}
INFO {"id":6,"orderDate":"Oct 11, 2016 12:00:00
AM","discount":17.85,"delivered":false,...}
INFO {"id":7,"orderDate":"Feb 11, 2016 12:00:00
AM","discount":9.09,"delivered":false,...}
INFO {"id":8,"orderDate":"Dec 11, 2014 12:00:00
AM","discount":8.35,"delivered":false,...}
INFO {"id":9,"orderDate":"Oct 14, 2015 12:00:00
AM","discount":6.55,"delivered":false,...}
```



Note

The output may differ because the orders may not be processed ordered using the order ID number.

- 8. Stop the route.

Press **Ctrl+C** in the terminal window where the Camel Maven plug-in was started.

- 9. Close the project.

- 9.1. Return to JBoss Developer Studio, and in the **Project Explorer** view, right-click custom-converter project and click **Close Project**.

This concludes the guided exercise.

Combining Messages Through Aggregation

Objective

After completing this section, students should be able to merge multiple messages using the Aggregator pattern and customize the output to a traditional data format.

Introducing the Aggregator Pattern

The aggregator pattern is a mechanism allowing you to group fragmented data from multiple source messages into single a unique message. The aggregator pattern is suitable for use cases where fragmented data is not the best way to deliver information. For example, when you want to batch process data that you receive in fragments, such as combining individual orders that need to be fulfilled by the same vendor. Using this pattern, you can define custom aggregation behavior to control how Camel uses the source data fragments to build the final aggregated message.

To build the final message, there are three pieces of information needed:

The correlation value

You must define a correlation value which defines an expression or predicate that Camel uses to match each **exchange** object captured by an aggregator pattern implementation. The correlation value is a field obtained using an **expression** to group messages together for aggregation. A common strategy used to group messages is to use an exchange header, but it could also refer to a message body field using **xpath** or **jsonpath**.

Logic for how to compile the final message

You must provide Java code to build the final message exchange sent by the aggregator. This can be as simple as concatenating the exchange bodies together, or much more complex custom business logic. To build this, implement the **AggregationStrategy** interface which builds the exchange payload with the messages captured using the correlation value.

The complete condition

You must define the complete condition which uses a predicate or time condition to instruct Camel to check when the final message **exchange** object built from the individual incoming exchanges should be sent out of the aggregator.

To use this pattern in your Camel route, use the **aggregate** DSL method, which requires two parameters:

```
.aggregate(correlationExpression, AggregationStrategyImpl)
```

Additionally, a completion condition is needed to define when to send the aggregated exchange. This is identified using methods from the Java domain-specific language (DSL) which will be discussed later in this section.

For example, in the following route, the messages with a matching header field named **destination** are aggregated using the **MyNewStrategy AggregationStrategy** implementation.:

```
from("file:in")
    .aggregate(header("destination"), new MyNewStrategy()...
    .to("file:out");
```

The AggregationStrategy Interface

The **AggregationStrategy** is a required implementation when merging multiple messages into a single message. It declares a single method (**aggregate**) and requires the following guidelines:

- The **aggregate** method requires two exchange parameters, but the first parameter will always be **null** for the first message. This is because when you receive the first message exchange you have not yet created the aggregated message. Therefore, an **if** clause must exist to check whether the first exchange is **null**, and instantiate the aggregated message and return it.
- The exchange object that is returned by the **aggregate** method is automatically passed into the next execution of the **AggregationStrategy** implementation. In the **AggregationStrategy**, implementation an **exchange** object is expected to be returned by the method execution, with body contents that represent the aggregation of the two exchange objects passed into the **aggregate** method execution.

```
final class BodyAggregationStrategy implements AggregationStrategy {
    @Override
    public Exchange aggregate(Exchange oldExchange ①, Exchange newExchange ②) {
        if (oldExchange == null){③
            return newExchange;
        }
        String newBody = newExchange.getIn().getBody(String.class);
        String oldBody = oldExchange.getIn().getBody(String.class); ④
        newBody = newBody.concat(oldBody); ⑤
        newExchange.getIn().setBody(newBody); ⑥
        return newExchange; ⑦
    }
}
```

- ① The **exchange** object that was previously processed and returned by this **AggregationStrategy** implementation.
- ② The **exchange** object containing the newest **exchange** object received by this **AggregationStrategy** implementation.
- ③ Mandatory condition to check if this is the first message processed by this **AggregationStrategy** implementation. The first invocation of the **aggregate** method will always have a **null** value for the **oldExchange** parameter.
- ④ Retrieves the body contents from the **exchange** object sent by the previous execution of this **AggregationStrategy** implementation and transform it into a **String**.
- ⑤ Merge the body content from both exchanges. This implementation uses a simple string concatenation to merge the two exchange bodies.
- ⑥ Updates the body of the **exchange** object with the merged body content to be sent to the next execution of this **AggregatorStrategy** implementation.
- ⑦ Sends the updated **exchange** object to the next execution of this **AggregationStrategy** implementation.

Controlling the Size of the Aggregation

When using the aggregator pattern, Camel requires that developers identify the conditions under which the aggregated message exchange must be sent to the remainder of the route. Below are

six of the most commonly used methods that are provided by Camel to identify the complete condition:

completionInterval(long completionInterval)

Build the aggregated message after a certain time interval (in milliseconds).

completionPredicate(Predicate predicate)

Build the aggregated message if the predicate is true.

completionSize(int completionSize)

Build the aggregated message when the number of messages defined in the **completionSize** is reached.

completionSize(Expression completionSize)

Build the aggregated message when the number of messages processed by a Camel expression is reached.

completionTimeout(long completionTimeout)

Build the aggregated message when there are no additional messages for processing and the **completionTimeout** (in milliseconds) is reached.

completionTimeout(Expression completionTimeout)

Build the aggregated message when there are no additional messages for processing and the timeout defined by a Camel expression is reached.

In the following route, the **completionSize** method is used to trigger the aggregated message creation:

```
from("file:in")
    .aggregate(header("destination"), new MyNewStrategy())
    .completionSize(5)
    .to("file:out");
```

**Note**

The **completionSize** method waits until the number of messages defined in the **completionSize** is reached. If this number is not reached, the **aggregate** method will hang when you try to stop the Camel context. To avoid this, in the **AggregateStrategy**, a header value called **AGGREGATION_COMPLETE_ALL_GROUPS** must be set manually to **true**:

```
newExchange.getIn().setHeader
(Exchange.AGGREGATION_COMPLETE_ALL_GROUPS, true).
```

It is also possible to use multiple completion conditions, as shown in the following example:

```
from("file:in")
    .aggregate(header("destination"), new MyNewStrategy())
    .completionInterval(10000)
    .completionSize(5)
    .to("file:out");
```

When multiple completion conditions are defined, whichever condition is met first, will trigger the completion of the aggregation. In the previous example, for completion of the batch to occur, either five total exchanges are processed, or 10 seconds have passed, whichever occurs first.

Using a Processor to Implement the Message Translator Pattern

Another simple way to implement the message translator pattern is to use a custom processor. With this approach you can do any modification that might be necessary to the **exchange**, and you have easy access to Java's wealth of APIs. You can invoke custom business logic, or use a proprietary API to transform the message content.

To create a custom processor, Camel provides the **Processor** interface, which requires a single method, **process**, with no return type. An example processor is included below:

```
public class DateProcessor implements Processor {  
    public void process(Exchange exchange) throws Exception {  
        exchange.getIn().setHeader("orderDate", new Date());  
    }  
}
```

In this example, the **DateProcessor** adds an exchange header called **orderDate** with the current date.

After you create the processor, you can use it inside a route by declaring the bean in Spring; for example, using the XML DSL:

```
<bean id="dateProcessor" class="com.redhat.training.jb421.DateProcessor"/>  
<camelContext id="jb421Context" xmlns="http://camel.apache.org/schema/spring">  
    <route>  
        <from uri="activemq:myQueue"/>  
        <process ref="dateProcessor"/>  
        <to uri="mock:myProducer"/>  
    </route>  
</camelContext>
```

Or by instantiating a new instance of it using the Java DSL:

```
public void configure() throws Exception {  
    from("activemq:myQueue")  
        .process(new DateProcessor())  
        .to("mock:myProducer");  
}
```

Demonstration: Implementing the Aggregator Pattern

Use Case: A large quantity of fixed-length order data is sent to the application via the file system. To make this order data more manageable, split the data into individual orders, and then aggregate those orders into batches of 25. Also, each order needs to be transformed into XML format and a **<batch>** wrapper element needs to be added around the 25 individual orders. Additionally, each order needs to be logged in XML format to the console individually.

1. Start JBoss Developer Studio and import the **aggregator-pattern** project from the `/home/student/JB421/labs/` directory.
2. Review the sample data.

Open `/home/student/JB421/data/ordersFixedLength/orders.txt` with a text editor:

```
32292711-26-1415.00832361
56853312-07-2966.99578186
10963401-22-8962.99254216
73300606-12-3231.00966662
17528603-29-6394.99700673
47991512-06-6054.99816174
74864211-28-5360.99854391
```



Note

The data file contains 2500 lines. Each line in the file may appear to be a long string of numbers and symbols, but each one actually represents an order. The positions and lengths of each piece of data inside the fixed-length file are specified in the bindy annotations on the model class.

3. Review the `com.redhat.training.jb421.model.Order` model class with Bindy annotations:

```
@FixedLengthRecord(length=25)
@XmlRootElement
public class Order implements Serializable {

    private static final long serialVersionUID = 1L;
    @DataField(pos = 1,length=6)
    private Integer id;
    @DataField(pos = 2, length=8, pattern="MM-dd-YY")
    private Date orderDate = new Date();
    @DataField(pos = 3, length=5, pattern="##.##")
    private BigDecimal discount;
    @DataField(pos = 5, length=6)
    private Integer customerId;
```

4. Add a splitter with streaming and tokenization.

Because the data file is large, it is necessary to split the data as it comes into the Camel route.

Update the Camel route to split the fixed-length data into separate records using tokenization to break on each new line in the file.

Open the `com.redhat.training.jb421.AggregateRouteBuilder` class.

Update the route definition to replace the comment `//TODO split fixed length data by new lines here and use streaming` with the following DSL snippet:

```
.split()
.tokenize("\n")
.streaming()
```

5. Review the **ArrayListAggregationStrategy** class that has been provided.

Open the **com.redhat.training.jb421.ArrayListAggregationStrategy** class.

```
public class ArrayListAggregationStrategy implements AggregationStrategy {

    public ArrayListAggregationStrategy() {
        super();
    }

    public Exchange aggregate(Exchange oldExchange, Exchange newExchange) {
        Message newIn = newExchange.getIn();
        Object newBody = newIn.getBody();
        ArrayList<Object> list = null;
        if (oldExchange == null) {
            list = new ArrayList<Object>();
            list.add(newBody);
            newIn.setBody(list);
            return newExchange;
        } else {
            Message in = oldExchange.getIn();
            list = in.getBody(ArrayList.class);
            list.add(newBody);
            return oldExchange;
        }
    }
}
```

6. Add the aggregator pattern to the route definition in order to combine batches of 25 orders into a single exchange.

Re-open the **AggregateRouteBuilder** class.

Look for the comment **//TODO aggregate into batches of 25 orders here using the ArrayListAggregationStrategy provided** and replace it with the following DSL snippet:

```
.aggregate(constant(true), new ArrayListAggregationStrategy())
.completionSize(25)
.completeAllOnStop()
```

7. Review the **BatchXMLProcessor** class.

This processor class allows us to create XML similar to the following:

```
<batch>
    <order>
        <id>1</id>
        <discount>5.99</discount>
```

```
<customerId>1</customerId>
<orderDate>11-18-16</orderDate>
</order>
<order>
<id>2</id>
<discount>8.59</discount>
<customerId>5</customerId>
<orderDate>01-08-16</orderDate>
</order>
</batch>
```

To achieve this XML structure, you must first marshal each order into XML fragments, and then add the batch elements around them.

Open the **com.redhat.training.BatchXMLProcessor** class.

```
public class BatchXMLProcessor implements Processor {

    @SuppressWarnings("unchecked")
    @Override
    public void process(Exchange exchange) throws Exception {

        // Load JAXB Context for Order
        JAXBContext jaxbContext = JAXBContext.newInstance(Order.class);
        Marshaller marshaller = jaxbContext.createMarshaller();
        // This option prevents JAXB from including the XML header
        marshaller.setProperty(Marshaller.JAXB_FRAGMENT, true);

        // Buffer to hold batch XML string
        StringBuilder batchXML = new StringBuilder();

        // Create opening tag
        batchXML.append("<batch>");

        List<Order> orderBatch = exchange.getIn().getBody(List.class);
        for (Order order : orderBatch) {
            StringWriter sw = new StringWriter();
            marshaller.marshal(order, sw);
            String orderXML = sw.toString();
            sw.close();
            batchXML.append(orderXML);
        }

        // Create closing tag
        batchXML.append("</batch>");

        // Set the result as the new exchange body
        exchange.getIn().setBody(batchXML.toString());
    }
}
```

8. Add the processor to transform aggregated list of orders to XML batch of orders.

Re-open the **AggregateRouteBuilder** class.

Look for the comment **//TODO use the BatchXMLProcessor to marshal the results w/ JAXB and add a <batch> tag around the results** and replace it with the following DSL snippet:

```
.process(new BatchXMLProcessor())
```

9. Intercept each batch to a separate endpoint.

Continue editing the **AggregateRouteBuilder.java** file, and look for the comment **// TODO add wire tap to direct:orderLogger route below** and replace it with the following DSL snippet:

```
.wireTap("direct:orderLogger")
```

10. Split the contents from the intercepted endpoint and log them to the console.

Continue editing the **AggregateRouteBuilder.java** file, and look for the comment **// TODO add split using tokenizeXML here to split up individual orders and log them** and replace it with the following DSL snippet:

```
.split()  
.tokenizeXML("order")
```

You can see for this use of the **split** DSL method, **tokenizeXML** is used. This is ideal because you know the exchange body will be XML data. Also, the **order** element is specified as the XML element you wish to split the exchange on.

Save your updates.

Solution:

```
from("file://"+SRC_FOLDER)  
.split()  
.tokenize("\n")  
.streaming()  
.unmarshal(bindy)  
.aggregate(constant(true), new ArrayListAggregationStrategy())  
.completionSize(25)  
.completeAllOnStop()  
.process(new BatchXMLProcessor())  
.wireTap("direct:orderLogger")  
.to("file://"+OUTPUT_FOLDER+"?  
fileName=output.xml&fileExist=Append", "mock:result");  
  
from("direct:orderLogger")  
.split()  
.tokenizeXML("order")  
.log("${body}");
```

11. Execute the route using the Camel Maven plug-in.
12. Verify the results in the output file.

In a new terminal window, run the following command to execute the provided bash script to copy the fixed-length file into the **orders/incoming** folder.

```
[student@workstation aggregator-pattern]$ ./setup-data.sh
```

In the Maven log, you should see output similar to the following (the log lines have been shortened to fit on the page) if the route is working properly:

```
INFO  <order><customerId>531438</customerId>...
INFO  <order><customerId>885558</customerId>...
INFO  <order><customerId>137718</customerId>...
```

Now check the output XML batches.

Return to the JBoss Developer Studio window, and in the **Project Explorer** view, right-click the **aggregator-pattern** project and then click **Refresh** to update the contents of the project with the new output file.

Expand the **orders/outgoing** directory and open the new **output.xml** file. This file should contain the XML batches of order data output by the route.

13. Clean up.

Stop the Camel Maven plug-in and close the **aggregator-pattern** project in JBoss Developer Studio.

This concludes the demonstration.



References

Aggregator pattern

<http://camel.apache.org/aggregator2.html>

XSLT documentation

<https://www.w3.org/TR/xslt>

Camel in Action, Second Edition - Chapter Enterprise Integration Patterns

► Guided Exercise

Aggregating Messages

In this exercise, you will aggregate **Order** objects into batches using a custom strategy, and then you will marshal those batches to XML and apply an XSLT transform to that XML.

Outcomes

You should be able to implement a custom aggregation strategy to aggregate batches of orders based on the value of a header on each incoming exchange using a completion interval to create the batches every five seconds.

Before You Begin

A starter project is provided for you to get started. This project includes a **CamelContext** definition and a **RouteBuilder** class. It also includes a style sheet that can be used with the XSLT endpoint to update order XML data to include the dollar (\$) symbol as a prefix to all prices.

Run the following command to download the starter project and data files used by this exercise:

```
[student@workstation ~]$ lab aggregate-messages setup
```

Steps

- 1. Start the Red Hat AMQ 7.0 broker.

Open a terminal window (**Applications** → **System Tools** → **Terminal**) and execute the following commands:

```
[student@workstation ~]$ cd ~/JB421/labs/aggregate-messages\
/broker1/bin
[student@workstation bin]$ ./artemis run
```

When the broker finishes booting up, the terminal contains the following message:

```
[org.apache.activemq.artemis] AMQ241001: HTTP Server started at http://
localhost:8161
[org.apache.activemq.artemis] AMQ241002: Artemis Jolokia REST API available at
http://localhost:8161/console/jolokia
[org.apache.activemq.artemis] AMQ241004: Artemis Console available at http://
localhost:8161/console
```

- 2. Open JBoss Developer Studio and import the starter Maven project.

- 2.1. Open JBoss Developer Studio by double-clicking the JBoss Developer Studio icon on the workstation desktop. JBoss Developer Studio requests a default workspace. Leave the default configuration (**/home/student/workspace**) and click **Launch**.

- 2.2. Import the Maven project by selecting from the JBoss Developer Studio main menu **File → Import**.
- 2.3. From the **Import** dialog box, select **Maven → Existing Maven Projects** and click **Next >**.
- 2.4. Click **Browse** and choose **/home/student/JB421/labs/aggregate-messages** and click **OK** to choose the Maven project. Click **Finish** to complete the import process in JBoss Developer Studio.
A new project named **aggregate-messages** will be listed in the **Project Explorer** view.

► 3. Review the project contents.

- 3.1. In the **Project Explorer** view, expand **aggregate-messages → src/main/java → com.redhat.training.jb421**. Double-click the **AggregateRouteBuilder.java** file.

```
from("activemq:queue:orderInput?username=admin&password=admin")
    .marshal().jaxb()
    // TODO add aggregate by header("orderType") here using
    // OrderBatchAggregationStrategy()

    // TODO add xslt using provided style sheet for updating
    // extPrice and price to include the $ symbol

    .log("${body}")
    .to("mock:fulfillmentSystem");
```

The route listens on the **orderInput** queue for new orders, then marshals the orders to XML with JAXB, logs the body to the console, and then forwards that XML on to a mock endpoint representing the fulfillment system.

- 3.2. Review the model classes.

In the **Project Explorer** view, expand **aggregate-messages → src/main/java → com.redhat.training.jb421.model**. Double-click the **Order.java** file.

```
@XmlRootElement
public class Order implements Serializable {

    private static final long serialVersionUID = 1L;

    private Integer id;
    private Date orderDate = new Date();
    private BigDecimal discount;
    private Boolean delivered=false;
    private Customer customer;
    @XmlElementWrapper(name="orderItems")
    @XmlElement(name="orderItem")
    private List<OrderItem> orderItems = new ArrayList<OrderItem>();

    ...
}
```

Order is the model class received by the consumer at the beginning of our route. Notice it is sparsely annotated with JAXB annotations to dictate the specifics of the marshal or unmarshal to XML.

In the **Project Explorer** view, in the package **com.redhat.training.jb421.model** double-click **OrderBatch.java** file.

```
@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class OrderBatch implements Serializable{

    private static final long serialVersionUID = 1L;

    @XmlElementWrapper(name="orders")
    @XmlElement(name="order")
    private List<Order> orders;
    private String orderType;
```

OrderBatch serves as a wrapper around a list of orders, also storing the **orderType** as you will aggregate batches based on this value in the exchange header.

3.3. Review the XSL style sheet.

In the **Project Explorer** view, expand **src/main/resources** and double-click **CurrencySymbol.xsl** to review the style sheet.

```
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:template match="*">
        <xsl:copy>
            <xsl:copy-of select="@*"/>
            <xsl:apply-templates/>
        </xsl:copy>
    </xsl:template>
    <xsl:template name="price"
        match="orders/order/orderItems/orderItem/item/price/text()">
        <xsl:value-of select="concat('$',.)"/>
    </xsl:template>
    <xsl:template name="extPrice"
        match="orders/order/orderItems/orderItem/extPrice/text()">
        <xsl:value-of select="concat('$',.)"/>
    </xsl:template>
</xsl:stylesheet>
```

This XSL style sheet applies a simple transformation to concatenate a dollar (\$) sign in front of the **price** and **extPrice** fields.

► 4. Combine the orders into batches.

4.1. Create a new Java class for the aggregation strategy.

To create a new class, in the **Project Explorer** view, right-click the **com.redhat.training.jb421** package inside **src/main/java** folder and select **New → Class**.

In the **New Class** wizard, use the following values:

- **Class name:** **OrderBatchAggregationStrategy**
- **Modifiers:** leave the default value.

- **Superclass:** leave the default value.

For **Interfaces**, click **Add** and another window will appear. In the **Choose interfaces** text box, type **AggregationStrategy** until you see that interface appear in the list, then double-click the **AggregationStrategy** entry to add it to the list and then click **Ok**.

Click **Finish** to complete the new class creation.

4.2. Update the contents of the newly created aggregation strategy.

Update the generated body of the **aggregate** method from:

```
public Exchange aggregate(Exchange oldExchange, Exchange newExchange) {
    // TODO Auto-generated method stub
    return null;
}
```

to:

```
public Exchange aggregate(Exchange oldExchange, Exchange newExchange) {
    Message newIn = newExchange.getIn();
    Order newBody = newIn.getBody(Order.class);
    OrderBatch batch = null;
    if (oldExchange == null) { ①
        batch = new OrderBatch();
        batch.getOrders().add(newBody);
        batch.setOrderType(newIn.getHeader("orderType", String.class)); ②
        newIn.setBody(batch);
        return newExchange;
    } else { ③
        Message in = oldExchange.getIn();
        batch = in.getBody(OrderBatch.class);
        batch.getOrders().add(newBody);
        return oldExchange;
    }
}
```

- ① If the **oldExchange** parameter is **null** then this must be the first order in a new batch. In that case, you create a new **OrderBatch** object, add the incoming **Order** to the batch, and then update the body to contain the **OrderBatch** you just created.
- ② Set the order type from the **header** on the **newIn** which was attached to the **newExchange**.
- ③ If **oldExchange** is not null, then this is not the first order in a new batch, and you can get the existing **OrderBatch** from the **oldExchange**, and add the latest order to it.

The **aggregate-method.txt** is provided with the method source code. Import the missing classes using the JBoss Developer Studio shortcut **Ctrl+Shift+O**.

**Important**

Make sure you import the correct **Order** and **Message** class implementations, these include **com.redhat.training.jb421.model.Order** and **org.apache.camel.Message**.

- 5. Aggregate **Order** instances with completion interval of five seconds.

Returning to editing **AggregateRouteBuilder.java**. Look for the comment
//TODO add aggregate by header("orderType") here using OrderBatchAggregationStrategy() and replace it with the following DSL snippet:

```
.aggregate(header("orderType") ①, new OrderBatchAggregationStrategy() ②)
.completionInterval(BATCH_COMPLETION_INTERVAL) ③
```

- ① **header("orderType")** is the correlation value for this aggregation, aggregating messages that have the same value in their exchange header named **orderType**.
- ② **OrderBatchAggregationStrategy** is the aggregation strategy that is used by Camel to determine exactly how to aggregate two exchanges.
- ③ **completionInterval** is used to specify how long camel should add orders to each batch before sending those batches through the remainder of the route. In this case, **BATCH_COMPLETION_INTERVAL** is defined to be 5000 ms or five seconds.

Save your updates to the route definition using **Ctrl+S**.

- 6. Format the output from the route with XSLT.

Edit **AggregateRouteBuilder.java**. Look for the comment **//TODO add xslt using provided style sheet for updating extPrice and price to include the \$ symbol** and replace it with the following DSL snippet:

```
.to("xslt:CurrencySymbol.xsl")
```

This uses the **xslt** endpoint to run the XML through the **CurrencySymbol.xsl** file you reviewed previously to update the **price** and **extPrice** fields to contain the dollar (\$) sign at the beginning of their values.

At this point, your route should resemble the following:

```
from("activemq:queue:orderInput?username=admin&password=admin")
.marshal().jaxb()
//TODO add aggregate by header("orderType") here using
OrderBatchAggregationStrategy()
.aggregate(header("orderType"), new OrderBatchAggregationStrategy())
.completionInterval(BATCH_COMPLETION_INTERVAL)
//TODO add xslt using provided style sheet for updating extPrice and price to
include the $ symbol
.to("xslt:CurrencySymbol.xsl")
.log("${body}")
.to("mock:fulfillmentSystem");
```

Save your updates to the route definition using **Ctrl+S**.

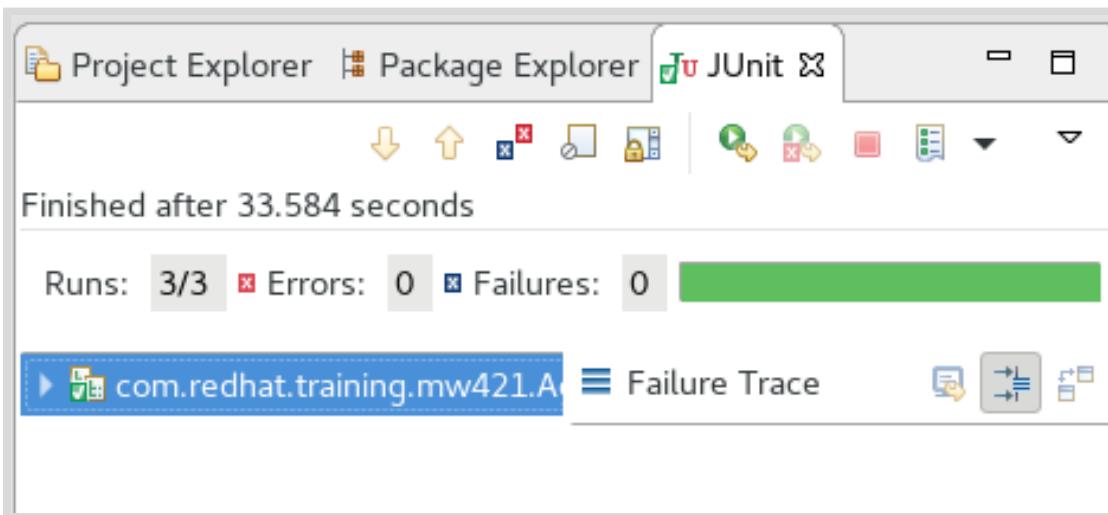
- 7. Run the provided JUnit test to verify your route.

- 7.1. Run the unit test.

Right-click the **com.redhat.training.jb421.AggregateRouteTest** class and select **Run As → JUnit Test** to run the test.

- 7.2. Check the JUnit test outcome.

Open the **JUnit** tab, if all tests passed, it resembles the following screen shot:



- 7.3. Review the console log.

Open the **Console** tab in the bottom pane of JBoss Developer Studio, if the route is functioning properly it ends with the following text output:

```
INFO Testing done:  
testDroppingMixedOrders(com.redhat.training.jb421.AggregateRouteTest)  
INFO Took: 11.038 seconds (11038 millis)
```

► 8. Close the project and stop the Red Hat AMQ Broker.

- 8.1. In the **Project Explorer** view, right-click **aggregate-messages** and click **Close Project**.
- 8.2. Return to the terminal window where Red Hat AMQ is running, and press **Ctrl+C** to stop it.

This concludes the guided exercise.

Accessing Databases with Camel Routes

Objectives

After completing this section, students should be able to:

- Use the **jdb**, **jpa** and **sql** components in Camel to retrieve data from, or persist data into an external database.
- Schedule route execution using the **timer** and **quartz** components.

Accessing Databases Using the JDBC, JPA and SQL Components

A common use case in enterprise integration is retrieving data from a relational database such as MySQL, Oracle database, or others. Often when building an integration system you may even need to retrieve, store or update data in multiple different databases. Camel implements multiple components that allow access to your relational database technologies:

JDBC

Provided by the **camel-jdbc** library. A SQL query can be executed against a database using JDBC (Java Database Connectivity). The response message contains the full result of the query.

SQL

Provided by the **camel-sql** library. Allows you to work with databases using JDBC queries. The difference between this component and **jdbc** component is that in case of SQL the query is a property of the endpoint and it uses message payload as parameters passed to the query.

Java Persistence API

Provided by the **camel-jpa** library. Similar to Hibernate JPA implementation, it can be used to manage database data using an ORM (Object Relational Mapping) layer. ORM tools provide an easy way to map your database tables to your model classes, and translate operations done on your model classes directly to SQL for you, drastically simplifying your application code for database communication.

These components are useful to capture data to be used by other systems, enhance message exchanges with data from a database, or to keep an external data store based on incoming message exchanges. These components simplify access to databases from your Camel routes and provide an easy-to-use mechanism to transfer data between systems.

Reviewing the JDBC Component

The **jdb** component enables you to access databases through Java Database Connectivity (JDBC) using SQL queries (**SELECT**) or operations (**INSERT**, **UPDATE**, etc). The **jdb** component expects the message body to contain SQL that it can execute against your database.

To use it in a Maven-based project, a dependency must be declared:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-jdbc</artifactId>
</dependency>
```

The URI syntax when using JPA component is: `jdbc:[dataSourceName][?options]`.

The **dataSourceName** parameter is required and refers to a data source defined in the Spring configuration file. An example data source configuration is:

```
<bean id="dataSource"
    class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="${jdbc.driverClassName}"/>
    <property name="url" value="${jdbc.url}"/>
    <property name="username" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
</bean>

<context:property-placeholder location="jdbc.properties"/>
```

The **jdbc** component supports a large number of options in the URI. Some commonly used options are included in the following table:

JDBC Options

Option name	Default value	Description
statement.<xxx>	null	Sets additional options on the java.sql.Statement that is used behind the scenes to execute the queries. For example, statement.maxRows=10 . For detailed documentation, see the java.sql.Statement Javadoc documentation.

Option name	Default value	Description
outputType	SelectList	<p>outputType='SelectList', for consumer or producer, outputs a List of Map. outputType='SelectOne' outputs single Java object in the following way:</p> <ul style="list-style-type: none"> If the query result contains only a single column (such as SELECT COUNT(*) FROM PROJECT), then the route returns a Long object. If the query has more than one column, then it returns a Map of that result. If the outputClass is set, then it converts the query result into a Java bean object by calling all the setters that match the column names. It assumes your class has a default constructor to create instance with. If the query resulted in more than one row, it throws a non-unique result exception.
outputClass	null	Set the full package and class name you want the camel-jdbc component must convert the results to when the outputType is set to SelectOne .

By default, the result is returned in the **Out** body as an instance of **ArrayList<HashMap<String, Object>>**. The **List** object contains the list of rows and the **Map** objects contain each row with the **String** map key as the column name and the **Object**map value as the column contents. You can use the option **outputType** to control the result.



Note

The **jdbc** component only supports producer endpoints.

Reviewing the SQL Component

The **sql** component also enables access to databases through JDBC using native SQL queries. Unlike the **jdbc** component, this component does not expect the query in the body of the message, but rather it expects the query to be defined in the endpoint URI, and the message content can be used to specify parameters of the query dynamically. Additionally this component uses **spring-jdbc** internally instead of plain JDBC which is leveraged by the **jdbc** component.

To use it in a Maven-based project, a dependency must be declared:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-sql</artifactId>
</dependency>
```

The URI syntax when using the **sql** component is: `sql:select * from table where id=:#myId order by name[?options]`.

In that example URI, the query is a simple SQL **SELECT** query, and Camel will attempt to load the **myId** parameter from the message body if it is of type **Map** otherwise, it will look for a header on the exchange with the same name. If the named parameter cannot be resolved, an exception is thrown.

The **sql** component supports a few commonly used options, included in the following table:

SQL Component Options

Option name	Default value	Description
dataSource		Sets the DataSource to use to communicate with the database.
outputType	SelectList	<p>outputType='SelectList', for consumer or producer, outputs a List of Map. outputType='SelectOne' outputs single Java object in the following way:</p> <ul style="list-style-type: none"> If the query result contains only a single column (such as SELECT COUNT(*) FROM PROJECT), then the route returns a Long object. If the query has more than one column, then it returns a Map of that result. If the outputClass is set, then it converts the query result into a Java bean object by calling all the setters that match the column names. It assumes your class has a default constructor to create instance with. If the query resulted in more than one row, it throws a non-unique result exception.
outputClass	null	Set the full package and class name you want the camel-sql component must convert the results to when the outputType is set to SelectOne .

Reviewing the JPA Component

The Java Persistence API defines an *object relational mapping* (ORM) library that allows access to a database using a unique set of commands that the JPA library translates to SQL for you. JPA

was designed with flexibility in mind and also allows low-level access under certain circumstances; for example, to use specific features (functions and procedures) from a database.

JPA is implemented by a set of providers such as Hibernate, OpenJPA, and Toplink, and it has been increasingly enhanced to improve performance and embrace database-specific functionalities.

Camel implements a component that can access a database using JPA, and then use this data in route processing. Access is provided using an entity-based approach, and data can be added to or extracted from a database.

To use the **jpa** component in a Maven-based project, declare the following dependency:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-jpa</artifactId>
</dependency>
```

The URI syntax when creating an endpoint which uses the JPA component is: **jpa : [entityClassName] [?options]**.

The **entityclassname** option is required to retrieve data from a JPA producer, but is optional when inserting data with a JPA consumers.

JPA Options

Option name	Default value	Description
persistenceUnit	camel	The persistence unit name.
consumeDelete	true	If true , the entity is deleted after it is consumed.
consumeLockEntity	true	Specifies whether or not to set an exclusive lock on each entity bean while processing the results from polling.
maximumResults	-1	JPA consumer only: Set the maximum number of results to retrieve on the query.
consumer.namedQuery	N/A	To use a named query when consuming data.

For example, a JPA producer can be used to generate a serialized object from a database table for the **Order** entity. The following example route, consumes rows from the **Order** table, transforms the rows to Java objects, unmarshals those objects to XML using JAXB and writes the resulting rows to files in the **out** directory:

```
from("jpa:com.redhat.training.entity.Order")
    .unmarshal().jaxb()
    .to("file:out");
```

To avoid removal of rows from a table, use the **consumeDelete** option:

```
from("jpa:com.redhat.training.entity.Order?consumeDelete=false")...
```

To specify a persistence unit, other than the default, use the **persistenceUnit** option. The persistence unit is associated to a **persistence.xml** file in your Java project, and provides the connections to your database:

```
from("jpa:com.redhat.training.entity.Order?  
consumeDelete=false&persistenceUnit=mysql")...
```

To identify the persistence unit name from a JPA-based project, evaluate the **persistence.xml** file:

```
<?xml version="1.0" encoding="UTF-8"?>  
<persistence version="2.0"  
    xmlns="http://java.sun.com/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/  
    XMLSchema-instance"  
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/  
    xml/ns/persistence/persistence_2_0.xsd">  
    <persistence-unit name="mysql①" transaction-type="RESOURCE_LOCAL">  
        ...  
    </persistence-unit>  
</persistence>
```

① The persistence unit name.

Scheduling Routes with Timers and Quartz

Often companies decide to defer the execution of batch processes to after regular business hours to avoid any issues of high-load taking down systems and impacting end users during their work day. Most of these batch processes are triggered by the OS or proprietary systems, but they do not provide a fine-grained mechanism to manage failures and retries.

Camel implements timer components to allow developers to trigger route processing at any time, at regular intervals, without external dependencies. This feature is implemented using two main libraries:

Java native timer features using the **timer** component

Implemented by the **camel-core** library, it uses basic timer features from the Java APIs.

Quartz framework features using the **camel-quartz** library

Implemented by the **camel-quartz** library, it allows advanced execution features, such as cron-based syntax and management of the number of executions.

Both components only support being used as a producers by their nature, and cannot be called within a **to** method.

Timer Component

The timer component is provided by the camel-core library and requires the following dependency in the **pom.xml** file:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-core</artifactId>
</dependency>
```

The endpoint URI format is: `timer:<timerName>?options`.

This timer creates an empty **exchange** that should be filled by a developer or during the route processing. The timer customization can be achieved using the included options.

Timer Options

Option name	Default value	Description
period	1000	Time in milliseconds between route executions. The time can also be specified using s (seconds), m (minutes), or h (hours). For example, an entry of 1h will be transformed by Camel to 3,600,000 ms.
delay	0	Length of time that Camel should wait until the first execution.

In the following URI, the route is triggered each hour, starting as soon as the Camel context is started:

```
from("timer:hourlyTimer?period=1h")
```

The **camel-timer** component does not provide a way to start processes at a certain time.

Quartz Component

Quartz is a timer execution framework which supports more advanced scheduling using CRON, which is an enterprise standard.

To use camel-quartz in a project, it must be imported by the **pom.xml** file:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-quartz</artifactId>
</dependency>
```

The endpoint URI format is: `quartz:<timerName>?options`.

Similar to the timer, it is an empty exchange that should be filled by a developer or during the route processing. Its customization can be achieved using the included options. The component uses either a **CronTrigger** or a **SimpleTrigger**. If no CRON expression is provided, the component uses a simple trigger.

**Important**

If you specify a CRON expression using the **cron** option, you cannot use the trigger options simultaneously.

Quartz Options

Option name	Default value	Description
cron	NA	Specifies a CRON expression to schedule route executions.
trigger.repeatCount	0	Number of times this route should be executed.
trigger.repeatInterval	0	The amount of time in milliseconds between repeated triggers.

For example the following quartz timer uses a CRON expression to fire a message every five minutes from 12pm to 6pm only on weekdays:

```
from("quartz://myGroup/myTimerName?cron=0+0/5+12-18+?+*+MON-FRI")
.to("activemq:startProcess");
```

**References****camel-jdbc**

<https://camel.apache.org/jdbc.html>

camel-sql

<http://camel.apache.org/sql.html>

camel-jpa

<http://camel.apache.org/jpa.html>

Timer

<http://camel.apache.org/timer.html>

Quartz Framework

<http://www.quartz-scheduler.org>

camel-quartz

<http://camel.apache.org/quartz.html>

Camel in Action, Second Edition - Chapter Enterprise integration patterns

Camel in Action, Second Edition - Chapter Transforming Data with Camel

► Guided Exercise

Scheduling Routes that Access a Database

In this exercise, you will use the **camel-jpa** and **camel-sql** components to read and write data in a relational database.

Outcomes

You should be able to implement a route that retrieves order data from the database using the **jpa** component, receiving one **Order** object every three seconds.

Before You Begin

A starter project is provided for you, and includes a **CamelContext** configuration file, a **RouteBuilder** implementation, and an integration test so that you can verify your work.

Run the following command to download the starter project and data files used by this exercise:

```
[student@workstation ~]$ lab schedule-database setup
```

Steps

- 1. Open JBoss Developer Studio and import the starter Maven project.
 - 1.1. Open JBoss Developer Studio (**Applications** → **Programming** → **JBoss Developer Studio**). JBoss Developer Studio will request a default workspace. Keep the default configuration (**/home/student/workspace**) and click **Launch**.
 - 1.2. Import the Maven project by selecting from the JBoss Developer Studio main menu **File** → **Import**.
 - 1.3. From the **Import** dialog box, select **Maven** → **Existing Maven Projects** and click **Next >**.
 - 1.4. Click **Browse** and choose **/home/student/JB421/labs/schedule-database** and click **Ok** to choose the Maven project. Click **Finish** to complete the import process in JBoss Developer Studio.
A new project named **schedule-database** will be listed in the **Project Explorer** view.



Important

The project imports with errors, but this is expected. You will resolve these errors in the subsequent steps of this exercise.

- 2. Review project contents.

- 2.1. In the **Project Explorer** view, expand **schedule-database** → **src/main/resources** → **META-INF/spring**. Double-click **bundle-camel-context.xml** to review the Spring configuration for the starter project.

Notice the usual Camel context defined with a single route builder. Also notice the data source defined as **mysqlDataSource**; this is the data source you need to connect to using the **jpa** component. The database necessary for this exercise is already running and populated with the necessary reference data.

- 2.2. In the **Project Explorer** view, expand **src/main/java** → **com.redhat.training.jb421.model**. Double-click the **Order.java** file.

```
...  
@Entity  
 @Table(name = "order_")  
 @NamedQuery(name="getUndeliveredOrders", query="select o"  
 + "from Order o where o.delivered = false")  
 public class Order implements Serializable {  
 ...
```

The **Order** model class is annotated with standard JPA annotations, and it will be the entity that needed to retrieve the order data from the database. Notice specifically the **@NamedQuery** as this is the query you use to pull unfulfilled orders.

- 2.3. In the **Project Explorer** view, expand **src/main/java** → **com.redhat.training.jb421**. Double-click the **DatabaseRouteBuilder.java** file.

Notice that the route definition is incomplete; as you progress through the exercise the route is going to be fixed.

► 3. Pull orders from the database every three seconds.

- 3.1. Add the **camel-jpa** dependency.

Return to the **Project Explorer** view and open the **pom.xml** file.

Look for the following comment:

```
<!--TODO Add camel-jpa dependency here -->
```

Replace it with the following dependency definition:

```
<dependency>  
 <groupId>org.apache.camel</groupId>  
 <artifactId>camel-jpa</artifactId>  
</dependency>
```

- 3.2. Press **Ctrl+S** to save your updates to the **pom.xml**.

- 3.3. Return to editing the **DatabaseRouteBuilder** class, add the following DSL at the beginning of the **configure** method, after the comment **//TODO use jpa consumer**, replace the existing mock consumer with a JPA consumer to implement the required database look up:



Note

The **jpa-route.txt** file is provided to minimize typing.

```
from("jpa:com.redhat.training.jb421.model.Order?"
    + "persistenceUnit=mysql" 1
    + "&consumeDelete=false" 2
    + "&consumer.namedQuery=getUndeliveredOrders" 3
    + "&maximumResults=1" 4
    + "&consumer.delay=3000" 5
    + "&consumeLockEntity=false" 6)
```

- 1** References the name of the persistence unit for the **jpa** component to use, this was defined in the Spring XML data source configuration.
- 2** Tells the consumer not to delete the rows as they are consumed.
- 3** Refers to the named query on the **Order** entity the consumer should use when executing.
- 4** Controls the maximum number of results the consumer should return.
- 5** Controls how often the consumer will poll the database, in milliseconds.
- 6** Prevents the **jpa** consumer from locking the **Order** rows as they are consumed. This is necessary because later in the route you are going to update the **Order** rows.

Press **Ctrl+S** to save your updates to the route builder.

▶ **4.** Transform **Order** objects by populating a **fulfilledBy** value and a **fulfilledDate** value and set the **orderId** value as an exchange header.

4.1. Create a new Java class for the custom processor.

In the **Project Explorer** view, right-click the *com.redhat.training.jb421* package and click **New → Class** to launch the new class creation wizard. Enter **OrderFulfillmentProcessor** as the class name and click **Finish** to create the class.



Note

The **processor.java.txt** file is provided for copy and paste.

Once the class is open for editing, update its contents to match the following:

```
package com.redhat.training.jb421;

import java.util.Date;

import org.apache.camel.Exchange;
import org.apache.camel.Processor;

import com.redhat.training.jb421.model.Order;

public class OrderFulfillmentProcessor implements Processor{

    @Override
    public void process(Exchange exchange) throws Exception {
        Order incoming = exchange.getIn().getBody(Order.class);
        exchange.getIn().setHeader("orderId", incoming.getId());
        incoming.setDateFulfilled(new Date());
```

```
    incoming.setFulfilledBy("admin");
}
}
```

This class implements the **Processor** interface provided by Camel and overrides the **process** method defined in that interface. This processor transforms the **Order** by setting two values, **dateFulfilled** and **fulfilledBy** on the **Order**. Additionally the **orderId** is set as an exchange header for easier processing later in the Camel route.

Press **Ctrl+S** to save your changes.

4.2. Add a call to the new processor.

Return to editing the **DatabaseRouteBuilder** class and after the comment **// TODO add processor** add the following DSL to call the processor you just created to transform the order data.

```
.process(new OrderFulfillmentProcessor())
```

4.3. Press **Ctrl+S** to save your changes to the route definitions.

▶ 5. Include a second direct route to update the database using the **camel-sql** component.

5.1. Add the **camel-sql** dependency.

Return to the **Project Explorer** view and open the **pom.xml** file.

Look for the following comment:

```
<!--TODO Add camel-sql dependency here -->
```

Replace it with the following dependency definition:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-sql</artifactId>
</dependency>
```

Press **Ctrl+S** to save your updates to the **pom.xml** file.

5.2. Return to editing the **DatabaseRouteBuilder** class and after the comment **// TODO add sql producer** add the following DSL to call the processor you just created to transform the order data.

```
from("direct:updateOrder")
    .log("Order delivered: ${header.orderId}")
    //TODO add sql producer
    .to("sql:update order_ set delivered = 1 where id=:#orderId");
```

This DSL uses the **camel-sql** component to run a SQL query to update the row in **order_** table for each order that it processes, using its **orderId** value. The **:#orderId** placeholder is automatically replaced by the exchange header with the same name as a feature of the **camel-sql** component.

5.3. Press **Ctrl+S** to save your changes to the route definitions.

▶ 6. Evaluate the routes with some test data.

- 6.1. Use the Camel Maven plug-in to start the route.

Open a new terminal window (**Applications** → **System Tools** → **Terminal**) and execute the following commands:

```
[student@workstation ~]$ cd JB421/labs/schedule-database/  
[student@workstation schedule-database]$ mvn camel:run -DskipTests
```

The following log is displayed once the route is started:

```
INFO Total 2 routes, of which 2 are started.  
INFO Apache Camel 2.21.0.fuse-000077-redhat-1 (CamelContext: jb421Context)  
started in 0.464 seconds
```

- 6.2. Create two test orders in the system.

Open a new terminal window (**Applications** → **System Tools** → **Terminal**) and execute the following commands:

```
[student@workstation ~]$ cd ~/JB421/labs/schedule-database/  
[student@workstation schedule-database]$ sh setup-data.sh
```

You should see the following output:

```
Resetting database data...  
Done!  
Sending two test orders...  
Orders sent!
```

- 6.3. Check the output in the Camel log.

Return to the terminal window where Maven is running the routes. You should see something similar to the following output if the orders were processed successfully. Note that this output has been truncated for brevity's sake.

```
INFO Order sent to fulfillment: 1  
INFO Order delivered: 1  
INFO Order sent to fulfillment: 2  
INFO Order delivered: 2
```

- 6.4. Stop the Maven Camel plug-in execution.

In the terminal window running Camel, press **Ctrl+C** to end the execution of the plug-in.

▶ 7. Close the project.

In the **Project Explorer** view, right-click **schedule-database** and click **Close Project**.

This concludes the guided exercise.

► Lab

Transforming Data

In this lab, you will complete an integration project using a Camel route that reads order data from a database, splits that order data into separate order line items and then aggregates those line items into a reservation XML document based on their catalog item ID. The route should then output the reservation XML, sorting the files so that each catalog item ID's reservation is written to a separate file on the local file system and update the orders in the database as "delivered".

Outcomes

After completing the lab, you should be able to read data from a database using the JPA component, marshal order data to and from a variety of data formats, and implement the wiretap, splitter, and aggregator enterprise integration patterns, all using built-in Camel components.

Before You Begin

A starter project is provided as a starter including annotated (JAXB and JPA annotations included) model classes and a preconfigured Camel context including the necessary data source configuration and a stubbed route builder. The starter project includes a unit test to check if the generated correct number of reservation files are created by the route in the correct locations.

- Run the following command to download the starter project used by this lab:

```
[student@workstation ~]$ lab transform-data setup
```

- Import the starter project into JBoss Developer Studio that is located at **/home/student/JB421/labs/transform-data**.

Steps

- Inspect the Spring configuration file in the provided starter project.

From the **Project Explorer** view, expand **src/main/resources → /META-INF/spring** and open the **bundle-camel-context.xml** file. Select the **Source** tab and look at the components it defines, including a persistence unit named **mysql**.

Also, review the model classes, located in the *com.redhat.training.jb421.model* package, specifically the **Order** class, which you will use to retrieve orders from the database, and the **Reservation** class, which you will use to aggregate order data.

- Process all undelivered orders every 10000 milliseconds.

In the **TransformRouteBuilder** route builder, create a consumer endpoint that uses JPA to read order data for undelivered orders from the bookstore database every 10000 ms. The number of milliseconds is provided as the variable **BATCH_TIMEOUT**.

Hint: Be sure to keep Camel from deleting the order records as it consumes them, or locking the rows, as you will be updating them as delivered later in the route.

Hint: The **Order** class provides a named query **getUndeliveredOrders** which can be used to retrieve all the current undelivered orders in the database.



Warning

You will see compilation errors in JBoss Developer Studio during some of these steps as the route is developed. These errors are safe to ignore until the consumer for the route is defined and the Java DSL statement is terminated with a semicolon (;), later in this lab.

3. Marshal the order data retrieved to XML using JAXB.

4. Divide the orders into separate order items.

Hint: The XPath expression **order/orderItems/orderItem** will return each of the **orderItem** XML elements from the **order** XML element separately.

5. Review the **Reservation** model class and the **ReservationAggregationStrategy** implementation.

```
@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class Reservation implements Serializable{

    private static final long serialVersionUID = 1L;

    private Integer id;

    private Integer catalogItemId;

    private Integer quantity;

    private Date reservationDate;
```

Notice the **Reservation** model class represents a reservation of inventory for a given catalog item.

```
public Exchange aggregate(Exchange oldExchange, Exchange newExchange) {
    OrderItem newBody = newExchange.getIn().getBody(OrderItem.class);
    Reservation reservation = null;
    if (oldExchange == null) {
        reservation = new Reservation();
        reservation.setCatalogItemId(newBody.getCatalogItem().getId());
        reservation.setReservationDate(new Date());
        reservation.setQuantity(newBody.getQuantity());
        newExchange.getIn().setBody(reservation);
        return newExchange;
    } else {
        reservation = oldExchange.getIn().getBody(Reservation.class);
```

```
        reservation.setQuantity(reservation.getQuantity() +  
newBody.getQuantity());  
        return oldExchange;  
    }  
}
```

Notice the **ReservationAggregationStrategy** implementation either creates a new **Reservation** object or updates the quantity of an existing reservation.

6. Aggregate order items into a reservation.

Add aggregation of the order items by catalog item id to the route. Use an XPath expression to specify the catalog item id as the aggregation correlation value, and use the **ReservationAggregationStrategy** class that was provided for you to aggregate the order items into a **Reservation** object by incrementing the quantity on the reservation for each incoming order item.

Hint: Be sure that the **completionInterval** value matches the delay on the JPA producer so that the aggregator creates new reservations for each batch of orders retrieved from the database. Also make sure any remaining order data is processed when the route is stopped.

Hint: The XPath expression **orderItem/catalogItem/id** returns the catalog item ID of the **orderItem** XML elements.

7. Log the reservation XML data created by the route to the console for debugging purposes. Add the **log** DSL method, using the Simple expression language (Simple EL) to log the body of the exchange.
8. Output all reservations to a single output folder.

Create a producer endpoint that creates a new XML file on the local file system using the **OUTPUT_FOLDER** variable provided as the top-level folder name. Camel should output one XML file for each reservation.

Use the catalog item id associated with each reservation as the folder name to sort the files and include a time stamp in the file name of each reservation XML file.

The catalog item id must be stored in the header with the **CatalogItemId** key.



Note

You can set a header on the exchange using an XPath expression to reference data from the XML. An example would be:

```
.setHeader("HeaderName", xpath(/path/to/element/text()))
```

You can use an exchange header in a dynamic URI using Simple EL. Simple EL also supports dynamic inserting of dates and times: An example of using both of the functionalities at the same time would be:

```
.to("file:outgoing"?fileName=${header.headerName}/order-$date:now:yyyy-MM-  
dd_HH-mm-ss}.xml")
```

9. Update the orders as delivered that were consumed using a wire tap.

Use a **wireTap** to create a second route to update the order in the database as "delivered". Implement this wire tap directly following the JPA consumer that retrieves the order data from the database.

Because the JPA consumer retrieves all currently undelivered orders, updating the orders as delivered after they are consumed is necessary to prevent retrieving the same order multiple times.



Note

The approach taken in this lab of using a wire tap to update the database could create data integrity issues. If an exception occurred during the route execution after the wire tap had already updated the order as delivered in the database, there is potential for an order to be marked as delivered without a proper reservation XML corresponding to it.

Hint: The new route should use the JPA component to update the order records as delivered. The **DeliverOrderProcessor** is provided to update the Java **Order** instances as delivered before they are sent to the JPA component.

10. Verify the route by running the tests with Maven using the terminal.

11. Grade your work. Execute the following command:

```
[student@workstation transform-data]$ lab transform-data grade
```

If you do not get a **PASS** grade, review your work. The **~/JB421/labs/transform-data/grade.log** file contains the Maven messages produced by the grading process. This may be helpful in debugging any issues.

12. Clean up.

Close the **transform-data** project in JBoss Developer Studio to conserve memory.

This concludes the lab.

► Solution

Transforming Data

In this lab, you will complete an integration project using a Camel route that reads order data from a database, splits that order data into separate order line items and then aggregates those line items into a reservation XML document based on their catalog item ID. The route should then output the reservation XML, sorting the files so that each catalog item ID's reservation is written to a separate file on the local file system and update the orders in the database as "delivered".

Outcomes

After completing the lab, you should be able to read data from a database using the JPA component, marshal order data to and from a variety of data formats, and implement the wiretap, splitter, and aggregator enterprise integration patterns, all using built-in Camel components.

Before You Begin

A starter project is provided as a starter including annotated (JAXB and JPA annotations included) model classes and a preconfigured Camel context including the necessary data source configuration and a stubbed route builder. The starter project includes a unit test to check if the generated correct number of reservation files are created by the route in the correct locations.

- Run the following command to download the starter project used by this lab:

```
[student@workstation ~]$ lab transform-data setup
```

- Import the starter project into JBoss Developer Studio that is located at **/home/student/JB421/labs/transform-data**.

Steps

- Inspect the Spring configuration file in the provided starter project.

From the **Project Explorer** view, expand **src/main/resources → /META-INF/spring** and open the **bundle-camel-context.xml** file. Select the **Source** tab and look at the components it defines, including a persistence unit named **mysql**.

Also, review the model classes, located in the *com.redhat.training.jb421.model* package, specifically the **Order** class, which you will use to retrieve orders from the database, and the **Reservation** class, which you will use to aggregate order data.

- Process all undelivered orders every 10000 milliseconds.

In the **TransformRouteBuilder** route builder, create a consumer endpoint that uses JPA to read order data for undelivered orders from the bookstore database every 10000 ms. The number of milliseconds is provided as the variable **BATCH_TIMEOUT**.

Hint: Be sure to keep Camel from deleting the order records as it consumes them, or locking the rows, as you will be updating them as delivered later in the route.

Hint: The **Order** class provides a named query **getUndeliveredOrders** which can be used to retrieve all the current undelivered orders in the database.

Insert the following Java DSL in place of the comment **//TODO add jpa consumer** in the **com.redhat.training.jb421.TransformRouteBuilder** class:

```
from("jpa:com.redhat.training.jb421.model.Order?persistenceUnit=mysql"
+ "&consumeDelete=false"
+ "&consumer.namedQuery=getUndeliveredOrders"
+ "&consumer.delay="+BATCH_TIMEOUT
+ "&consumeLockEntity=false")
```



Warning

You will see compilation errors in JBoss Developer Studio during some of these steps as the route is developed. These errors are safe to ignore until the consumer for the route is defined and the Java DSL statement is terminated with a semicolon (;), later in this lab.

3. Marshal the order data retrieved to XML using JAXB.

Add the following Java DSL to the route definition in the **TransformRouteBuilder** class in place of the comment **//TODO marshal order to XML with JAXB**.

```
.marshal().jaxb()
```

4. Divide the orders into separate order items.

Hint: The XPath expression **order/orderItems/orderItem** will return each of the **orderItem** XML elements from the **order** XML element separately.

After the data is marshaled to XML, add the following Java DSL to the route definition in **TransformRouteBuilder** in place of the comment **//TODO split the order into individual order items**.

```
.split(xpath("order/orderItems/orderItem"))
```

5. Review the **Reservation** model class and the **ReservationAggregationStrategy** implementation.

```
@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class Reservation implements Serializable{

    private static final long serialVersionUID = 1L;

    private Integer id;

    private Integer catalogItemId;
```

```
private Integer quantity;

private Date reservationDate;
```

Notice the **Reservation** model class represents a reservation of inventory for a given catalog item.

```
public Exchange aggregate(Exchange oldExchange, Exchange newExchange) {
    OrderItem newBody = newExchange.getIn().getBody(OrderItem.class);
    Reservation reservation = null;
    if (oldExchange == null) {
        reservation = new Reservation();
        reservation.setCatalogItemId(newBody.getCatalogItem().getId());
        reservation.setReservationDate(new Date());
        reservation.setQuantity(newBody.getQuantity());
        newExchange.getIn().setBody(reservation);
        return newExchange;
    } else {
        reservation = oldExchange.getIn().getBody(Reservation.class);
        reservation.setQuantity(reservation.getQuantity() +
newBody.getQuantity());
        return oldExchange;
    }
}
```

Notice the **ReservationAggregationStrategy** implementation either creates a new **Reservation** object or updates the quantity of an existing reservation.

6. Aggregate order items into a reservation.

Add aggregation of the order items by catalog item id to the route. Use an XPath expression to specify the catalog item id as the aggregation correlation value, and use the **ReservationAggregationStrategy** class that was provided for you to aggregate the order items into a **Reservation** object by incrementing the quantity on the reservation for each incoming order item.

Hint: Be sure that the **completionInterval** value matches the delay on the JPA producer so that the aggregator creates new reservations for each batch of orders retrieved from the database. Also make sure any remaining order data is processed when the route is stopped.

Hint: The XPath expression **orderItem/catalogItem/id** returns the catalog item ID of the **orderItem** XML elements.

Add the following Java DSL to the route definition in **TransformRouteBuilder** in place of the comment **//TODO aggregate the order items based on their catalog item ID.**

```
.aggregate(xpath("orderItem/catalogItem/id"),
    new ReservationAggregationStrategy())
    .completionInterval(BATCH_TIMEOUT)
    .completeAllOnStop()
```

7. Log the reservation XML data created by the route to the console for debugging purposes. Add the `log` DSL method, using the Simple expression language (Simple EL) to log the body of the exchange.

After the aggregation, add the following Java DSL to the route definition in `TransformationRouteBuilder` in place of the comment `//TODO log the reservation XML to the console`.

```
.log("${body}")
```

8. Output all reservations to a single output folder.

Create a producer endpoint that creates a new XML file on the local file system using the `OUTPUT_FOLDER` variable provided as the top-level folder name. Camel should output one XML file for each reservation.

Use the catalog item id associated with each reservation as the folder name to sort the files and include a time stamp in the file name of each reservation XML file.

The catalog item id must be stored in the header with the `CatalogItemId` key.



Note

You can set a header on the exchange using an XPath expression to reference data from the XML. An example would be:

```
.setHeader("HeaderName", xpath(/path/to/element/text()))
```

You can use an exchange header in a dynamic URI using Simple EL. Simple EL also supports dynamic inserting of dates and times: An example of using both of the functionalities at the same time would be:

```
.to("file:outgoing"?fileName=${header.headerName}/order-${date:now:yyyy-MM-dd_HH-mm-ss}.xml")
```

After the log statement, add the following Java DSL to the route definition in `TransformationRouteBuilder`, in place of the comment `//TODO add file producer`.

```
.setHeader("CatalogItemId", xpath("/reservation/catalogItemId/text()"))
.to("file:"+OUTPUT_FOLDER+"?fileName=${header.CatalogItemId}"/
    + "reservation-${date:now:yyyy-MM-dd_HH-mm-ss}.xml");
```

9. Update the orders as delivered that were consumed using a wire tap.

Use a **wireTap** to create a second route to update the order in the database as "delivered". Implement this wire tap directly following the JPA consumer that retrieves the order data from the database.

Because the JPA consumer retrieves all currently undelivered orders, updating the orders as delivered after they are consumed is necessary to prevent retrieving the same order multiple times.



Note

The approach taken in this lab of using a wire tap to update the database could create data integrity issues. If an exception occurred during the route execution after the wire tap had already updated the order as delivered in the database, there is potential for an order to be marked as delivered without a proper reservation XML corresponding to it.

Hint: The new route should use the JPA component to update the order records as delivered. The **DeliverOrderProcessor** is provided to update the Java **Order** instances as delivered before they are sent to the JPA component.

Add the following Java DSL to the route definition in **TransformRouteBuilder** immediately following the JPA producer, replacing the comment **//TODO add wire tap to second route**.

```
.wireTap("direct:updateOrder")
```

Add the following Java DSL to create a new route to update the database in place of the comment **//TODO add second route to update order in the database**.

```
from("direct:updateOrder")
    .process(new DeliverOrderProcessor())
    .to("jpa:com.redhat.training.jb421.model.Order?persistenceUnit=mysql");
```

10. Verify the route by running the tests with Maven using the terminal.

From the terminal window, run the unit test using Maven to check if the route is working. The test takes some time to run, and sends a total of five orders, each with two order items with a total of three unique catalog items attached to them.

```
[student@workstation ~]$ cd JB421/labs/transform-data  
[student@workstation transform-data]$ mvn test
```

The **mvn: test** should produce output similar to the following:

Results:

```
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
```

The log output should show six reservation XML outputs similar to the following one:

```
<reservation>  
  <catalogItemId>21</catalogItemId>  
  <quantity>6</quantity>  
  <reservationDate>2018-07-18T00:56:36.902-05:00</reservationDate>  
</reservation>
```

11. Grade your work. Execute the following command:

```
[student@workstation transform-data]$ lab transform-data grade
```

If you do not get a **PASS** grade, review your work. The **~/JB421/labs/transform-data/grade.log** file contains the Maven messages produced by the grading process. This may be helpful in debugging any issues.

12. Clean up.

Close the **transform-data** project in JBoss Developer Studio to conserve memory.

This concludes the lab.

Summary

In this chapter, you learned:

- Camel supports data in a variety of formats and provides tools for working with or translating between these formats.
- Camel uses a set of type converters at runtime to automatically convert exchange bodies from one format to another. A set of these is provided by Camel, but customer type converters can also be created for automatic message translation from one type to another.
- The message translator pattern is implemented in Camel in a variety of ways, including **transform**, customer processors, XSLT transformation, and type conversion.
- The splitter pattern can be used in Camel and supports a variety of splitting mechanisms, including tokenization. When splitting large amounts of data, streaming can also be used to split the data more effectively.
- The aggregator pattern is provided in Camel, and a correlation expression is used to determine which messages should be aggregated. An implementation of the **AggregationStrategy** interface is used to combine all the message exchanges for a single correlation key into a single message exchange.
- Camel provides the **jdb**c and **jpa** components to support connectivity to external databases. Both of these components can retrieve and create data.

Chapter 4

Creating Tests for Routes and Error Handling with Camel

Goal

Develop reliable routes by developing route tests and handling errors.

Objectives

- Develop tests for Camel routes with Camel Test Kit.
- Create realistic test cases with mock components.
- Create reliable routes that handle errors gracefully.

Sections

- Testing Routes with Camel Test Kit (and Guided Exercise)
- Developing Routes with Mock Components (and Guided Exercise)
- Handling Errors in Camel (and Guided Exercise)

Lab

Testing and Error Handling with Camel

Testing Routes with Camel Test Kit

Objectives

After completing this section, students should be able to:

- Develop tests for Camel routes with Camel Test Kit
- Debug Camel routes defined in XML files with JBoss Developer Studio Integration Stack

Testing Camel Routes

An important aspect of developing reliable integration solutions is thoroughly testing your Camel routes. One difficulty when testing integration solutions is that the routes inherently require external dependencies. To resolve this issue, Camel provides a set of classes and libraries to test routes and change endpoints during runtime to avoid starting up all services and dependencies needed by a route, such as a database or a message queue. These libraries are provided in the following modules:

- **camel-test** provides a number of helpers for writing JUnit tests for Camel routes.
- **camel-test-spring** provides additional helpers for writing JUnit tests for routes using the Spring Framework or Java fluent interface (Java DSL).

Implementing Tests with Camel

A Camel-enabled test is created by extending one of the Camel Test Kit supporting classes. For example:

- **org.apache.camel.test.CamelTestSupport**: This allows test support to Camel routes and components. A number of helper methods are provided to test an **Endpoint** instance state and a **Predicate** instance filtering capabilities and for other route testing tasks.

Normally, the **CamelTestSupport** class is used to test simple routes that do not depend on services that need to be externally started and configured, such as message queues or databases. When these are used, it is preferred to use a subclass that integrates with an inversion of control (IoC) framework such as Spring or CDI.

- **org.apache.camel.test.spring.CamelSpringTestSupport**: This is a common approach to test Camel elements. The library extends **CamelTestSupport** to add helper methods to read route definitions from a Spring beans configuration file.

Another way to create a Camel-enabled test is by using one of the JUnit Runner implementations provided by the Camel Test Kit, for example, **org.apache.camel.test.spring.CamelSpringRunner**, which initializes the Camel context from a Spring beans configuration file. This approach is useful when you need to use JUnit extensions that require you to create a test subclass in addition to the **CamelSpringTestSupport** class.

**Note**

CamelTestSupport and related classes and annotations are part of the **camel-test** module. The subclasses featuring Spring framework integration, such as **CamelSpringTestSupport**, are part of the **camel-test-spring** module.

The following sample uses **CamelTestSupport** helper methods to test a route:

```
public class PredicatesTest extends CamelSpringTestSupport {❶

    @Override
    protected AbstractApplicationContext createApplicationContext() {❷
        return new ClassPathXmlApplicationContext
            ("META-INF/spring/bundle-context.xml");
    }

    @Test
    public void testRoute(){
        super.template.sendBodyAndHeader("file:in", "file", Exchange.FILE_NAME,
"testFile.txt");❸
        GenericFile receiveBody = (GenericFile)
super.consumer.receiveBody("file:out");❹
        String content = receiveBody.getFileNameOnly();
        assertEquals("testFile.txt",content);

    }
}
```

- ❶ Creates a test case that extends the **CamelSpringTestSupport** class to inherit helper methods such as start and stop, a Camel context instance, or attributes such as the **CamelContext** instance.
- ❷ Overrides a method to read the Spring beans configuration file that is used by the test.
- ❸ Invokes the **sendBodyAndHeader** method from the template inherited attribute to send contents using the **file:in** endpoint. The method adds the **file** string to the body, and names the file **testFile.txt**.
- ❹ Evaluates the final destination from the route, whether the file was evaluated and consumed by the route, and verifies that it was delivered to the **file:out** endpoint.

When a test case is executed, the test execution creates a **CamelContext** instance and reads the XML file defined by the **createApplicationContext** overridden method. Also the test case instantiates attributes inherited by the **CamelSpringTestSupport** class, such as the template attribute that gives developers access to any endpoint that starts a route execution. After all tests are executed, the context instance stops.

**Important**

Route processing may take some time to execute. You may receive misleading errors in your tests as the exchange might have not been completely processed. The following section discusses how to avoid them by using the **NotifyBuilder** class.

Installing Camel Routes with JBoss Developer Studio Integration Stack

Red Hat JBoss Developer Studio Integration Stack provides some extensions to debug and deploy Camel routes. The tools are available as part of the JBoss Fuse Tools plug-ins group during the installation process and they are not installed by default.

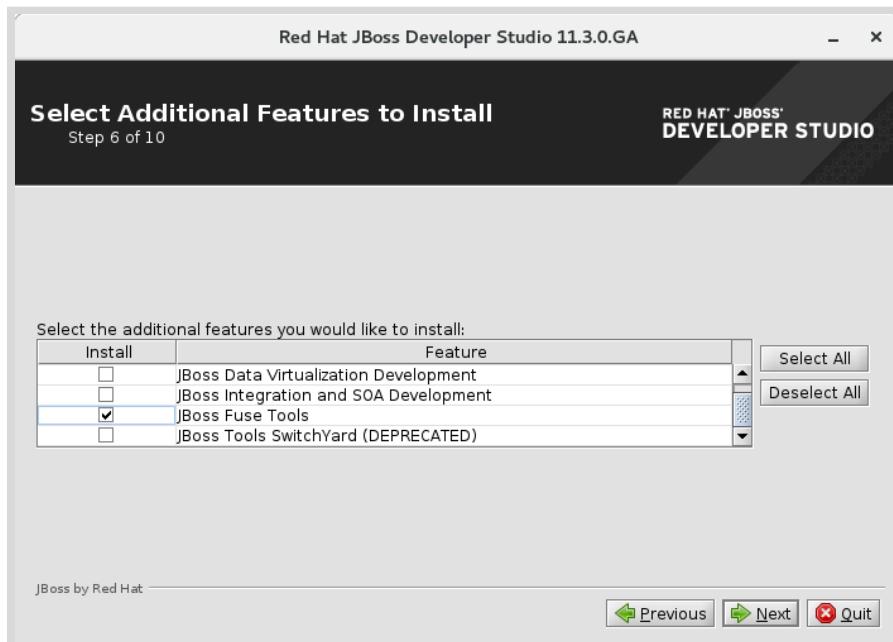


Figure 4.1: Selecting Camel Tools Support during JBoss Developer Studio installation

Running Camel Routes with JBoss Fuse Tooling

To run Camel routes defined in an XML file with JBoss Developer Studio Integration Stack, you must right-click the project with Camel routes and select **Run As → Local Camel Context** to start any route defined in the XML file. As a requirement to run the route, you must have the Camel Maven plug-in defined in the project's `pom.xml` file. If you have JUnit test cases that are broken, you may select **Run As → Local Camel Context (without tests)** to skip the test execution.

Debugging Camel Routes with JBoss Fuse Tooling

To debug an XML route and inspect the contents stored in the **Exchange** objects, use the **Debug As → Local Camel Context** option instead. To create breakpoints in the route to evaluate the **Exchange** objects, open the **Design** tab from the XML Camel route definition and right-click one of the boxes from the route and select **Set breakpoint**. As soon as the route processes an **Exchange** object, the **Debug** perspective opens.



References

Camel Testing

<https://camel.apache.org/testing.html>



References

For more information, refer to the *Installing JBoss Developer Studio Integration Stack Using the Standalone Installer* chapter in the *Red Hat JBoss Developer Studio Integration Stack 11.3* at

https://access.redhat.com/documentation/en-us/red_hat_jboss_developer_studio_integration_stack/11.3/html/installation_guide/standalone

► Guided Exercise

Testing Routes with Camel Test Kit

In this exercise, you will start a Spring DSL-based route in Red Hat JBoss Developer Studio using Camel plug-ins. You will implement a test case with Camel Test Kit to read a Spring DSL route definition and test a Camel route.

Outcomes

You should be able to start a Camel route defined in an XML definition file in JBoss Developer Studio and implement a test case using the **CamelSpringTestSupport** class to verify a Camel route functionality.

Before You Begin

A starter project that already includes the Spring DSL definition file is provided for you.

Run the following command to download the starter project used by this exercise:

```
[student@workstation ~]$ lab test-kit setup
```

► 1. Import the starter Maven project.

- 1.1. Open JBoss Developer Studio (**Applications** → **Programming** → **JBoss Developer Studio**). JBoss Developer Studio requests a default workspace; keep the default configuration (**/home/student/workspace**) and click **Launch**.
- 1.2. Import the Maven project by selecting from the JBoss Developer Studio main menu **File** → **Import**.
- 1.3. From the **Import** dialog box, select **Maven** → **Existing Maven Projects** and click **Next >**.
- 1.4. Click **Browse** and choose **/home/student/JB421/labs/test-kit** and click **Ok** to choose the Maven project. Click **Finish** to complete the import process in JBoss Developer Studio.

A new project named **test-kit** is listed in the **Project Explorer** view.

► 2. Complete the project Maven dependencies.

- 2.1. Open the project POM file.

In the **Project Explorer** view, expand **test-kit** and double-click **pom.xml**.

A new JBoss Developer Studio editor tab opens with the **test-kit/pom.xml** file in a Maven POM Editor.

Click the **pom.xml** tab at the bottom of the editor to see the POM raw source code.

- 2.2. Look for this comment:

```
<!-- TODO Add camel-test dependencies here -->
```

Replace it with the following dependency definitions:

```
<dependency>
<groupId>org.apache.camel</groupId>
<artifactId>camel-test-spring</artifactId>
<scope>test</scope>
</dependency>
```

The **camel-test-spring** dependency is required to use Camel's support classes for testing purposes.

- 2.3. Press **Ctrl+S** to save your updates to the **pom.xml** file.
- ▶ 3. Review the existing route definition.
 - 3.1. Open the route builder class for editing.
In the **Project Explorer** view, expand **src/main/resources** and then expand the **META-INF/spring** directory and double-click on **bundle-context.xml** to open the Spring DSL definition. Select the **Source** tab from the editor.
 - 3.2. Review the route definition:

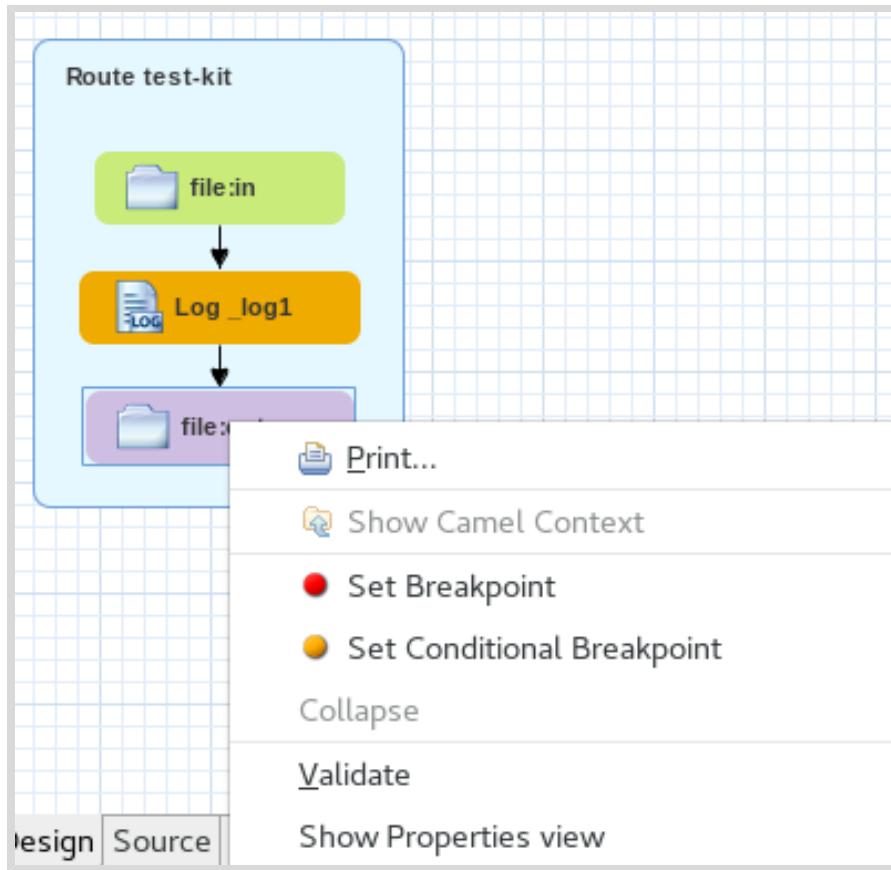
```
<?xml version="1.0" encoding="UTF-8"?>
<beans
  ...
>
  <camelContext id="jb421Context"
    xmlns="http://camel.apache.org/schema/spring">
    <route id="test-kit">
      <from id="_from1" uri="file:in"/>
      <log id="_log1" message="${body}"/>
      <to id="_to1" uri="file:out"/>
    </route>
  </camelContext>
</beans>
```

The route reads files from the project's **in** directory and forwards them to the **out** directory. During execution, the route prints the content of the file in the console.

- ▶ 4. Debug the route using Camel's plug-in from JBoss Developer Studio.
- 4.1. In JBoss Developer Studio, right-click the test-kit project and select **Debug As → Local Camel Context**. It triggers the Camel plug-in to run a route locally. Camel starts and displays the following message in the **Console** view:

```
INFO Apache Camel 2.21.0.fuse-000077-redhat-1 (CamelContext: jb421Context)
started in 0.930 seconds
INFO Starting beans in phase 2147483646
```

- 4.2. Define a breakpoint in the route to suspend the execution while the route is running. In the **bundle-context.xml** editor, select the **Design** tab. Right-click the **file:out** box and choose **Set breakpoint**. The box displays a red circle to indicate that a breakpoint was added to the route.



4.3. Deploy a file to test the route.

From a terminal window, run the following commands to create a file that is consumed by the Camel route.

```
[student@workstation ~] cd JB421/labs/test-kit
[student@workstation test-kit] touch in/test.txt
```

JBoss Developer Studio opens the **Debug** perspective and displays information about variables. Look for **Message** → **Message Headers** and collapse it. Inspect the information about the exchange, the body, and other information about the file.

To resume the route execution, press **F8**. Stop the route execution clicking the **Terminate** from the **Console** tab.

4.4. Return to the **JBoss** perspective in JBoss Developer Studio.

In the menu, select **Window** → **Perspective** → **Open Perspective** → **Other...** and choose **JBoss (default)** in the **Open Perspective** dialog box. Click **Open** to change back to the original set of tabs used for development.

► 5. Implement the test case for the route.

5.1. In the **Project Explorer** tab, collapse the test-kit project. Right-click **src/test/java** folder and choose **New** → **Other...**. Select **Java** → **JUnit** → **JUnit Test Case** and click **Next >**.

Use the following values in the **New JUnit Test Case** dialog box:

- Package: **com.redhat.training.camel.test**
- Name: **BundleContextXmlTest**

Leave the remaining fields with the default value and click **Finish**.



Note

There is an option to create a **Camel Test Case** among the options in the wizard but it creates a working test case with some items that were not introduced yet.

- 5.2. Remove the dummy test method from the class.

Delete the **test** method declared in the created test case.

- 5.3. Update the class to become a test case that supports Camel.

Define the **BundleContextXmlTest** class as a subclass of **CamelSpringTestSupport** parent class.

```
import org.apache.camel.test.spring.CamelSpringTestSupport;
...
public class BundleContextXmlTest extends CamelSpringTestSupport {
```



Note

This step creates a compilation error that is fixed in the later steps.

- 5.4. Override the **createApplicationContext** method from the **CamelSpringTestSupport** superclass to read the Spring DSL XML file.

Create a method **createApplicationContext** method annotated with the **@Override** annotation. It returns an instance of a **ClassPathXmlApplicationContext** object that reads the file located at **META-INF/spring/bundle-context.xml**.

```
import org.springframework.context.support.ClassPathXmlApplicationContext;
...
public class BundleContextXmlTest extends CamelSpringTestSupport {
...
@Override
protected ClassPathXmlApplicationContext createApplicationContext() {
    return new ClassPathXmlApplicationContext
        ("META-INF/spring/bundle-context.xml");
}
...
}
```

- 5.5. Implement a method that deletes any existing file in the origin directory before the test is executed.

Before the **createApplicationContext** method declaration, create a method named **before** that invokes the **deleteDirectory** method from the parent class. Annotate it with the **@Before** annotation.

```
...
import org.junit.Before;
import java.io.IOException;
...

public class BundleContextXmlTest extends CamelSpringTestSupport {

    @Before
    public void before() {
        deleteDirectory("in");
    }

    protected ClassPathXmlApplicationContext createApplicationContext() {
    ...
    }
    ...
}
```

- 5.6. Implement a method that deletes any existing file in the destination directory after the test is executed.

Create a method named **after** that invokes the **deleteDirectory** method from the parent class. Declare it after the method named **before** and annotate it with the **@After** annotation.

```
...
import org.junit.After;
...

public class BundleContextXmlTest extends CamelSpringTestSupport {
    ...

    @After
    public void after() {
        deleteDirectory("out");
    }

    ...
}
```

- 5.7. Create the test method named **testCamelRoute** to evaluate whether the route is processing the files and the **out** directory receives the file.

Use the **template** inherited attribute to create a file in the **in** directory. The **testFile.txt** file must have the **text** String as its content.

```
import org.apache.camel.Exchange;
import org.junit.Test;
...

public class BundleContextXmlTest extends CamelSpringTestSupport {
    ...

    @Test
    public void testCamelRoute() throws Exception {
```

```
template.sendBodyAndHeader("file:in", "file", Exchange.FILE_NAME,  
"testFile.txt");  
}  
...  
}
```

- 5.8. Implement the test to verify that the route processed the file and delivered it to the **out** directory.

In the **testCamelRoute** method, use the **consumer** inherited attribute to verify whether the **out** directory stored the file after route execution.

```
import org.apache.camel.component.file.GenericFile;  
...  
public class BundleContextXmlTest extends CamelSpringTestSupport {  
    ...  
    @Test  
    public void testCamelRoute() throws Exception {  
        template.sendBodyAndHeader("file:in", "file", Exchange.FILE_NAME,  
"testFile.txt");  
        GenericFile receiveBody = (GenericFile) consumer.receiveBody("file:out");  
        String content = receiveBody.getFileNameOnly();  
        assertEquals("testFile.txt", content);  
    }  
    ...  
}
```

- 5.9. Check that the test works.

From JBoss Developer Studio, right-click the **BundleContextXmlTest** class from **com.redhat.training.camel.test** package and select **Run As → JUnit Test** to start the test. The test passes with a green bar displayed in the **JUnit** tab.

This concludes this guided exercise.

Developing Routes with Mock Components

Objective

After completing this section, students should be able to create realistic test cases with mock components.

Developing with Mock Endpoints

By its nature, integration depends on multiple systems. This can be challenging when testing routes that depend on AMQ brokers or an FTP server, for example. To simulate these endpoints, Camel provides the **mock** component to check the output of a route execution, and to act as a placeholder for any external systems that are not yet available. To use this component in a route, include a **mock** URL as a producer, as shown in the following example:

```
from("file:/tmp/orders")
    .to("mock:inventoryManagement")
```

In the previous route, the test may access the **/tmp/orders** directory and store test files. The test method can access any mock endpoint and inspect the final outcome stored in the **mock:inventoryManagement** endpoint.

Mock endpoints also have a variety of uses when you are testing your routes. Some of the benefits of mock endpoints is that they allow for inspection of what it processes, including:

- the number of exchanges received
- the contents of an exchange
- the headers from a message

After executing the route, the mock endpoint must check if these expectations were accomplished by calling the **assertIsSatisfied** method.

Camel also supports mechanisms to update the route definition during test execution to avoid hard coding a mock in a route. To do this, a test must change the route prior to starting the context by using the **adviceWith** method.

Testing Routes with Camel Annotations

To further facilitate developing test methods for routes and capture information from endpoints in a test, Camel supports testing with annotations. Due to its power and simplicity, annotations are the preferred approach to develop tests.

Camel provides the following classes to help with sending test messages to an endpoint, and to check the messages received by an endpoint:

- **ProducerTemplate**: enables developers to create exchanges, specifying headers and the body contents that are sent to a route.

- **MockEndpoint**: provides methods to set expectations and assertions about the output from a route processing; that is, the exchanges were sent through a route to a consumer endpoint.

Camel provides the following annotations to inject components:

- **@Produce**: attribute-level annotation that injects a **ProducerTemplate** instance in a test. The ProducerTemplate instance is an endpoint that you may send custom exchange to the route without the need to manually access it using the **CamelContext** objects.
- **@EndpointInject**: attribute-level annotation that injects an endpoint managed by Camel. Usually it is useful to access **MockEndpoint** instances and access the information stored in the endpoint, such as the exchange received, the header values from the exchange, and the number of messages received by that endpoint.

In the following **RouteBuilder** definition, a file endpoint reads information from a directory and sends them to a mock endpoint:

```
public class FileRouteBuilder extends RouteBuilder{
    public void configure() throws Exception{
        from("file:/tmp/orders")
            .routeId("process")
            .to("mock:inventoryManagement");
    }
}
```

In the following test case, the previous **RouteBuilder** instance is used by the overridden **createRouteBuilder** method. Access the **file:/tmp/orders** and the **mock:inventoryManagement** endpoints with the **ProducerTemplate** and the **MockEndpoint** instances respectively:

```
public class RouteTest extends CamelTestSupport {
    @Produce(uri = "file:/tmp/orders")
    private ProducerTemplate template;

    @EndpointInject(uri = "mock:inventoryManagement")
    private MockEndpoint mock;
    @Override
    protected RouteBuilder createRouteBuilder() {
        return new FileRouteBuilder();
    }

    @Test
    public void testRoute(){
        template.sendBodyAndHeader("file", Exchange.FILE_NAME, "testFile.txt");
        GenericFile receiveBody = (GenericFile) mock.receiveBody();
        //Assertions
    }
}
```

Substituting Endpoints in a Route for Testing Purposes

Any route that sends messages to an external system needs to be able to be tested without the external system. To solve this problem, endpoints can be substituted with mocks during unit test execution.

Camel supports changes to an existing route by calling the **adviceWith** method and passing, as an argument, an **AdviceRouteBuilder** instance to describe the changes that must be made before a test runs.

In the following **RouteBuilder** definition, a file endpoint reads information from a directory and sends them to a direct endpoint:

```
public class DirectRouteBuilder extends RouteBuilder{
    public void configure() throws Exception{
        from("file:/tmp/orders")
            .routeId("process")
            .to("direct:transformComma");
    }
}
```

In the following code, the inherited attribute **context** captures a route definition and changes the route to divert messages to a different endpoint (**mock:direct:transformComma**).

```
public class RouteTest extends CamelTestSupport {

    @Produce(uri = "file:/tmp/orders")
    private ProducerTemplate template;

    @Override
    public boolean isUseAdviceWith() { ❶
        return true;
    }

    @EndpointInject(uri = "mock:direct:transformComma")
    private MockEndpoint mock;
    @Override
    protected RouteBuilder createRouteBuilder() {
        return new DirectRouteBuilder();
    }

    @Test
    public void testRoute(){
        context.getRouteDefinition("process")
            .adviceWith(modelCamelContext, new AdviceWithRouteBuilder() {
                @Override
                public void configure() throws Exception {
                    interceptSendToEndpoint("direct:transformComma")❷
                        .skipSendToOriginalEndpoint()❸
                        .to("mock:direct:transformComma");❹
                }
            });
    }
}
```

- ➊ Tells Camel that you intend to use **adviceWith** in this unit test.
- ➋ Intercepts calls to an endpoint named **direct:transformComma**
- ➌ Skips the original endpoint
- ➍ Redirects to an endpoint named **mock:direct:transformComma**

The test case using **AdviceWithRouteBuilder** instances must override the **isUseAdviceWith** returning **true** in order to manually start the Camel context.

This tells the test runner to *not* start the Camel context automatically before starting the test methods. This way, a test can change the context before starting it, and prevents the routes processing exchanges before they are changed by the test.

A state left in the Camel context by one test may interfere with other tests in the same test suite. To avoid this, you must stop the Camel context on each test method.

Expecting Exchanges with NotifyBuilder

The exchanges sent through a route can take some time to be processed, especially when the destination is an external system such as an FTP server or a message queue. For that reason, the route processing may require waiting during the route test execution until the exchange arrives to the final endpoint.

Camel uses **NotifyBuilder** to deal with delays. **NotifyBuilder** can be used as an assert function, because it can wait for any number of conditions in the context and its routes to be true, such as the quantity of exchanges to be fully processed.

A time limit can also be specified, and if the **NotifyBuilder** conditions are not satisfied during this time, it returns false.

The following code waits for one message for 5 seconds:

```
NotifyBuilder builder = new NotifyBuilder().whenDone(1).create();
Assert.assertTrue(builder.matches(5, TimeUnit.SECONDS));
```

Implementing Tests

When writing a test using the Camel Test Kit, it is important to do operations in a strict order to avoid race conditions and other timing issues. The following code follows the guideline:

```
@EndpointInject(uri="mock:cdg")
private MockEndpoint mockCdg;
@Test
public void testFileCamelRoute() throws Exception {
    AdviceWithRouteBuilder mockRoute = new AdviceWithRouteBuilder() {
        @Override
        public void configure() throws Exception {
            interceptSendToEndpoint(
                "file:orders/dest/cdg").skipSendToOriginalEndpoint().to("mock:cdg");
            ...
        }
    };
    context.getRouteDefinition("process").adviceWith(context, mockRoute); ①
    context.start(); ②

    NotifyBuilder builder = new NotifyBuilder(context).whenDone(1).create(); ③
```

```

builder.matches(2, TimeUnit.SECONDS); ④

fileOrders.sendBodyAndHeader(wholeContent, Exchange.FILE_NAME, "file.xml"); ⑤
mockCdg.expectedMessageCount(1); ⑥
assertMockEndpointsSatisfied(); ⑦
context.stop(); ⑧
}

```

- ①** Change routes using the **adviceWith** method.
- ②** Start the **CamelContext** instance.
- ③** Create and configure an instance of a **NotifyBuilder**.
- ④** Use **NotifyBuilder** to wait and assert its conditions were matched.
- ⑤** Send test exchanges using a **ProducerTemplate** or any other means.
- ⑥** Set expectations on all mock endpoints.
- ⑦** Check whether the mock expectations were satisfied.
- ⑧** Stop the **CamelContext** instance.

Not all of the above operations need to be done in all test cases, but when they are, it is recommended to follow the order of operations described.

Testing Predicates Using Mocks

The **MockEndpoint** class has three methods that allow predicate testing:

Method Name
assertPredicate(Predicate predicate, Exchange exchange, Boolean expected)
assertPredicateDoesNotMatch(Predicate predicate, Exchange exchange)
assertPredicateMatches(Predicate predicate, Exchange exchange)

Each method is inherited by the **CamelTestSupport** class and is a static method.

There are also the name of methods start with the string `expected` can be used to set the expectations about the number of messages, exchange content, and other valid evaluations that are useful for checking the messages exchanged. Refer to Camel API documentation.

Demonstration: Testing Routes with Mock Components

1. Start JBoss Developer Studio and import the **test-demo** project from the `/home/student/JB421/labs/` directory.
2. Evaluate the JAXB annotations in the model classes.

The model classes in this project are annotated with JAXB annotations to support marshaling and unmarshaling of these classes to and from XML.

Open the **com.redhat.training.camel.test.model.Order** model class.

```
...  
@XmlRootElement  
public class Order implements Serializable {  
...  
}
```

This is the only JAXB annotation used in all of the model classes included in this project. It defines the **Order** object as the root element of our XML data.



Note

If you want further control over how JAXB names the XML elements, you can explicitly specify the element names, attributes, and so on.

3. Review the route builder defined in the Camel context file.

Open **src/main/resources/META-INF/spring/bundle-context.xml**.

```
...  
<bean class="com.redhat.training.camel.test.TransformRouteBuilder"  
      id="transformRouteBuilder"/>  
<camelContext id="jb421Context" xmlns="http://camel.apache.org/schema/spring">  
  <routeBuilder ref="transformRouteBuilder"/>  
</camelContext>  
...
```

The Camel context has a single route builder.

4. Review the current route definition in the route builder class provided.

Review the route currently configured in the **com.redhat.training.camel.test.TransformRouteBuilder** file:

```
@Override  
public void configure() throws Exception {  
    from("direct:orderInput")  
    //TODO Define routeId to process  
    //TODO add filter  
    .to("file:orders/admin");  
}
```

The current route processes contents and sends them to the **orders/admin** directory.

5. Review the Camel Test Kit annotations used in the test class.

Open the **com.redhat.training.camel.test.TransformRouteTest** test class.

```
...  
public class TransformRouteTest extends CamelTestSupport {❶  
  
    @Produce(uri="direct:orderInput")❷  
    private ProducerTemplate producer;
```

```
@EndpointInject(uri = "mock:admin")③  
private MockEndpoint destination;  
  
...
```

- ① Every test case needs to be a subclass of the **CamelTestSupport** class.
- ② Injects the **direct:orderInput** endpoint to send messages to the route.
- ③ Injects the mock endpoint used to evaluate whether the messages were processed. The endpoint does not exist in the original route because it was changed by the test method.

6. Review the **isUseAdviceWith** method.

```
@Override  
public boolean isUseAdviceWith() {  
    return true;  
}
```

The method indicates that your test changes the original route and you need to manually manage the Camel context to allow a change to the route before executing the test.

7. Review the **before()** method.

```
@Before  
public void before() throws Exception {  
    deleteDirectory("orders");①  
    AdviceWithRouteBuilder mockRoute = new AdviceWithRouteBuilder() {②  
        @Override  
        public void configure() throws Exception {  
            interceptSendToEndpoint("file:orders/admin")③  
                .skipSendToOriginalEndpoint()④  
                .to("mock:admin");⑤  
        }  
    };  
    context.getRouteDefinition("process").adviceWith(context, mockRoute);⑥  
    context.start();  
}  
...
```

- ① Deletes the directory to avoid any wrong output
- ② Creates a deviation to the route
- ③ Intercepts the calls from the **file:orders/admin** endpoint
- ④ Deviates the calls from the **file:orders/admin** endpoint
- ⑤ Defines the new destination for the route
- ⑥ Updates an existing route with the detour

8. Check the **testNonAdminOrder** test method.

```
@Test  
public void testNonAdminOrder() throws Exception {  
    NotifyBuilder builder = new NotifyBuilder(context).whenDone(1).create();①  
    builder.matches(2000, TimeUnit.MILLISECONDS);②  
  
    Order testNonAdminOrder = createTestOrder(false);
```

```

String nonAdminXML = getExpectedXmlString(testNonAdminOrder);

producer.sendBodyAndHeader(nonAdminXML, Exchange.FILE_NAME, "output.xml"); ③
destination.expectedMessageCount(1); ④
assertMockEndpointsSatisfied(); ⑤
}

```

- ①** Instantiates a **NotifyBuilder** object that expects that only one message is processed by the route.
 - ②** Waits for two seconds until the test is considered processed. This allows time for Camel to process the contents.
 - ③** Creates an exchange that is processed by the route.
 - ④** Defines how many messages are processed the mock endpoint.
 - ⑤** Evaluates the expectations from the mock endpoint.
9. Run the **TransformRouteTest** test class. The test fails with a **NullPointerException** exception because the route that must be updated could not be found.
 10. Define the route identification to allow the route updates by the test case.

Add the following identification to the route definition in the **TransformRouteBuilder** class.

```

@Override
public void configure() throws Exception {
    from("direct:orderInput")
        //TODO Define routeId to process
        .routeId("process")
        //TODO add filter
        .to("file:orders/admin");
}

```

11. Test the route again.

Re-run the **com.redhat.training.camel.test.TransformRouteTest** test class. This time the **testNonAdmin** test method passes but the **testAdminOrder** method fails.

12. Review the **Predicate** that has been provided for you.

Open **com.redhat.training.camel.test.AdminOrderFilter**. Review the **matches()** method specifically because it controls the behavior of the filter.

```

@Override
public boolean matches(Exchange exchange) {
    Order order = exchange.getIn().getBody(Order.class);
    if(order != null && order.getCustomer() != null && !
    order.getCustomer().isAdmin()){
        log.info("Filtering out non admin order!");
        return true;
    }
    return false;
}

```

13. Add functionality to the route to filter out orders by admin users. Put this filter after the wire tap method call so these orders are logged, but then they are not sent to the fulfillment system. Use the **AdminOrderFilter** predicate class to implement the filter.

Open **com.redhat.training.camel.test.TransformRouteBuilder**. Update the first route definition to include a filter as follows:

```
from("direct:orderInput")
    .routeId("process")
    .filter(new AdminOrderFilter())
    .to("file:orders/admin");
```

Save your updates.

14. Run the **TransformRouteTest** test class again.

All the test cases passes this time because administrator's orders are not received by the **mock:admin** endpoint.

15. Clean up.

Close the **test-demo** project in JBoss Developer Studio.

This concludes the demonstration.



References

AdviceWith

<http://camel.apache.org/advicewith.html>

Message filter pattern

<http://camel.apache.org/message-filter.html>

Camel Test Kit

<http://camel.apache.org/camel-test.html>

► Guided Exercise

Verifying Route Processing with Mocks Components

In this exercise, you will develop a JUnit test case to verify Camel routes that use the splitter pattern and the content based router to break down an XML into multiple parts and forward those parts to different destinations.

Outcomes

You should be able to:

- Use Camel mock framework to update an existing route
- Run tests in an environment without the whole set of services running.

Before You Begin

A starter project is provided for you to start that already includes the Java DSL route definition.

Run the following command to download the starter project used by this exercise:

```
[student@workstation ~]$ lab test-mock setup
```

► 1. Import the starter Maven project.

- 1.1. Open JBoss Developer Studio (**Applications** → **Programming** → **JBoss Developer Studio**). JBoss Developer Studio requests a default workspace; keep the default configuration (**/home/student/workspace**) and click **Launch**.
- 1.2. Import the Maven project by selecting from the JBoss Developer Studio main menu **File** → **Import**.
- 1.3. From the **Import** dialog box, select **Maven** → **Existing Maven Projects** and click **Next >**.
- 1.4. Click **Browse** and choose **/home/student/JB421/labs/test-mock** and click **Ok** to choose the Maven project. Click **Finish** to complete the import process in JBoss Developer Studio.

A new project named **test-mock** is listed in the **Project Explorer** view.

► 2. Review the existing route definition.

- 2.1. Inspect the route builder class.
In the **Project Explorer** view, expand **src/main/java** → **com.redhat.training.camel.mock**. Double-click on **SplitterRouteBuilder** class.

```

public class SplitterRouterBuilder extends RouteBuilder {

    @Override
    public void configure() throws Exception {
        from("file:orders/
incoming").split().tokenizeXML("orderItem").to("direct:process");

        from("direct:process")
            .setHeader("company",
                xpath("/orderItem/orderItemPublisherName/text()"))
            .log("Company: ${header.company}")
            .choice()
                .when(header("company").
                    isEqualTo("ABC Company"))
                    .to("file:orders/dest/abc")
                .when(header("company")
                    .isEqualTo("ORly"))
                    .to("file:orders/dest/orly")
                .when(header("company")
                    .isEqualTo("Namming"))
                    .to("file:orders/dest/namming")

            .otherwise().to("file:others");
    }
    ...
}

```

The route reads data from the **order/incoming** directory. The route then uses the splitter pattern to break an XML order file into individual order items and forwards the individual items to a destination named **direct:process**.

The **direct:process** route reads the publisher name from the **orderItemPublisherName** element and stores it in the header with the key named **company**. The content-based router uses this value to send the data to separate directories.

2.2. Start the route.

From a terminal window, run the following commands to start the route:

```

[student@workstation ~]$ cd JB421/labs/test-mock
[student@workstation test-mock]$ mvn camel:run -DskipTests
...
INFO  Route: route1 started and consuming from: file://orders/incoming
INFO  Route: route2 started and consuming from: direct://process
INFO  Total 2 routes, of which 2 are started
INFO  Apache Camel 2.21.0.fuse-000077-redhat-1 (CamelContext: jb421Context)
      started in 0.459 seconds
INFO  Starting beans in phase 2147483646

```

2.3. Test the route.

Open a new terminal window and run the following commands to copy files that are processed by the route:

```
[student@workstation ~]$ cd JB421/labs/test-mock  
[student@workstation test-mock]$ ./setup-data.sh  
Preparing test folder:  
Cleaning test folder...  
Copying sample data files...  
Preparation complete!
```

2.4. Evaluate the output from the route.

In the terminal window running the route, inspect the message output. As the files are processed, each of the generate a log entry:

```
INFO Created default XPathFactory  
com.sun.org.apache.xpath.internal.jaxp.XPathFactoryImpl@5f2aff2  
INFO Company: ORly  
INFO Company: ORly  
INFO Company: ABC Company  
INFO Company: Namming  
INFO Company: ABC Company  
INFO Company: ORly  
INFO Company: ABC Company  
INFO Company: Namming  
INFO Company: ORly  
INFO Company: ABC Company  
INFO Company: Namming  
INFO Company: ORly  
INFO Company: ORly  
INFO Company: ABC Company  
INFO Company: Namming  
...
```



Note

Orders may be processed in a different order than displayed in the previous output.

2.5. Inspect the directory where the files processed by the route are stored.

From the terminal window that you used to run the command to copy the files, run the following command:

```
[student@workstation test-mock]$ ls orders/dest/orly  
order-1.xml order-2.xml order-3.xml order-4.xml order-5.xml order-6.xml
```

2.6. Verify the contents of the **order-1.xml** file.

It contains an order item for the publisher ORly and the route generated the file after splitting it:

```
[student@workstation test-mock]$ cat orders/dest/orly/order-1.xml
<orderItem>
  <orderItemId>1</orderItemId>
  <orderItemQty>1</orderItemQty>
  <orderItemPublisherName>ORly</orderItemPublisherName>
  <orderItemPrice>10.59</orderItemPrice>
</orderItem>
```

- 2.7. Stop the route execution.

Press **Ctrl+C** on the terminal window running the Camel route.

- 3. Override the **SplitterXmlTest** test case methods from the **com.redhat.training.camel.mock** package to support mock usage and dynamically change the endpoints for testing purposes.

- 3.1. Support the endpoint updates requested by the test to avoid touching the existing route.

Override the **isUseAdviceWith** method to enable the test case to update the route for testing purposes. Right after the comment **//TODO Override the isUseAdviceWith method to return true**, add the following method definition:

```
public class SplitterXmlTest extends CamelTestSupport {
...
@Override
public boolean isUseAdviceWith() {
    return true;
}
```

- 3.2. Read the route definition from the **SplitterRouteBuilder** class from the **com.redhat.training.camel.mock** package to be used by the test.

Override the **createRouterBuilder** method to read the route definitions from the **SplitterRouteBuilder** class. Declare it after the comment **// TODO Override the createRouterBuilder method to return a SplitterRouterBuilder instance**.

```
import org.apache.camel.builder.RouteBuilder;

public class SplitterXmlTest extends CamelTestSupport {

...
@Override
protected RouteBuilder createRouteBuilder() {
    return new SplitterRouterBuilder();
}
...
```

- 4. Implement a method that is executed before the tests to update the route to use mock endpoints instead of the **file:orders/dest/*** endpoints. Also, start the Camel context to load the routes as you need to use a customized route than the one provided.

- 4.1. Annotate the **mockEndpoints** method definition to be executed before the test is started. Use the **@Before** annotation to declare the **mockEndpoints** method is executed before each test method.

```
import org.junit.Before;

public class SplitterXmlTest extends CamelTestSupport {
    @Before
    public void mockEndpoints() throws Exception {
    }
```

- 4.2. Instantiate an **AdviceWithRouteBuilder** object to update the route definition to support mocks. As the **AdviceWithRouteBuilder** is an interface declare an anonymous class that implements the **configure** method.

```
import org.apache.camel.builder.AdviceWithRouteBuilder;

public class SplitterXmlTest extends CamelTestSupport {
    @Before
    public void mockEndpoints() throws Exception {
        AdviceWithRouteBuilder mockRoute = new AdviceWithRouteBuilder() {
            @Override
            public void configure() throws Exception {
            }
        };
    }
}
```

- 4.3. Intercept the data sent to the **file:orders/dest/orly** endpoint and forward it to the **mock:orly** endpoint. In the **configure** method declared in the previous step, add the following instruction:

```
...
public class SplitterXmlTest extends CamelTestSupport {
    @Before
    public void mockEndpoints() throws Exception {
        AdviceWithRouteBuilder mockRoute = new AdviceWithRouteBuilder() {
            @Override
            public void configure() throws Exception {
                interceptSendToEndpoint(
                    "file:orders/dest/orly")
                    .skipSendToOriginalEndpoint()
                    .to("mock:orly");
            }
        };
    }
}
```

- 4.4. Intercept the data sent to the **file:orders/dest/namming** endpoint and forward it to the **mock:namming** endpoint. In the **configure** method, add the following instruction:

```
...
public class SplitterXmlTest extends CamelTestSupport {
    @Before
    public void mockEndpoints() throws Exception {
        AdviceWithRouteBuilder mockRoute = new AdviceWithRouteBuilder() {
            @Override
            public void configure() throws Exception {
                ...
                interceptSendToEndpoint(
                    "file:orders/dest/namming")
                    .skipSendToOriginalEndpoint()
                    .to("mock:namming");
            }
        };
    }
}
```

- 4.5. Intercept the data sent to the **file:orders/dest/abc** endpoint and forward it to the **mock:abc** endpoint. In the **configure** method, add the following instruction:

```
...
public class SplitterXmlTest extends CamelTestSupport {
    @Before
    public void mockEndpoints() throws Exception {
        AdviceWithRouteBuilder mockRoute = new AdviceWithRouteBuilder() {
            @Override
            public void configure() throws Exception {
                ...
                interceptSendToEndpoint
                    ("file:orders/dest/abc")
                    .skipSendToOriginalEndpoint()
                    .to("mock:abc");
            }
        };
    }
}
```

- 4.6. Intercept the data sent to the **file:others** endpoint and forward it to the **mock:others** endpoint. In the **configure** method declared in the previous step, add the following instruction:

```
import org.apache.camel.builder.AdviceWithRouteBuilder;
...
public class SplitterXmlTest extends CamelTestSupport {
    @Before
    public void mockEndpoints() throws Exception {
        AdviceWithRouteBuilder mockRoute = new AdviceWithRouteBuilder() {
            @Override
            public void configure() throws Exception {
                ...
                interceptSendToEndpoint("file:others")
            }
        };
    }
}
```

```
        .skipSendToOriginalEndpoint()
        .to("mock:others");
    }
}
}
```

- 4.7. Add an identifier to the route as a way to update the endpoints in the route from the test.

In the **SplitterRouterBuilder** class from **com.redhat.training.camel.mock** package, update the following route to include a **routeId** value:

```
    public class SplitterRouterBuilder extends RouteBuilder {  
  
        @Override  
        public void configure() throws Exception {  
            ...  
            from("direct:process").routeId("process")  
            ...  
        }  
    }
```

- 4.8. Update the route with the new mock endpoints in the test case.

Return to the **SplitterXmlTest** test case. Update the **mockEndpoints** method to update the correct route by adding the following code using the route ID **process** added in the previous step:

```
public class SplitterXmlTest extends CamelTestSupport {  
    ...  
    @Before  
    public void mockEndpoints() throws Exception {  
        AdviceWithRouteBuilder mockRoute = new AdviceWithRouteBuilder() {  
            @Override  
            public void configure() throws Exception {  
                ...  
                interceptSendToEndpoint("file:others")  
                    .skipSendToOriginalEndpoint()  
                    .to("mock:others");  
            }  
        };  
        context.getRouteDefinition("process").adviceWith(context, mockRoute);  
    }  
}
```

- #### 4.9. Start the Camel context to initialize the routes.

Add the following code to the **mockEndpoints** method:

```
...
public class SplitterXmlTest extends CamelTestSupport {
    ...
    @Before
    public void mockEndpoints() throws Exception {
        ...
        context.getRouteDefinition("process").adviceWith(context, mockRoute);
        context.start();
    }
    ...
}
```

- 4.10. Stop the Camel context after the test execution.

Annotate the **after** method with **@After**. In the method, invoke the **stop** method the **context** object.

```
import org.junit.After;
...
public class SplitterXmlTest extends CamelTestSupport {
    ...
    @After
    public void after() throws Exception {
        context.stop();
    }
    ...
}
```

- ▶ 5. Annotate the **SplitterXmlTest** test case from the **com.redhat.training.camel.mock** package.

The test case is a subclass of the **CamelTestSupport** class and provides some attributes and methods used by the JUnit test.

- 5.1. Annotate the **fileOrders** attribute to inject the **file:orders/incoming** endpoint into the test case.

Use the **@Produce** annotation to inject the endpoint as follows:

```
import org.apache.camel.Produce;
...
public class SplitterXmlTest extends CamelTestSupport {
    ...
    @Produce(uri="file:orders/incoming")
    private ProducerTemplate fileOrders;
```

- 5.2. Annotate the **mockOrly** attribute to inject the **mock:orly** endpoint into the test case.

The **mock:orly** endpoint is introduced later in the test case. It is used instead of **file:orders/dest/orly** during the test to evaluate the content processed by the route. Use the **@EndpointInject** annotation to inject the endpoint as follows:

```
import org.apache.camel.EndpointInject;
...
public class SplitterXmlTest extends CamelTestSupport {
...
@EndpointInject(uri = "mock:orly")
private MockEndpoint mockOrly;
```

- 5.3. Annotate the **mockAbc** attribute to inject the **mock:abc** endpoint into the test case.

The **mock:abc** endpoint is introduced later in the test case. It is used instead of **file:orders/dest/abc** during the test to evaluate the content processed by the route. Use the **@EndpointInject** annotation to inject the endpoint as follows:

```
...
public class SplitterXmlTest extends CamelTestSupport {
...
@EndpointInject(uri = "mock:abc")
private MockEndpoint mockAbc;
```

- 5.4. Annotate the **mockNamming** attribute to inject the **mock:namming** endpoint into the test case.

The **mock:namming** endpoint is introduced later in the test case. It is used instead of **file:orders/dest/namming** during the test to evaluate the content processed by the route. Use the **@EndpointInject** annotation to inject the endpoint as follows:

```
...
public class SplitterXmlTest extends CamelTestSupport {
...
@EndpointInject(uri = "mock:namming")
private MockEndpoint mockNamming;
```

- 5.5. Annotate the **mockOthers** attribute to inject the **mock:others** endpoint into the test case.

The **mock:others** endpoint is introduced later in the test case. It is used instead of **file:others** during the test to evaluate the content processed by the route. Use the **@EndpointInject** annotation to inject the endpoint as follows:

```
...
public class SplitterXmlTest extends CamelTestSupport {
...
@EndpointInject(uri = "mock:others")
private MockEndpoint mockOthers;
```

- 5.6. Run the test case to verify the test passes.

From JBoss Developer Studio, right-click the **SplitterXmlTest** class and select **Run As → JUnit Test**. A red bar is displayed in the **JUnit** tab indicating that the test is not working.

- ▶ 6. Implement the test method that evaluates the routes.

- 6.1. Instantiate a **NotifyBuilder** instance to monitor the route execution and guarantee that the test does not finish before the files are processed.

As the test processing may take some time to start up, you need to indicate Camel how many messages the route processes and how much time the route must wait until a message is received. In the **testFileCamelRoute** method, instantiate a **NotifyBuilder** that expects a single file to be processed. Configure it to wait for two seconds before the test fails.

```
import org.apache.camel.builder.NotifyBuilder;  
...  
public class SplitterXmlTest extends CamelTestSupport {  
    @Test  
    public void testFileCamelRoute() throws Exception {  
        NotifyBuilder builder = new NotifyBuilder(context).whenDone(1).create();  
        builder.matches(2, TimeUnit.SECONDS);  
        ...  
    }  
    ...  
}
```

- 6.2. Evaluate the **wholeContent** variable.

This **String** mimics the content of an XML file with three items. Each of them are from different companies and therefore they should be processed by different endpoints.

- 6.3. Set an expectation that the **mock:abc** mock endpoint receives a single message.

```
...  
public class SplitterXmlTest extends CamelTestSupport {  
    ...  
    @Test  
    public void testFileCamelRoute() throws Exception {  
        NotifyBuilder builder = new NotifyBuilder(context).whenDone(1).create();  
        builder.matches(2, TimeUnit.SECONDS);  
        mockAbc.expectedMessageCount(1);  
    }  
    ...  
}
```

- 6.4. Set an expectation that the **mock:namming** mock endpoint receives a single message.

```
...  
public class SplitterXmlTest extends CamelTestSupport {  
    ...  
    @Test  
    public void testFileCamelRoute() throws Exception {  
        NotifyBuilder builder = new NotifyBuilder(context).whenDone(1).create();  
        builder.matches(2, TimeUnit.SECONDS);  
        mockAbc.expectedMessageCount(1);  
        mockNamming.expectedMessageCount(1);  
    }  
}
```

```

    }
    ...
}
```

- 6.5. Set an expectation that the **mock:orly** mock endpoint receives a single message.

```

...
public class SplitterXmlTest extends CamelTestSupport {
    ...
    @Test
    public void testFileCamelRoute() throws Exception {
        NotifyBuilder builder = new NotifyBuilder(context).whenDone(1).create();
        builder.matches(2, TimeUnit.SECONDS);
        mockAbc.expectedMessageCount(1);
        mockNamming.expectedMessageCount(1);
mockOrly.expectedMessageCount(1);
    }
    ...
}
```

- 6.6. Set an expectation that the **mock:others** mock endpoint does not receive any messages.

```

...
public class SplitterXmlTest extends CamelTestSupport {
    ...
    @Test
    public void testFileCamelRoute() throws Exception {
        NotifyBuilder builder = new NotifyBuilder(context).whenDone(1).create();
        builder.matches(2, TimeUnit.SECONDS);
        mockAbc.expectedMessageCount(1);
        mockNamming.expectedMessageCount(1);
        mockOrly.expectedMessageCount(1);
mockOthers.expectedMessageCount(0);
    }
    ...
}
```

- 6.7. Send a file through the route. Use the **fileOrders** attribute to send the content from the **wholeContent** attribute in the test.

From the **testFileCamelRoute** method, invoke the **sendBodyAndHeader** method from the **fileOrders** attribute to send the **wholeContent** String. Define the file name header field with **file.xml** value.

```

...
public class SplitterXmlTest extends CamelTestSupport {
    ...
    @Test
    public void testFileCamelRoute() throws Exception {
        ...
        NotifyBuilder builder = new NotifyBuilder(context).whenDone(1).create();
        builder.matches(2, TimeUnit.SECONDS);
    }
}
```

```
mockAbc.expectedMessageCount(1);
mockNamming.expectedMessageCount(1);
mockOrly.expectedMessageCount(1);
mockOthers.expectedMessageCount(0);
fileOrders.sendBodyAndHeader(wholeContent, Exchange.FILE_NAME, "file.xml");
...
}
...
```

- 6.8. Request the test framework to evaluate the mocks.

Invoke the **assertMockEndpointsSatisfied** method to verify the expectations for the mocks were met.

```
...
public class SplitterXmlTest extends CamelTestSupport {
    ...
    @Test
    public void testFileCamelRoute() throws Exception {
        NotifyBuilder builder = new NotifyBuilder(context).whenDone(1).create();
        builder.matches(2, TimeUnit.SECONDS);
        mockAbc.expectedMessageCount(1);
        mockNamming.expectedMessageCount(1);
        mockOrly.expectedMessageCount(1);
        mockOthers.expectedMessageCount(0);
        fileOrders.sendBodyAndHeader(wholeContent, Exchange.FILE_NAME, "file.xml");
        assertMockEndpointsSatisfied();
    }
}
```

- 6.9. Press **Ctrl+S** to save the changes to the JUnit test.

- 6.10. Run the JUnit test. This time the test passes because the expectations from the mocks are satisfied.

- 7. Clean up: close the project in JBoss Developer Studio to conserve memory.

In the **Project Explorer** view, right-click **test-mock** and click **Close Project**.

This concludes this guided exercise.

Handling Errors in Camel

Objective

After completing this section, students should be able to create reliable routes that handle errors gracefully.

Understanding Error Types in Camel

Camel routes may capture errors by nature of an external system, such as network outages, file system permissions, and services unavailability due to maintenance. Error handling is a powerful method provided by Camel to avoid data corruption or provide alternative services (such as backup databases) to solve these problems.

Errors that occur during Camel route execution can be classified into two categories:

Recoverable errors

Errors raised due to temporary problems that can be solved by retrying the request, such as network instability or a database connection timeout. Normally the failure generates a Java exception and is stored as part of the **Exchange** object transmitted by a route.

Irrecoverable errors

Failures without an immediate solution, such as file system failures or database corruption. These errors enable a flag from the outgoing **Message** in the **Exchange** object transmitted by a route.

Both error types should occur within the route processing and not outside, such as an endpoint. Usually failures from an endpoint are not even evaluated by Camel error handling facilities; however, some components, such as camel-file and camel-jpa, manage error handling internally.

Camel identifies recoverable errors during route processing, and manages these errors automatically, stopping the execution and propagating the error to the caller. This behavior is implemented by a Camel error handler implementation, but it can be customized to support redelivery attempts.

By default, irrecoverable errors are not handled by the Camel error handlers, but they can be enabled using a flag. This error is translated into an exception, similar to the recoverable errors. A Camel user can set this flag manually to halt the processing of an exchange and tell Camel to not attempt any redelivery or other error recovery behavior.

Camel error handlers are executed within each step in a route and are part of a **Channel**. If a failure happens in one step of a route, the previous **Channel** captures the error and transmits it to the error handler that is configured.



Important

Any modifications made to the **Exchange** object before the failing step are left unchanged.

Customizing Camel Error Handling

You may define an error handler that can be used in all routes. By default, Camel provides a default error handler that does not redeliver exchanges, and notifies the caller about the failure.

The error handler can be customized for each route, calling the **errorHandler** method in the route definition:

```
public class MyRoute extends RouteBuilder{
    public void configure() throws Exception{
        from("file:inputDir")
            .errorHandler(...)
            .to(...)
    }
}
```

For a Spring-based configuration:

```
<errorHandler id="myErrorHandler" type="ErrorHandlerName"/>
<route id="firstRoute" errorHandlerRef="myErrorHandler">
    ...
</route>
```

Alternatively, the error handler can be set for a context scope, calling the **errorHandler** method inherited by the **RouteBuilder** class:

```
public class MyRoute extends RouteBuilder{
    public void configure() throws Exception{
        errorHandler(...);
        from("file:inputDir")
            .to(...)
    }
}
```

For a Spring-based configuration:

```
<camel:errorHandler id="myErrorHandler" type="ErrorHandlerName"/>
<camel:camelContext errorHandlerRef="myErrorHandler"
    xmlns="http://camel.apache.org/schema/spring">
    ...
</camel:camelContext>
```

Camel implements four error handlers that you can use in your routes:

DefaultErrorHandler

Does not redeliver exchanges, and notifies the caller about the failure.

LoggingErrorHandler

Logs the exception to the default output.

NoErrorHandler

Disables the error handler mechanism.

DeadLetterChannel

Implements the dead letter channel enterprise integration pattern (EIP).

DefaultErrorHandler

The **DefaultErrorHandler** class does not require any configuration or code to activate. By default, the error raised during the route execution is sent back to the caller, with the changes made during the successful step. If any other error handler is configured as the default, the route can be customized to use it, calling the **errorHandler(defaultErrorHandler())** within a route declaration.

```
from("....")
.errorHandler(defaultErrorHandler())
.to("....");
```

For a Spring-based configuration, the **errorHandler** type is: **DefaultErrorHandler**.

LoggingErrorHandler

The **LoggingErrorHandler** class is an error handler where all exceptions are sent to the logging facility. All errors are sent to a category named **org.apache.camel.processor.LoggingErrorHandler** and with an **ERROR** level. To activate it:

```
from("....")
.errorHandler(loggingErrorHandler())
.to("....");
```

For a Spring-based configuration, the **errorHandler** type is: **LoggingErrorHandler**.

NoErrorHandler

The **NoErrorHandler** class is an error handler without any error management, and is a workaround to disable the mandatory error handling imposed by Camel. To activate it:

```
from("....")
.errorHandler(noErrorHandler())
.to("....");
```

For a Spring-based route, the **errorHandler** type is: **NoErrorHandler**.

DeadLetterChannel

The **DeadLetterChannel** class is an error handler using the dead letter channel EIP. The differences from the **DefaultErrorHandler** are:

- The exchange with problems is sent to a different destination from the caller.
- The exchange does not store the generated exception.

The exchange with problems can be evaluated by another route.

To activate it:

```
from("....")
.errorHandler(deadLetterChannel("destination"))
.to("....");
```

For a Spring-based route, the configuration requires the URI with the destination for exchanges with errors:

```
<camel:errorHandler id="myErrorHandler" type="DeadLetterChannel"
deadLetterUri="destination"/>
<camel:route id="firstRoute" errorHandlerRef="myErrorHandler">
...
</camel:route>
```

Controlling Error Handling Using the `onException` Exception Clause

Camel error handlers represent a standard approach to handle every error thrown by the route. Unfortunately, this approach is not useful in most integration systems, since it does not allow custom error handling based on a specific condition. For example, a business rule error should not be re-executed, but a network instability error can be re-executed.

Camel implements a case-by-case exception management policy, customizing routing and redelivery policies, using the **onException** method call.

In the following route, if a **LogException** is raised, Camel sends the exchange to a different route, otherwise the default routing logic is used.

```
public class JavaRouteBuilder extends RouteBuilder {

    @Override
    public void configure() throws Exception {
        onException(LogException.class).to("file:log");
        from("file:in")
        ...
    }
}
```

The **onException** method accepts alternative redelivery mechanisms:

```
public class JavaRouteBuilder extends RouteBuilder {

    @Override
    public void configure() throws Exception {
        onException(LogException.class)
            .to("file:log")
            .maximumRedeliveries(3);
        from("file:in?noop=true")
        ...
    }
}
```

In the previous example, if a **LogException** is thrown during the routing execution, it attempts the redelivery three times, which is different from the default error handler.

To avoid multiple declarations, Camel allows multiple exceptions in a unique **onException** method call:

```
onException(LogException.class, DeliveryException.class)
    .to("file:error");
```

Similarly to use **onException** in a Spring XML Camel route use the **<onException>** XML tag as shown in the following example:

```
<camelContext xmlns="http://camel.apache.org/schema/spring">
    <onException>
        <exception>javax.xml.xpath.XPathException</exception>
        <exception>javax.xml.transform.TransformerException</exception>
        <to uri="log:xml?level=WARN"/>
    </onException>
    <onException>
        <exception>java.io.IOException</exception>
        <exception>java.sql.SQLException</exception>
        <exception>javax.jms.JMSException</exception>
        <redeliveryPolicy maximumRedeliveries="5" redeliveryDelay="3000"/>
    </onException>
</camelContext>
```

Marking an Exception as Handled

Unless you specifically mark an exception as handled, it will always be returned to the caller. In order to prevent this behavior Camel provides the **handled** DSL method, which supports a predicate parameter to which can be set to true explicitly or set based on a dynamic expression. The following example shows marking an exception as handled:

```
onException(Exception.class)
    .handled(true)
    .setHeader(Exchange.HTTP_RESPONSE_CODE, constant(500))
    .setBody(simple("${exception.message}\n"));
```

Similarly in Spring XML DSL:

```
<onException>
    <exception>java.lang.Exception</exception>
    <handled><constant>true</constant></handled>
    <setHeader headerName="Exchange.HTTP_RESPONSE_CODE">
        <constant>500</constant>
    </setHeader>
    <setBody>
        <simple>${exception.message}\n</simple>
    </setBody>
</onException>
```

Exception selection

Camel always looks for the specific exception among the **onException** declarations. If it is not found, Camel looks for the exception hierarchy in a similar fashion to the Java exception management. For example, in the following exception hierarchy:

```
Exception.class
+-DeliveryException.class
  +-AddressNotFoundException.class
```

If an **AddressNotFoundException** exception is thrown and there is only exception handling to route a **DeliveryException** exception, Camel uses the exception handling for a **DeliveryException** exception.

Redelivery management

For each **onException** method call, Camel uses the default error handler values, even if other values were specified for the route itself. For example, in the following **onException** method call:

```
public class JavaRouteBuilder extends RouteBuilder {

    @Override
    public void configure() throws Exception {
        onException(LogException.class)
            .to("file:log");
        from("file:in?noop=true")
            .errorHandler(defaultErrorHandler().maximumRedeliveries(5))
            ...
    }
}
```

Even though for the route there is a maximum of five redeliveries, the **onException** method call does not consider five as the maximum number of re-deliveries. It uses the default values from the default error handler management.

Finally, it is possible to mix **onException** definitions with more general **errorHandler** definitions. In this case, exceptions that are not explicitly handled by an **onException** block will fall to the generic **errorHandler** behavior. The following example shows using both in a Java DSL Camel route:

```
errorHandler(defaultErrorHandler()
    .maximumRedeliveries(3)
    .redeliveryDelay(2000));

onException(IOException.class)
    .maximumRedeliveries(5);

from("file://orderservice")
    .to("netty4:tcp://service.example.com:4444?textline=true")
    .bean("orderBean", "prepareReply");
```

Try/Catch DSL

When a **RouteBuilder** is declared, Camel executes it only once during the route building process. For each piece of data processed by Camel, the **RouteBuilder** is not executed as ordinary Java code. Therefore, the ordinary try/catch/finally Java mechanism does not work.

However, there is an alternative way to declare the exception management, using a syntax similar to Java code: **doTry/doCatch/doFinally**:

```
public class JavaRouteBuilder extends RouteBuilder {

    @Override
    public void configure() throws Exception {
        from("file:in?noop=true")
            .doTry()
                .process(new LogProcessor())
            .doCatch(LogException.class)
                .process(new FileProcessor())
            .end();
        ...
    }
}
```

The **doTry/doCatch/doFinally** approach is a useful way to implement logic to solve the problem that is not only diverting the route to another destination. In the previous example, the **FileProcessor** may update the file processed by the **file:in** endpoint to change the file name to something like **filename-error.xml**.

Unlike the **catch** clause in Java, Camel's **doCatch** supports two important features. First, **doCatch** allows multiple exception types to be defined in a single block as shown below:

```
from("direct:start")
    .doTry()
        .process(new ProcessorFail())
        .to("mock:result")
    .doCatch(IOException.class, IllegalStateException.class)
        .to("mock:catch")
    .doFinally()
        .to("mock:finally")
.end();
```

Additionally, **doCatch** checks the exception hierarchy when it matches a thrown exception against the **doCatch** block. This is important because many times the original caused exception is wrapped by other wrapper exceptions, typically transposing the exception from a checked to a runtime exception.

For example, Camel itself does this by wrapping exceptions occurring during route execution into a **CamelRuntimeException** exception. Therefore, if the original exception was an **IOException** exception, then the **doCatch** defined for **IOException** still matches, even though it is wrapped by a **CamelRuntimeException** exception.

Spring DSL also supports **doTry**, **doCatch**, and **doFinally**, as in the following example:

```
<route>
  <from uri="direct:start"/>
  <!-- here the try starts. its a try .. catch .. finally just as regular java
code -->
  <doTry>
    <process ref="processorFail"/>
    <to uri="mock:result"/>
    <doCatch>
      <!-- catch multiple exceptions -->
      <exception>java.io.IOException</exception>
      <exception>java.lang.IllegalStateException</exception>
      <to uri="mock:catch"/>
    </doCatch>
    <doFinally>
      <to uri="mock:finally"/>
    </doFinally>
  </doTry>
</route>
```



References

Exception Clause

<http://camel.apache.org/exception-clause.html>

Try, Catch, Finally

<http://camel.apache.org/try-catch-finally.html>

► Guided Exercise

Handling Errors in Camel

In this exercise, you will develop mechanisms to handle errors raised during Camel route processing.

Outcomes

You should be able to use Camel's **doTry/doCatch** blocks, error handlers, and **onException** mechanisms to capture Java exceptions raised during route execution to provide exception management capabilities.

Before You Begin

A starter project is provided for you to start that already includes the Java DSL route definition.

Run the following command to download the starter project used by this exercise:

```
[student@workstation ~]$ lab test-error setup
```

► 1. Import the starter Maven project.

- 1.1. Open JBoss Developer Studio (**Applications** → **Programming** → **JBoss Developer Studio**). JBoss Developer Studio requests a default workspace; keep the default configuration (**/home/student/workspace**) and click **Launch**.
- 1.2. Import the Maven project by selecting from the JBoss Developer Studio main menu **File** → **Import**.
- 1.3. From the **Import** dialog box, select **Maven** → **Existing Maven Projects** and click **Next >**.
- 1.4. Click **Browse** and choose **/home/student/JB421/labs/test-error** and click **Ok** to choose the Maven project. Click **Finish** to complete the import process in JBoss Developer Studio.

A new project named **test-error** is listed in the **Project Explorer** view.

► 2. Review the existing route definition.

- 2.1. Inspect the route builder class.

In the **Project Explorer** view, expand **src/main/java** and then expand the **com.redhat.training.camel.error** package and double-click on **ErrorHandlingRouteBuilder** class.

```
...
```

```
public class ErrorHandlingRouteBuilder extends RouteBuilder {
    @Override
    public void configure() throws Exception {
```

```

from("file:orders/incoming") ①
    .routeId("errorProcess")
    .process(new AmountProcessor()) ②
    .to("direct:process"); ③

from("direct:process")
    .routeId("directoryPermissionError")
    .process(new PriceProcessor()) ④
    .to("file:orders/root/dest");

}
}

```

- ①** Reads data from the **order/incoming** directory.
- ②** Calculates the amount of items from each XML file and stores it in the header.
- ③** Forwards order items to a destination named **direct:process**.
- ④** Sums up the item prices and store the total value in an exchange header.

The upstream application that generates the XML files that are consumed by these routes was upgraded and it introduced a problem to the route processing. Furthermore, the system administrators changed some permissions to the file system that affected the application too.

2.2. Start the route.

From a terminal window, run the following commands to start the route:

```

[student@workstation ~]$ cd JB421/labs/test-error
[student@workstation test-error]$ mvn camel:run -DskipTests
...
INFO Route: route1 started and consuming from: file://orders/incoming
INFO Route: route2 started and consuming from: direct://process
INFO Total 2 routes, of which 2 are started
INFO Apache Camel 2.21.0.fuse-000077-redhat-1 (CamelContext: jb421Context)
started in 0.459 seconds
INFO Starting beans in phase 2147483646

```

2.3. Test the route.

Open a new terminal window, and run the following commands to copy files that are processed by the route:

```

[student@workstation ~]$ cd JB421/labs/test-error
[student@workstation test-error]$ ./setup-data.sh
Preparing test folder:
Cleaning test folder...
Copying sample data files...
Preparation complete!

```

The script copies six XML files with the content processed by the Camel routes.

2.4. Evaluate the output from the route.

In the terminal window running the route, inspect the message output. As the files are processed, each file generates an error log:

```

...
Message History
-----
RouteId          ProcessorId      ...
[errorProcess    ] [errorProcess  ]...
[errorProcess    ] [process1     ]...

Stacktrace
-----
java.lang.NumberFormatException: For input string: "1.5"
...
Message History
-----
RouteId          ProcessorId      ...
[errorProcess    ] [errorProcess  ]...
[errorProcess    ] [process1     ]...
[errorProcess    ] [to1          ]...
[directoryPermissio] [process2     ]...

Stacktrace
-----
java.lang.NumberFormatException: For input string: "NA"
...

```

As the output shows, two routes raise **NumberFormatException** exceptions during the processing. As they are different issues, each one is managed using a different approach.

2.5. Stop the route execution.

Press **Ctrl+C** in the terminal window running the Camel route.

- ▶ 3. Inspect the **AmountProcessor** processor to identify the reason that the first route is raising the exception.

In JBoss Developer Studio, open the **AmountProcessor** class from the **com.redhat.training.camel.error** package.

The processor reads the field from the **orderItemQty** element from an XML content, sums up the value, and stores it in the **totalAmount** header field. Due to the nature of the orders (books), the processor consider that only integer values are used. Unfortunately, the changes made to the external system generating the XML files updated the field to accept fractional numbers.

To minimize problems, the XML files with invalid values must be moved to the **orders/error** directory.

- 3.1. Update the **ErrorHandlingRouteBuilder** class from **com.redhat.training.camel.error** package to identify the exception and move the files using wrong data format to a separate directory.

Open the **ErrorHandlingRouteBuilder** class and update the route to catch the **NumberFormatException** exception and move the files to the **orders/error** directory.

```
public class ErrorHandlingRouteBuilder extends RouteBuilder {
    ...
    @Override
    public void configure() throws Exception {
        from("file:orders/incoming")
            .routeId("errorProcess")
            .doTry()
                .process(new AmountProcessor())
                .to("direct:process")
            .doCatch(NumberFormatException.class)
                .to("file:orders/error")
            .endDoTry();
    }
}
```

- 3.2. Test the route to check that the exception handling is working.

Right-click the **test-error** project in JBoss Developer Studio and select **Run As → JUnit Test**. The **JUnit** tab opens and the tests are executed. The **JUnit** tab displays a red bar but the **testFileWithQtyNotIntegerRoute** test method shows a green tick. That means that the **doTry/doCatch** mechanism captured the exception and the file is moved to the right.

- 3.3. Test the route using files.

From the terminal window, run the following command:

```
[student@workstation test-error]$ mvn camel:run -DskipTests
...
INFO  Apache Camel 2.21.0.fuse-000077-redhat-1 (CamelContext: jb421Context)
started in 0.565 seconds
```

Open a new terminal window and run the following commands:

```
[student@workstation ~]$ cd ~/JB421/labs/test-error
[student@workstation test-error]$ ./setup-data.sh
Preparing test folder:
Cleaning test folder...
Copying sample data files...
Preparation complete!
```

- 3.4. Verify that the files from the **incoming** directory were processed and sent to the correct directory. From the existing terminal window, run the following commands to check that the files were moved to the directory.

```
[student@workstation test-error]$ ls orders/error
order-1.xml  order-2.xml
```

Two files were processed but only one file raised the **NumberFormatException** exception.

- 3.5. Stop the route execution.

Press **Ctrl+C** in the terminal window running the Camel route.

- 4. Implement an exception for an issue from the **PriceProcessor** processor.

In JBoss Developer Studio, open the **PriceProcessor** class from the **com.redhat.training.camel.error** package. The processor calculates the total price but there are some XML files that have the **NA** string defined in the **orderItemPrice** element, which causes the **NumberFormatException** exception to be raised.

- 4.1. Update the **ErrorHandlingRouteBuilder** class from **com.redhat.training.camel.error** package to provide an exception handler that captures the **NumberFormatException** exception raised in any other route and send them to the **orders/tmp** directory. Name the route as **numberFormatException** as it is used by the unit tests. Use the following code as a reference:

```
public class ErrorHandlingRouteBuilder extends RouteBuilder {  
    ...  
    @Override  
    public void configure() throws Exception {  
        onException(NumberFormatException.class)  
            .routeId("numberFormatException")  
            .to("file:orders/tmp")  
            .handled(true);  
  
        from("file:orders/incoming")  
            .routeId("errorProcess")  
            .doTry()  
                .process(new AmountProcessor())  
                .to("direct:process")  
            .doCatch(NumberFormatException.class)  
                .to("file:orders/error")  
            .endDoTry();  
    }  
}
```

- 4.2. Test the route to check that the error handler is working.

Right-click the **test-error** project in JBoss Developer Studio and select **Run As → JUnit Test**. The **JUnit** tab opens and the tests are executed. The **JUnit** tab displays a red bar but the **testFileWithQtyNotIntegerRoute** and the **testFileCamelNotAvailableRoute** test methods show a green tick.

- 4.3. Test the route using files.

From the terminal window, run the following commands:

```
[student@workstation test-error]$ mvn camel:run -DskipTests  
...  
INFO Apache Camel 2.21.0.fuse-000077-redhat-1 (CamelContext: jb421Context)  
started in 0.565 seconds
```

Open a new terminal window and run the following commands:

```
[student@workstation ~]$ cd ~/JB421/labs/test-error
[student@workstation test-error]$ ./setup-data.sh
Preparing test folder:
Cleaning test folder...
Copying sample data files...
Preparation complete!
```

- 4.4. Evaluate that the files from the **incoming** directory were processed. From the existing terminal window, run the following commands to check that the route processed the files.

```
[student@workstation test-error]$ ls orders/incoming
order-3.xml    order-4.xml    order-5.xml    order-6.xml
```

The route processed only two files. The file is stored in the **orders/tmp** directory, but the remaining files were left in the **incoming** directory.

- 4.5. Review the output from the route processing to identify the root cause that prevented the files to be processed.

From the terminal window running the Camel routes, inspect the outputs. The following output is listed:

```
...
Message History
-----
RouteId          ProcessorId   ...
[errorProcess]    ] [errorProcess]  ]...
[errorProcess]    ] [doTry1      ]...
[errorProcess]    ] [process1    ]...
[errorProcess]    ] [to1         ]...
[directoryPermissio] [process2    ]...
[directoryPermissio] [to3         ]...

Stacktrace
-----
org.apache.camel.component.file.GenericFileOperationFailedException: Cannot store
file: orders/root/dest/order-5.xml
...
```

In the stacktrace output, you can see that the file could not be stored in the **orders/root/dest** directory due to permission issues.

- 4.6. Stop the route execution.

Press **Ctrl+C** on the terminal window running the Camel route.

► 5. Implement an error handler for the permission error identified during the previous step.

In JBoss Developer Studio, open the **ErrorHandlingRouteBuilder** class from the **com.redhat.training.camel.error** package.

- 5.1. Update the **ErrorHandlingRouteBuilder** class from **com.redhat.training.camel.error** package to provide an error handler that captures any exception raised and send them to the **orders/trash** directory.

Open the **ErrorHandlingRouteBuilder** class and add an error handler that sends all the files to the **orders/trash** directory.

```
public class ErrorHandlingRouteBuilder extends RouteBuilder {  
    ...  
    @Override  
    public void configure() throws Exception {  
  
        errorHandler(  
            deadLetterChannel("file:orders/trash")  
                .disableRedelivery()  
        );  
        ...  
    }  
}
```

- 5.2. Test the route to check that the error handler is working.

Right-click the **test-error** project in JBoss Developer Studio and select **Run As** → **JUnit Test**. The **JUnit** tab opens and the tests are executed. The **JUnit** tab displays a green bar because all the tests passed.

- 5.3. Test the route using files.

From the terminal window, run the following commands:

```
[student@workstation test-error]$ mvn camel:run -DskipTests  
...  
INFO Apache Camel 2.21.0.fuse-000077-redhat-1 (CamelContext: jb421Context)  
started in 0.565 seconds
```

Open a new terminal window and run the following commands:

```
[student@workstation ~]$ cd ~/JB421/labs/test-error  
[student@workstation test-error]$ ./setup-data.sh  
Preparing test folder:  
Cleaning test folder...  
Copying sample data files...  
Preparation complete!
```

- 5.4. Verify that the files from the **incoming** directory were processed. From the existing terminal window, run the following commands to check that the route processed the files.

```
[student@workstation test-error]$ ls orders/incoming
```

The route processed all the files.

The files are stored in the **orders/trash** directory because the **orders/root** directory is owned by the **root** user. Run the following command to verify that the processed files are stored in the **orders/trash** directory.

```
[student@workstation test-error]$ ls orders/trash  
order-3.xml order-4.xml order-5.xml order-6.xml
```

- 5.5. Stop the route execution.

Press **Ctrl+C** in the terminal window running the Camel route.

- 5.6. Close the project.

In the **Project Explorer** view, right-click **test-error** and click **Close Project**.

This concludes this guided exercise.

▶ Lab

Testing and Error Handling with Camel

Performance Checklist

In this lab, you will manage exceptions during route execution and implement a unit test that checks whether the route executes correctly.

Outcomes

You should be able to:

- Use exception management capabilities to catch exceptions raised during the route execution
- Implement a test method that evaluates whether the route is executed as expected.

A preconfigured Camel context, including necessary route builder, is provided. This starter project includes a unit test to check whether routes support exception management.

- Run the following command to download the starter project used by this lab:

```
[student@workstation ~]$ lab test-final setup
```

- Import the starter project into JBoss Developer Studio.
 1. Import the starter Maven project.
 - 1.1. Open JBoss Developer Studio (**Applications** → **Programming** → **JBoss Developer Studio**). JBoss Developer Studio requests a default workspace; keep the default configuration (**/home/student/workspace**) and click **Launch**.
 - 1.2. Import the Maven project by selecting from the JBoss Developer Studio main menu **File** → **Import**.
 - 1.3. From the **Import** dialog box, select **Maven** → **Existing Maven Projects** and click **Next >**.
 - 1.4. Click **Browse** and choose **/home/student/JB421/labs/test-final** and click **Ok** to choose the Maven project. Click **Finish** to complete the import process in JBoss Developer Studio.

A new project named **test-final** is listed in the **Project Explorer** view.

2. Review the existing route definition.

- 2.1. Inspect the route builder class.

In the **Project Explorer** view, expand **src/main/java** and then expand the **com.redhat.training.camel.error** package and double-click on **ErrorHandlingRouteBuilder** class.

```

...
public class ErrorHandlingRouteBuilder extends RouteBuilder {
@Override
public void configure() throws Exception {

    from("file:orders/incoming")
        .routeId("errorProcess")
        .choice()
        .when(
            xpath("/order/customer/shippingAddress/state/text() = 'AK'"))
            .to("file:orders/output/AK")
        .when(
            xpath("/order/customer/shippingAddress/state/text() = 'MA'"))
            .to("file:orders/output/MA")
        .otherwise()
        .to("direct:auditing");

    from("direct:auditing")
        .routeId("auditing")
        .process(new ValueHeaderProcessor())
        .to("file:orders/root/dest")
}
}

```

The first route reads data from the **order/incoming** directory. The route uses the content-based router pattern to send orders that include a shipping state of **AK** or **MA** to **orders/output/AK** or **orders/output/MA** directories, respectively. Orders from other states must be sent to the **direct:auditing** endpoint.

Orders sent to the **direct:auditing** endpoint are evaluated by a processor. The **totalInvoice** value must be stored as a header named **totalInvoice** in the **Exchange** payload.

The latest upgrade changed the **totalInvoice** value to support non-numeric content which should be discarded from the route and stored in the **orders/trash** directory.

Furthermore, system administrators changed some permissions on the **orders/root/problems** directory that affected the application too.

3. Implement a JUnit test to verify orders shipped to the MA state. The **ErrorHandlingTest** class from **com.redhat.training.camel.error** package implements all the test methods from the project.

The test must use **maStateContent** attribute as the XML input used by the test and implement the logic in the **testFileWithMARoute** test method.

The test already provides the **mock:MA** endpoint injected as the **mockMa** attribute and it must verify whether a single message is processed when the **maStateContent** attribute is sent to the route.

- 3.1. Instantiate an **AdviceWithRouterBuilder** anonymous class in the **testFileWithMARoute** method from **com.redhat.training.camel.error.ErrorHandlingTest** class. The anonymous class must override the **configure** method and intercept request sent to the **file:orders/output/MA** endpoint to the **mock:MA** endpoint.

- 3.2. Update the route definition to use the **AdviceWithRouterBuilder** instance created in the previous step.
Invoke the **adviceWith** method to update the **errorProcess** route using the new route definition.
 - 3.3. Start the context to use the updated route definition.
 - 3.4. Instantiate a **NotifyBuilder** instance to monitor the route execution and guarantee that the test does not finish before the files are processed.
As the test processing may take some time to start up, you need to indicate Camel how many messages the route processes and how much time the route must wait until a message is received. Instantiate a **NotifyBuilder** that expects that a single file is processed and it waits for two seconds until the test fails.
 - 3.5. Send an **Exchange** instance in the **fileOrders** attribute using the **sendBodyAndHeader** method. Use the **maStateContent** attribute as the **Exchange** body.
 - 3.6. Define an expectation for the **mock:MA** endpoint to receive a single message.
Use the **mockMa** injected attribute to check whether a single message was received by the endpoint.
 - 3.7. Invoke the assertion method to inspect that the expectations were met.
 - 3.8. Stop the context.
Invoke the stop method from the **context** inherited attribute.
 - 3.9. Remove the **fail** method invocation.
 - 3.10. Save your changes by pressing **Ctrl+S**.
 - 3.11. Test the route to verify that the route is working.
Right-click the **test-final** project in JBoss Developer Studio and select **Run As → JUnit Test**. The **JUnit** tab opens and the tests are executed. The **JUnit** tab displays a red bar but the **testFileWithMARoute** test method shows a green tick mark. That means that the route testing passed.
4. Implement an exception management for an issue from the **ValueHeaderProcessor** processor.
In JBoss Developer Studio, open the **ValueHeaderProcessor** class from the **com.redhat.training.camel.error** package. The processor stores the **totalInvoice** element value but there are some XML files that have the **NA** string defined in the **totalInvoice** element, which causes the **NumberFormatException** exception to be raised.
 - 4.1. Update the **ErrorHandlingRouteBuilder** class from **com.redhat.training.camel.error** package to use a **doTry/doCatch** block that captures the **NumberFormatException** exception raised in the route whose starting endpoint is **direct:auditing** and send them to the **orders/trash** directory.
 - 4.2. Test the route to check that the error handler is working.
Right-click the **test-final** project in JBoss Developer Studio and select **Run As → JUnit Test**. The **JUnit** tab opens and the tests are executed. The **JUnit** tab displays

a red bar but the **testFileWithNonAvailableStateAndNATotalInvoiceRoute** and the **testFileWithNonAvailableStateRoute** test methods show a green tick mark.

5. Implement an error handler for the permission error identified during the previous step.

In JBoss Developer Studio, open the **ErrorHandlingRouteBuilder** class from the **com.redhat.training.camel.error** package.

- 5.1. Update the **ErrorHandlingRouteBuilder** class from **com.redhat.training.camel.error** package to provide an error handler that captures any exception raised and send them to the **orders/problems** directory.

- 5.2. Test the route to verify that the error handler is working.

Right-click the **test-final** project in JBoss Developer Studio and select **Run As** → **JUnit Test**. The **JUnit** tab opens and the tests are executed. The **JUnit** tab displays a green bar because all the tests passed.

6. Grade your work. Execute the following command:

```
[student@workstation ~]$ lab test-final grade
```

If you do not get a **PASS** grade, review your work. The **~/JB421/labs/test-final/grade.log** file contains the Maven messages produced by the grading process. This may be helpful in debugging any issues.

7. Clean up.

Close the **test-final** project in JBoss Developer Studio to conserve memory.

This concludes the lab.

► Solution

Testing and Error Handling with Camel

Performance Checklist

In this lab, you will manage exceptions during route execution and implement a unit test that checks whether the route executes correctly.

Outcomes

You should be able to:

- Use exception management capabilities to catch exceptions raised during the route execution
- Implement a test method that evaluates whether the route is executed as expected.

A preconfigured Camel context, including necessary route builder, is provided. This starter project includes a unit test to check whether routes support exception management.

- Run the following command to download the starter project used by this lab:

```
[student@workstation ~]$ lab test-final setup
```

- Import the starter project into JBoss Developer Studio.
 1. Import the starter Maven project.
 - 1.1. Open JBoss Developer Studio (**Applications** → **Programming** → **JBoss Developer Studio**). JBoss Developer Studio requests a default workspace; keep the default configuration (**/home/student/workspace**) and click **Launch**.
 - 1.2. Import the Maven project by selecting from the JBoss Developer Studio main menu **File** → **Import**.
 - 1.3. From the **Import** dialog box, select **Maven** → **Existing Maven Projects** and click **Next >**.
 - 1.4. Click **Browse** and choose **/home/student/JB421/labs/test-final** and click **Ok** to choose the Maven project. Click **Finish** to complete the import process in JBoss Developer Studio.

A new project named **test-final** is listed in the **Project Explorer** view.

2. Review the existing route definition.

- 2.1. Inspect the route builder class.

In the **Project Explorer** view, expand **src/main/java** and then expand the **com.redhat.training.camel.error** package and double-click on **ErrorHandlingRouteBuilder** class.

```

...
public class ErrorHandlingRouteBuilder extends RouteBuilder {
@Override
public void configure() throws Exception {

    from("file:orders/incoming")
        .routeId("errorProcess")
        .choice()
        .when(
            xpath("/order/customer/shippingAddress/state/text() = 'AK'"))
            .to("file:orders/output/AK")
        .when(
            xpath("/order/customer/shippingAddress/state/text() = 'MA'"))
            .to("file:orders/output/MA")
        .otherwise()
        .to("direct:auditing");

    from("direct:auditing")
        .routeId("auditing")
        .process(new ValueHeaderProcessor())
        .to("file:orders/root/dest")
}
}

```

The first route reads data from the **order/incoming** directory. The route uses the content-based router pattern to send orders that include a shipping state of **AK** or **MA** to **orders/output/AK** or **orders/output/MA** directories, respectively. Orders from other states must be sent to the **direct:auditing** endpoint.

Orders sent to the **direct:auditing** endpoint are evaluated by a processor. The **totalInvoice** value must be stored as a header named **totalInvoice** in the **Exchange** payload.

The latest upgrade changed the **totalInvoice** value to support non-numeric content which should be discarded from the route and stored in the **orders/trash** directory.

Furthermore, system administrators changed some permissions on the **orders/root/problems** directory that affected the application too.

3. Implement a JUnit test to verify orders shipped to the MA state. The **ErrorHandlingTest** class from **com.redhat.training.camel.error** package implements all the test methods from the project.

The test must use **maStateContent** attribute as the XML input used by the test and implement the logic in the **testFileWithMARoute** test method.

The test already provides the **mock:MA** endpoint injected as the **mockMa** attribute and it must verify whether a single message is processed when the **maStateContent** attribute is sent to the route.

- 3.1. Instantiate an **AdviceWithRouterBuilder** anonymous class in the **testFileWithMARoute** method from **com.redhat.training.camel.error.ErrorHandlingTest** class. The anonymous class must override the **configure** method and intercept request sent to the **file:orders/output/MA** endpoint to the **mock:MA** endpoint.

```

...
@Test
public void testFileWithMARoute() throws Exception {

    AdviceWithRouteBuilder mockRoute = new AdviceWithRouteBuilder() {
        @Override
        public void configure() throws Exception {
            interceptSendToEndpoint("file:orders/output/MA")
                .skipSendToOriginalEndpoint()
                .to("mock:MA");
        }
    };
}

```

- 3.2. Update the route definition to use the **AdviceWithRouteBuilder** instance created in the previous step.

Invoke the **adviceWith** method to update the **errorProcess** route using the new route definition.

```

public class ErrorHandlingTest extends CamelTestSupport {
    ...
    @Test
    public void testFileWithMARoute() throws Exception {

        AdviceWithRouteBuilder mockRoute = new AdviceWithRouteBuilder() {
            ...
            context.getRouteDefinition("errorProcess").adviceWith(context, mockRoute);
        };
    }
}

```

- 3.3. Start the context to use the updated route definition.

```

public class ErrorHandlingTest extends CamelTestSupport {
    ...
    @Test
    public void testFileWithMARoute() throws Exception {
        ...
        context.getRouteDefinition("errorProcess").adviceWith(context, mockRoute);
        context.start();
    }
}

```

- 3.4. Instantiate a **NotifyBuilder** instance to monitor the route execution and guarantee that the test does not finish before the files are processed.

As the test processing may take some time to start up, you need to indicate Camel how many messages the route processes and how much time the route must wait until a message is received. Instantiate a **NotifyBuilder** that expects that a single file is processed and it waits for two seconds until the test fails.

```

public class ErrorHandlingTest extends CamelTestSupport {
    ...
    @Test
    public void testFileWithMARoute() throws Exception {

```

```

...
    context.start();
    NotifyBuilder builder = new NotifyBuilder(context).whenDone(1).create();
    builder.matches(2, TimeUnit.SECONDS);
}
}

```

- 3.5. Send an **Exchange** instance in the **fileOrders** attribute using the **sendBodyAndHeader** method. Use the **maStateContent** attribute as the **Exchange** body.

```

public class ErrorHandlingTest extends CamelTestSupport {
    ...
    String maStateContent = "<order>\n" +
        " <orderId>1</orderId>\n" +
        " <orderDate>2016-12-10T12:01:00-05:00</orderDate>\n" +
        " <totalInvoice>NA</totalInvoice>\n" +
        " <customer>\n" +
        "   <shippingAddress>\n" +
    ...
    @Test
    public void testFileWithMARoute() throws Exception {
        ...
        builder.matches(2, TimeUnit.SECONDS);
        fileOrders.sendBodyAndHeader(maStateContent, Exchange.FILE_NAME, "file.xml");
        ...
    }
}

```

- 3.6. Define an expectation for the **mock:MA** endpoint to receive a single message. Use the **mockMa** injected attribute to check whether a single message was received by the endpoint.

```

public class ErrorHandlingTest extends CamelTestSupport {
    ...
    @Test
    public void testFileWithMARoute() throws Exception {
        ...

        fileOrders.sendBodyAndHeader(maStateContent, Exchange.FILE_NAME, "file.xml");
        mockMa.expectedMessageCount(1);
        ...
    }
}

```

- 3.7. Invoke the assertion method to inspect that the expectations were met.

```

public class ErrorHandlingTest extends CamelTestSupport {
    ...

```

```

@Test
public void testFileWithMARoute() throws Exception {
    ...
    mockMa.expectedMessageCount(1);
    assertMockEndpointsSatisfied();
    ...
}
}

```

- 3.8. Stop the context.

Invoke the stop method from the **context** inherited attribute.

```

public class ErrorHandlingTest extends CamelTestSupport {
    ...
    @Test
    public void testFileWithMARoute() throws Exception {
        ...
        assertMockEndpointsSatisfied();
        context.stop();
    }
}

```

- 3.9. Remove the **fail** method invocation.

```
fail("Not implemented");
```

Delete this line.

- 3.10. Save your changes by pressing **Ctrl+S**.

- 3.11. Test the route to verify that the route is working.

Right-click the **test-final** project in JBoss Developer Studio and select **Run As → JUnit Test**. The **JUnit** tab opens and the tests are executed. The **JUnit** tab displays a red bar but the **testFileWithMARoute** test method shows a green tick mark. That means that the route testing passed.

- 4.** Implement an exception management for an issue from the **ValueHeaderProcessor** processor.

In JBoss Developer Studio, open the **ValueHeaderProcessor** class from the **com.redhat.training.camel.error** package. The processor stores the **totalInvoice** element value but there are some XML files that have the **NA** string defined in the **totalInvoice** element, which causes the **NumberFormatException** exception to be raised.

- 4.1.** Update the **ErrorHandlingRouteBuilder** class from **com.redhat.training.camel.error** package to use a **doTry/doCatch** block that captures the **NumberFormatException** exception raised in the route whose starting endpoint is **direct:auditing** and send them to the **orders/trash** directory.

Use the following code as a reference:

```
public class ErrorHandlingRouteBuilder extends RouteBuilder {
...
@Override
public void configure() throws Exception {

    from("direct:auditing")
        .routeId("auditing")
        .doTry()
            .process(new ValueHeaderProcessor())
            .to("file:orders/root/dest")
        .doCatch(NumberFormatException.class)
            .to("file:orders/trash")
        .endDoTry();
}
}
```

- 4.2. Test the route to check that the error handler is working.

Right-click the **test-final** project in JBoss Developer Studio and select **Run As → JUnit Test**. The **JUnit** tab opens and the tests are executed. The **JUnit** tab displays a red bar but the **testFileWithNonAvailableStateAndNATotalInvoiceRoute** and the **testFileWithNonAvailableStateRoute** test methods show a green tick mark.

5. Implement an error handler for the permission error identified during the previous step.

In JBoss Developer Studio, open the **ErrorHandlingRouteBuilder** class from the **com.redhat.training.camel.error** package.

- 5.1. Update the **ErrorHandlingRouteBuilder** class from **com.redhat.training.camel.error** package to provide an error handler that captures any exception raised and send them to the **orders/problems** directory.

Open the **ErrorHandlingRouteBuilder** class and add an error handler that sends all the files to the **orders/problems** directory.

```
public class ErrorHandlingRouteBuilder extends RouteBuilder {
...
@Override
public void configure() throws Exception {

    errorHandler(
        deadLetterChannel("file:orders/problems")
        .disableRedelivery()
    );
...
}
}
```

- 5.2. Test the route to verify that the error handler is working.

Right-click the **test-final** project in JBoss Developer Studio and select **Run As → JUnit Test**. The **JUnit** tab opens and the tests are executed. The **JUnit** tab displays a green bar because all the tests passed.

6. Grade your work. Execute the following command:

```
[student@workstation ~]$ lab test-final grade
```

If you do not get a **PASS** grade, review your work. The **~/JB421/labs/test-final/grade.log** file contains the Maven messages produced by the grading process. This may be helpful in debugging any issues.

7. Clean up.

Close the **test-final** project in JBoss Developer Studio to conserve memory.

This concludes the lab.

Summary

In this chapter, you learned:

- Camel provides the Camel test kit to provide mechanisms for injecting endpoints and testing routes without the need to start every external service.
- To use the testing capability of Camel test kit, add the **camel-test** or **camel-spring-test** Maven dependencies to your project's dependency.
- To create tests that uses **camel-test** dependency and support context and dependency injection (CDI), extend **org.apache.camel.test.CamelTestSupport**.
- Mock endpoints are useful to identify how many messages, errors, and the results of a test without the need to start any associated infrastructure such as a message queue or a database.
- To produce tests that do not require the need to inject mock endpoints, change the routes during the tests with the **AdviceWithRouteBuilder** instances that update existing route endpoints with mock endpoints.
- Camel provides a **NotifyBuilder** class to define the number of exchanges sent during testing as well as the amount of time each route should take to execute.
- To manage exceptions, Camel implements **doTry**, **doCatch**, and **doFinally**.

Chapter 5

Routing with Java Beans

Goal

Create dynamic routes in Camel using Java Beans

Objectives

- Create Java Beans for use in routing and transforming messages.
- Implement routes that include dynamic routing with CDI.
- Execute bean methods within DSL predicates.

Sections

- Developing Routes with Java Beans and Bean Registries (and Guided Exercise)
- Creating Dynamic Routes with CDI (and Guided Exercise)
- Using Beans in Predicates (and Guided Exercise)

Lab

Routing with Java Beans

Developing Routes with Java Beans and Bean Registries

Objectives

After completing this section, students should be able to:

- Create Java Beans for use in routing and transforming messages.
- Call beans to manipulate exchange information from Camel.
- Implement the recipient list enterprise integration pattern in Camel.

Invoking Beans

In many cases, developers need a way to call bean methods within a Camel route. This approach is useful for abstracting business logic required within a route, such as a method that calculates the tax for a product. By abstracting out this method, the Camel route becomes more readable and more maintainable. A Camel route can invoke a bean using multiple mechanisms, either as an endpoint or using a processor. There are two major mechanisms to invoke any bean in Camel: bean DSL and the bean component.

Bean DSL

Camel provides a method that can be used to invoke any bean. It is usually called in a route, and provides two overloaded versions that are commonly used:

- **bean(<Class>)**: Accepts a class as a parameter to refer to any existing class available in the class path. If multiple methods are declared in the class, Camel executes the one that can handle a Camel body.
- **bean(String)**: Accepts a String as a parameter to refer to a class in the bean registry.

Either method has another overload with a second String argument, which refers to the method that should be called.

Bean Component

Camel supports a call to a bean with the bean component. It provides a semantic similar to any component, and it may be called in a **to** method call. In the following example, the bean named "test" is invoked in the route execution.

```
to("bean://test");
```

Declaring Java Beans

As mentioned previously, Camel is able to reference beans through one of several different approaches. By not adhering to a single approach, such as just using CDI, Camel makes integration easier. As a result, it is not required for Camel developers to be able to use each of these methods, rather stick with the approach that is best suited for the existing development environment. The following are some of the approaches Camel supports for bean initialization:

ApplicationContextRegistry

This is the default approach for a bean registry when using Camel with Spring. Using the **camelContext** element requires no additional work to declare or initialize beans, because Spring handles the bean lookup without any configuration.

Java Import

Camel can refer to regular Java classes by using the **bean** method call. In the following example, the **TransformBean** class is used by Camel, even though it is not configured in the Spring beans configuration file.

```
from("file://originSelectMethod")
.bean(TransformBean.class)
...
```

CDI

When using Context Dependency Injection (CDI), Camel is able to recognize any bean annotated with one of the CDI annotations. The following is an example CDI bean:

```
@Singleton
@Named("taxBean")
public class TaxCalculator{

    public String processTax(String data) throws Exception {
        return "value";
    }

}
```

The **taxBean** is called in this route without requiring additional configuration:

```
@Singleton
public class TaxRoute extends RouteBuilder {

    public void configure() throws Exception {
        from("file:in?noop=true")
            .bean("taxBean")
            .to("file:out")
    }
}
```

Spring XML

Camel can use any bean declared in its configuration file by ID. In the following example, the configuration file refers to a bean:

```
<beans ...>
    <bean id="configFile" class="com.redhat.training.jb421.ConfigurationFile" />
</beans>
```

It may be referred by a route using the following syntax:

```
.to("bean://configFile")
```

Calling Bean Methods

To invoke a Java bean in a Camel route, use the **bean** method. The syntax for this method is the following:

```
bean(beanName, "beanMethodName")
```

The first parameter refers to the name of the bean and the second optional parameter refers to the name of the method to call in the bean. Camel bean method invocation looks for a method that processes a compatible input. It is therefore not required to specify a method name if only a single method is available that matches the compatible input. If any conflict is found, Camel throws an **AmbiguousMethodCallException**. To remove this ambiguity, specify the method name within the **bean** method.

In the following bean, two methods are declared. Both are similar due to their parameters and return type.

```
public class TaxCalculator{

    public String processTax(String data) throws Exception {
        return "value";
    }

    public String processTotalValue(String data) throws Exception {
        return "result";
    }
}
```

Camel solves this ambiguity by using the following method call that specifically references the desired **processTotalValue** method:

```
public class TaxRoute extends RouteBuilder {

    @Autowired
    private TaxCalculator calculatorBean;

    public void configure() throws Exception {
        from("file:in?noop=true")
            .bean(calculatorBean, "processTotalValue")
            .to("file:out")
    }
}
```

The XML DSL provides a similarly easy syntax for calling a bean method from a route:

```
<bean id="calculatorBean" class="TaxCalculator">
    ...
</route>
```

```
<from uri="file:in?noop=true"/>
<bean ref="calculatorBean" method="processTotalValue"/>
<to uri="file:out"/>
</route>
...
```

In order for Camel to actually access the bean methods, the bean must be available to Camel in a registry. Camel provides several approaches for finding and initializing beans, such as the CDI bean registry, JNDI, and others. After the bean is looked up by Camel, the bean can be referenced by name within the route. These registries are discussed in more detail later in this section.

Executing Methods on Java Objects Inside a Camel Route

Executing a bean method from a route without parameters is very limiting. Camel provides an approach that allows users to pass in a parameter to a bean method, providing further flexibility within routes. In our previous route, for example, a method is called to calculate total cost of a product. In order to do this, the bean method must be able to access the message data in order to determine the cost of the product.

The argument that is passed to the method bean must meet the following requirements:

- The parameter is compatible with the exchange processing body.
- The bean's method return is compatible with Camel's transformation classes.

By default, Camel binds the body of the message to the first parameter in the invoked bean method. This means that the **bean** method header must declare a parameter that is able to accept the message body, such as **String messageBody**.

In the following example, a single method is declared in the Java bean. Camel sends the message body as the method argument and returns the resulting processing back to the route.

```
package com.redhat.training;

public class TaxCalculator{

    public String processTax(String data) throws Exception {
        double value=Double.parseDouble(data)*0.06;
        return ""+value;
    }
}
```

The class is configured as a bean in Spring beans configuration file:

```
<beans... >
    <bean id="taxCalculator"
          class="com.redhat.training.TaxCalculator" />
    ...
</beans>
```

The bean method is then referenced in the following XML route definition:

```
<beans ...>
    <camelContext xmlns="http://camel.apache.org/schema/spring">
        <route>
            <from uri="file:in"/>
            <bean ref="taxCalculator" ❶
                  method="processTax" ❷/>
            <to uri="file:out"/>
        </route>
    </camelContext>
</beans>
```

- ❶ The ID referring to the bean with the method executed.
- ❷ The method in the class to be called.

The following is the same route defined in the Java DSL using the bean component:

```
from("file:in?noop=true")
    .to("bean://taxCalculator?method=processTotalValue") ❶
```

- ❶ **taxCalculator** is the bean id in the Sprint beans configuration file.

Parameter Binding

A method with one argument only processes the body of the **Message** object. If you want to either pass in an additional parameter or process other exchange information, such as message headers, you must use Camel parameter binding.

Camel automatically binds the following values that can be leveraged within a method call:

- **Exchange**: The exchange used to transport data in a Camel route.
- **Message**: The input message.
- **CamelContext**: The **CamelContext** instance.
- **TypeConverter**: To use a specific data converter during the route execution in Camel.
- **Registry**: To obtain during the route execution beans stored in the registry.
- **Exception**: To capture exceptions raised during the route execution.

In the following method, a message is passed as an argument:

```
public class TaxCalculator {
    public String processTotalValue(Message message)
        throws Exception {
        ...
    }
}
```

Camel provides annotations to extract data from a message instead of using the Camel API inside the bean class.

```
public class TaxCalculator {

    public String processTotalValue(@Body String data①,
        @Header("CamelFileNameOnly") String filename ②)
        throws Exception {
        ...
    }
}
```

- ① Captures the body of a message and passes it as a parameter for this method.
- ② Captures the value **CamelFileNameOnly** from the message header and passes it as a parameter for this method.

The following annotations are available:

- **@Attachments**: Associates the method parameter to the message attachments.
- **@Body**: Binds the parameter to the message body.
- **@Header(name)** Binds the parameter to the given message header.
- **@Headers**: Binds the parameter to all the input headers.
- **@OutHeaders**: Binds the parameter to the output headers.

The **@Attachments**, **@Headers**, **@OutHeaders**, and **@Properties** annotated parameter types should be a **java.util.Map**.

Camel also allows for message contents to be processed using annotations.

- **@Bean**: Invokes a method on a bean.
- **@Simple**: Evaluates a Simple EL.
- **@XPath**: Evaluates an XPath expression.

Implementing Recipient List Pattern

The recipient list EIP is an integration pattern wherein a message is routed to multiple destinations in parallel. Similar to the Content-based Router, this pattern determines the destination based on the content of the message, however that destination is a dynamic list of recipients. A use case for this pattern is when a customer processes returns and the route needs to check with nearby warehouses for stock availability. The recipient list is generated based on the location of the customer processing the return and each warehouse is sent the return order to see if room is available for storage.

In Camel, the list of recipients is stored in the message header. The following example shows a file sent to a number of recipients, according to the contents of its header **destination**.

```
public class JavaRouteBuilder extends RouteBuilder {

    @Override
    public void configure() throws Exception {
        from("file:in")
            .recipientList(header("destinations")); ①
    }
}
```

- ① A list of recipients can be obtained directly from the header named **destinations**.

As mentioned previously, the list of recipients is dynamically generated based on the content of the message. A **Processor** class can be used to create the header contents to create this list of recipients:

```
public class DestinationBean implements Processor{

    public void process(Exchange exchange) throws Exception {
        Message in = exchange.getIn();
        String recipientList="";
        String fileName = (String)in.getHeader("CamelFileNameOnly");
        String[] destination = fileName.split("-");
        if ("toyota".equals(destination[0])){
            recipientList = recipientList.concat("file:toyota");
        } else if("gmc".equals(destination[0])){
            recipientList = recipientList.concat("file:gmc");
        } else {
            recipientList = recipientList.concat("file:gmc,file:toyota");
        }
        in.getHeaders().put("destinations", recipientList);
    }
}
```

The logic in the **Processor** class determines the destination of the file based on the initial file name. If the pattern is not found, the file is published to all destinations available.

The following code executes the route and uses the recipient list component and the **destinations** property on the message header to determine the list of recipients:

```
public class JavaRouteBuilder extends RouteBuilder {

    @Override
    public void configure() throws Exception {
        from("file:in")
            .bean(DestinationBean.class)
            .recipientList(header("destinations"));
    }
}
```

Demonstration: Implementing the Recipient List Pattern

1. Import the recipient-list project into JBoss Developer Studio.
2. Inspect the **com.redhat.training.jb421.OrderRouteBuilder** class.

The class has three routes that reads from a file and send it to different endpoints. They support the following requirements:

- The first route processes files from the **origin** folder and transforms all commas (,) to semicolons (;).

- The second route processes files from the **originSelectMethod** folder and transforms all commas (,) to colons (:).
 - The last route reads the seventh field from the message and forwards it to a destination using the recipient list EIP.
3. Call a bean in the first route execution.

Call the **BodyTransformBean** in the route. Uncomment the bean method call in the **OrderRouteBuilder** route as follows:

```
.bean(BodyTransformBean.class, "replaceCommaWithSemiColon")
```

4. Check that the message was processed by the first route with the **com.redhat.training.jb421.RouteWithSemiColonTest** test class.

Run the JUnit test. Overall the tests fail. However, the **testRouteFileWithSemiColon** method has a green check, indicating that one of the tests passed.

5. Call the bean end point in the second route execution.

Uncomment the following code from the **OrderRouteBuilder** route code:

```
.to("bean:bodyTransformBean?method=replaceCommaWithColon")
```

6. Check that the message was processed by the first route.

Run the **com.redhat.training.jb421.RouteWithSemiColonTest** test class. A green bar should be presented.

7. Call the recipient list EIP in the third route execution.

To call the bean and set the recipient list, uncomment the third route from the **OrderRouteBuilder** route code:

```
.bean("destinationBean")
```

8. Check that each destination received one message from the test.

In the **RouteRecipientListTest**, uncomment:

```
//TODO add expectations
amqEndpoint.expectedMessageCount(1);
ftpEndpoint.expectedMessageCount(1);
testEndpoint.expectedMessageCount(1);
```

This code checks that each destination receives one message for each one.

9. Test the routes.

Run the **RouteRecipientListTest** JUnit test. A green bar is presented.

10. Close the project in JBoss Developer Studio.

This concludes this demonstration.



References

Recipient List pattern

<https://camel.apache.org/recipient-list.html>

Bean Component

<https://camel.apache.org/bean.html>

Camel in Action, Second edition - Chapter Routing with Camel

Camel in Action, Second edition - Chapter Using Beans with Camel

► Guided Exercise

Invoking Beans in Camel Routes

In this exercise, you will invoke beans inside Camel routes to transform message contents and implement the recipient list Enterprise Integration Pattern.

Outcomes

You should be able to develop and configure a Camel bean that intercepts a route, customize the message body to update the content format, and use the recipient list EIP implementation from Camel.

Before You Begin

Run the following command to download the starter project and data files used by this exercise. Click **Applications** → **Favorites** → **Terminal** in the desktop menu and type:

```
[student@workstation ~]$ lab invoke-beans setup
```

- ▶ 1. Import the invoke-beans project into JBoss Developer Studio.
 - 1.1. Open JBoss Developer Studio (**Applications** → **Programming** → **JBoss Developer Studio**). JBoss Developer Studio requests a default workspace; keep the default configuration (**/home/student/workspace**) and click **Launch**.
 - 1.2. Import the Maven project by selecting the JBoss Developer Studio main menu **File** → **Import**.
 - 1.3. From the **Import** dialog box, select **Maven** → **Existing Maven Projects** and click **Next >**.
 - 1.4. Click **Browse** and choose **/home/student/JB421/labs/invoke-beans** and click **OK** to choose the Maven project. Click **Finish** to complete the import process in JBoss Developer Studio.

A new project named invoke-beans will be listed in the **Project Explorer** view.

- ▶ 2. Inspect the starter project.

The project has three routes that support the following requirements:

- The first route processes files from the **origin** folder and transforms all commas (,) to backquotes (`).
- The second route is linked to the first route and transforms non-ASCII characters to question marks (?). This change avoids processing issues in the downstream systems that consume the CSV contents.
- The third route reads the output from the second route and forwards to a destination using the recipient list EIP.

The routes call Java beans to make these transformations. These Java beans are already implemented for you:

- **BodyTransformBean** with two methods to support the changes required by the downstream integrated system.
 - **DestinationBean** with a method to implement the recipient list pattern.
- 2.1. Open the **OrderRouteBuilder** class by expanding the **invoke-beans** item in the **Project Explorer** tab in the left pane of JBoss Developer Studio, and then click **invoke-beans → src/main/java → com.redhat.training.jb421** to expand it. Double-click the **OrderRouteBuilder.java** file.
- Review the two routes and how they are implemented. The first one is a complete route, the following route is partially implemented, and you must implement the third route.
- 2.2. Open the **BodyTransformBean** class by expanding the **invoke-beans** item in the **Project Explorer** tab in the left pane of JBoss Developer Studio, then click **invoke-beans → src/main/java → com.redhat.training.jb421.beans** to expand it. Double-click the **BodyTransformBean.java** file.
- The **BodyTransformBean** class has two methods used to transform the contents of the body:
- **replaceCommaWithBacktick**: Supports the first requirement from this step.
 - **replaceNonASCIIWithQuestionMark**: Supports the second requirement from this step.

► 3. Inspect the first route.

The route whose **routeId** is named **transformComma** is provided for evaluation purposes.

3.1. Inspect the route.

The **replaceCommaWithBacktick** method from the **BodyTransformBean** is used to replace all commas (,) with backquotes (^):

```
from("file:origin")
.routeId("transformComma")
.bean(BodyTransformBean.class,"replaceCommaWithBacktick")
.to("direct:transformChars");
```

3.2. Run the JUnit test

Open the **src/test/java** folder and expand the **com.redhat.training.jb421** package.

Right-click the **RouteTransformationTest** test class and select **Run As → JUnit Test**. A green check is presented beside the method named **testRouteWithComma**.

Only the test method named **testRouteWithComma** should pass. The other tests will pass as you continue with this exercise.

► 4. Update the second route.

To update the contents, use the method **replaceNonASCIIWithQuestionMark** from the class **com.redhat.training.jb421.bean.BodyTransformBean**.

4.1. Complete the second route to support the body change.

Call the method named **replaceNonASCIIWithQuestionMark** from the **com.redhat.training.jb421.bean.BodyTransformBean** bean.

Invoke the bean component using the method option in the route as follows:

```
from("direct:transformChars")
.routeId("transformChars")
//TODO send to the bean component
.to("bean://bodyTransformBean?"+  
"method=replaceNonASCIIWithQuestionMark")
.to("direct:recipientList");
```

Save the changes by pressing **Ctrl+S**.

4.2. Configure **BodyTransformBean** as a Spring bean.

Open the **src/main/resources** folder, and edit the **META-INF/spring/bundle-context.xml** file by double-clicking it. Open the **Source** tab and add, right after the comment that reads **TODO** comment:

```
<!-- TODO Configure bodyTransformBean -->
<bean class="com.redhat.training.jb421.beans.BodyTransformBean"
id="bodyTransformBean"/>
```

Save the changes by pressing **Ctrl+S**.

4.3. Run the test.

Right-click the **RouteTransformationTest** test case and select **Run As → JUnit Test**. Two green checks are presented. However, the bar is still red because the last test is not working yet.

▶ 5. Create the last route to implement a recipient list EIP.

The route is tied to the previous routes but it breaks down the CSV content to multiple exchanges and send the messages to multiple destinations. A bean named **DestinationBean** is provided to define alternative routes depending on the type of the item.

5.1. Create the third route to support the recipient list EIP.

Create the route to:

- Split each line as a new exchange
- Use the **DestinationBean** class to identify the final destination.

Add, just after the **TODO** comment, a new route to:

- Read from the **direct:recipientList** endpoint, using the splitter EIP
- Identify the route with a **routeId** value of **recipientList**.
- Invoke the **calculateDesintation** method from the **DestinationBean** class to read the message contents, and add the destination to each item's header.
- Call the recipient list EIP that gets the destination using the header key named **destination**

```
//TODO implement the third route
from("direct:recipientList")
.routeId("recipientList")
.split(body().convertToString().tokenize("\n"))
.setHeader("destination",
    method(DestinationBean.class,"calculateDestination"))
.recipientList(header("destination"));
```

Save the changes by pressing **Ctrl+S**.

5.2. Test the third route.

Right-click the **RouteTransformationTest** test case and select **Run As → JUnit Test**. A green bar is presented. Now all tests pass.

► 6. Clean up.

Close the project in JBoss Developer Studio to conserve memory.

In the **Project Explorer** view, right-click invoke-beans project and click **Close Project**.

This concludes the guided exercise.

Dynamic Routing with CDI

Objectives

After completing this section, students should be able to implement routes with CDI that include dynamic routing.

Camel CDI Support

Contexts and Dependency Injection (CDI) is a Java specification introduced in Java EE 5 that delegates the object lifecycle management to the runtime environment. Camel supports CDI using a separate dependency, which introduces additional annotations that are integrated to CDI. CDI and its annotations provide features such as auto route detection and Camel primitive types such as **endpoint** to facilitate integration development. One convenient feature, for example, is that CDI automatically detects and instantiates any beans that extend **RouteBuilder** and automatically adds them to a Camel context. For example, after adding the necessary Camel CDI dependencies, the following is all that is required to run a simple route:

```
class RouteBean extends RouteBuilder {

    @Override
    public void configure() {
        from("jms:in")
            .to("log:out");
    }
}
```

To add CDI to a Maven-enabled project, add the following dependency and ensure that a **beans.xml** file exists in the **META-INF** directory:

```
<dependency>
    <groupId>javax.enterprise</groupId>
    <artifactId>cdi-api</artifactId>
    <scope>provided</scope>①
    <version>1.2</version>
</dependency>
<dependency>
    <groupId>org.apache.camel</groupId>②
    <artifactId>camel-cdi</artifactId>
</dependency>
```

- ^① The scope of CDI is usually **provided** because CDI is part of the runtime Java EE container.
- ^② The package for Camel CDI annotations.

Camel CDI Annotations

The following Camel CDI annotations are useful for creating the Camel context and defining endpoints:

@org.apache.camel.cdi.ContextName

A class-level annotation used to provide the name of the Camel context that the **RouteBuilder** routes use. When this annotation is omitted, the **RouteBuilder** routes are added to a default context created and started by Camel-CDI. Using this annotation allows a large project to isolate routes in different contexts.

If all **RouteBuilder** implementations in the same project use the same Camel context name, Camel-CDI just uses this name for its default context and there is no need for the application to explicitly instantiate a context using the required name.

```
@ContextName("context")
class RouteBean extends RouteBuilder {

    @Override
    public void configure() {
        //Route here
    }
}
```

CDI also supports creating multiple Camel contexts, useful in large projects with lots of routes that can be organized into different contexts. To create an additional context, create a class that extends the **DefaultCamelContext** class and annotate it with **@ContextName** and with the CDI annotation **@ApplicationScoped**. In the class that contains your routes, refer to your desired Camel context by using the **@ContextName** annotation.

@org.apache.camel.cdi.Uri

An annotation used to inject endpoints in a route. The **URI** annotation allows an application to provide custom configuration for the endpoint component and is commonly used by unit tests. After injecting and declaring the **Endpoint** object, the object can be referenced by any route by its declared name (**endpoint** in the following example).

```
@Inject
@Uri("direct:in")
Endpoint endpoint;
```

If using multiple Camel contexts, you can refer to your desired context when instantiating the **Endpoint** with the following annotation:

```
@Inject
@Uri("direct:in")
@CamelContext("context1")
Endpoint endpoint;
```

Binding Bean Parameters

Any bean can be invoked by a Camel route by simply using the **bean** method. To execute a specific method in a bean with multiple methods, the **bean** DSL accepts a second parameter that defines the method name invoked.

The following is an example of calling a bean from within a route using CDI with the **bean** method and the **URI** annotations:

```
public class TaxRoute extends RouteBuilder {

    @Inject
    @Uri("file:in")
    Endpoint in;

    @Inject
    @Uri("log:out")
    Endpoint out;

    @Override
    public void configure() throws Exception {
        from(in)
            .bean("taxBean", "calculateTax")
            .to(out);
    }
}
```

Routing with Routing Slip

In a route, there are occasionally multiple steps required to complete route processing. If these intermediate steps are endpoints, the whole route is considered a *pipeline*.

One common integration problem is that this pipeline isn't static and is not always predetermined. For example, consider a pipeline that processes security checks for a credit card company. A message containing purchase information sent from the same zip code as the user does not require additional security checks, but an international purchase requires additional endpoints in the pipeline to verify that the purchase is not fraudulent.

The **Routing Slip EIP** is an integration pattern used to route a message consecutively through a series of processing steps where the sequence of steps is not known at design time and can vary for each message. The pattern calculates the pipeline in a single method call based on the message. To support this pattern, Camel has a **routingSlip** method. The method requires a predefined **header** containing a comma-separated list of endpoints that are to be used in the exchange processing:

```
from("jms:inbox").routingSlip(header("destination"));
```

In this example, the header must have a header with a key named **destination**. If that header has a value of "**file://output,ftp://infrastructure/labs**", then the routing slip processes the file endpoint first and then the FTP endpoint.

Routing with Dynamic Router

Similar to the Routing Slip, the dynamic router EIP supports a dynamic route based on a message. A dynamic router, however, does not require this list to be predetermined, as with the routing slip.

To enable the dynamic router, create a method to calculate the route URL that returns a **String**. Each time the endpoint process finishes, the route is recalculated by the same method.

```
dynamicRouter(method(DestinationBeans.class, "execute"));
```

Routing with `toD()` DSL

If the route needs to be dynamic for only a single destination, Camel supports an easier approach with the `toD` DSL. In this method, the parameter can be a **String** or an Simple EL expression that resolves to a destination.

For instance, the following method resolves the key named destination from the exchange header to identify the destination.

```
toD("${header.destination}") ;
```

This example requires that the header contains a key named **destination** with a value that resolves to an endpoint URL.

Demonstration: Developing Routes with CDI

1. Import the `develop-cdi` project into JBoss Developer Studio.
2. Inspect the `com.redhat.training.jb421.OrderRouteBuilder`.

The route reads files from the **orders** folder and processes them. First, each file is split into individual lines and each line is sent in a separate message exchange. Each exchange is evaluated by a bean to read the body and identify the correct destination.

Initially, all the orders are sent to a folder, but eventually the route sends them to different destinations according to the order type.

3. Inspect the `com.redhat.training.jb421.beans.DestinationBean`.

This class is responsible for defining the endpoint where the exchange is sent. A single method is provided (**processDestination**), which reads the seventh field from the CSV file and add to the header the destination.

4. Inspect the `pom.xml` file.

In the `pom.xml` file, there are no references to any CDI API, which is needed to compile and deploy CDI-compliant applications. Also, the Camel CDI APIs are also missing.

5. Add CDI dependencies to the project.

Remove the comment element after the comment `<!-- TODO add dependency for camel-cdi and cdi -->`:

6. Enable CDI in the project.

Using the JBoss Developer Studio **beans.xml file** wizard, create a `beans.xml` file. Update the file to use the **bean-discovery-mode** as **all**.

7. Update the Camel Maven plug-in configuration to support CDI.

Remove the comment element right after the comment `<!-- add CDI support -->`:

8. Annotate the **DestinationBean** to become managed by CDI.

To activate **DestinationBean** in CDI, uncomment the `@Named` class-level annotation.

9. Test the environment to determine whether CDI is enabled.

Run Maven with `camel:run -DskipTests` as a goal. The following output is expected:

```
[g.apache.camel.cdi.Main.main()] DefaultCamelContext INFO Total 1 routes, of which  
1 are started.
```

Click **Stop** from the **Console** view to stop the route.



Note

The **Run As → Local Camel context (without tests)** does not work because there is no Spring beans configuration file.

10. Associate the route with a **CamelContext**.

Uncomment the **@ContextName** class-level annotation in the **OrderRouteBuilder** class.

11. Update the route to work with dynamic routes.

There are two routes commented out that work with dynamic routes: **toD** and **dynamicRouter**. Comment out the **.to("file:destination")** method call and uncomment the **toD** method call.

Test the route with the new **com.redhat.training.jb421.RouteTransformationTest** class. A green bar means that the test passed.

12. Uncomment the method responsible for calculate the dynamic router EIP. Uncomment the **processDynamicRouterDestination** method.

Test the route with the new **com.redhat.training.jb421.RouteTransformationTest** class. A green bar means that the test passed.

13. Test the routing slip EIP.

Uncomment the **routingSlip** method call from the **OrderRouteBuilder** and comment out the **dynamicRouter**.

Rerun the **RouteTransformationTest** test. A green bar is displayed which means that the test passed.

This concludes this demonstration.



References

Camel CDI component

<http://camel.apache.org/cdi.html>

CDI bean-discovery-mode attribute

<https://docs.oracle.com/javaee/7/tutorial/cdi-adv001.htm>

Camel CDI Testing components

<http://camel.apache.org/cdi-testing.html>

Routing Slip EIP

<http://camel.apache.org/routing-slip.html>

Dynamic Router EIP

<http://camel.apache.org/dynamic-router.html>

toD DSL

<http://camel.apache.org/message-endpoint.html>

Camel in Action, Second edition - Chapter Using Beans with Camel

Camel in Action, Second edition - Chapter Testing

► Guided Exercise

Implementing Dynamic Routing with CDI

In this exercise, you will implement routes that can change according to an algorithm using Camel CDI extensions.

Outcomes

You should be able to implement dynamic routes with either **toD** or **routingSlip** methods from routes using CDI extensions.

Before You Begin

Run the following command to download the starter project and data files used by this exercise. Click **Applications** → **Favorites** → **Terminal** in the desktop menu and type:

```
[student@workstation ~]$ lab dynamic-route-cdi setup
```

► 1. Import the starter Maven project.

- 1.1. Open JBoss Developer Studio (**Applications** → **Programming** → **JBoss Developer Studio**). JBoss Developer Studio requests a default workspace; keep the default configuration (**/home/student/workspace**) and click **Launch**.
- 1.2. Import the Maven project by selecting the JBoss Developer Studio main menu **File** → **Import**.
- 1.3. From the **Import** dialog box, select **Maven** → **Existing Maven Projects** and click **Next >**.
- 1.4. Click **Browse** and choose **/home/student/JB421/labs/dynamic-route-cdi** and click **OK** to choose the Maven project. Click **Finish** to complete the import process in JBoss Developer Studio.

A new project named **dynamic-route-cdi** is listed in the **Project Explorer** view.

► 2. Complete the project Maven dependencies.

- 2.1. Open the project POM file.
In the **Project Explorer** view, expand **dynamic-route-cdi** and double-click **pom.xml**.
A new JBoss Developer Studio editor tab opens with the **dynamic-route-cdi/pom.xml** file in a Maven POM Editor.
Click the **pom.xml** tab at the bottom of the editor to see the POM raw source code.
- 2.2. Look for the **<! -- add dependency for CDI -->** in the **pom.xml** file, and add the following dependencies:

**Note**

The **pom-cdi.xml.txt** file is provided to copy and paste the contents.

```
<dependency>
  <groupId>javax.enterprise</groupId>
  <artifactId>cdi-api</artifactId>
  <scope>provided</scope>
</dependency>
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-cdi</artifactId>
</dependency>
```

2.3. Add CDI support for the Camel plug-in.

Look for the **<!-- add dependency for Camel CDI plugin -->** comment and add the following configuration to support Maven Camel plug-in CDI support:

**Note**

The **pom-plugin.xml.txt** file is provided to copy and paste the contents.

```
<configuration>
  <useCDI>true</useCDI>
</configuration>
<dependencies>
  <dependency>
    <groupId>org.apache.deltaspike.cdictrl</groupId>
    <artifactId>deltaspike-cdictrl-weld</artifactId>
    <version>1.8.1</version>
  </dependency>
  <dependency>
    <groupId>org.jboss.weld.se</groupId>
    <artifactId>weld-se</artifactId>
    <version>2.4.6.Final</version>
  </dependency>
</dependencies>
```

The **deltaspike-cdictrl-weld** and **weld-se** dependencies provide the same CDI implementation used by JBoss EAP.

2.4. Press **Ctrl+S** to save your updates to the **pom.xml** file.

► 3. Include the CDI configuration file.

The **beans.xml** file must be added to enable CDI support.

Expand the **src/main/resources/META-INF** folder and right-click it. Select **New → Other...** and choose **CDI (Contexts and Dependency injection) → beans.xml File**. Do not change any of the values and click **Finish**. The **beans.xml** file is created. In the newly created tab, update it to use the **Bean-Discovery-Mode as all**.

Press **Ctrl+S** to save your updates to the **beans.xml** file.

► 4. Inspect the route.

- 4.1. Open the **OrderRouteBuilder** class by expanding the **dynamic-route-cdi** item in the **Project Explorer** tab in the left pane of JBoss Developer Studio, then click **dynamic-route-cdi → src/main/java → com.redhat.training.jb421** to expand it. Double-click the **OrderRouteBuilder.java** file.

The file has a single route that reads from a database all the undelivered orders and sends them to the file system. It is updated later to support a dynamic route approach.

- 4.2. Run the JUnit test.

Expand **src/test/java → com.redhat.training.jb421**.

Right-click the **RouteTransformationTest** test case and select **Run As → JUnit Test**.

No test methods should pass. The tests will pass as you continue with this exercise.

► 5. Create a bean to support dynamic routing.

A bean will be responsible for defining the destination based on the following rule:

- Comic books are sent to a FTP server (**ftp://services.lab.example.com/**) with the user: **student** and password: **student**.
- Books are sent to a JMS queue named **comics** running on an ActiveMQ.
- Memorabilia are sent the order item details to a directory named **others**.

- 5.1. Evaluate the example content that will be consumed by the bean.

In a previous step, an XML file was generated by the route, and it should be evaluated to check which output should be manipulated. Open any of the files from the **orders/outcoming** directory. It must be similar to the following example:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<order>
  ...
  <orderItems>
    <orderItem>
      <catalogItem>
        <category>memorabilia</category>
      </catalogItem>
    </orderItem>
  </orderItems>
</order>
```

Notice that the category name is located at the following XPath expression: `/order/orderItems/orderItem/catalogItem/category/text()`. You must be used this expression in class that calculates the route URL.

- 5.2. Create a new class to be used as a Camel bean with CDI.

Right-click the project and select **New → Class**. Set the package name to **com.redhat.training.jb421.beans** and change the **Name** to **DestinationBean**. Leave the remaining fields with the default values. Click **Finish** to create the class.

- 5.3. Define the class a CDI component.

Over the class declaration, add the **@Named** annotation.

```
import javax.inject.Named;  
...  
  
@Named  
public class DestinationBean {  
...}
```

This allows a class to be identified as a CDI component.

- 5.4. Implement the logic for the bean processing.

A method will be added to the **DestinationBean** class to define the destination.

Camel allows XML processing using an XPath expression in an exchange body.

Implement the following method call:



Note

To minimize typing, the method source code is provided in the **method-bean.txt** file. To import the missing classes use the shortcut **Ctrl+Shift+O**.

```
public void processDestination(@XPath(value="/order/orderItems/orderItem/  
catalogItem/category/text()") String type, @Headers Map<String, Object> headers) {  
    String destination;  
    if("comics".equals(type)){  
        destination = "ftp://services.lab.example.com?  
username=student&password=student";  
    }else if("book".equals(type)){  
        destination = "jms:books";  
    }else{  
        destination = "file://others";  
    }  
    headers.put("destination", destination);  
}
```

Notice that the method call receives two parameters:

- An XPath expression: This is processed by Camel. Camel reads the body and looks for the value using the XPath expression. For this purpose, the XPath expression is identical to the one identified previously.
- A **java.util.Map** argument: This provides mechanisms to update the header from the **Exchange**.

The logic from the code identifies the destination based on the type.

► 6. Implementing **toD** method call.

- 6.1. The destination must be calculated by a bean and added as a header to the Camel message. Unfortunately, calculating the destination is not so straightforward because the component used depends on the content type. Thus, the route name must be processed by a Java class.

Using the **DestinationBean** class created in the previous step, read a field from the XML file and process it. Add to the **OrderRouteBuilder** class the bean call to trigger the destination bean.

Open the **OrderRouteBuilder**, right after the **unmarshal** method call the bean call:

```
.unmarshal(jaxbDataFormat)  
.bean("destinationBean")
```

Notice that the bean name is similar to the **DestinationBean** class name, using an initial lowercase. This is a CDI-specific feature.

- 6.2. Comment the **to** method and invoke the **toD** method to send to the correct destination as a parameter. Recall that destination was stored in the header with the key named **destination**. Add right after the bean method call the **toD** method call:

```
.bean("destinationBean")  
//.to("file://orders/outcoming");  
.toD("${header.destination}");
```

- 6.3. Right-click the **RouteTransformationTest** test case and select **Run As → JUnit Test**. A green bar is presented. All tests pass.

► 7. Implementing **routingSlip** method call.

- 7.1. Comment the **toD** method and invoke the **routingSlip** method to define the destination using the bean. To identify the header, use the header helper method as follows:

```
.bean("destinationBean")  
//.to("file://orders/outcoming");  
//.toD("${header.destination}");  
.routingSlip(header("destination"));
```

- 7.2. Right-click the **RouteTransformationTest** test case and select **Run As → JUnit Test**. A green bar is presented. All tests pass.

► 8. Clean up.

Close the project in JBoss Developer Studio to conserve memory.

In the **Project Explorer** view, right-click dynamic-route-cdi project and click **Close Project**.

This concludes the guided exercise.

Executing Bean Methods in Predicates

Objective

After completing this section, students should be able to execute bean methods within DSL predicates.

Executing Beans in Predicates

There are several EIP supported by Camel that handle complex routes and pipelines. In many instances, enterprises need routing to be dynamic or dependent on each individual message. Routes that leverage this capability are much more maintainable and reduce code duplication because they do not need to create multiple routes and pipelines to handle each individual use case, rather a single pipeline can support multiple uses cases.

In a similar manner, the filter EIP is a pattern that allows messages to be filtered out to an output channel rather than continuing along the route. To determine which messages are filtered out, the filter requires a condition, such as filtering out order forms with malformed email addresses. In many cases, however, the condition needs to be determined dynamically from an external system. For instance, a logistics company may not have distribution centers within a set of zip codes, therefore any exchanges sent to addresses within those zip codes must be refused. The acceptable zip codes, however, are subject to constant changes that a traditional filter does not support.

In Camel, a bean is commonly used to create dynamic predicates that access external systems, such as a database or a file. To become a predicate, a bean must meet the following requirements:

- The predicate must be declared as a bean: In Camel, this can be achieved by using the XML configuration file, CDI annotations, or using references to the class.
- The predicate must have a method returning a boolean value that the Camel component uses to evaluate if the condition is true or false.

To work with beans as predicates, use the DSL method named **method**:

```
from("direct:example")
.filter(method("validAddress","isValidAddress"))
...
```

In the previous snippet, the Camel route looks for the **validAddress** bean based on the first parameter, and invokes the **isValidAddress** method based on the second parameter:

```
@Named("validAddress")
public class ValidAddressBeanPredicate{
...
    public boolean isValidAddress(Exchange message){
    ...
}
}
```

In this example, the **filter** EIP executes the **isValidAddress** method by passing in the message as a **Exchange** argument. The method returns a boolean value to determine whether the message contains a valid address. If the method returns **false**, the message is filtered out by the filter EIP. If the method returns **true**, then the message continues along the route.

In addition to the **filter** EIP, the previously introduced content-based router EIP also heavily relies on predicates to route messages based on content. For example, the following route evaluates whether an order is placed by a VIP customer using the **isVIP** method in the **vipBean** component:

```
from("jms:orders")
    .choice()
        .when(method("vipBean","isVIP"))
            .to("jms:vipOrders")
                .when(method("vipBean","isRegular"))
                    .to("jms:regularOrders")
                .otherwise()
                    .to("jms:errorOrders")
            .endChoice();
```

Compound Predicates

A filter rule can be compounded with **and**, **or**, and **not** boolean operators using the **PredicateBuilder** class helper methods to create more sophisticated filtering.

For instance, to check if a header value is either **null** or empty, the following predicate can be used in a **filter** method:

```
PredicateisNullHeader = PredicateBuilder.or(header("email").isNull(),
                                             simple(" ${header.email==''} "));
```

After creating the **Predicate** object named **nullHeader**, add the predicate to a filter, content-based router or other component:

```
from("jms:in")
    .filter(isNullHeader)
    .to("jms:out")
```



Important

The **PredicateBuilder** class is supported only by Java DSL, not Spring DSL.

By creating compound predicates in the Java DSL, routes become more readable and maintainable because the complex logic is contained within the predicates rather than being embedded in the actual routes.



References

Camel Predicates

<http://camel.apache.org/predicate.html>

Camel in Action, Second edition - Chapter Using Beans with Camel

► Guided Exercise

Using Beans in Predicates

In this exercise, you will invoke beans with Predicates.

Outcomes

You should be able to customize a route using a bean to mimic a predicate.

Before You Begin

Run the following command to download the starter project and data files used by this exercise. Click **Applications** → **Favorites** → **Terminal** in the desktop menu and type:

```
[student@workstation ~]$ lab bean-predicates setup
```

► 1. Import the starter Maven project.

- 1.1. Open JBoss Developer Studio (**Applications** → **Programming** → **JBoss Developer Studio**). JBoss Developer Studio requests a default workspace; keep the default configuration (**/home/student/workspace**) and click **Launch**.
- 1.2. Import the Maven project by selecting the JBoss Developer Studio main menu **File** → **Import**.
- 1.3. From the **Import** dialog box, select **Maven** → **Existing Maven Projects** and click **Next >**.
- 1.4. Click **Browse** and choose **/home/student/JB421/labs/bean-predicate** and then click **OK** to choose the Maven project. Click **Finish** to complete the import process in JBoss Developer Studio.

A new project named **bean-predicate** is listed in the **Project Explorer** view.



Important

The project imports with errors, this is expected. You will resolve these errors in the subsequent steps of this exercise.

► 2. Create a predicate.

A bean predicate will be responsible for defining the destination based using the following rule:

- Orders from the **Orly** company that the invoice value is greater than \$ 0 are sent to a JMS queue named **approved** deployed on an ActiveMQ broker.
- Orders from other companies that the invoice value is greater than \$0 and less than \$ 100 are sent to a JMS queue named **approved** deployed on an ActiveMQ broker.
- Orders from other companies that the invoice value is equal or greater than \$ 100 are sent to a JMS queue named **needsApproval** deployed on an ActiveMQ broker.

- 2.1. Create a new class that will be used as a Camel bean predicate with CDI.

Right-click the project and select **New → Class**. Set the package name to `com.redhat.training.jb421.predicates` and change the **Name** to **ApprovalPredicate**. Leave the remaining fields with the default values. Click **Finish** to create the class.

- 2.2. Define the class a CDI component.

Over the class declaration, add the `@Named` annotation to identify it as a CDI component.

```
import javax.inject.Named;  
...  
@Named  
public class ApprovalPredicate {
```

- 2.3. Add three methods to define the destination according to the rule.



Note

To minimize typing, the method source code is provided in the **method-predicate.txt** file. To import the missing classes use the shortcut **Ctrl+Shift+O**.

```
private static final BigDecimal HUNDRED = new BigDecimal("100");  
  
public boolean isFromOrlyCompany(@XPath("/order/company") String company,  
    @XPath("/order/totalInvoice") String totalInvoice) {  
    return "Orly".equalsIgnoreCase(company)  
    && new BigDecimal(totalInvoice).compareTo(BigDecimal.ZERO) > 0;  
}  
  
public boolean isLessThan100(@XPath("/order/totalInvoice") String totalInvoice){  
    return new BigDecimal(totalInvoice).compareTo(HUNDRED) < 0  
    && new BigDecimal(totalInvoice).compareTo(BigDecimal.ZERO) > 0;  
}  
  
public boolean isGreaterThanOrEqual100(@XPath("/order/totalInvoice") String totalInvoice)  
{  
    return new BigDecimal(totalInvoice).compareTo(HUNDRED) >= 0;  
}
```

- 2.4. Press **Ctrl+S** to save your updates to the **ApprovalPredicate.java** file.

▶ 3. Update the **com.redhat.training.jb421.OrderRouteBuilder** class

The route evaluates the total order invoice and order company, as an elements from an XML file. The route invokes various beans as predicates to check which range the total value of the order is in. To support the use case, a **choice/when** clause is used.

- 3.1. Update the first **when** clause to invoke the **isFromOrlyCompany** method from the bean named **approvalPredicate**. Sent these orders to the **approved** queue.

```
//TODO Use the isFromOrlyCompany from the approvalPredicate predicate
.when(method("approvalPredicate","isFromOrlyCompany"))
.to("jms:approved")
```

- 3.2. Update the second **when** clause to invoke the **isLessThan100** method from the bean named **approvalPredicate**. Sent these orders to the **approved** queue.

```
//TODO Use the isLessThan100 from the approvalPredicate predicate
.when(method("approvalPredicate","isLessThan100"))
.to("jms:approved")
```

- 3.3. Update the third **when** clause to invoke the **isGreaterThan100** method from the bean named **approvalPredicate**. Sent these orders to the **needsApproval** queue.

```
//TODO Use the isGreaterThan100 from the approvalPredicate predicate
.when(method("approvalPredicate","isGreaterThan100"))
.to("jms:needsApproval")
```

- 3.4. If none of the **when** clauses are match, then update the **otherwise** method to invoke the **InvalidValueExceptionProcessor** processor.

```
//TODO Invoke the InvalidValueExceptionProcessor processor
.process(new InvalidValueExceptionProcessor())
```

- 4. Run the provided JUnit test case to verify your route.

Right-click the **com.redhat.training.jb421.BeanPredicateTest** test case and select **Run As → JUnit Test** to run it.



Note

These test methods can take a few minutes to complete. Wait until the final JUnit results are shown for each test.

A green bar should be presented. All tests should pass.

- 5. Clean up.

Close the project in JBoss Developer Studio to conserve memory.

In the **Project Explorer** view, right-click bean-predicate project and click **Close Project**.

This concludes the guided exercise.

▶ Lab

Routing with Java Beans

Performance Checklist

In this lab, you will implement routes that can change according to an algorithm using Camel CDI extensions and Java beans.

Outcomes

You should be able to implement dynamic routes using CDI extensions.

Before You Begin

Run the following command to download the starter project and data files used by this exercise. Click **Applications** → **Favorites** → **Terminal** in the desktop menu and type:

```
[student@workstation ~]$ lab beans-review setup
```

1. Import the beans-review project as an existing Maven project into JBoss Developer Studio. The project is available in the **/home/student/JB421/labs/beans-review** directory.



Important

The project imports with errors, but this is expected. You will resolve these errors in the subsequent steps of this exercise.

2. **Complete the project Maven dependencies.**

Add the **cdi-api** and the **camel-cdi** dependencies as provided in the **pom.xml** file. Also, configure the Maven Camel plug-in to support CDI. The **pom-plugin.xml.txt** file is provided for that purpose.

3. Include the CDI configuration file.

Include the **beans.xml** file to enable CDI support.

Press **Ctrl+S** to save your updates to the **beans.xml** file.

4. Evaluate the example content files provided with the **setup-data.sh** script.

- 4.1. Go back to the terminal window and run the provided shell script to populate the **orders/incoming** folder:

```
[student@workstation ~]$ cd ~/JB421/labs/beans-review
[student@workstation beans-review]$ ./setup-data.sh
...
'Preparation complete!'
```

- 4.2. Evaluate the order file to check if a order has the **backorder** element:

```
[student@workstation beans-review]$ cat orders/incoming/order-1.xml
<order>
<orderId>1</orderId>
<backorder>false</backorder>
<orderItems>
<orderItem>
<orderItemId>1</orderItemId>
<orderItemQty>1</orderItemQty>
<orderItemPublisherName>ORLy</orderItemPublisherName>
<orderItemPrice>10.59</orderItemPrice>
</orderItem>
</orderItems>
</order>
```

4.3. Inspect the orders **backorder** elements:

```
[student@workstation beans-review]$ grep backorder orders/incoming/*
orders/incoming/order-1.xml: <backorder>false</backorder>
orders/incoming/order-2.xml: <backorder>true</backorder>
orders/incoming/order-3.xml:      <backorder>false</backorder>
orders/incoming/order-4.xml:      <backorder>false</backorder>
orders/incoming/order-5.xml:      <backorder>false</backorder>
orders/incoming/order-6.xml:      <backorder>true</backorder>
```

There should be two orders with the **backorder** element as **true**.

5. Create a bean to support dynamic routing.

This bean is responsible for defining the destination based on the following rule:

- Orders that have the **backorder** element as **true** are sent to a JMS queue named **backorder** deployed on an AMQ broker.
- Orders that have **backorder** element as **false** are sent to a JMS queue named **shipment** deployed on an AMQ broker.

Create a new header named **destination**. The header value is **backorder** if the order has the **backorder** element value as **true**, and **shipment** if the **backorder** element value is **false**. You need this header later in the route. The code for the bean method is available at [/home/student/JB421/labs/bean-review/method-bean.txt](#).

6. Update the **com.redhat.training.jb421.OrderRouteBuilder** class.

The route should invoke the **bean** method to use the **DestinationBean** class, which sets a header with the destination. Use the **toD** method to produce a message to the correct AMQ queue.

7. Test that the route works as expected.

- Open a terminal window and execute the following commands to start the Red Hat AMQ 7.0 broker:

```
[student@workstation ~]$ cd ~/JB421/labs/beans-review/broker1/bin
[student@workstation bin]$ ./artemis run
```

When the broker finishes booting up, the terminal contains the following message:

```
[org.apache.activemq.artemis] AMQ241001: HTTP Server started at http://localhost:8161
[org.apache.activemq.artemis] AMQ241002: Artemis Jolokia REST API available at http://localhost:8161/console/jolokia
[org.apache.activemq.artemis] AMQ241004: Artemis Console available at http://localhost:8161/console
```

- 7.2. Open a terminal window and inspect the **shipment** queue. It should have no messages:

```
[student@workstation ~]$ cd ~/JB421/labs/beans-review/broker1/bin
[student@workstation bin]$ ./artemis browser --destination queue://shipment
...
Consumer ActiveMQQueue[shipment], thread=0 browsed: 0 messages
...
```

- 7.3. Inspect the **backorder** queue. It should have no messages:

```
[student@workstation bin]$ ./artemis browser --destination queue://backorder
...
Consumer ActiveMQQueue[backorder], thread=0 browsed: 0 messages
...
```

- 7.4. Open a terminal window and start the route by using the **camel:run** Maven goal:

```
[student@workstation ~]$ cd ~/JB421/labs/beans-review
[student@workstation beans-review]$ mvn clean camel:run -DskipTests
```

Wait for the route to fully start before proceeding to the next step.

- 7.5. Return to the terminal window that you are inspecting the queues and inspect the **backorder** queue. It should have two messages:

```
[student@workstation bin]$ ./artemis browser --destination queue://backorder
...
Consumer ActiveMQQueue[backorder], thread=0 browsed: 2 messages
...
```

- 7.6. Inspect the **shipment** queue. It should have four messages:

```
[student@workstation bin]$ ./artemis browser --destination queue://shipment
...
Consumer ActiveMQQueue[shipment], thread=0 browsed: 4 messages
...
```

- 7.7. Return to the terminal window running Camel and stop it by pressing **Ctrl+C**.

- 7.8. Return to the terminal window running the Red Hat AMQ broker and stop it by pressing **Ctrl+C**.

8. Grade the lab.

```
[student@workstation beans-review]$ lab beans-review grade
```

9. Clean up: close the project in JBoss Developer Studio to conserve memory.
In the **Project Explorer** view, right-click beans-review project and click **Close Project**.
This concludes the lab.

► Solution

Routing with Java Beans

Performance Checklist

In this lab, you will implement routes that can change according to an algorithm using Camel CDI extensions and Java beans.

Outcomes

You should be able to implement dynamic routes using CDI extensions.

Before You Begin

Run the following command to download the starter project and data files used by this exercise. Click **Applications** → **Favorites** → **Terminal** in the desktop menu and type:

```
[student@workstation ~]$ lab beans-review setup
```

1. Import the beans-review project as an existing Maven project into JBoss Developer Studio. The project is available in the **/home/student/JB421/labs/beans-review** directory.



Important

The project imports with errors, but this is expected. You will resolve these errors in the subsequent steps of this exercise.

- 1.1. Open JBoss Developer Studio (**Applications** → **Programming** → **JBoss Developer Studio**). JBoss Developer Studio requests a default workspace; keep the default configuration (**/home/student/workspace**) and click **Launch**.
- 1.2. Import the Maven project by selecting the JBoss Developer Studio main menu **File** → **Import**.
- 1.3. From the **Import** dialog box, select **Maven** → **Existing Maven Projects** and click **Next >**.
- 1.4. Click **Browse** and choose **/home/student/JB421/labs/beans-review** and click **OK** to choose the Maven project. Click **Finish** to complete the import process in JBoss Developer Studio.

A new project named **beans-review** is listed in the **Project Explorer** view.

2. **Complete the project Maven dependencies.**

Add the **cdi-api** and the **camel-cdi** dependencies as provided in the **pom.xml** file. Also, configure the Maven Camel plug-in to support CDI. The **pom-plugin.xml.txt** file is provided for that purpose.

- 2.1. Expand the **beans-review** item in the **Project Explorer** pane on the left, and then double-click the **pom.xml** file.

- 2.2. Click the **pom.xml** tab at the bottom of the file to view the contents of the **pom.xml** file.
- 2.3. Add the **cdi-api** and **camel-cdi** dependencies to the project.

```
<!-- add dependency for CDI -->
<dependency>
  <groupId>javax.enterprise</groupId>
  <artifactId>cdi-api</artifactId>
  <scope>provided</scope>
</dependency>
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-cdi</artifactId>
</dependency>
```

- 2.4. Add CDI support for the Maven Camel plug-in.

Look for the **<!-- add dependency for Camel CDI plugin -->** comment and add the following configuration to support Maven Camel plug-in CDI support:



Note

The **pom-plugin.xml.txt** file is provided to copy and paste the contents.

```
<configuration>
  <useCDI>true</useCDI>
</configuration>
<dependencies>
  <dependency>
    <groupId>org.apache.deltaspike.cdictrl</groupId>
    <artifactId>deltaspike-cdictrl-weld</artifactId>
    <version>1.8.1</version>
  </dependency>
  <dependency>
    <groupId>org.jboss.weld.se</groupId>
    <artifactId>weld-se</artifactId>
    <version>2.4.6.Final</version>
  </dependency>
</dependencies>
```

- 2.5. Press **Ctrl+S** to save your updates to the **pom.xml** file.

3. Include the CDI configuration file.

Include the **beans.xml** file to enable CDI support.

Expand the **src/main/resources/META-INF** folder and right-click it. Select **New → Other...** and choose **CDI (Contexts and Dependency injection) → beans.xml file** and click **Next >**. Do not change any of the values and click **Finish**. The **beans.xml** file is created. In the newly created tab, update it to use the **Bean-Discovery-Mode as all**.

Press **Ctrl+S** to save your updates to the **beans.xml** file.

4. Evaluate the example content files provided with the **setup-data.sh** script.

- 4.1. Go back to the terminal window and run the provided shell script to populate the **orders/incoming** folder:

```
[student@workstation ~]$ cd ~/JB421/labs/beans-review
[student@workstation beans-review]$ ./setup-data.sh
...
'Preparation complete!'
```

- 4.2. Evaluate the order file to check if a order has the **backorder** element:

```
[student@workstation beans-review]$ cat orders/incoming/order-1.xml
<order>
  <orderId>1</orderId>
  <backorder>false</backorder>
  <orderItems>
    <orderItem>
      <orderItemId>1</orderItemId>
      <orderItemQty>1</orderItemQty>
      <orderItemPublisherName>ORly</orderItemPublisherName>
      <orderItemPrice>10.59</orderItemPrice>
    </orderItem>
  </orderItems>
</order>
```

- 4.3. Inspect the orders **backorder** elements:

```
[student@workstation beans-review]$ grep backorder orders/incoming/*
orders/incoming/order-1.xml: <backorder>false</backorder>
orders/incoming/order-2.xml: <backorder>true</backorder>
orders/incoming/order-3.xml:           <backorder>false</backorder>
orders/incoming/order-4.xml:           <backorder>false</backorder>
orders/incoming/order-5.xml:           <backorder>false</backorder>
orders/incoming/order-6.xml:           <backorder>true</backorder>
```

There should be two orders with the **backorder** element as **true**.

5. Create a bean to support dynamic routing.

This bean is responsible for defining the destination based on the following rule:

- Orders that have the **backorder** element as **true** are sent to a JMS queue named **backorder** deployed on an AMQ broker.
- Orders that have **backorder** element as **false** are sent to a JMS queue named **shipment** deployed on an AMQ broker.

Create a new header named **destination**. The header value is **backorder** if the order has the **backorder** element value as **true**, and **shipment** if the **backorder** element value is **false**. You need this header later in the route. The code for the bean method is available at **/home/student/JB421/labs/bean-review/method-bean.txt** file.

- 5.1. Create a new class that will be used as a Camel bean with CDI.

Right-click the project and select **New → Class**. Set the package name to **com.redhat.training.jb421.beans** and change the **Name** to

DestinationBean. Leave the remaining fields with the default values. Click **Finish** to create the class.

- 5.2. Define the **com.redhat.training.jb421.beans.DestinationBean** class as a CDI bean.

Add the **@Named** annotation above the class declaration.

```
import javax.inject.Named;
@Named
public class DestinationBean {
```

This allows a class to be identified as a CDI component.

- 5.3. Implement the logic for the bean processing.

A method will be added to the **DestinationBean** class to define the destination. Camel allows XML processing using XPath expression in an exchange body. Implement the following method call:



Note

To minimize typing, the method source code is provided in the **method-bean.txt** file. To import the missing classes use the shortcut **Ctrl+Shift+O**.

```
public void processDestination(
    @XPath(value="/order/backorder/text()") String backorder,
    @Headers Map<String, Object> headers) {
    String destination;
    if (Boolean.valueOf(backorder)) {
        destination = "backorder";
    } else {
        destination = "shipment";
    }
    headers.put("destination", destination);
}
```

- 5.4. Press **Ctrl+S** to save your updates to the file.

6. Update the **com.redhat.training.jb421.OrderRouteBuilder** class.

The route should invoke the **bean** method to use the **DestinationBean** class, which sets a header with the destination. Use the **toD** method to produce a message to the correct AMQ queue.

- 6.1. Open the **OrderRouteBuilder** class, just after the **routeId** method invoke the bean:

```
.routeId("destination")
.bean("destinationBean")
```

- 6.2. Call the **toD** method to send to the correct destination as a parameter. Recall that the destination was stored in the header with the key named **destination**. Add the **toD** method call just after the bean method call:

```
.bean("destinationBean")
.to("activemq:queue:${header.destination}?username=admin&password=admin");
```

7. Test that the route works as expected.

- 7.1. Open a terminal window and execute the following commands to start the Red Hat AMQ 7.0 broker:

```
[student@workstation ~]$ cd ~/JB421/labs/beans-review/broker1/bin
[student@workstation bin]$ ./artemis run
```

When the broker finishes booting up, the terminal contains the following message:

```
[org.apache.activemq.artemis] AMQ241001: HTTP Server started at http://localhost:8161
[org.apache.activemq.artemis] AMQ241002: Artemis Jolokia REST API available at http://localhost:8161/console/jolokia
[org.apache.activemq.artemis] AMQ241004: Artemis Console available at http://localhost:8161/console
```

- 7.2. Open a terminal window and inspect the **shipment** queue. It should have no messages:

```
[student@workstation ~]$ cd ~/JB421/labs/beans-review/broker1/bin
[student@workstation bin]$ ./artemis browser --destination queue://shipment
...
Consumer ActiveMQQueue[shipment], thread=0 browsed: 0 messages
...
```

- 7.3. Inspect the **backorder** queue. It should have no messages:

```
[student@workstation bin]$ ./artemis browser --destination queue://backorder
...
Consumer ActiveMQQueue[backorder], thread=0 browsed: 0 messages
...
```

- 7.4. Open a terminal window and start the route by using the **camel:run** Maven goal:

```
[student@workstation ~]$ cd ~/JB421/labs/beans-review
[student@workstation beans-review]$ mvn clean camel:run -DskipTests
```

Wait for the route to fully start before proceeding to the next step.

- 7.5. Return to the terminal window that you are inspecting the queues and inspect the **backorder** queue. It should have two messages:

```
[student@workstation bin]$ ./artemis browser --destination queue://backorder
...
Consumer ActiveMQQueue[backorder], thread=0 browsed: 2 messages
...
```

- 7.6. Inspect the **shipment** queue. It should have four messages:

```
[student@workstation bin]$ ./artemis browser --destination queue://shipment  
...  
Consumer ActiveMQQueue[shipment], thread=0 browsed: 4 messages  
...
```

- 7.7. Return to the terminal window running Camel and stop it by pressing **Ctrl+C**.
- 7.8. Return to the terminal window running the Red Hat AMQ broker and stop it by pressing **Ctrl+C**.
- 8.** Grade the lab.

```
[student@workstation beans-review]$ lab beans-review grade
```

- 9.** Clean up: close the project in JBoss Developer Studio to conserve memory.
In the **Project Explorer** view, right-click beans-review project and click **Close Project**.
This concludes the lab.

Summary

In this chapter, you learned:

- Camel supports invoking the **bean** method from a route by using the bean component.
- When using context dependency injection (CDI), Camel is able to recognize any bean annotated with CDI annotations for use in a route without additional configuration.
- In Camel, a bean is commonly used to create dynamic predicates that access external systems, such as a database or a file.
- The recipient list EIP is an integration pattern wherein a message is routed to multiple destinations in parallel.
- You must use the **toD** DSL method for any endpoint URLs which must be calculated at runtime by a bean method or other mechanism.
- The **routingSlip** DSL method allows you to route a message to series of destinations in order.
- The **dynamicRouter** is similar to the routing slip except it can calculate the list of destinations at runtime using a bean method.

Chapter 6

Implementing REST Services

Goal

Enabling REST support in Camel with the Java REST DSL

Objectives

- Create a route that hosts a REST Service using the REST DSL.
- Consume HTTP resources to implement the content enricher pattern.
- Document a REST service using the REST DSL integration with Swagger.

Sections

- Implementing a REST Service with REST DSL (and Guided Exercise)
- Consuming Web Services with the HTTP Component (and Guided Exercise)
- Implementing REST Service Documentation with Swagger (and Guided Exercise)

Lab

Implementing REST Services

Implementing a REST Service with the REST DSL

Objectives

After completing this section, students should be able to:

- Create a route that hosts a REST service using the REST DSL.
- Customize a REST service to use various data bindings.

Introducing the REST DSL API

Beginning in version 2.14, Camel offers a REST DSL which can be used in Java or XML route definitions to build REST web services. The REST DSL allows end users to define REST services in Camel routes using verbs that align with the REST HTTP protocol, such as **GET**, **POST**, **DELETE**, and so on. This drastically reduces the amount of development time necessary to build REST services into your Camel routes by eliminating a lot of the boilerplate networking code, and allowing you to focus on the business logic that supports the REST service.

The REST DSL is a facade supporting multiple REST component implementations, including **camel-spark-rest**, **camel-undertow**, and others. The DSL builds REST endpoints as consumers for Camel routes. The actual REST service implementation is provided by other Camel components, such as Restlet, Spark, and others that include REST integration.

The following is an example of the REST DSL in use in a Java route definition:

```
public class HelloRoute extends RouteBuilder {  
  
    public void configure() throws Exception {  
        restConfiguration()  
            .component("spark-rest")  
            .port(8080);  
  
        rest("/speak")  
            .get("/hello")  
            .transform().constant("Hello World");  
    }  
}
```

The REST DSL works as an extension to the existing Camel routing DSL, using specialized keywords to more closely resemble the underlying REST and HTTP technologies. These keywords are mapped directly into the existing Camel DSL, meaning that the REST DSL provides a simple syntax that extends Camel's existing DSL. This also means that all existing functionality of a normal Camel route is available inside of a REST DSL defined route, enabling REST developers to leverage EIPs and other Camel features to implement their service.

Configuring the REST DSL

When using the REST DSL, you need to specify which of the REST DSL capable components handle the requests made to the REST services. Each of the underlying implementations is different, but their functionality as it relates to the REST DSL is largely the same. Each component

offers slightly different options, depending on your use case, you may find one to work better than another. This course focuses on **camel-spark-rest**, but all of the concepts taught here should apply to the other REST DSL supported components. The following are some of the components that currently support the REST DSL:

- **camel-jetty**: Uses the Jetty HTTP server
- **camel-restlet**: Uses the Restlet library
- **camel-spark-rest**: Uses the Java Spark library
- **camel-undertow**: Uses the JBoss Undertow HTTP server

In order to specify the REST implementation used by the REST DSL, and other behavior as well, a separate DSL element, the **restConfiguration** DSL method is provided. Using this element, a number of options can be set to control the resulting REST service created by Camel, as shown in the following example:

```
restConfiguration()  
    .component("spark-rest")  
    .contextPath("/restService")  
    .port(8080);
```

Or similarly in the XML DSL:

```
<restConfiguration component="spark-rest" contextPath="/restService" port="8080"/>
```

Because the REST DSL is not an implementation, only a subset of the options common to all implementations, most options are specific to the REST component used by the DSL. The following is a table of the common options across all components:

REST DSL Common Configuration Options

Option	Description
component	The Camel component to use as the HTTP server. Options include jetty , restlet , spark-rest , and undertow .
scheme	The HTTP scheme to use, HTTP or HTTPS. HTTP is the default.
hostname	The host name or IP where the HTTP server is bound.
port	The port number to use for the HTTP server.
contextPath	The base context path for the HTTP server.

You can also set options on the **restConfiguration** DSL element to configure the component, endpoint, and consumer specifically. Because these can vary from component to component, to set these options you need to use a generic DSL element shown in the following table:

REST Configuration Generic Options

Option Type	DSL Element
component	componentProperty
endpoint	endpointProperty
consumer	consumerProperty

To use any of these properties, you must set a key and value, where the key corresponds to the name of a property available for that component, endpoint, or consumer. An example of when this could be used is to set the minimum and maximum threads for the Jetty server as shown in the following example:

```
restConfiguration()
    .component("jetty").port(8080)
    .componentProperty("minThreads", "1")
    .componentProperty("maxThreads", "8");
```



Note

Ensure that any component, endpoint, or consumer properties you set are using key values that match the component, endpoint, or consumer available options. If you try to set an option that does not exist, then the route compiles but an error is thrown at runtime.

Developing with the REST DSL

After you have configured the component for the REST DSL to use, adding a REST service definition to your Camel route is simple. First, define the set of services with the **rest** element, and set the context path that is specific to this set of services.

You can then define individual services using REST DSL methods such as **get**, **post**, **put**, and **delete**. You can also define path parameters using the `{}` syntax for each service. The following example **RouteBuilder** class shows a set of four services defined with the REST DSL:

```
public class OrderRoute extends RouteBuilder {

    public void configure() throws Exception {

        restConfiguration()
            .component("spark-rest").port(8080);

        rest("/orders")
            .get("{id}")①
                .to("bean:orderService?method=getOrder(${header.id})")
            .post()②
                .to("bean:orderService?method=createOrder")
            .put()③
                .to("bean:orderService?method=updateOrder")
            .delete("{id}")④
    }
}
```

```

        .to("bean:orderService?method=cancelOrder(${header.id})");
    }
}

```

- ① Maps to any HTTP GET requests received at `http://localhost:8080/orders/id`
- ② Maps to any HTTP POST requests received at `http://localhost:8080/orders/`
- ③ Maps to any HTTP PUT requests received at `http://localhost:8080/orders/`
- ④ Maps to any HTTP DELETE requests received at `http://localhost:8080/orders/id`

Customizing the REST Payload with Data Binding

The REST DSL supports automatic binding of XML and JSON data to POJOs using Camel's data formats. This means incoming JSON or XML data is automatically unmarshaled into model objects so that any processing done inside the service can use your Java model classes instead of raw JSON or XML data. For example, a service that consumes new order records in JSON format can automatically unmarshal that JSON into the **Order** model class for easier processing by subsequent components in the Camel route.

Binding with the REST DSL supports the modes listed in the following table and they are defined in the **org.apache.camel.model.rest.RestBindingMode** enumeration:

REST DSL Binding Modes

Mode	Description
off	Binding is turned off. This is the default.
auto	Binding is automatic, assuming the necessary data formats are available on the class path. Typically based on the Content-Type header.
json	Binding to and from JSON is enabled and a JSON compatible data format is required, such as camel-jackson .
xml	Binding to and from XML is enabled, requires camel-jaxb on the class path.
json_xml	Binding to and from JSON and XML are both enabled. Requires both data formats to be available on the class path.

Similar to other configurations for REST DSL, the binding mode is set on the **restConfiguration** element, as shown in the following example:

```

restConfiguration()
    .component("spark-rest").port(8080)
    .bindingMode(RestBindingMode.json)
    .dataFormatProperty("prettyPrint", "true");

```

Also, similar to component or endpoint properties, data format properties specific to the data format you are using can be set generically using **dataFormatProperty**. In the previous example, Jackson's **prettyPrint** option is set to **true** using a data format property that formats the JSON output in a human-readable format.

Implementing REST APIs with Spark

The **spark-rest** component allows Camel to define REST endpoints using the Spark REST Java library using REST DSL. To use this component, include the following Maven dependency in your **pom.xml** file:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-spark-rest</artifactId>
</dependency>
```

To enable the spark-rest component with REST DSL, specify it in the **component** option on the **restConfiguration** DSL element as shown in the following example:

```
restConfiguration()
    .component("spark-rest").port(8080);
```

Managing Errors Raised by REST APIs using onException

When you develop a REST API, sometimes it is necessary to catch exceptions that occur during processing and return the correct HTTP error code along with an optional message with more information.

For example, if a service supports getting a customer by ID, and an ID is passed in that does not match any of the customers in the database, an HTTP 204 code (not found) must be returned. This is simple with Camel's built-in **onException** DSL element. You are only required to define the error code and body to be returned for each exception you want to handle.

The following example demonstrates handling errors in a REST DSL service:

```
restConfiguration()
    .component("undertow").port(8080)
    .bindingMode(RestBindingMode.json)
    .dataFormatProperty("prettyPrint", "true");

onException(CustomerNotFoundException.class)
    .handled(true)
    .setHeader(Exchange.HTTP_RESPONSE_CODE, constant(204))
    .setBody(constant("No customer found."));
```

Demonstration: Implementing a REST Service with the REST DSL API

Use Case: Create a REST service with four endpoints:

- Get an existing order by its ID
- Create an empty order.
- Add an order item to the order using XML data.
- Delete the order and its items.

1. Start JBoss Developer Studio and import the **implement-rest-dsl** project from the **/home/student/JB421/labs/** directory.
2. Evaluate the **pom.xml** file and the current Maven dependencies defined for the project.
3. Add a dependency for the **camel-spark-rest** library to the **pom.xml** file following the **TODO** comment.
4. Evaluate the **com.redhat.training.jb421.OrderItemProcessor** class.

This class is used by the HTTP GET method route, to combine an order with its order items. It throws a **NoResultException** exception if the HTTP GET method is called with an order ID that does not exist.

5. Evaluate the **com.redhat.training.jb421.OrderRouteBuilder** class.

At the beginning of the route definition the exception handling mechanism catches any **NoResultException** exception thrown when you call the HTTP GET method with an ID that does not exist.

```
onException(NoResultException.class)
    .handled(true)
    .setHeader(Exchange.HTTP_RESPONSE_CODE, constant(400))
    .setHeader(Exchange.CONTENT_TYPE, constant("text/plain"))
    .setBody().constant("No order found with ID!");

// configure rest-dsl
restConfiguration()
    // to use spark-rest component and run on port 8080
    .component("spark-rest").port(8080)
    .bindingMode(RestBindingMode.json);

// rest services under the orders context-path
rest("/orders")
    .get("{id}")
        .to("direct:getOrder")
    .post("{id}").type(Order.class)
        .to("direct:createOrder")
    .put("{id}").type(OrderItem.class)
        .to("direct:updateOrder")
    .delete("{id}").type(Order.class)
        .to("direct:cancelOrder");
```

The REST DSL route defines HTTP verbs to four distinct services. Each of these services uses a **direct** endpoint to connect the REST DSL service to the back-end business logic.

6. Execute the route using the Maven Camel plug-in skipping tests.

Wait for the route to fully start before proceeding to the next step.

7. Install the RESTClient plug-in for Firefox.

Open the **/home/student/restclient.xpi** file in your Firefox browser to install the plug-in.

8. Test the **createOrder** REST endpoint using the RESTClient plug-in for Firefox.

Set a custom header with the name of **Content-Type** and the value of **application/json**.

Send an HTTP POST request to `http://localhost:8080/orders/orderId` replacing **orderId** with any positive integer

9. Verify the results in the database.

Use the MySQL CLI to select from the **order_** table and verify that the order was created. The **check-orders.sh** script provides you the command.

10. Test the **updateOrder** REST endpoint using the RESTClient plug-in.

Send an HTTP PUT request to `http://localhost:8080/orders/orderId` replacing **orderId** with the number used in step 8. Use the content of `/home/student/JB421/data/orderItemJSON/orderItem.json` as the body of the request.

11. Verify the results in the database.

Use the MySQL CLI to select from the **OrderItem** table and verify that the order item was created. The **check-items.sh** script provides you the command.

12. Test the **getOrder** REST endpoint using the RESTClient plug-in.

Send an HTTP GET request to `http://localhost:8080/orders/orderId` replacing **orderId** with the number used in step 8. Make sure the body of the request is empty. You should get a **200 OK** status with the details of the order displayed as response headers.

13. Test the **cancelOrder** REST endpoint using the RESTClient plug-in.

Send an HTTP DELETE request to `http://localhost:8080/orders/orderId` replacing **orderId** with the number used in step 8.

14. Verify the results in the database.

Use the MySQL CLI to select from the **order_** table and verify that the order was deleted. The **check-orders.sh** script provides you the command.

Use the MySQL CLI to select from the **OrderItem** table and verify that the order item was deleted. The **check-items.sh** script provides you the command.

15. Test the **getOrder** REST operation exception handling using the RESTClient plug-in.

Send an HTTP GET method to `http://localhost:8080/orders/orderId` replacing **orderId** with the number used in step 8. Make sure the body of the request is empty. You should get an HTTP 400 status code since this order has been deleted. You can see the error message in the **Response** section of the RESTClient.

16. Clean up.

Stop the execution of the Maven plug-in, and close the **implement-rest-dsl** project in JBoss Developer Studio.

This concludes the demonstration.



References

REST DSL

<http://camel.apache.org/rest-dsl.html>

Spark-REST component

<http://camel.apache.org/spark-rest.html>

Camel in Action, Second Edition - Chapter REST and Web Services

► Guided Exercise

Guided Exercise: Implementing a REST Service with the REST DSL

In this exercise, you will implement a REST service using the REST DSL that uses the **sql** component to read order information from the database.

Outcomes

You should be able to implement a route that hosts a REST service that implements three use cases:

- Retrieve the shipping address for an order by its ID.
- Calculate the total for an order by its ID.
- Retrieve the list of items for an order by its ID.

Before You Begin

A starter project is provided for you. It includes a **CamelContext** configured in a Spring configuration file, a **RouteBuilder** subclass, and an integration test for you to use to verify your work.

Run the following command to download the starter project and data files used by this exercise:

```
[student@workstation ~]$ lab rest-dsl setup
```

Steps

- 1. Open JBoss Developer Studio and import the starter Maven project.
 - 1.1. Open JBoss Developer Studio (**Applications** → **Programming** → **JBoss Developer Studio**). JBoss Developer Studio requests a default workspace; keep the default configuration (**/home/student/workspace**) and click **Launch**.
 - 1.2. Import the Maven project by selecting JBoss Developer Studio main menu **File** → **Import**.
 - 1.3. From the **Import** dialog box, select **Maven** → **Existing Maven Projects** and click **Next**.
 - 1.4. Click **Browse** and choose **/home/student/JB421/labs/rest-dsl** and click **OK** to choose the Maven project. Click **Finish** to complete the import process in JBoss Developer Studio.
A new project named **rest-dsl** will be listed in the **Project Explorer** view.
- 2. Review the Spring and Camel configurations.

In the **Project Explorer** view, expand **rest-dsl** → **src/main/resources** → **/META-INF/spring**. Double-click **bundle-camel-context.xml** file to review the Spring configuration for the starter project.

Notice the following components are defined:

- The element named **camelContext** is the single route builder in the Camel context. Nested as part of the element, there is another element named **routeBuilder** which refers to the single route builder.
- The data source identified by **mysqlDataSource** is the data source used by Camel to connect using the sql component. The database necessary for this exercise is already running and populated with the necessary tables.
- The entity manager factory connects the persistence context with the **mysqlDataSource** bean to provide JPA persistence functionality. A transaction manager is also defined for use with JPA persistence.

► 3. Review the route definition.

In the **Project Explorer** view, expand **src/main/java** → **com.redhat.training.jb421**. Double-click the **OrderRouteBuilder.java** file.

- 3.1. Review the implementation of the business logic of the services using the sql component as follows:

```
// routes that implement the REST services
from("direct:shipAddress")
    .log("Retrieving shipping address for order with id ${header.id}")
    .to("sql:select * from bookstore.Address where id IN (
        + "select ship_addr_id from bookstore.Customer where id IN "
        + "(" select cust_id from bookstore.order_ where id = :#id }}"
        + "?dataSource=mysqlDataSource&outputType=SelectOne"
        + "&outputClass=com.redhat.training.jb421.model.Address");

from("direct:orderTotal")
    .log("Retrieving order total for order with id ${header.id}")
    .to("sql:select * from bookstore.OrderItem where order_id = :#id"
        + "?dataSource=mysqlDataSource&outputType=SelectList"
        + "&outputClass=com.redhat.training.jb421.model.OrderItem")
    .bean(OrderTotalBean.class, "getOrderTotal(${body})");

from("direct:itemList")
    .log("Retrieving order items for order with id ${header.id}")
    .to("sql:select * from bookstore.OrderItem where order_id = :#id"
        + "?dataSource=mysqlDataSource&outputType=SelectList"
        + "&outputClass=com.redhat.training.jb421.model.OrderItem");
```

Notice the **direct:orderTotal** route uses the **OrderTotalBean** class to calculate the total cost of an order.

- 3.2. Review the three REST endpoints defined in REST DSL as follows:

```
@Override
public void configure() throws Exception {
    ...
}
```

```
// configure rest-dsl
restConfiguration();
    //TODO set to use spark-rest component and run on port 8080

    //TODO update the binding to support JSON
    ...
    // rest services under the orders context-path
rest("/orders")
    .get("/shipAddress/{id}")
        .to("direct:shipAddress")
    .get("/orderTotal/{id}")
        .to("direct:orderTotal")
    .get("/itemList/{id}")
        .to("direct:itemList");
    ...

```

Notice the following important pieces of the REST DSL definition:

- REST DSL defines three endpoints.
- All three endpoints are mapped to HTTP GET requests.
- Each service uses its own unique context path (defined as the argument for the **get** DSL method).
- All three services rely on a path parameter named **id**, which is declared between braces (**{}**).

3.3. Review the exception handling defined in the route.

```
onException(NonUniqueResultException.class)
    .handled(true)
    .setHeader(Exchange.HTTP_RESPONSE_CODE, constant(500))
    .setHeader(Exchange.CONTENT_TYPE, constant("text/plain"))
    .setBody().constant("Data error finding shipping address with ID!");
```

The sql component can throw a **NonUniqueResultException** exception when it returns more than one result and the **outputType** parameter is set to the **SelectOne** value. In this route, the **onException** method at the top of the route definition handles the **NonUniqueResultException** exception using to return a HTTP 500 status code to the client.

► 4. Add the missing dependencies to enable the REST DSL and API documentation.

In the **pom.xml** file, add the **camel-spark-rest** dependency to the project to allow the **spark-rest** component to create a REST service using the REST DSL.

Under the **<!-- TODO add camel-spark-rest dependency -->** comment in the **pom.xml** file add the following dependency:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-spark-rest</artifactId>
</dependency>
```

- 5. Finish the configuration of the REST DSL from the **OrderRouteBuilder** class using the **restConfiguration** DSL method.

- 5.1. Set **spark-rest** as the component to implement the REST services.

Just after the **restConfiguration** fluent interface method, directly following the comment **//TODO set to use spark-rest component and run on port 8080**, add the following DSL to set the component.

```
.component("spark-rest")
```

Remove the semi-colon from the **restConfiguration** method invocation.

- 5.2. Configure the services to run on port 8080.

Set the **port** option, appended to the **component** DSL element as follows:

```
.component("spark-rest").port(8080)
```

- 5.3. Configure the services to automatically bind JSON data to Java objects.

In the **restConfiguration** DSL method, directly following the comment **//TODO updating bind to support JSON** add the following DSL to enable automatic JSON binding.

```
.bindingMode(RestBindingMode.json);
```

- 5.4. Press **Ctrl+S** to save your changes to the route definition.

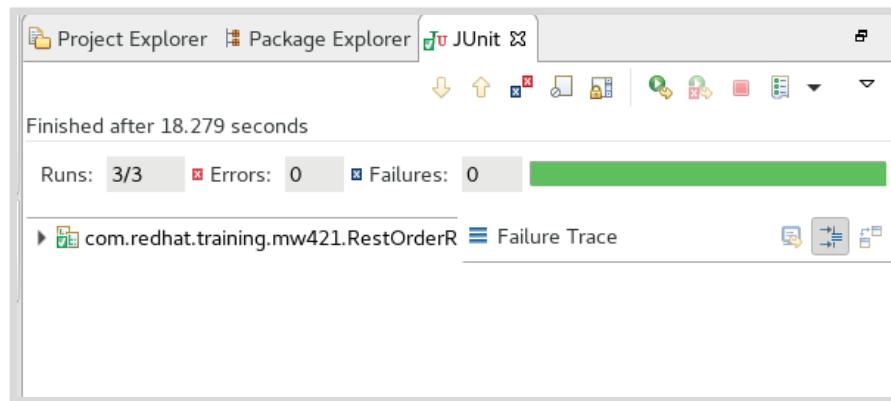
- 6. Run the provided JUnit test to verify your route.

- 6.1. Run the unit test.

Right-click the **RestOrderRouteTest** test case from the **com.redhat.training.jb421** package and select **Run As → JUnit Test** to run the test.

- 6.2. Check the JUnit test outcome.

Open the **JUnit** tab, if all tests passed, it resembles the following screen shot:



- 6.3. Review the console log.

Open the **Console** tab in the bottom pane of JBoss Developer Studio, if the route is functioning properly it ends with the following text output:

```
RestOrderRouteTest - 
*****
RestOrderRouteTest - Testing done:
  testShipAddress(com.redhat.training.jb421.RestOrderRouteTest)
RestOrderRouteTest - Took: 3.240 seconds (3240 millis)
RestOrderRouteTest - 
*****
```

► 7. Create test data using the provided script.

- 7.1. Review the script that creates the test data.

In JBoss Developer Studio, open the **setup-data.sh** file at the root of the **rest-dsl** project and review the SQL statements.

Notice that the script inserts a new order with one item, a new customer for that order, and a new address for the shipping address. This test data will be retrieved later in the exercise while testing the REST services.

- 7.2. Execute the script to add the data.

Open a new terminal window, navigate to **/home/student/JB421/labs/rest-dsl** and run **./setup-data.sh**:

```
[student@workstation ~]$ cd ~/JB421/labs/rest-dsl
[student@workstation rest-dsl]$ ./setup-data.sh
Data setup complete!
```

► 8. Start the routes for testing.

In the same terminal window, run the following command to start the route using the Camel Maven plug-in:

```
[student@workstation rest-dsl]$ mvn camel:run -DskipTests
```

Wait until the route is started. Search in the terminal window for the following output before proceeding:

```
INFO Total 6 routes, of which 6 are started.
INFO Apache Camel 2.21.0.fuse-000077-redhat-1 (CamelContext: jb421Context)
      started in 0.908 seconds
```

► 9. Test the route using RESTClient plug-in for Firefox.

- 9.1. Open a new Firefox window **Applications → Internet → Firefox Web Browser**.

- 9.2. Install the RESTClient plug-in for Firefox.

In the Firefox window, press **Ctrl+O** to open the **Open File** dialog. Navigate to the **/home/student/** directory and select the **restclient.xpi** file. Press **Open** to install the plug-in.

Press **Install** to confirm the RESTClient plug-in installation.

- 9.3. Open the plug-in.

In the Firefox window, look at the icons on the right side of the toolbar at the top of the window. Click the red icon shown in the following screen capture to launch the RESTClient plug-in:

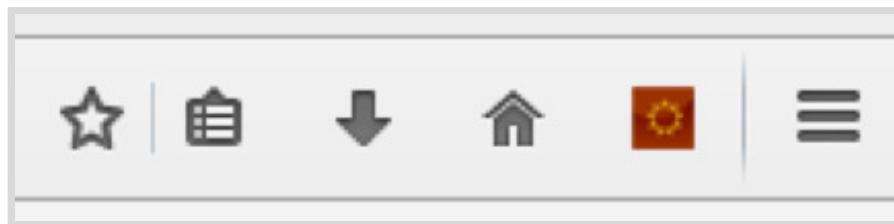


Figure 6.2: RESTClient plug-in icon is shown second from the right.

9.4. Get the shipping address of an order.

In the RESTClient window, under the **Request** section, set the method to GET and the URL to `http://localhost:8080/orders/shipAddress/99`.

Click the **SEND** button to send the request to service. If successful, you see an HTTP status code **200** in the **Response** section.

Check the raw response body by clicking **Response** and you should see output similar to the following:

```
{"id":99,  
"streetAddress1":"100 East Davie Street",  
"streetAddress2": "",  
"streetAddress3": "",  
"city":"Raleigh",  
"state":"NC",  
"postalCode":"27601",  
"country":"USA"}
```

9.5. Get the total cost of an order by its ID.

In the RESTClient window, under the **Request** section, set the method to GET and the URL to `http://localhost:8080/orders/orderTotal/99`.

Click the **SEND** button to send the request to service. If successful, you see an HTTP status code **200** in the **Response** section.

Check the raw response body by clicking **Response** and you should see output identical to the following:

31.98

9.6. Retrieve the list of items for an order by its ID.

In the RESTClient window, under the **Request** section, set the method to GET, the URL to `http://localhost:8080/orders/itemList/99`.

Click the **SEND** button to send the request to service. If successful, you see an HTTP status code **200** in the **Response** section.

Check the raw response body by clicking on the **Response** and you should see output similar to the following:

```
[{"id":99,"quantity":2,"extPrice":15.99,"catalogItem":null}]
```

- 9.7. Stop the Maven Camel plug-in execution.

In the terminal window running Camel, press **Ctrl+C** to end the execution of the plug-in.

- 10. Close the project.

In the **Project Explorer** view, right-click rest-dsl project and click **Close Project**.

This concludes the guided exercise.

Consuming REST Services with the HTTP Component

Objective

After completing this section, students should be able to develop a Camel route that uses Camel's HTTP component to enrich a message exchange.

Implementing the Content Enricher Pattern

When sending messages or data from one system to another, it is common for the target or receiving system to require more information than the source or sending system can provide. For example, the source system's data may only contain a customer ID, but the receiving system actually requires the customer name and address. Additionally, an order message sent by the order management system may only contain an order number, but you need to find the items associated with that order, so that you can pass it to the order fulfillment system. In these situations, use the *content enricher pattern* in your Camel route to enrich or enhance your message exchange with the required additional data.

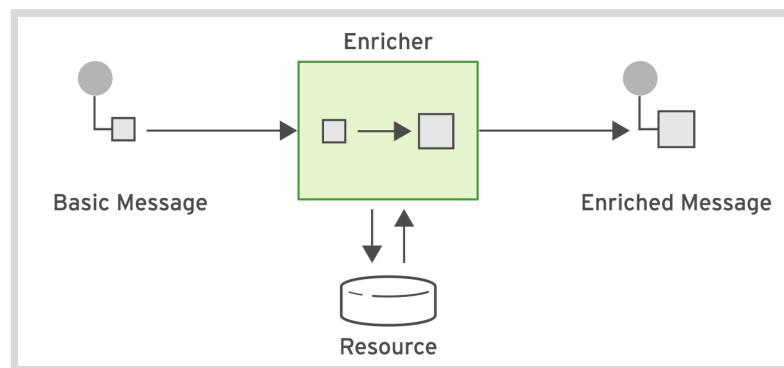


Figure 6.3: The Content Enricher Pattern

Camel supports the content enricher pattern using the **enrich** DSL method to enrich the message as shown in the following example:

```

from("direct:start")
    .enrich("direct:resource"①, aggregationStrategy②)
    .to("direct:result");

from("direct:resource")
    ...
  
```

- ① The **enrich** DSL method has two parameters, the first is the URI of the producer Camel must invoke to retrieve the enrichment data.
- ② The second parameter is an optional instance of the **AggregationStrategy** implementation that you must provide for Camel to use to combine the original message exchange with the enrichment data. If you do not provide an aggregation strategy, Camel uses the body obtained from the resource as the enriched message exchange.

The **enrich** DSL method uses a Camel producer to obtain some additional data. This producer is invoked synchronously. The **enrich** method retrieves additional data from a *resource endpoint*

in order to enrich an incoming message (contained in the *original exchange*). The **enrich** method then uses an aggregation strategy to combine the original exchange and the *resource exchange*.

The first parameter of the **aggregate** method from the **AggregationStrategy** implementation corresponds to the original exchange and the second parameter to the resource exchange. Here is an example template for implementing an aggregation strategy to use with the **enrich** DSL method:

```
public class ExampleAggregationStrategy implements AggregationStrategy {

    public Exchange aggregate(Exchange original, Exchange resource) {
        Object originalBody = original.getIn().getBody();
        Object resource = resource.getIn().getBody();
        Object mergeResult = ....// combine original body and resource response
        if (original.getPattern().isOutCapable()) {
            original.getOut().setBody(mergeResult);
        } else {
            original.getIn().setBody(mergeResult);
        }
        return original;
    }
}
```

Consuming Web Resources and REST Services Using the HTTP Component

Hypertext Transfer Protocol (HTTP) is widely used by websites to distribute content. Due to its simple nature and format, it has been widely used to integrate systems, and its use is required when connecting to remote web services. Camel provides the **http** component to integrate with other technologies over HTTP. This component is able to consume contents from an HTTP server, or even use **GET** and **POST** HTTP methods with a REST service to retrieve or create data.

The **http** component is provided by the **camel-http** library and its endpoint URI format is `http[s]://hostname[:port][/resourceURI][?options]`.

To import the **camel-http** library include the following configuration in **pom.xml** file:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-http</artifactId>
</dependency>
```

Camel always uses the **InOut** message exchange protocol due to the HTTP protocol nature (based on a request/response paradigm). Additionally, the library supports HTTP over Secure Socket Layer (SSL) to use encryption over the HTTP protocol.

The endpoint URI format is:

```
http:[hostname][:port][/resourceUri][?param1=value1][&param2=value2]
```

**Note**

You can only produce to endpoints generated by the **http** component. Therefore it should never be used as input into your Camel routes.

Camel uses the following algorithm to determine if either the **GET** or **POST** HTTP method should be used:

1. The method provided in header field named **Exchange.HTTP_METHOD**.
2. **GET** if query string is provided in header **Exchange.HTTP_QUERY**.
3. **GET** if endpoint is configured with a query string.
4. **POST** if there is data to send (body is not null).
5. **GET** otherwise.

Therefore, by default, depending on the content contained in the body of the **inMessage** object on the exchange, Camel either:

- Sends an HTTP **POST** request to the URL using the exchange body as the body of the HTTP request and returns the HTTP response as the **outMessage** object on the exchange, if there is message content.
- If the body is **null**, sends an HTTP **GET** request to the URL and returns the response as the **outMessage** object on the exchange.

URI parameters can either be set directly on the endpoint URI or as a header, as follows:

```
from("direct:start")
    .to("http://example.com?order=123&detail=short");
```

```
from("direct:start")
    .setHeader(Exchange.HTTP_QUERY, constant("order=123&detail=short"))
    .to("http://example.com");
```

When you use the **camel-http** component, Camel can throw an exception depending on the HTTP response code returned by the external resource:

- If the response code is from 100 to 299, then Camel regards it as a success.
- If the response code is 300 or greater, then Camel regards it as a redirection response or a server failure, and throws an **HttpOperationFailedException** exception with any error messages attached to the response.

The option **throwExceptionOnFailure** can be set to **false** to prevent the **HttpOperationFailedException** exception from being thrown for failed response codes. This option allows you to get any response code from the remote server without Camel throwing an exception. The following example route demonstrates this:

```
from("direct:start")
.setHeader(Exchange.HTTP_QUERY, constant("order=123&detail=short"))
.to("http://example.com&throwExceptionOnFailure=false");
```

Enriching a Message Exchange Using the HTTP Component

You can easily use the **camel-http** component in conjunction with the content enricher pattern to update your message exchanges with data from an external web resource. You could use this to retrieve some relevant data from an external system exposed over HTTP. This approach is especially helpful in a microservices-based environment. The following example implements this use case:

```
from("activemq:orders")
.enrich("direct:enrich"①, new HttpAggregationStrategy())②
.log("Order sent to fulfillment: ${body}")
.to("mock:fulfillmentSystem");

from("direct:enrich")③
.setBody(constant(null))④
.to("http://webservice.example.com");⑤
```

- ① The URI for the producer that the **enrich** DSL element invokes to retrieve the resource message.
- ② The **AggregationStrategy** implementation that the **enrich** DSL element uses to combine the original message exchange and the resource message.
- ③ The URI for the consumer that the **enrich** DSL element invokes.
- ④ Set the body of the exchange to **null** so that the HTTP component sends an HTTP GET request to the resource.
- ⑤ The URI for the HTTP component producer is the address of the external web service.

In the Camel route from the previous example, the implementation of **HttpAggregationStrategy** that Camel uses to create the enriched message is shown in the following example:

```
public class HttpAggregationStrategy implements AggregationStrategy{

@Override
public Exchange aggregate(Exchange original, Exchange resource) {
    Order originalBody = original.getIn().getBody(Order.class);①
    String resourceResponse = resource.getIn().getBody(String.class);②
    originalBody.setFulfilledBy(resourceResponse);③
    return original;
}
```

- ① Retrieve the original message exchange body as an instance of the **Order** model class.
- ② Retrieve the resource message exchange body as a Java **String**.
- ③ Set the response as the **fufilledBy** property on the **Order** object.



References

camel-http

<https://camel.apache.org/http.html>

Camel in Action, Second Edition - Chapter Enterprise integration patterns

► Guided Exercise

Consuming a REST Service with the HTTP Component

In this exercise, you will implement the content enricher pattern using the **camel-http** component.

Outcomes

You should be able to implement the content enricher pattern using the **camel-http** component to consume HTTP resources to enrich a message exchange.

Before You Begin

A starter project is provided for you. It includes a **CamelContext** configured in a Spring configuration file, a **RouteBuilder** subclass, and an integration test for you to use to verify your work.

Run the following command to download the starter project and data files used by this exercise:

```
[student@workstation ~]$ lab enrich-http setup
```

Steps

- ▶ 1. Open JBoss Developer Studio and import the starter Maven project.
 - 1.1. Open JBoss Developer Studio (**Applications** → **Programming** → **JBoss Developer Studio**). JBoss Developer Studio requests a default workspace. Leave the default configuration (**/home/student/workspace**) and click **Launch**.
 - 1.2. Import the Maven project by selecting the JBoss Developer Studio main menu **File** → **Import**.
 - 1.3. From the **Import** dialog box, select **Maven** → **Existing Maven Projects** and click **Next >**.
 - 1.4. Click **Browse** and choose **/home/student/JB421/labs/enrich-http** and click **OK** to choose the Maven project. Click **Finish** to complete the import process in JBoss Developer Studio.
Dismiss the prompt about problems importing the project. You will fix these problems during this exercise.
A new project named **enrich-http** is listed in the **Project Explorer** view.
- ▶ 2. Review the Spring and Camel configurations.
In the **Project Explorer** view, expand the **enrich-http** → **src/main/resources/META-INF/spring** directory. Double-click **bundle-camel-context.xml** file to review the Spring configuration for the starter project.
Notice that the following components are defined:

- The Camel context is defined with a single route builder in an element named **camelContext**. Nested as part of the element, there is another element named **routeBuilder**, which refers to the single route builder.
- The Camel route uses the data source identified as **mysqlDataSource** to receive new order data. The database necessary for this exercise is already running and populated with the necessary tables.
- The entity manager factory connects the persistence context with the **mysqlDataSource** bean to provide JPA persistence functionality. A transaction manager is also defined for use with JPA persistence.

► 3. Review the route definition.

- 3.1. In the **Project Explorer** view, expand **src/main/java** → **com.redhat.training.jb421**. Double-click the **EnrichRouteBuilder.java** file.
- 3.2. Review the implementation of the Camel route:

```
from("jpa:com.redhat.training.jb421.model.Order?persistenceUnit=mysql"
+ "&consumeDelete=false"
+ "&consumer.namedQuery=getUndeliveredOrders"
+ "&maximumResults=1"
+ "&consumer.delay=3000"
+ "&consumeLockEntity=false")
    //TODO add enrich call

.log("Order sent to fulfillment: ${body}")
.to("mock:fulfillmentSystem")
.to("direct:updateOrder");

from("direct:enrich")
    //TODO set message exchange body null

    //TODO call http://classroom.example.com

from("direct:updateOrder")
.log("Order delivered: ${header.orderId}")
.to("sql:update order_ set delivered = 1 where id=:#orderId");
```

► 4. Add the missing dependency to enable the camel-http component.

In the **pom.xml** file, add the **camel-http** dependency to the project to allow the Camel route to make HTTP requests using the **http:** producer.

Under the **<!-- TODO add camel-http dependency -->** comment in the **pom.xml** file add the following dependency:

```
<dependency>
<groupId>org.apache.camel</groupId>
<artifactId>camel-http</artifactId>
</dependency>
```

Press **Ctrl+S** to save your changes.

► 5. Finish the **HttpAggregationStrategy** implementation.

- 5.1. In the **Project Explorer** view, expand **src/main/java** → **com.redhat.training.jb421**. Double-click the **HttpAggregationStrategy.java** file.

- 5.2. Implement the **AggregationStrategy** interface:

```
//TODO implement the AggregationStrategy interface
public class HttpAggregationStrategy implements AggregationStrategy {
```

Camel provides this interface to define a strategy for aggregating two exchanges together into a single exchange. When an aggregation strategy is used with the **enrich** DSL method the first exchange is the original message exchange that was passed to the **enrich** call, the second exchange is the result of the enrich route, or in this case the result from the **direct:enrich** route which is not yet implemented.

- 5.3. Implement the **aggregate** method required by the **AggregationStrategy** interface:

```
//TODO implement the aggregate method
@Override
public Exchange aggregate(Exchange original, Exchange resource) {
}
```

- 5.4. Add the logic to combine the two exchanges, setting the resource result as the **fulfilledBy** field on the **Order** object.

First get the **original** exchange body as an instance of **Order** so you can access its fields:

```
@Override
public Exchange aggregate(Exchange original, Exchange resource) {
    Order originalBody = original.getIn().getBody(Order.class);
}
```

Retrieve the response from the **direct:enrich** endpoint as a **String**:

```
@Override
public Exchange aggregate(Exchange original, Exchange resource) {
    Order originalBody = original.getIn().getBody(Order.class);
    String resourceResponse = resource.getIn().getBody(String.class);
}
```

Set the **fulfilledBy** field on the **Order** instance using the **resourceResponse** variable:

```

@Override
public Exchange aggregate(Exchange original, Exchange resource) {
    Order originalBody = original.getIn().getBody(Order.class);
    String resourceResponse = resource.getIn().getBody(String.class);
    originalBody.setFulfilledBy(resourceResponse);
}

```

- 5.5. Add an exchange header called **orderId** to the **original** exchange. The **direct:updateOrder** endpoint uses this header to update the database.

```

@Override
public Exchange aggregate(Exchange original, Exchange resource) {
    Order originalBody = original.getIn().getBody(Order.class);
    String resourceResponse = resource.getIn().getBody(String.class);
    originalBody.setFulfilledBy(resourceResponse);
    original.getIn().setHeader("orderId", originalBody.getId());
}

```

- 5.6. Return the **original** exchange, which references the modified **Order** object, and contains the **orderId** exchange header:

```

@Override
public Exchange aggregate(Exchange original, Exchange resource) {
    Order originalBody = original.getIn().getBody(Order.class);
    String resourceResponse = resource.getIn().getBody(String.class);
    originalBody.setFulfilledBy(resourceResponse);
    original.getIn().setHeader("orderId", originalBody.getId());
    return original;
}

```

- 5.7. Press **Ctrl+S** to save your changes.

- ▶ 6. Update the **EnrichRouteBuilder** class to include an **enrich** DSL method call and use the **HttpAggregationStrategy** class to combine the resource response and the original exchange.
- 6.1. Add the **enrich** DSL method call set the first parameter to the URI of the **direct:enrich** endpoint and the second parameter as a new instance of the **HttpAggregationStrategy** class:

```

from("jpa:com.redhat.training.jb421.model.Order?persistenceUnit=mysql"
    + "&consumeDelete=false"
    + "&consumer.namedQuery=getUndeliveredOrders"
    + "&maximumResults=1"
    + "&consumer.delay=3000"
    + "&consumeLockEntity=false")
//TODO add enrich call
.enrich("direct:enrich", new HttpAggregationStrategy())
.log("Order sent to fulfillment: ${body}")
.to("mock:fulfillmentSystem")
.to("direct:updateOrder");

```

6.2. Press **Ctrl+S** to save your changes to the route definition.

- 7. Finish the implementation of the **direct:enrich** endpoint to use the **http:** producer to retrieve some data using an HTTP **GET** request.

7.1. To initiate an HTTP **GET** request, clear the body of the message exchange. The HTTP producer automatically sends a **GET** request to the URL when the message exchange body is empty.

```
from("direct:enrich")
//TODO set message exchange body null
.setBody(constant(null))
```

7.2. Add the HTTP producer call:

```
from("direct:enrich")
//TODO set message exchange body null
.setBody(constant(null))
//TODO call http://classroom.example.com
.to("http://classroom.example.com");
```

7.3. Press **Ctrl+S** to save your changes to the route definition.

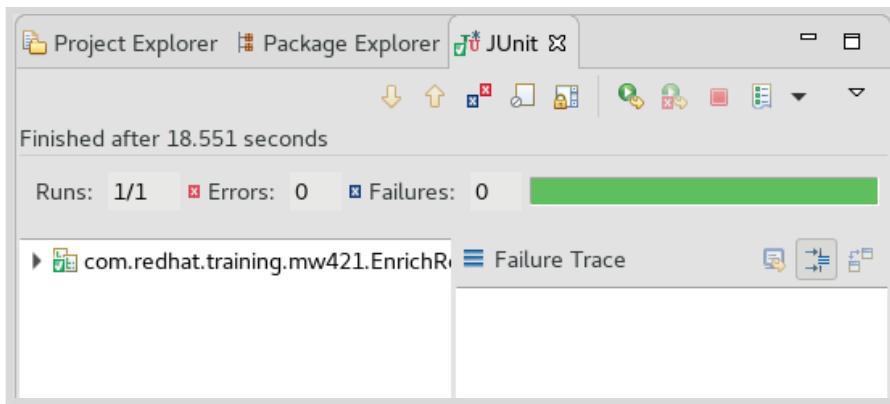
- 8. Run the provided JUnit test to verify your route.

8.1. Run the unit test.

Right-click the **EnrichRouteTest** test case from the **com.redhat.training.jb421** package and select **Run As → JUnit Test** to run the test.

8.2. Check the JUnit test outcome.

Open the **JUnit** tab, if all tests passed, it resembles the following screen shot:



8.3. Review the console log.

Open the **Console** tab in the bottom pane of JBoss Developer Studio, if the route is functioning properly it ends with the following text output:

```
EnrichRouteTest    INFO
*****
EnrichRouteTest    INFO Testing done:
testFulfillingOrders(com.redhat.training.jb421.EnrichRouteTest)
EnrichRouteTest    INFO Took: 12.929 seconds (12929 millis)
EnrichRouteTest    INFO
*****
```

- ▶ 9. Start the routes for testing and create test data using the provided **setup-data.sh** script.

- 9.1. In the same terminal window, run the following command to start the route using the Camel Maven plug-in:

```
[student@workstation ~]$ cd ~/JB421/labs/enrich-http
[student@workstation enrich-http]$ mvn camel:run -DskipTests
```

Wait until the route is started. Search in the terminal window for the following output before proceeding:

```
INFO Total 3 routes, of which 3 are started.
INFO Apache Camel 2.21.0.fuse-000077-redhat-1 (CamelContext: jb421Context)
started in 0.908 seconds
```

- 9.2. Create two test orders in the system.

Open a new terminal window (**Applications** → **System Tools** → **Terminal**) and execute the following commands:

```
[student@workstation ~]$ cd ~/JB421/labs/enrich-http/
[student@workstation enrich-http]$ sh setup-data.sh
```

You should see the following output:

```
Resetting database data...
Done!
Sending two test orders...
Orders sent!
```

- 9.3. Check the output in the Camel log.

Return to the terminal window where Maven is running the routes. You should see something similar to the following output if the orders were processed successfully. Note that this output has been truncated for brevity's sake.

```
INFO Order sent to fulfillment: Order [id=1, ... =<h1>Welcome to Red Hat
Training!</h1>
]
INFO Order delivered: 1
INFO Order sent to fulfillment: Order [id=2, ... =<h1>Welcome to Red Hat
Training!</h1>
]
INFO Order delivered: 2
```

Ensure that the **fulfilledBy** value contains the message **<h1>Welcome to Red Hat Training!</h1>** which is the result of the **http** producer route. This means our enrichment and aggregation is functioning properly.

- 9.4. Stop the Maven Camel plug-in execution.

In the terminal window running Camel, press **Ctrl+C** to end the execution of the plug-in.

- 10. Close the project.

In the **Project Explorer** view, right-click **enrich-http** and click **Close Project**.

This concludes the guided exercise.

Implementing REST Service Documentation with Swagger

Objective

After completing this section, students should be able to document REST services using Swagger.

Introducing Swagger for REST documentation

Swagger, also known as the Open API Specification, standardizes the REST API documentation in a human-readable format. As organizations rely more heavily on REST services and microservices, easily accessible and readable service documentation becomes more important. Swagger provides this capability with very minimal developer overhead.

Swagger gives developers tools that allow them to document their REST services inline. Swagger then takes this metadata and generates a standard set of documentation available in human and machine readable JSON format at a separate URL from the REST service itself. This drastically reduces the effort required of you to properly document your API, and makes updating documentation much easier. The resulting documentation is easily readable by both developers and business users and provides clarity to available endpoints and parameter requirements.

Documenting Camel REST DSL with Swagger

The Camel REST DSL includes integration with Swagger by default. To enable Swagger with the REST DSL, include the following dependency in your **pom.xml** file.

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-swagger-java</artifactId>
</dependency>
```

After the **camel-swagger-java** library is available on the class path, set the context path where the Swagger API service is accessible. This is done using the **restConfiguration** method inherited from the **RouteBuilder** class and invoking the **apiContextPath** method. To configure the port where you may access the service, invoke the **port** method as shown in the following example:

```
restConfiguration()
    .component("spark-rest")
    .port(8080)
    .bindingMode(RestBindingMode.json_xml)
    .apiContextPath("api");
```

In the previous example, the REST API documentation is accessible at `http://localhost:8080/api`.

Camel's Swagger integration includes set of elements that are available in the REST DSL. These elements control the contents of the API documentation. You can use these elements to document methods and parameters with descriptions and types. The following table includes some of the commonly used documentation elements:

REST DSL Binding Modes

Element	Description
description	This element provides a summary of the REST service and each operation. You can use this element at the service and method levels.
param	This element documents input parameters. Supports sub-elements including name and description .
responseMessage	This element documents possible responses from the service. It also supports sub-elements including code and message .
type	This element documents parameter types. You can use it at the operation level to document the Java type of the body parameter, or at the parameter (param) level, to document the parameter type of individual parameters. The following parameter types are supported for individual parameters: <ul style="list-style-type: none"> • body • formData • header • path • query
dataType	This element documents the Java type of a given parameter.
outType	This element documents the return type of a given operation.

The following is an example REST DSL route definition that includes many of the documentation elements:

```

restConfiguration()
    .component("spark-rest")
    .port(8080)
    .bindingMode(RestBindingMode.json_xml)
    .apiContextPath("api");

rest("/customers")
    .description("Customer services")①

    .get("{id}").outType(Customer.class)②
        .description("Get customer by id")③
        .param()④
            .name("id")⑤
  
```

```

    .description("The customer id")⑥
    .type(RestParamType.path)⑦
    .endParam()
.to("bean:customerService?method=getCustomer(${header.id})")

.post().type(Customer.class)⑧.outType(String.class)
.description("Create a new customer")
.responseMessage()
.code(200)⑨
.message("The created customer id")⑩
.endResponseMessage()
.to("bean:customerService?method=createCustomer")

```

- ① Set a description at the service level of "Customer services".
- ② Set the return type of the HTTP **GET** method as a **Customer** instance.
- ③ Set a description at the HTTP method level of "Get custom by id".
- ④ Start a new **param** to document the **id** path parameter.
- ⑤ Set the name of the **param** to "id".
- ⑥ Set the description of the **param** to "The customer id".
- ⑦ Set the type of the **param** to "path" using the **RestParamType** helper class provided by Camel.
- ⑧ Specify the expected type of the HTTP **POST** request body to **Customer** objects.
- ⑨ Define a possible response associated with HTTP status code 200.
- ⑩ Define a message to describe the response associated with HTTP status code 200.

The example REST API shown previously produces the following JSON output with the REST API documentation defined in the previous source code:

```
{
  "swagger" : "2.0",
  "info" : { },
  "host" : "0.0.0.0:8080",
  "tags" : [ {
    "name" : "customers",
    "description" : "Customer services"
  }],
  "schemes" : [ "http" ],
  "paths" : {
    "/customers" : {
      "post" : {
        "tags" : [ "customers" ],
        "summary" : "Create a new customer",
        "operationId" : "route2",
        "parameters" : [ {
          "in" : "body",
          "name" : "body",
          "required" : true,
          "schema" : {
            "$ref" : "#/definitions/Customer"
          }
        }],
        "responses" : {
          "200" : {
            "description" : "The created customer id",

```

```

        "schema" : {
            "type" : "string"
        }
    }
},
"/customers/{id}" : {
    "get" : {
        "tags" : [ "customers" ],
        "summary" : "Get customer by id",
        "operationId" : "route1",
        "parameters" : [ {
            "name" : "id",
            "in" : "path",
            "description" : "The customer id",
            "required" : true,
            "type" : "string"
        }],
        "responses" : {
            "200" : {
                "description" : "Output type",
                "schema" : {
                    "$ref" : "#/definitions/Customer"
                }
            }
        }
    },
    "definitions" : {
        "Address" : {
            "type" : "object",
            "properties" : {
                "id" : {
                    "type" : "integer",
                    "format" : "int32",
                    "xml" : {
                        "attribute" : true
                    }
                },
                "streetAddress1" : {
                    "type" : "string"
                },
                ...output omitted...
                "country" : {
                    "type" : "string"
                }
            }
        },
        "Customer" : {
            "type" : "object",
            "properties" : {
                "id" : {
                    "type" : "integer",
                    "format" : "int32"
                }
            }
        }
    }
}

```

```
"firstName" : {  
    "type" : "string"  
    ...output omitted...  
    "billingAddress" : {  
        "$ref" : "#/definitions/Address"  
    },  
    "shippingAddress" : {  
        "$ref" : "#/definitions/Address"  
    }  
}  
}  
}
```



References

Camel Swagger component

<http://camel.apache.org/swagger.html>

Camel in Action, Second Edition - Chapter REST and Web Services

► Guided Exercise

Implementing Swagger Documentation with Camel

In this exercise, you document a REST service using the REST DSL Swagger integration to provide metadata about the service endpoints.

Outcomes

You should be able to update a route built with the Rest DSL to use Swagger documentation to provide API documentation for the service at the **/api-doc** context path.

Before You Begin

A starter project is provided for you. It includes a **CamelContext** configured in a Spring configuration file and a **RouteBuilder** subclass.

Run the following command to download the starter project and data files used by this exercise:

```
[student@workstation ~]$ lab rest-swagger setup
```

Steps

- ▶ 1. Open JBoss Developer Studio and import the starter Maven project.
 - 1.1. Open JBoss Developer Studio (**Applications** → **Programming** → **JBoss Developer Studio**). JBoss Developer Studio requests a default workspace; keep the default configuration (**/home/student/workspace**) and click **Launch**.
 - 1.2. Import the Maven project by selecting JBoss Developer Studio main menu **File** → **Import**.
 - 1.3. From the **Import** dialog box, select **Maven** → **Existing Maven Projects** and click **Next**.
 - 1.4. Click **Browse** and choose **/home/student/JB421/labs/rest-swagger** and click **OK** to choose the Maven project. Click **Finish** to complete the import process in JBoss Developer Studio.
Dismiss the prompt about problems importing the project. You will fix these problems during this exercise.
A new project named **rest-swagger** is listed in the **Project Explorer** view.
- ▶ 2. Review the Spring and Camel configurations.
In the **Project Explorer** view, expand **rest-swagger** → **src/main/resources** → **/META-INF/spring**. Double-click **bundle-camel-context.xml** file to review the Spring configuration for the starter project.
Notice the following components are defined:

- The Camel context is defined with a single route builder. It is defined by an element named **camelContext**. Nested as part of the element, there is another element named **routeBuilder** which refers to the single route builder.
- The data source defined as **mysqlDataSource**, this is the data source used by the sql component. The database necessary for this exercise is already running and populated with the necessary tables. The data source is referred as a bean element identified by **mysqlDataSource**.
- The entity manager factory connects the persistence context with the **mysqlDataSource** bean to provide JPA persistence functionality. A transaction manager is also defined for use with JPA persistence.

► 3. Add the missing dependencies to enable the REST DSL and API documentation in the **pom.xml** file.

In the **Project Explorer** view, double-click **pom.xml** file.

Add the **camel-swagger-java** dependency to the project to allow the **spark-rest** component to document the REST services with Swagger.

Look for the `<!-- add camel-swagger dependency -->` comments in the **pom.xml** file, and add the following dependency:

```
<dependency>
<groupId>org.apache.camel</groupId>
<artifactId>camel-swagger-java</artifactId>
</dependency>
```

► 4. Configure the service level Swagger options to set the context path, API title, and version elements.

In the **Project Explorer** view, expand **src/main/java → com.redhat.training.jb421**. Double-click the **OrderRouteBuilder.java** file.

4.1. Set the context path to **/api-doc**.

Remove the semi-colon after the **bindingMode** method call and invoke the **apiContextPath** method using **/api-doc** as the argument.

```
restConfiguration()
// use spark-rest component and run on port 8080
.component("spark-rest").port(8080)
.bindingMode(RestBindingMode.json)
//TODO set API context path to '/api-doc'
.apiContextPath("/api-doc")
```

4.2. Set the API title to "Order REST Service API" using the **api.title** API property:

```
restConfiguration()
    // use spark-rest component and run on port 8080
    .component("spark-rest").port(8080)
    .bindingMode(RestBindingMode.json)
    //TODO set API context path to '/api-doc'
    .apiContextPath("/api-doc")
    //TODO set an API property of 'api.title' to 'Order REST Service API'
    .apiProperty("api.title", "Order REST Service API")
```

4.3. Set the API version to **1** using the **api.version** API property:

```
restConfiguration()
    // use spark-rest component and run on port 8080
    .component("spark-rest").port(8080)
    .bindingMode(RestBindingMode.json)
    //TODO set API context path to '/api-doc'
    .apiContextPath("/api-doc")
    //TODO set an API property of 'api.title' to 'Order REST Service API'
    .apiProperty("api.title", "Order REST Service API")
    //TODO set an API property of 'api.version' to '1'
    .apiProperty("api.version", "1");
```

4.4. Press **Ctrl+S** to save your changes to the route definition.

► 5. Add Swagger documentation to the REST DSL defined **/shipAddress/{id}** endpoint.

5.1. Set the return value type of the REST endpoint as an **Address** object.

Set the **outType** parameter to **Address.class** to specify this endpoint outputs an instance of the **Address** model class:

```
//TODO add outType of Address
.get("/shipAddress/{id}").outType(Address.class)
```

5.2. Document the **id** path parameter for the REST endpoint.

Specify the path parameter name as **id**, its type as **path** using the **RestParamType** enum, and its description using the provided **PARAM_DESCRIPTION** variable:

```
//TODO add outType of Address
.get("/shipAddress/{id}").outType(Address.class)
//TODO add param named "id" with a type of "path" and use the PARAM_DESCRIPTION
for the description
.param().name("id").type(RestParamType.path)
.description(PARAM_DESCRIPTION).endParam()
```

5.3. Specify a description for the endpoint of **Get the shipping address for an order by its ID**:

```
//TODO add outType of Address
.get("/shipAddress/{id}").outType(Address.class)
//TODO add param named "id" with a type of "path" and use the PARAM_DESCRIPTION
for the description
.param().name("id").type(RestParamType.path)
.description(PARAM_DESCRIPTION).endParam()
//TODO set the endpoint description to "Get the shipping address for an order by
its ID"
.description("Get the shipping address for an order by its ID")
.to("direct:shipAddress")
```

5.4. Press **Ctrl+S** to save your changes to the route definition.

► 6. Add Swagger documentation to the REST DSL defined **/orderTotal/{id}** endpoint.

6.1. Set the return value type of the REST endpoint as an **Double** value.

Set the **outType** parameter to **Double.class** to specify this endpoint outputs an instance of the **Double** primitive type:

```
//TODO add outType of Double
.get("/orderTotal/{id}").outType(Double.class)
```

6.2. Document the **id** path parameter for the REST endpoint.

Specify its name as "**id**", its type as **path** using the **RestParamType** enum, and its description using the provided **PARAM_DESCRIPTION** variable:

```
//TODO add outType of Double
.get("/orderTotal/{id}").outType(Double.class)
//TODO add param named "id" with a type of "path" and use the PARAM_DESCRIPTION
for the description
.param().name("id").type(RestParamType.path)
.description(PARAM_DESCRIPTION).endParam()
```

6.3. Specify a description for the endpoint of "**Get the total cost for an order by its ID**"

```
//TODO add outType of Double
.get("/orderTotal/{id}").outType(Double.class)
//TODO add param named "id" with a type of "path" and use the PARAM_DESCRIPTION
for the description
.param().name("id").type(RestParamType.path)
.description(PARAM_DESCRIPTION).endParam()
//TODO set the endpoint description to "Get the total cost for an order by its
ID"
.description("Get the total cost for an order by its ID")
.to("direct:orderTotal")
```

6.4. Close the DSL with a semi-colon after the to method call:

```
.to("direct:orderTotal");
```

► 7. Start the routes for testing.

Open a new terminal window, run the following command to start the route using the Camel Maven plug-in:

```
[student@workstation ~]$ cd JB421/labs/rest-swagger
[student@workstation rest-swagger]$ mvn camel:run -DskipTests
```

Wait until the route is started. Search in the terminal window for the following output before proceeding:

```
INFO Total 7 routes, of which 7 are started.
INFO Apache Camel 2.21.0.redhat-630187 (CamelContext: jb421Context) started in
0.753 seconds
...output omitted...
INFO Started ServerConnector@3b61dec8{HTTP/1.1,[http/1.1]}{0.0.0.0:8080}
INFO Started @29766ms
```

► 8. Test the Swagger documentation using Firefox.

8.1. Open a new Firefox window **Applications → Internet → Firefox Web Browser**.

8.2. Enter the following URL in your browser <http://localhost:8080/api-doc>.

8.3. Observe the JSON output by Swagger for the service itself:

```
{
  "swagger" : "2.0",
  "info" : {
    "version" : "1",
    "title" : "Order REST Service API"
  },
  ...output omitted...
```

8.4. Verify the individual endpoints are documented properly:

```
"paths" : {
  "/orders/orderTotal/{id}" : {
    "get" : {
      "tags" : [ "orders" ],
      "summary" : "Get the total cost for an order by its ID",
      "operationId" : "route2",
      "parameters" : [ {
        "name" : "id",
        "in" : "path",
        "description" : "The ID of the order",
        "required" : true,
        "type" : "string"
      }],
      "responses" : {
        "200" : {
          "description" : "Output type",
          "schema" : {
            "type" : "number",
            "format" : "float"
          }
        }
      }
    }
  }
}
```

```
        "format" : "double"
...output omitted...
"/orders/shipAddress/{id}" : {
    "get" : {
        "tags" : [ "orders" ],
        "summary" : "Get the shipping address for an order by its ID",
        "operationId" : "route1",
        "parameters" : [ {
            "name" : "id",
            "in" : "path",
            "description" : "The ID of the order",
            "required" : true,
            "type" : "string"
        }],
        "responses" : {
            "200" : {}
        }
    }
}
```

8.5. Stop the Maven Camel plug-in execution.

In the terminal window running Camel, press **Ctrl+C** to end the execution of the plug-in.

▶ **9.** Close the project.

In the **Project Explorer** view, right-click rest-swagger project and click **Close Project**.

This concludes the guided exercise.

► Lab

Implementing REST Services

Performance Checklist

In this lab, you will use Camel to implement a REST service using the REST DSL and document it with Swagger.

The REST service you will build needs two REST endpoints that retrieve:

- The shipping address from an order
- The list of book titles in a given order

Both methods use a path parameter of the order ID as the only input. Both endpoints must output JSON data.

Outcomes

After completing the lab, you should be able to implement and document a REST service using Camel's REST DSL and Swagger.

Before You Begin

A skeleton project is provided as a starter, including annotated (JAXB and JPA annotations included) model classes, and a preconfigured Camel context, which includes the necessary data source configuration and a stubbed route builder.

The route builder stub includes two routes, which implement the necessary logic to retrieve the data from the database given a header containing the order ID, named **id**. You should connect your REST services to these routes by sending the message exchange to their direct endpoints.

The starter project also includes a unit test to verify whether the REST services are working properly. This test can evaluate any of the three implementations without modification.

- Run the following command to download the starter project used by this lab:

```
[student@workstation ~]$ lab rest-services setup
```

- Import the starter project into JBoss Developer Studio, located at **/home/student/JB421/labs/rest-services**.

Steps

1. Inspect the provided starter project.

Examine the Spring configuration file **src/main/resources/META-INF/spring/bundle-camel-context.xml** and the components it defines, including a data source named **mysqlDataSource**.

Examine the route builder stub **com.redhat.training.jb421.OrderRouteBuilder** and the two existing routes, named **direct:shipAddress** and **direct:bookTitles**. Each of these two routes expects a header named **id** to exist.

2. Update the project **pom.xml** file with the necessary dependencies to create and document your REST service.

Hint: To use the REST DSL, you need to add the dependency for the underlying implementation that you want to use with the REST DSL. For this lab, use the **camel-spark-rest** library as your REST DSL implementation.

3. Update the route definition to include the REST service definition.

Make sure that the REST service runs on the **localhost** host and listens to port **8080** and uses the URIs **/orders/shipAddress/*id*** and **/orders/bookTitles/*id*** with the HTTP GET method for the two services. It is very important to include the path parameter, which must be stored in a header named **id**.

Also make sure to connect the REST services with the existing Camel routes that retrieve the necessary data for the route using a direct producers evaluated in the previous step.

4. Add API documentation to your service at a context path of **/api-doc**.

Set the title as **Order REST Service API** and version **1** on the service.

Also, include the following information about each endpoint parameters:

- Name of **id**
- Description of **id**
- Type of **id**
- Endpoint description

5. Populate the orders table with a test order with an ID of **100**, using the **setup-data.sh** script provided.

6. Start the routes for testing using the Camel Maven plug-in.

7. Test the **/orders/shipAddress** REST endpoint using the Rest Client plug-in for Firefox. Use the service to retrieve information for order with an ID of **100**.



Note

The RESTClient plug-in file is located at **/home/student/restclient.xpi**. You can open this file in Firefox to install the plug-in if needed.

8. Test the **/orders/bookTitles** REST endpoint using the RestClient plug-in for Firefox. Use the service to retrieve book title information for order with an ID of **100**.
9. Test the Swagger documentation is available at the **http://localhost:8080/api-doc** URL.
10. Return to the terminal window where the Maven Camel plug-in is running and press **Ctrl+C** to stop the execution.

11. Grade your work. Execute the following command:

```
[student@workstation rest-services]$ lab rest-services grade
```

If you do not get a **PASS** grade, review your work. The **~/JB421/labs/rest-services/grade.log** file contains the Maven messages produced by the grading process. This may be helpful in debugging any issues.

12. Clean up.

Close the **rest-services** project in JBoss Developer Studio to conserve memory.

This concludes the lab.

► Solution

Implementing REST Services

Performance Checklist

In this lab, you will use Camel to implement a REST service using the REST DSL and document it with Swagger.

The REST service you will build needs two REST endpoints that retrieve:

- The shipping address from an order
- The list of book titles in a given order

Both methods use a path parameter of the order ID as the only input. Both endpoints must output JSON data.

Outcomes

After completing the lab, you should be able to implement and document a REST service using Camel's REST DSL and Swagger.

Before You Begin

A skeleton project is provided as a starter, including annotated (JAXB and JPA annotations included) model classes, and a preconfigured Camel context, which includes the necessary data source configuration and a stubbed route builder.

The route builder stub includes two routes, which implement the necessary logic to retrieve the data from the database given a header containing the order ID, named **id**. You should connect your REST services to these routes by sending the message exchange to their direct endpoints.

The starter project also includes a unit test to verify whether the REST services are working properly. This test can evaluate any of the three implementations without modification.

- Run the following command to download the starter project used by this lab:

```
[student@workstation ~]$ lab rest-services setup
```

- Import the starter project into JBoss Developer Studio, located at **/home/student/JB421/labs/rest-services**.

Steps

1. Inspect the provided starter project.

Examine the Spring configuration file **src/main/resources/META-INF/spring/bundle-camel-context.xml** and the components it defines, including a data source named **mysqlDataSource**.

Examine the route builder stub **com.redhat.training.jb421.OrderRouteBuilder** and the two existing routes, named **direct:shipAddress** and **direct:bookTitles**. Each of these two routes expects a header named **id** to exist.

2. Update the project **pom.xml** file with the necessary dependencies to create and document your REST service.

Hint: To use the REST DSL, you need to add the dependency for the underlying implementation that you want to use with the REST DSL. For this lab, use the **camel-spark-rest** library as your REST DSL implementation.

Open the **pom.xml** and add the following dependencies:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-spark-rest</artifactId>
</dependency>
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-swagger-java</artifactId>
</dependency>
```

3. Update the route definition to include the REST service definition.

Make sure that the REST service runs on the **localhost** host and listens to port **8080** and uses the URLs **/orders/shipAddress/{id}** and **/orders/bookTitles/{id}** with the HTTP GET method for the two services. It is very important to include the path parameter, which must be stored in a header named **id**.

Also make sure to connect the REST services with the existing Camel routes that retrieve the necessary data for the route using a direct producers evaluated in the previous step.

Configure the REST DSL with the underlying implementation with the **spark-rest** implementation, and set the **port** option to **8080**. Use REST DSL to define the two required services with the correct URLs, and a path parameter named **id**, and then route the existing direct routes as follows:

```
// configure rest-dsl
restConfiguration()
    // to use spark-rest component and run on port 8080
    .component("spark-rest").port(8080);

// rest services under the orders context-path
rest("/orders")
    .get("/shipAddress/{id}")
        .to("direct:shipAddress")
    .get("/bookTitles/{id}")
        .to("direct:bookTitles");
```

4. Add API documentation to your service at a context path of **/api-doc**.

Set the title as **Order REST Service API** and version **1** on the service.

Also, include the following information about each endpoint parameters:

- Name of **id**
- Description of **id**
- Type of **id**
- Endpoint description

Update the REST DSL to include the necessary Swagger methods to include the documentation metadata for the service itself as well as each endpoint:

```
// configure rest-dsl
restConfiguration()
    // to use spark-rest component and run on port 8080
    .component("spark-rest").port(8080)
    .apiContextPath("/api-doc")
    .apiProperty("api.title", "Order REST Service API")
    .apiProperty("api.version", "1");
rest("/orders")
    .get("/shipAddress/{id}")
        .description("Get the shipping address for an order by its ID")
        .param().name("id").type(RestParamType.path)
        .description("The order ID").endParam()
        .to("direct:shipAddress")
    .get("/bookTitles/{id}")
        .description("Get the book titles in an order by its ID")
        .param().name("id").type(RestParamType.path)
        .description("The order ID").endParam()
        .to("direct:bookTitles");
```

5. Populate the orders table with a test order with an ID of **100**, using the **setup-data.sh** script provided.

Use the provided script to populate the data:

```
[student@workstation ~]$ cd ~/JB421/labs/rest-services
[student@workstation rest-services]$ ./setup-data.sh
```

6. Start the routes for testing using the Camel Maven plug-in.

Run the following commands to start the route using the Camel Maven plug-in:

```
[student@workstation rest-services]$ mvn camel:run
```

Wait until the route is started. Search in the terminal window for the following output before proceeding:

```
INFO org.apache.camel.spring.SpringCamelContext - Total 5 routes, of which 5 are
started.
INFO Apache Camel 2.21.0.fuse-000077-redhat-1 (CamelContext: jb421Context) is
starting
```

7. Test the **/orders/shipAddress** REST endpoint using the Rest Client plug-in for Firefox. Use the service to retrieve information for order with an ID of **100**.

In the RESTClient window, under the **Request** section, set the method to **GET** and the URL to <http://localhost:8080/orders/shipAddress/100>.



Note

The RESTClient plug-in file is located at **/home/student/restclient.xpi**. You can open this file in Firefox to install the plug-in if needed.

Click **SEND** to send the request to service. If successful, you will see an HTTP status code **200** in the **Response** section.

8. Test the **/orders/bookTitles** REST endpoint using the RestClient plug-in for Firefox. Use the service to retrieve book title information for order with an ID of **100**.

In the RESTClient window, under the **Request** section, set the method to **GET** and the URL to <http://localhost:8080/orders/bookTitles/100>.

Click **SEND** to send the request to service. If successful, you see an HTTP status code **200** in the **Response** section.

9. Test the Swagger documentation is available at the <http://localhost:8080/api-doc> URL.

In the RESTClient window, under the **Request** section, set the method to **GET** and the URL to <http://localhost:8080/api-doc>.

Click **SEND** to send the request to service. If successful, you will see an HTTP status code **200** in the **Response** section.

10. Return to the terminal window where the Maven Camel plug-in is running and press **Ctrl+C** to stop the execution.

11. Grade your work. Execute the following command:

```
[student@workstation rest-services]$ lab rest-services grade
```

If you do not get a **PASS** grade, review your work. The **~/JB421/labs/rest-services/grade.log** file contains the Maven messages produced by the grading process. This may be helpful in debugging any issues.

12. Clean up.

Close the **rest-services** project in JBoss Developer Studio to conserve memory.

This concludes the lab.

Summary

In this chapter, you learned:

- Camel provides a number of components that can be used to implement REST services.
- The REST DSL allows developers to define REST services in Camel routes using REST style, with verbs that align with the HTTP protocol, such as **get**, **post**, **delete**, and so on.
- The REST DSL is a facade that supports multiple REST component implementations. The actual REST transport is provided by other Camel components such as Restlet, Spark-REST, and others that include REST integration.
- The content enricher pattern can be implemented in Camel using the **enrich** DSL. Typically a second route is used to enrich the existing exchange, often using some external resource such as a web service or database. The original exchange and resource result are combined using an implementation of the **AggregationStrategy** interface.
- The http component allows Camel to integrate with remote web resources, including REST services.
- Swagger can be used in conjunction with the REST DSL to provide documentation of the parameters and output for each REST service.

Chapter 7

Deploying Camel Routes

Goal

Package and deploy Camel applications for deployment with Red Hat Fuse

Objectives

- Deploy Camel integration projects to Karaf as an OSGi bundle.
- Deploy Camel integration projects to EAP as a Java EE archive.
- Deploy Camel integration projects to Spring Boot.

Sections

- Deploying Routes with Karaf (and Guided Exercise)
- Deploying Routes with JBoss EAP (and Guided Exercise)
- Deploying Routes with Spring Boot (and Guided Exercise)

Lab

Deploying Camel Routes

Deploying Routes with Karaf

Objectives

After completing this section, students should be able to deploy Camel integration projects to Karaf as an OSGi bundle.

Introducing Red Hat Fuse on Karaf

Red Hat Fuse supports three different deployment options: Apache Karaf, JBoss EAP, and Spring Boot. This section focuses on how to package and deploy Camel Routes on Apache Karaf.

Apache Karaf is a lightweight and enterprise-ready application container based on the OSGi specifications. Karaf can host multiple kinds of applications, such as OSGi, Spring, and Java Web Archive (WAR).

At the core of Apache Karaf is an OSGi framework, which can be either Apache Felix or Eclipse Equinox. Karaf extends the OSGi framework by adding a management console, hot deployment, dynamic configuration, centralized logging, remote management using either JMX or SSH, and security.

The *Open Service Gateway Initiative* (OSGi) specification defines a Java framework for developing and deploying modular software programs and libraries.

Class Loading Mechanisms for Java Applications

The OSGi specification was designed to allow multiple applications to share an application container without conflicting dependencies so that applications and libraries can be developed and updated independently of each other. At the same time, the OSGi specification allows applications to share dependent libraries in a safe way for runtime efficiency.

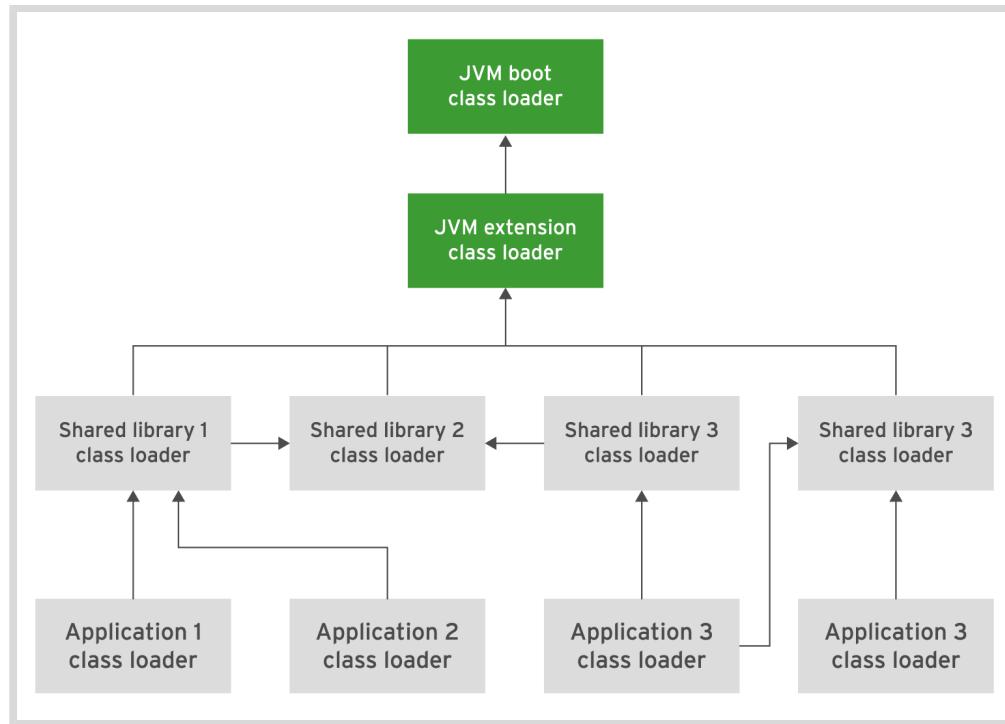
Traditional JVM forces all applications to share the same parent class loader, and consequently to use the same library dependencies, which means all applications need to be updated at the same time a shared dependent library is updated.

Some of these issues could be solved by creating multiple parent class loaders, and then segregating application class loaders as children of different parents to form a tree structure. Managing these parent and child relationships becomes very complex, and most application container products do not provide such fine-grained control over shared class loaders.

It is worth noting that the Java EE specifications provide no way to define class loaders shared between applications. By Java EE standards, any application that requires a module beyond the standard Java EE APIs would package this module in the application EAR file, leading to multiple copies of the same libraries and an increased memory footprint at runtime.

OSGi Class Loading

The OSGi specification solves the shared modules issues by organizing class loaders not in a tree, but in a directed graph. Each application or module, that is, each bundle, gets its own class loader, and each bundle class loader connects to the class loaders for its dependency bundles.

**Figure 7.1: OSGi class loading.**

A traditional JVM class loader differs from an OSGi class loader in the following ways:

- The class loader for an OSGi bundle is not restricted to connect to a single other (parent) class loader.
- Only a subset of the classes inside an OSGi bundle are exposed through the bundle class loader, while in the traditional JVM model, all classes are exposed.
- An OSGi class loader defines whether all classes are exposed by its dependency class loaders.

OSGi class loading solves most issues from traditional JVM class loading:

- A module can be shared between applications without needing to package and load the module multiple times.
- It is possible to install multiple versions of the same module. Each bundle sees only the version it requires.
- Managing relationships between bundles is easy because each bundle declares what it exports and what it imports without needing to manually build a map of all relationships. The OSGi runtime tracks dependencies and detects if a bundle declares its dependencies in a way that creates conflicts.

OSGi Bundles and OSGi Dependencies

The OSGi specification requires that applications and libraries are packaged as OSGi bundles. A *bundle* is a JAR file with a manifest file enriched with OSGi metadata. The OSGi metadata declares which Java packages the bundle exports to other bundles, and which ones the bundle imports from other bundles.

Here is a sample **MANIFEST.MF** file with OSGi metadata:

```

Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: hello: The Hello App bundle
Bundle-SymbolicName: com.redhat.training.jb421.hello
Bundle-Version: 1.0.0
Export-Package: com.redhat.training.jb421;uses:="org.apache.camel, org.slf4j, org.apache.camel.builder.xml";version="1.0.0"
Import-Package: org.apache.activemq, org.apache.activemq.camel.component; version="[5.11,6)", org.apache.camel;version="[2.17,3)", org.apache.camel.builder.xml;version="[2.17,3)", org.slf4j;version="[1.7,2)"

```

The standard Java JAR manifest is extensible, allowing OSGi to add directives to the manifest. These directives contain the **Bundle-** prefix, the **Export-Package** prefix, and the **Import-Package** prefix.

A developer can either add the OSGi metadata manually to **MANIFEST.MF** file or let tooling such as an IDE and Maven add the metadata automatically.

Building OSGi Bundles with the Felix Bundle Plug-in

The *Felix Bundle* plug-in for Maven makes it easier to package Java byte code and resources as an OSGi bundle. The plug-in provides a new packaging type called **bundle**, and the plug-in configuration allows specifying what components are to be exported or hidden by the bundle.

By default, the Felix Bundle plug-in uses the Maven POM dependencies and other configuration to generate the correct OSGi metadata for the project. The plug-in also allows overriding all metadata using the plug-in configuration. See the references at the end of this section for more information about the Felix Bundle plug-in configuration options.

The following declaration is a sample of a Felix Bundle plug-in declaration inside a Maven POM:

```

<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <extensions>true</extensions>
  <configuration>
    <instructions>
      <Private-Package>com.redhat.training.jb421</Private-Package>
    </instructions>
  </configuration>
</plugin>

```

The **<extensions>** value must be set to **true**. Without it, the Maven **package** target fails to create an OSGi bundle.

The Felix Bundle plug-in adds the necessary OSGi metadata to the JAR file so that the package becomes an OSGi bundle. The plug-in also uses Java introspection and Maven POM dependency information to automatically populate the **Import-Package** and **Export-Package** directives in the JAR manifest.

If the automatic population of OSGi metadata does not yield the expected results, or if some information needs to be manually added, a developer can use the plug-in configuration to override or augment the automatically generated values. For example, to manually import and export packages:

```

<plugin>
    <groupId>org.apache.felix</groupId>
    <artifactId>maven-bundle-plugin</artifactId>
    <extensions>true</extensions>
    <configuration>
        <instructions>
            <Export-Package>com.example.api</Export-Package>
            <Private-Package>com.example.api.util</Private-Package>
            <Import-Package>*;com.example.shared</Import-Package>
        </instructions>
    </configuration>
</plugin>

```

The **Import-Package** configuration directive includes an asterisk (*), which means to scan the project to find the proper values. The previous example does not use any automatic values for the **Export-Package** directive but adds a single package to the set that has been automatically discovered for the **Import-Package** directive.

There are many other configuration directives that could be added, allowing complete customization of the bundle manifest. Check the references on this section for more information.

After you add the Felix Bundle plug-in to a project's POM, the **package** Maven goal packages the project as an OSGi bundle. Apache Karaf can either consume the bundle JAR file directly, or it can fetch the bundle from a Maven repository. In this case, the **install** Maven goal installs the bundle JAR file in the local user's Maven repository in the **~/.m2/repository** folder.

Configuring Camel Applications for Apache Karaf

Older releases of Red Hat Fuse allowed Spring applications to be deployed directly to Apache Karaf without changes. These applications relied on the *Spring Dynamic Modules (Sprig DM)* library to connect Spring managed beans and Apache Karaf services together.

Red Hat Fuse 7 does not certify the Spring DM module anymore, and recommends that developers convert the Spring beans configuration file to an OSGi Blueprint configuration file. The *Blueprint* framework is the OSGi specification for dependency injection and mimics the Spring beans configuration file, including support for Camel XML routes.

The only changes required for most Camel applications are:

- Change the top-level element XML from **beans** to **blueprint**.
- Change the XML Schema ID declaration.
- Remove the Spring framework dependencies form the project's POM.

Using the Blueprint framework generates a smaller deployment on Apache Karaf and provides a stronger Service Level Agreement (SLA) from Red Hat because Spring framework libraries are only certified for Red Hat Fuse, and not directly supported by Red Hat.

The following listing shows the declaration of Blueprint configuration files that replaces the Spring beans declaration in a Camel application targeting Apache Karaf:

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

Creating Camel Projects with Maven Archetypes

Camel provides a number of Maven archetypes that make it easier to create new Camel project from scratch using the command line. Some of them target Apache Karaf as the target runtime environment, while some target others Fuse runtimes.

Red Hat recommends that you use the archetypes from Red Hat Fuse product and avoid the equivalent archetypes from the upstream Camel community, to get a POM that is configured for supported and certified dependencies, and to avoid a POM that targets a runtime unsupported by Red Hat Fuse.

Among the Maven archetypes provided by Camel are:

camel-archetype-blueprint

Creates a new project with an XML route configured using the OSGi Blueprint framework.

camel-archetype-cdi

Creates a new project with a route configured using the Java DSL and Camel components configured using the CDI API.

camel-archetype-java

Creates a new project with a sample route configured using the Java DSL.

camel-archetype-spring

Creates a new project with a sample XML route configured using the Spring framework.

See the references at the end of this section for a list of all Maven archetypes provided by Camel. As usual for Maven artifacts and plug-ins, Red Hat recommends using the product bill of materials (BOM) and the product repositories to get the same build releases used for the supported product.

Most archetypes include the Camel Maven plug-in configuration to start the Camel context and test the sample routes, except the **camel-archetype-java** archetype, which includes Java code to start the Camel context.

To use a Maven archetype provided by Camel, you must provide values for a number of system properties using the **-D** command-line option:

- **archetypeGroupId**: The group that provides the archetypes. For Camel archetypes, this is **org.apache.camel.archetypes**. For additional Fuse archetypes, this is **org.jboss.fuse.fis.archetypes**.
- **archetypeArtifactId**: The name of the archetype to be used.
- **archetypeVersion**: The version of the archetype. Use **2.21.0.fuse-000077-redhat-1** to match Red Hat Fuse 7.
- **groupId**: The package name for sample classes generated by the archetype and for the new project POM.

- **artifactId**: The Maven artifact name for the new project.

For example, to generate a Camel Blueprint-based OSGi application using Maven archetypes:

```
$ mvn archetype:generate \
-DarchetypeGroupId=org.apache.camel.archetypes \
-DarchetypeArtifactId=camel-archetype-blueprint \
-DarchetypeVersion=2.21.0.fuse-000077-redhat-1 \
-DgroupId=com.redhat.test \
-DartifactId=arch-test \
-Dversion=1.0.0
```

The Fuse Management CLI

Red Hat Fuse provides a sophisticated management console with web and command-line (CLI) interfaces. The console is used to manage OSGi bundles, services, and their configurations.

There are many ways to start the Apache Karaf container provided by Fuse, depending on whether it is a standalone instance or part of a cluster. The easiest way in a development environment is by running the **fuse** script, which starts the container and provides a Karaf shell prompt:

```
$ cd FUSE_HOME/bin/
$ ./fuse
Please wait while Red Hat Fuse is loading...
...
karaf@root>
```

To shut down a Fuse instance started by the **fuse** script, either terminate the prompt using **Ctrl+D** or issue the **shutdown** command.

A user starting Red Hat Fuse over a local connection does not need to authenticate to have administrator rights, but a remote user needs login credentials with proper access rights. In the previous listing, the **root** prompt indicates that the current user has full administrator privileges over the Karaf instance.

Management commands have a prefix identifying the feature that provides them. Fuse and Karaf provides prefixes such as **osgi**: for managing OSGi bundles, and **log**: for managing the container and applications logs. If a management command does not include a prefix, then **osgi**: is assumed. For example, **osgi:list** and just **list** are the same command.

The **osgi:help** command lists all commands currently known by Fuse/Karaf, which can be a very long list. You can also use **help command** or **command --help** to get information about a specific command.

The Fuse/Karaf CLI supports piping the output of one command to another using normal UNIX syntax of a vertical bar (|). The **shell**: commands are made to work with pipes, for example **shell:tail** and **shell:grep** work much like their UNIX shell counterparts. The **shell**: prefix can also be omitted.

The console itself is extensible. A bundle can register management services and provide additional management commands. For example, the AMQ service adds the **activemq**: prefix for managing AMQ queues and subscriptions.

Managing Camel Routes Deployed to Karaf

To deploy and start Camel routes packaged as an OSGi bundle, use the **osgi:install** command passing the Maven coordinates as arguments:

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

If the bundle POM contains the following identifiers:

```
<groupId>com.example.integration</groupId>
<artifactId>routes</artifactId>
<version>1.0</version>
<name>Sample Bundle</name>
```

The **osgi:install** command becomes:

```
karaf@root> osgi:install mvn:com.example.integration/routes/1.0
```

The **osgi:install** command loads the bundle from the local Maven repository or if that fails, the command loads from any other repository referenced by the Maven **settings.xml** configuration file. The CLI returns a numeric bundle ID that is used by other management commands.



Note

The exact sequence of Maven repositories and configuration files used by Fuse and Karaf is found in **\$FUSE_HOME/etc/org.ops4j.pax.url.mvn.cfg** inside your Fuse installation.

The **osgi:install** command may also load a bundle from the local file system.

```
osgi:install file:path_to_bundle_jar
```

In the following example, **/home/user/integration** is a Maven project folder and **routes-1.0.jar** is a JAR that contains an OSGi bundle:

```
karaf@root> osgi:install file:/home/user/integration/target/routes-1.0.jar
```

The numeric bundle ID is found with the **osgi:list** command:

```
karaf@root> osgi:list
...
217 | Active | 80 | 7.0.0 | OPS4J Pax Web - Runtime
219 | Active | 80 | 7.0.0 | OPS4J Pax Web - Undertow
220 | Active | 80 | 1.0 | Sample Bundle
```

Use the **osgi:start** command to start an installed bundle:

```
karaf@root> osgi:start 220
```

The **-s** option may be used to install and start a bundle at the same time:

```
karaf@root> osgi:install -s mvn:com.example.integration/routes/1.0
```

Stop a bundle using the **osgi:stop** command:

```
karaf@root> osgi:stop 220
```

Finally, a bundle can be uninstalled using the **osgi:uninstall** command:

```
karaf@root> osgi:uninstall 220
```

All the work performed in the background by Fuse can be seen in the Karaf logs:

```
karaf@root> log:display
...
12:23:21.571 INFO [Blueprint Event Dispatcher: 1] Attempting to start
  CamelContext: example-context
12:23:21.574 INFO [Blueprint Event Dispatcher: 1] Apache Camel 2.21.0.fuse-000077-
redhat-1 (CamelContext: example-context) is starting
...
```

How Karaf Locates Bundles

The Fuse CLI **osgi:install** command allows for installing a bundle from the local file system, from a Maven repository, and a few other sources, such as HTTP servers.

When installing from the file system, the URI schema is **file:** and the context path is the file system path. When installing from a Maven repository, the URI is **mvn:** and the context path is the Maven artifact ID.

Contrary to what a developer might expect, when a bundle is installed from a Maven repository, Red Hat Fuse and Karaf does not automatically follow the bundle dependencies and does not try to fetch them from Maven repositories. This happens because the bundle's symbolic name might not match the Maven module name. This mean that any dependencies of the bundle must be installed manually.

A bundle specifies its dependencies using the package names, which do not match either the bundle's symbolic name or the Maven artifact ID. For that reason, there is no way for Karaf to find the correct artifacts in a Maven repository.

Use Fuse CLI commands, such as **osgi:list** and **log:display** to find installed bundles and to get information about installation and start up errors, including missing dependencies.

The bundle status from **osgi:list** shows if a bundle had its dependencies resolved or not. Here are the states, as defined by the OSGi specification:

- **Installed:** The OSGi runtime knows the bundle exists and where to find it for download.
- **Resolved:** All bundle dependencies are available to the OSGi runtime. All of them come from other installed bundles.
- **Starting:** The bundle is being initialized and started.
- **Active:** The bundle is running. Its classes are available to other bundles.

- **Stopping:** The bundle is being stopped.
- **Uninstalled:** The bundle was uninstalled. Karaf forgets about the bundle at this point, so this state should rarely be shown by **osgi:list**.

By default, Karaf defers to the Maven settings in `~/.m2/settings.xml` to configure the dependency repository, including any remote repositories specified there. For a developer, this behavior is desirable for its configuration simplicity, but a systems administrator might want more control in a production environment.

Karaf stores settings in a number of Java properties files in the `$FUSE_HOME/etc` folder. The `config.properties` file includes a few settings that change Karaf behavior regarding Maven repositories:

- **org.ops4j.pax.url.mvn.settings:** provides an alternative location for a Maven `settings.xml` file. If this is not found, Karaf attempts to find a repository the same way the `mvn` command would, by deferring to the `~/.m2/settings.xml` file configuration.
- **org.ops4j.pax.url.mvn.defaultRepositories:** a list of local Maven repositories searched by Karaf. The default list includes the standard user repository in `~/.m2/repository` and the internal Karaf repository in `$FUSE_HOME/data/repository`. The latest one is required for Karaf to work.
- **org.ops4j.pax.url.mvn.repositories:** a list of remote Maven repositories searched by Karaf. The default list includes a few Red Hat and community repositories, so a Fuse installation works even if a user has not configured the Maven `settings.xml` file.



Warning

Do not try to make configuration changes by editing the `FUSE_HOME/etc/org.ops4j.pax.url.mvn.cfg` file. These settings are used only the first time Fuse is started. Afterwards, the settings in `FUSE_HOME/etc/config.properties` take precedence.

Camel Dependencies in Karaf

Installing a Camel route packaged as an OSGi bundle would normally require importing the packages for each Camel module required by the route, as it would for any other dependency. However, Fuse adds services to Karaf that inspect each installed bundle and search for signs of Camel usage, such as a Blueprint configuration file with a `camelContext` definition or Camel CDI annotations, and then adds the dependencies of Camel bundles.

The Camel bundles are pre-installed during the Fuse startup procedure, as can be seen by listing all installed bundles just after starting Red Hat Fuse:

```
karaf@root> osgi:list | grep camel
56 | Active | 50 | 2.21.0.fuse-000077-redhat-1 | camel-blueprint
57 | Active | 80 | 2.21.0.fuse-000077-redhat-1 | camel-commands-core
58 | Active | 50 | 2.21.0.fuse-000077-redhat-1 | camel-core
...
```

Each Camel component module is installed as a different bundle; for example, `camel-ftp` and `camel-jms`. Other Camel components not provided by the Fuse installation can be added by the developer if they are packaged as a bundle, but their usage is outside the Fuse product support scope.



References

Red Hat Fuse

<https://www.redhat.com/en/technologies/jboss-middleware/fuse>

Apache Karaf community

<http://karaf.apache.org/>

Apache Aries community that provides the Blueprint and other OSGi libraries used on Apache Karaf

<http://aries.apache.org/modules/blueprint.html>

OSGi Architecture

<https://www.osgi.org/developer/architecture/>

Apache Felix community

<http://felix.apache.org/>

Apache Felix Maven plug-in

<http://felix.apache.org/documentation/subprojects/apache-felix-maven-bundle-plugin-bnd.html>

Camel Maven archetypes

<http://camel.apache.org/camel-maven-archetypes.html>

► Guided Exercise

Deploying Camel Projects with Red Hat Fuse

In this exercise, you will deploy a Camel-based application on Red Hat Fuse.

Outcomes

You should be able to deploy a Camel based application as a bundle on Red Hat Fuse.

Before You Begin

You must have Red Hat Fuse installed at `/opt/karaf` on the **workstation** VM.

A starter project is provided for you, including a route that prints a message to the console every second, and a REST DSL based route that returns a simple textual greeting.

Run the following command to download the starter project and data files used by this exercise:

```
[student@workstation ~]$ lab camel-fuse setup
```

- 1. Open JBoss Developer Studio and import the starter Maven project.
 - 1.1. Open JBoss Developer Studio (**Applications** → **Programming** → **JBoss Developer Studio**). JBoss Developer Studio requests a default workspace. Leave the default configuration (`/home/student/workspace`) and click **Launch**.
 - 1.2. Import the Maven project by selecting the JBoss Developer Studio main menu **File** → **Import**.
From the **Import** dialog box, select **Maven** → **Existing Maven Projects** and click **Next >**.
 - 1.3. Click **Browse** and choose `/home/student/JB421/labs/camel-fuse`. Click **OK** to choose the Maven project. Click **Finish** to complete the import process in JBoss Developer Studio.
A new project named **camel-fuse** is listed in the **Project Explorer** view. Ignore errors in the starter project. You will fix these errors as you progress through the exercise.
- 2. Enable the bill of materials (BOM) for deploying Camel applications on Red Hat Fuse.
In the **pom.xml** file, add the BOM for Fuse Karaf in the **dependencyManagement** section. Replace the **CHANGE_ME** value as follows:

```
...
<dependencyManagement>
  <dependencies>
    <dependency>
```

```
<groupId>org.jboss.redhat-fuse</groupId>
<!-- TODO: Add the Fuse Karaf BOM -->
<artifactId>fuse-karaf-bom</artifactId>
<version>${jboss.fuse.bom.version}</version>
<type>pom</type>
<scope>import</scope>
</dependency>
</dependencies>
</dependencyManagement>
...
```

Press **Ctrl+S** to save your changes.

After you save the changes to the Maven POM, the project is rebuilt and the errors disappear.

► 3. Review the rest of the **pom.xml** file.

Briefly review the Maven dependencies listed in the **dependencies** section of the POM file. The application uses the basic **camel-core** and **camel-blueprint** components, and some logging-related dependencies.

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

Note the Maven bundle plug-in declaration in the **plugins** section, which is used to create an OSGi bundle of the application that can be deployed to Red Hat Fuse:

```
...
<plugin>
  <groupId>org.apache.felix</groupId>
  <artifactId>maven-bundle-plugin</artifactId>
  <version>${version-maven-bundle-plugin}</version>
  <extensions>true</extensions>
  <configuration>
    <instructions>
      <Bundle-SymbolicName>camel-fuse</Bundle-SymbolicName>
      <Bundle-Name>Camel Deployment on Fuse [camel-fuse]</Bundle-Name>
    </instructions>
  </configuration>
</plugin>
...
```

- 4. Inspect the route configuration in the blueprint XML configuration file.

In the **Project Explorer** view, expand the **camel-fuse** → **src/main/resources/OSGI-INF/blueprint** directory. Double-click the **blueprint.xml** file.

Click the **Source** tab in the bottom of the editor. Note how the **log-route** is configured to print a hello greeting every second to the console:

```
...
<route id="log-route">
    <from id="message-timer" uri="timer:foo?period=1s"/>
    <setBody id="set-message">
        <simple>Hello from Camel!</simple>
    </setBody>
    <log id="log-message" message="">>>> ${body} : ${id}">
</route>
...
```

Also note the **packageScan** declaration under the Camel context named **fuse-example-context**. This makes Camel scan the classes under the **com.redhat.training.jb421** folder automatically, and look for route builder declarations:

```
...
<camelContext id="fuse-example-context" xmlns="http://camel.apache.org/schema/blueprint">
    <packageScan>
        <package>com.redhat.training.jb421</package>
    </packageScan>
...
...
```

- 5. Inspect the route builder class.

In the **Project Explorer** view, expand **src/main/java** → **com.redhat.training.jb421**. Double-click the **RestRouteBuilder.java** file.

Review the implementation of the Camel route. It exposes a single HTTP GET endpoint at the URL **/camel/hello/{name}**, and responds with a simple text greeting and the host name where the route is deployed:

```
...
@Override
public void configure() throws Exception {

    rest("/hello").get("{name}").produces("application/json").to("direct:sayHello");

    from("direct:sayHello").routeId("HelloREST")
        .setBody().simple("{\n"
        + "    greeting: Hello, ${header.name}\n"
        + "    server: " + System.getenv("HOSTNAME") + "\n"
        + "}\n");
}
```

► 6. Start the Red Hat Fuse instance.

In a terminal window on the **workstation** VM, run the following command to start Fuse:

```
[student@workstation ~]$ cd /opt/karaf/bin  
[student@workstation bin]$ ./fuse
```

Wait until the Fuse instance is started. You should see the following during start up, after which you are dropped into the Karaf shell:

```
...  
Red Hat Fuse starting up. Press Enter to open the shell now...  
100% [=====]  
  
Karaf started in 4s...  
...  
Red Hat Fuse (7.0.0.fuse-000191-redhat-1)  
...  
karaf@root()>
```

► 7. Deploy the application to Fuse.

- 7.1. Open a new terminal on the **workstation** VM, and run the following command from the **~/JB421/labs/camel-fuse** folder to package and install the application to the local Maven repository. You use the Karaf shell to deploy the application to Fuse later in the exercise:

```
[student@workstation ~]$ cd ~/JB421/labs/camel-fuse  
[student@workstation camel-fuse]$ mvn clean install
```

- 7.2. Install the **camel-servlet** feature in Fuse. The application the feature to expose the HTTP REST endpoint declared in the route builder class. Switch to the terminal window where the Karaf shell is running, and run the following command:

```
karaf@root()> feature:install camel-servlet
```

Run the **list** command in the Karaf shell to verify that the **camel-servlet** bundle is installed and active:

```
karaf@root()> list | tail -n 3  
217 | Active | 80 | 7.2.2 | OPS4J Pax Web - Runtime  
219 | Active | 80 | 7.2.2 | OPS4J Pax Web - Undertow  
223 | Active | 50 | 2.21.0.fuse-710018-redhat-00001 | camel-servlet
```

- 7.3. Deploy the bundle to the running Fuse instance. Switch to the terminal window running the Karaf shell, and run the following command in a single line:

```
karaf@root()> install -s file:///home/student/JB421/labs/camel-fuse/target/camel-fuse-1.0.0.jar  
Bundle ID: 226
```

Make note of the bundle ID printed on the console. You will need it later in the exercise to stop and uninstall the bundle from Fuse.

Run the **list** command in the Karaf shell again to verify that the **camel-fuse** bundle is deployed and active:

```
karaf@root()> list | tail -n 3
219 | Active | 80 | 7.2.2                                | OPS4J Pax Web - Undertow
223 | Active | 50 | 2.21.0.fuse-710018-redhat-00001 | camel-servlet
225 | Active | 80 | 1.0.0                                | Camel Deployment on Fuse
[camel-fuse]
```

► 8. Test the Camel route.

- 8.1. Run the **log:display** command on the Karaf shell to verify that the **log-route** in the application prints a message to the console every second:

```
karaf@root()> log:display -n 3
... timer://foo] >>> Hello from Camel! : ID-workstation-lab-example-
com-1534151768492-0-634
... timer://foo] >>> Hello from Camel! : ID-workstation-lab-example-
com-1534151768492-0-636
... timer://foo] >>> Hello from Camel! : ID-workstation-lab-example-
com-1534151768492-0-638
```

- 8.2. Test the REST API endpoint exposed by the application.

From a terminal window on the **workstation** VM, use the **curl** command to test the route:

```
[student@workstation ~]$ curl http://localhost:8181/camel/hello/Luke
{
  greeting: Hello, Luke
  server: workstation.lab.example.com
}
```

► 9. Clean up.

- 9.1. Stop and uninstall the application.

Run the **stop** command in the Karaf shell to stop the application. Pass the bundle ID of the **camel-fuse** bundle as the argument:

```
karaf@root()> stop 226
```

Run the **uninstall** command in the Karaf shell to uninstall the application. Pass the bundle ID of the **camel-fuse** bundle as the argument:

```
karaf@root()> uninstall 226
```

- 9.2. Stop the Red Hat Fuse instance.

In the terminal window running the Karaf shell, press **Ctrl+D** to stop the Fuse instance and exit to the operating system shell.

- 9.3. Close the project.

In the **Project Explorer** view of the IDE, right-click **camel-fuse** and click **Close Project**.

This concludes the guided exercise.

Deploying Routes with JBoss EAP

Objectives

After completing this section, students should be able to deploy Camel integration projects to JBoss EAP as a Java EE archive.

Introducing Red Hat Fuse on JBoss EAP

The JBoss Enterprise Application Platform, or JBoss EAP, is a Java Enterprise Edition (Java EE) certified application server. It is based on the WildFly open source project and serves as the foundation for multiple JBoss middleware products from Red Hat.

When Red Hat Fuse is run on JBoss EAP, it leverages many EAP features such as distributed transaction management, clustering, security, messaging, and access to Java EE APIs such as JPA, EJB, CDI, and more.

Red Hat Fuse is installed on EAP as a layered product that adds new subsystems to EAP and is integrated with EAP management features.

A JBoss EAP subsystem extends the application server to recognize and automatically configure new kinds of applications. Red Hat Fuse on JBoss EAP recognizes Camel applications that use either the Spring framework or CDI.

Running Red Hat Fuse on EAP requires downloading and running the JBoss EAP installer as a first step, and then downloading and running the Red Hat Fuse on EAP installer over the existing JBoss EAP installation.

Installing Fuse on JBoss EAP

Assuming that you have a JBoss EAP 7.1 installed at the location **EAP_HOME**, download the Fuse installer for EAP JAR file, and copy it to the **EAP_HOME** folder.

Run the Fuse installer for JBoss EAP as follows to install the Fuse subsystem on EAP:

```
$ cd $EAP_HOME
$ java -jar ~/Downloads/fuse-eap-installer-7.0.0.fuse-000085-redhat-1.jar
...
Processing config for: camel
  Writing 'layers=fuse' to: ./modules/layers.conf
  Enable camel configuration in: ./standalone/configuration/standalone.xml
  Enable camel configuration in: ./standalone/configuration/standalone-full.xml
  Enable camel configuration in: ./standalone/configuration/standalone-full-ha.xml
  Enable camel configuration in: ./standalone/configuration/standalone-ha.xml
Red Hat Fuse Integration Platform - Version 7.0.0.fuse-000085-redhat-1
```

Methods of Deploying Routes on EAP

The community Camel project offers multiple portable ways to package and start a Camel context for Java EE application servers such as JBoss EAP. The following are the most popular options:

- Package the routes and their supporting classes as a Java EE Web Archive (WAR) and use Servlet initialization code with CDI. Provide Camel classes and its dependencies inside the WAR file as JAR files embedded in the **WEB-INF/lib** folder.
- Package routes and their supporting classes as a standard Java Archive (JAR) and use EJB initialization code with CDI. Provide Camel classes and its dependencies as JAR modules inside an EAR package, or as part of the application server class path.

Red Hat Fuse on JBoss EAP provides an even easier alternative to the ones above: package the routes, without any Camel-related dependencies, as a standard Java Archive (a JAR file) and let the Camel EAP subsystem perform the following tasks:

- Inspect each new deployment for files named **META-INF/spring/*-camel-context.xml**.
- Inspect each new deployment for CDI beans that use Camel classes such as **RouteBuilder**, and annotations such as **@CamelContext**.
- If any of the previous two conditions is true, start the declared Camel context and adds any Camel dependencies to the deployment.

Deploying to JBoss EAP Using Maven

Red Hat provides a bill of materials (BOM) for developing Camel applications that are going to be deployed on EAP. Using the BOM simplifies adding Camel modules to your application, because you do not have to manage individual versions of each Camel component. Add the BOM to your Maven POM file as follows:

```
...
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.jboss.redhat-fuse</groupId>
      <artifactId>fuse-eap-bom</artifactId>
      <version>${fuse.eap.bom.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
...
```

The WildFly open source project, which is the upstream community project upon which JBoss EAP is based, provides a Maven plug-in to automate deployment tasks, based on the JBoss EAP management API.

Using the WildFly Maven plug-in features is easier and more robust than relying on file copy and marker files, and also has the advantage of supporting JBoss managed domain mode.

The following declaration is an example of a WildFly plug-in declaration inside a Maven POM:

```
...
<plugin>
  <groupId>org.wildfly.plugins</groupId>
  <artifactId>wildfly-maven-plugin</artifactId>
  <version>${maven.wildfly.version}</version>
  <configuration>
```

```
<hostname>127.0.0.1</hostname>
<port>9999</port>
<username>admin</username>
<password>secret</password>
</configuration>
</plugin>
...
```

If your JBoss EAP instance has been secured, the WildFly plug-in configuration can take administrative credentials. In a development environment where the EAP instance runs as the user that does the Maven build, however, there is no need to provide credentials.

To package and deploy the Camel routes as a JAR, EAR or WAR deployment, use the **wildfly:deploy** Maven goal. The **-DskipTests** option can be used to make the operation faster by not running unit tests:

```
$ mvn wildfly:deploy -DskipTests
```

Similarly, use the **wildfly:undeploy** Maven goal to undeploy the routes.

The WildFly Maven plug-in also provides features to run EAP management commands, for example to create data sources and queues so that any deployment prerequisite configuration can be fully automated. Check the references at the end of this section for more information.

Using Camel XML Routes With JBoss EAP

Camel applications targeting JBoss EAP are not required to use the Spring framework to define XML routes. All that is required is using the **@ImportResource** annotation from the Camel CDI module, as shown by the following listing:

```
import org.apache.camel.cdi.ImportResource;
...
@ImportResource("camel-context.xml")
@ApplicationScoped
class MyRoutes extends RouteBuilder {
```

The **camel-context.xml** should be accessible as a classpath resource. The easiest way to get that is saving the file in the **src/main/resources** folder of a Maven project.

JBoss Developer Studio Tools for EAP

JBoss Developer Studio has tools for running, debugging and deploying applications to JBoss EAP. These tools work with Camel-enabled applications as well as with standard Java EE applications.

The Server view is used to configure server definitions, then start and stop these servers. For JBoss EAP with Fuse, use the **Red Hat JBoss EAP 7.1** server type.

A Camel integration project can be deployed and undeployed with a JBoss EAP instance configured inside JBoss Developer Studio by using the **Add and remove** context menu choice.

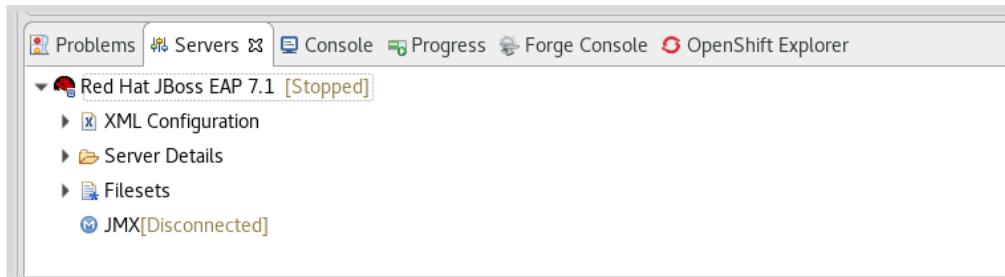


Figure 7.2: A JBoss EAP instance configured inside JBoss Developer Studio

After the server has been started, its logging output can be seen in the Console view.

The screenshot shows the JBoss Developer Studio interface with the 'Console' tab selected in the top navigation bar. The main workspace displays the log output for a JBoss EAP 7.1 instance. The logs show the server starting up, including the configuration of Camel endpoints and the deployment of Camel routes. The log ends with the message 'Started 561 of 788 services (359 services are la...'.

```

Red Hat JBoss EAP 7.1 [JBoss Application Server Startup Configuration] /usr/lib/jvm/java-1.8.0-openjdk-1.8.0.181-7.b13.fc28.x86_64/bin/java (Aug 17, 2018, 2:39:32 PM)
14:39:41,329 INFO [org.wildfly.extension.undertow] (ServerService Thread Pool -- 65) WFLYUT0021: Registered web context: '/hawtio' for server 'default-server'
14:39:41,334 INFO [org.wildfly.extension.camel] (ServerService Thread Pool -- 65) Add Camel endpoint: http://127.0.0.1:8080/hawtio
14:39:42,151 INFO [org.apache.camel.impl.DefaultCamelContext] (MSC service thread -1) MSC service thread -1) Activating Camel Subsystem
14:39:42,150 [org.apache.camel.impl.DefaultCamelContext] (MSC service thread -1) MSC service thread -1) CamelContext: camel-eap-context (core=1, classpath: 2791)
14:39:42,303 INFO [org.apache.camel.impl.HeadersMapFactoryResolver] (MSC service thread -1) HeadersMapFactoryResolver detected and using custom HeadersMapFactory: org.apache.camel.component.headersmap.FastHeadersMapFacto
14:39:42,323 WARN [org.wildfly.extension.camel] (MSC service thread -1) Ignoring configured host: http://0.0.0.0/api/hello/%7Bname%7D?httpMethodRestrict=GET%2COPTIONS&matchOnUriPrefix=false
14:39:42,364 INFO [org.wildfly.extension.camel] (MSC service thread -1) Add Camel endpoint: http://127.0.0.1:8080/api/hello/{name}
14:39:42,364 INFO [org.apache.camel.impl.DefaultCamelContext] (MSC service thread -1) Route: routel started and consuming from: http://0.0.0.0/api/hello/%7Bname%7D?httpMethodRestrict=GET%2COPT
14:39:42,365 INFO [org.apache.camel.impl.DefaultCamelContext] (MSC service thread -1) Total 1 routes, of which 1 are started
14:39:42,400 INFO [org.apache.camel.impl.DefaultCamelContext] (MSC service thread -1) Apache Camel 2.22.0 (camel-eap-context) started in 1.551 seconds
14:39:42,400 [org.wildfly.extension.camel] (ServerService Thread Pool -- 65) WFLYUT0021: Registered web context: '/camel-eap' for server 'default-server'
14:39:42,430 INFO [org.wildfly.extension.camel] (ServerService Thread Pool -- 65) Add Camel endpoint: http://127.0.0.1:8080/camel-eap
14:39:42,438 INFO [org.jboss.as.server] (Controller Boot Thread) WFLYSRV0010: Deployed "camel-eap.war" [runtime-name : camel-eap.war"]
14:39:42,438 INFO [org.jboss.as.server] (ServerService Thread Pool -- 38) WFLYSRV0010: Deployed "hawtio-wildfly-2.0.0.fuse-000172-redhat-1.war" [runtime-name : hawtio-wildfly-2.0.0.fuse-000172-r
14:39:42,532 INFO [org.jboss.as.server] (Controller Boot Thread) WFLYSRV0010: Resuming server
14:39:42,537 INFO [org.jboss.as] (Controller Boot Thread) WFLYSRV0060: Http management interface listening on http://127.0.0.1:9990/management
14:39:42,538 INFO [org.jboss.as] (Controller Boot Thread) WFLYSRV0051: Admin console listening on http://127.0.0.1:9990
14:39:42,538 INFO [org.jboss.as] (Controller Boot Thread) WFLYSRV0023: JBoss EAP 7.1.0.GA (WildFly Core 3.0.10.Final-redhat-1) started in 9981ms - Started 561 of 788 services (359 services are la...

```

Figure 7.3: A JBoss EAP instance output inside the Console view

The JBoss Developer Studio tools for running a server also work with JBoss Fuse/Karaf. Choose the **Red Hat Fuse 7.0 Server** server type. JBoss Developer Studio also automatically starts an SSH session with the JBoss Fuse server so the developer can use Fuse management commands.

Managing Camel Routes Deployed on EAP

The easiest way for a developer to start a JBoss EAP instance from the command line is using the **standalone.sh** script. It starts a single server instance in standalone mode.

```
$ cd EAP_HOME/bin/
$ ./standalone.sh
...
...[org.wildfly.extension.camel] (MSC service thread 1-3) Activating Camel Subsystem
...
...JBoss EAP 7.1.0.GA (WildFly Core 3.0.10.Final-redhat-1) started in 9981ms
```

If messaging capabilities are required, the **-c** option must be used to specify the **standalone-full.xml** configuration file, which includes, among other capabilities, the AMQ messaging subsystem.

```
$ cd EAP_HOME/bin/
$ ./standalone.sh -c standalone-full.xml
```

To stop the JBoss EAP server instance, press **Ctrl+C**.

Managing a running EAP instance is done using the **jboss-cli.sh** script that starts the CLI shell. The **--connect** option connects the shell to a local running standalone instance:

```
$ cd EAP_HOME/bin/  
$ ./jboss-cli.sh --connect  
[standalone@localhost:9999 /]
```

The JBoss EAP management CLI works in a similar way to a file system, where management objects can be browsed, created, removed, and changed. Each management object provides new attributes and operations.

A few commands are provided by the CLI shell itself. Among them, the **deploy** command is used to deploy a packaged application. For example:

```
[standalone@localhost:9999 /] deploy ~/myroutes/target/myroutes-1.0.jar
```

The effect of a **deploy** command can be seen in the EAP server logs. For example:

```
17:00:55,949 INFO [org.jboss.as.server] (Controller Boot Thread) WFLYSRV0010:  
Deployed "myroutes-1.0.jar" (runtime-name : "myroutes-1.0.jar")
```

If the deployment was recognized by the EAP Camel subsystem, then the EAP server logs should also show messages about starting the Camel context and its routes. For example:

```
17:00:54,300 INFO [org.jboss.as.server.deployment] (MSC service thread  
1-3) WFLYSRV0027: Starting deployment of "myroutes-1.0.jar" (runtime-name:  
"myroutes-1.0.jar")  
...  
17:00:55,203 INFO [org.apache.camel.spring.SpringCamelContext] (MSC service  
thread 1-3) Apache Camel 2.21.0.fuse-000077-redhat-1 (CamelContext:  
processorContext) is starting  
...  
17:00:55,539 INFO [org.apache.camel.spring.SpringCamelContext] (MSC service  
thread 1-3) Total 1 routes, of which 1 are started.  
17:00:55,540 INFO [org.apache.camel.spring.SpringCamelContext] (MSC  
service thread 1-3) Apache Camel 2.21.0.fuse-000077-redhat-1 (CamelContext:  
processorContext) started in 0.336 seconds  
...  
17:00:55,949 INFO [org.jboss.as.server] (management-handler-thread - 3)  
WFLYSRV0009: Deployed "myroutes-1.0.jar" (runtime-name : "myroutes-1.0.jar")
```

Each deployment is identified, by default, by its file name. A list of current deployments can be listed using the **/deployment** namespace:

```
[standalone@localhost:9999 /] ls /deployment  
myroutes-1.0.jar hawtio-wildfly-2.0.0.fuse-000172-redhat-1.war
```

The CLI shell also provides the **undeploy** command:

```
[standalone@localhost:9999 /] undeploy myroutes-1.0.jar
```

The effect of an **undeploy** command can also be seen in the EAP server logs. For example:

```
17:02:34,025 INFO [org.jboss.as.server] (management-handler-thread - 6)
JBAS015858: Undeployed "myroutes-1.0.jar" (runtime-name: "myroutes-1.0.jar")
```



References

WildFly Maven plug-in

<https://docs.jboss.org/wildfly/plugins/maven/latest/>

For more information about packaging and deploying Camel routes into JBoss EAP:

Deploying into JBoss EAP

https://access.redhat.com/documentation/en-us/red_hat_fuse/7.0/html-single/deploying_into_jboss_eap/

For more information about JBoss EAP management:

JBoss EAP 7.1 product documentation

<https://access.redhat.com/documentation/en/red-hat-jboss-enterprise-application-platform/?version=7.1>

► Guided Exercise

Deploying Camel projects with JBoss EAP

In this exercise, you will deploy a Camel-based application on JBoss EAP as a WAR file.

Outcomes

You should be able to deploy a Camel-based application as a WAR file on JBoss EAP.

Before You Begin

You must have JBoss EAP installed at `/opt/jboss-eap-7.1` on the **workstation** VM. This JBoss EAP installation already has the Red Hat Fuse subsystem installed in it, and is ready for route deployment.

A starter project is provided for you to start from that already includes a **RouteBuilder** implementation, and a Java Bean class with a single method that returns a simple textual greeting.

Run the following command to download the starter project and data files used by this exercise:

```
[student@workstation ~]$ lab camel-eap setup
```

- ▶ 1. Open JBoss Developer Studio and import the starter Maven project.
 - 1.1. Open JBoss Developer Studio (**Applications** → **Programming** → **JBoss Developer Studio**). JBoss Developer Studio requests a default workspace. Leave the default configuration (`/home/student/workspace`) and click **Launch**.
 - 1.2. Import the Maven project by selecting from the JBoss Developer Studio main menu **File** → **Import**.
From the **Import** dialog box, select **Maven** → **Existing Maven Projects** and click **Next >**.
 - 1.3. Click **Browse** and choose `/home/student/JB421/labs/camel-eap` and click **OK** to choose the Maven project. Click **Finish** to complete the import process in JBoss Developer Studio.
A new project named **camel-eap** will be listed in the **Project Explorer** view. Ignore errors in the starter project. You will fix these errors as you progress through the exercise.
- ▶ 2. Enable the bill of materials (BOM) for deploying Camel applications on JBoss EAP.
In the **pom.xml** file, add the BOM for Fuse EAP in the **dependencyManagement** section. Replace the **CHANGE_ME** value as follows:

```

...
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.jboss.redhat-fuse</groupId>
      <!-- TODO: Add Fuse EAP BOM -->
      <artifactId>fuse-eap-bom</artifactId>
      <version>${fuse.eap.bom.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
...

```

Press **Ctrl+S** to save your changes.

Once you save the changes to the Maven POM, the project will be rebuilt and the errors should disappear.

► 3. Review the rest of the `pom.xml` file.

Briefly review the Maven dependencies listed in the **dependencies** section of the POM file. The application uses the **camel-cdi** component and a number of Java EE API dependencies. Note the **provided** scope declaration for all the dependencies. The JBoss EAP container provides all the APIs at run time, and the dependencies are not bundled in the application WAR file.

```

...
<dependencies>
  <dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-cdi</artifactId>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>org.jboss.spec.javaee.ws.rs</groupId>
    <artifactId>jboss-jaxrs-api_2.0_spec</artifactId>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>javax.enterprise</groupId>
    <artifactId>cdi-api</artifactId>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>org.jboss.spec.javaee.ejb</groupId>
    <artifactId>jboss-ejb-api_3.2_spec</artifactId>

```

```
<scope>provided</scope>
</dependency>
</dependencies>
...
```

Note the WildFly Maven plug-in declaration in the **plugins** section, which is used to deploy the WAR file of the application to JBoss EAP:

```
...
<plugin>
  <groupId>org.wildfly.plugins</groupId>
  <artifactId>wildfly-maven-plugin</artifactId>
  <version>${version-wildfly-maven-plugin}</version>
</plugin>
...
```

► 4. Inspect the **HelloBean** class.

In the **Project Explorer** view, expand the **camel-eap → src/main/java/com/redhat/training/fuse** directory. Double-click **HelloBean.java** file.

HelloBean is a simple plain old Java object (POJO) class with a single method called **hello**, that takes a name as the parameter, and returns a greeting with the input name and the host name where the application is deployed:

```
...
@Named("helloBean")
public class HelloBean {

    public String hello(String name) throws Exception {
        return "Hello " + name + "!. My hostname is " +
        InetAddress.getLocalHost().getHostName() + "\n";
    }
}
```

► 5. Inspect the route builder class.

In the **Project Explorer** view, expand the **camel-eap → src/main/java/com/redhat/training/fuse** directory. Double-click the **HelloRoute.java** file.

Review the implementation of the Camel route. It exposes a single HTTP GET endpoint at the URL **/api/hello/{name}**, and forwards the **name** parameter from the request to the **hello** method of the **HelloBean** class:

```
...
@ApplicationScoped
@ContextName("camel-eap-context")
public class HelloRoute extends RouteBuilder {

    @Override
    public void configure() throws Exception {

        restConfiguration()
            .component("undertow")
            .contextPath("api");
```

```

        rest("/hello")
            .get("{name}")
            .produces(MediaType.TEXT_PLAIN)
            .route()
                .bean(HelloBean.class, "hello(${header.name})")
                .log("Received greeting request...")
            .endRest();
    }
}

```

► 6. Start the JBoss EAP instance.

In a terminal window on the **workstation** VM, run the following command to start JBoss EAP:

```
[student@workstation ~]$ cd /opt/jboss-eap-7.1/bin
[student@workstation bin]$ ./standalone.sh
```

Wait until JBoss EAP is started. You should see the following during start up:

```
...
...WFLYSRV0049: JBoss EAP 7.1.0.GA (WildFly Core 3.0.10.Final-redhat-1) starting
...
...[org.wildfly.extension.camel] (MSC service thread 1-2) Activating Camel
Subsystem
...
...WFLYSRV0025: JBoss EAP 7.1.0.GA (WildFly Core 3.0.10.Final-redhat-1) started in
9525ms...
...
```

► 7. Deploy the application to JBoss EAP.

Open a new terminal window on the **workstation** VM, and run the following command from the **~/JB421/labs/camel-eap** folder:

```
[student@workstation ~]$ cd ~/JB421/labs/camel-eap
[student@workstation camel-eap]$ mvn clean wildfly:deploy
[INFO] -----
[INFO] Building Camel EAP example 1.0.0
[INFO] -----
...
[INFO] Webapp assembled in [30 msecs]
...
[INFO] Building war: /home/student/JB421/labs/camel-eap/target/camel-eap.war
...
[INFO] --- wildfly-maven-plugin:1.2.2.Final:deploy (default-cli) @ camel-eap ---
...
INFO: ELY00001: WildFly Elytron version 1.1.7.Final
```

Chapter 7 | Deploying Camel Routes

```
[INFO] -----  
[INFO] BUILD SUCCESS  
[INFO] -----  
...
```

Switch to the terminal window running JBoss EAP, and note the deployment of the WAR file:

```
...  
...WFLYSRV0027: Starting deployment of "camel-eap.war"...  
...  
...Camel CDI is starting Camel context [camel-eap-context]  
...Apache Camel 2.21.0.fuse-000077-redhat-1 (CamelContext: camel-eap-context) is  
starting  
...  
...Camel context starting: camel-eap-context  
...  
...Add Camel endpoint: http://127.0.0.1:8080/api/hello/{name}  
...  
WFLYSRV0010: Deployed "camel-eap.war" (runtime-name : "camel-eap.war")
```

► **8.** Test the Camel route.

From a terminal window on the **workstation** VM, use the **curl** command to test the route:

```
[student@workstation ~]$ curl http://localhost:8080/api/hello/Luke  
Hello Luke!. My hostname is workstation.lab.example.com
```

Switch to the terminal window running JBoss EAP, and observe the log message indicating that request was received at the route endpoint:

```
...[route2] (default task-1) Received greeting request...
```

► **9.** Clean up.

9.1. Undeploy the application.

Run the following command from the **~/JB421/labs/camel-eap** folder:

```
[student@workstation ~]$ cd ~/JB421/labs/camel-eap  
[student@workstation camel-eap]$ mvn clean wildfly:undeploy  
...  
[INFO] BUILD SUCCESS  
...
```

Switch to the terminal window running JBoss EAP, and note the undeployment of the WAR file:

```
...  
...Remove Camel endpoint: http://127.0.0.1:8080/api/hello/{name}  
...WFLYUT0022: Unregistered web context: '/camel-eap'...  
...Camel CDI is stopping Camel context [camel-eap-context]  
...
```

```
...Graceful shutdown of 1 routes completed in 0 seconds  
...  
...WFLYSRV0009: Undeployed "camel-eap.war"...  
...
```

- 9.2. Stop the JBoss EAP server.

In the terminal window running JBoss EAP, press **Ctrl+C** to stop it.

- 9.3. Close the project.

In the **Project Explorer** view, right-click **camel-eap** and click **Close Project**.

This concludes the guided exercise.

Deploying Routes with Spring Boot

Objectives

After completing this section, students should be able to deploy Camel integration projects to Spring Boot.

Introducing the Spring Boot Runtime

The *Spring Boot* framework makes it easy to create standalone applications based on the Spring framework. While traditional Spring framework applications are meant to be deployed into a web container such as Apache Tomcat or an application server such as JBoss EAP, Spring Boot applications are self-contained Java applications that either embed a web container or rely on other mechanisms, such as a messaging client or a reactive library, that sources events to the application.

Spring Boot builds upon the *Spring framework*. The Spring framework was created as an alternative to the Enterprise Java Beans (EJB) programming models of early Enterprise Java standards, before the release of EJB 3 and CDI. The Spring framework provides an *inversion of control* (*IoC*) container that supports the *dependency injection* (*DI*) programming model.

The Spring framework is comprised of a large ecosystem of libraries maintained by its corporate sponsor Pivotal and the larger developer community. Some of these libraries support standard Enterprise Java APIs, such as JPA, JMS, and JAX-RS, while others provide competing APIs, such as Spring Web. The Spring framework supports multiple implementations of the same APIs and does not prescribe an application architecture.

Spring Boot also relies heavily on the *Spring Cloud* libraries that implement common microservice architecture patterns. Spring Cloud started as a project to integrate Netflix Open Source Software (Netflix OSS) libraries into the Spring framework ecosystem and later grew to support alternative implementations of microservice architecture patterns.

Configuring a Spring Boot Application

Originally, the Spring framework used XML configuration files to set up both application components, named *managed beans*, and infrastructure components. Many Camel developers still rely in this XML configuration file to declare Camel XML routes and integrations with external systems such as messaging middleware and databases.

Later releases of the Spring framework favor an annotation-based approach to dependency injection, similar to CDI. You can still use Spring Beans XML configuration files, but the favored approach is using annotations, automatic configuration, and Spring Boot starters:

- Spring Boot applications are expected to use Spring framework annotations such as **@Component**, **@Autowired**, and **@Bean** wire Spring managed beans together, instead of using **<bean>** elements in a Spring Beans configuration file.
- Spring Boot favors convention over configuration, and automatically configures infrastructure components required by application managed beans based on the dependencies available on the class path.

- Spring Boot **starters** are Maven artifacts that provide default dependency graphs for common scenarios using Spring framework libraries. For example, using libraries for web applications, including web services, requires a web container.

Many Spring Boot starters require configuration parameters provided by the Java system properties, such as configuring TCP port to listen for HTTP connections and the data source URL. Spring Boot initializes these properties from the **application.properties** file and other configuration sources.

Spring Boot starters are organized around features. Among the starters provided by Spring Boot are:

- **spring-boot-starter-web**: starter to use Spring MVC with Apache Tomcat as the embedded web container
- **spring-boot-starter-data-jpa**: starter to use Spring Data JPA with Hibernate and JDBC data sources
- **spring-boot-starter-activemq**: starter to use JMS messaging with Apache ActiveMQ

The Camel community also provides a number of Spring Boot starters for your integration projects. Among the starters provided by Apache Camel are:

- **camel-spring-boot-starter**: starter required to initialize a Camel context and start processing Camel routes
- **camel-servlet-starter**: starter that provides the Camel Servlet for routes that use the REST DSL
- **camel-jackson-starter**: starter that provides JSON type converters
- **camel-hystrix-starter**: starter that provides the Hystrix implementations of the circuit-breaker microservices architecture pattern

A typical integration application requires multiple starter dependencies in its POM configuration file.

Using Camel Routes from a Spring Boot Application

Camel Java routes are declared as Spring managed beans.

To declare a Camel **RouteBuilder**, a Camel **TypeConverter**, or other Camel component as a Spring framework managed bean, use the **@Component** annotation, as in the following example:

```
import org.springframework.stereotype.Component;
...
@Component
public class RestRouteBuilder extends RouteBuilder {
    ...
}
```

A Spring Boot application must provide a class that is annotated as **@SpringBootApplication** and defines a static **main** method that invokes the **SpringApplication.run** static method:

Despite not using a Spring Beans configuration file, a Spring Boot application can use Camel XML routes by adding the **@ImportResource** annotation to the Spring Boot Application class, as in the following example:

```

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.ImportResource;

@SpringBootApplication
@ImportResource({"classpath:spring/camel-context.xml"})
public class Application {

    public static void main(String[] args) {
        org.springframework.context.ApplicationContext ctx =
            org.springframework.boot.SpringApplication.run(
                Application.class, args);
    }
}

```

Configuring Dependencies for a Spring Boot Application

Unlike the JBoss EAP and Apache Karaf runtimes, Red Hat does not provide an enterprise distribution of Spring Boot as part of the Red Hat Fuse product. Red Hat certifies community Spring Boot releases and a subset of its dependencies from the community repositories as compatible with each Red Hat Fuse release.

Red Hat does provide support according to the Red Hat Fuse subscription service-level agreement (SLA) for the libraries where Red Hat maintains an active participation in the upstream development, and so is able to influence feature development plans and provide timely fixes to critical issues.

Among the components of a Spring Boot application that Red Hat provides full support are:

- JPA using Hibernate
- JAX-RS using Apache CXF
- Web containers using Apache Tomcat and Undertow
- Spring Cloud integration with Kubernetes
- Camel libraries and starters

These artifacts must be downloaded from the JBoss Enterprise Maven repository to be eligible for production support.

For other Spring Boot libraries and dependencies certified for Red Hat Fuse, Red Hat only provides best effort support, because fixes depend on the upstream community, which is not tied to the Red Hat Fuse subscription SLA terms.

Red Hat Fuse developers targeting Spring Boot as the production runtime are advised to evaluate the risk involved in using libraries from Spring Boot that are not tested for compatibility with Red Hat Fuse. The complete list of libraries that are either supported or certified are available in the product documentation.

Red Hat recommends that the POM for a Spring Boot application uses the bill of materials (BOM) provided by the Red Hat Fuse product as an imported POM, as in the following example:

```

...
<properties>
  ...
  <jboss.fuse.bom.version>7.1.0.fuse-710019-redhat-00002</
jboss.fuse.bom.version>
<properties>
  ...
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.jboss.redhat-fuse</groupId>
      <artifactId>fuse-springboot-bom</artifactId>
      <version>${jboss.fuse.bom.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>

```

All integration applications require the Camel Spring Boot starter:

```

...
<dependencies>
  <dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-spring-boot-starter</artifactId>
  </dependency>
  ...
<dependencies>
  ...

```

Applications that use the Camel REST DSL need Spring Boot Starter Web, and Red Hat recommends Undertow as the web container instead of Apache Tomcat.

The following example provides the recommended dependencies for an application that uses the REST DSL and processes JSON data:

```

...
<dependencies>
  ...
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <exclusions>
      <exclusion>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-tomcat</artifactId>
      </exclusion>
    </exclusions>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-undertow</artifactId>
  
```

```

</dependency>
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-servlet-starter</artifactId>
</dependency>
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-jackson-starter</artifactId>
</dependency>
<dependencies>
...

```

Any other Camel, Spring Boot, Spring Cloud, or Spring framework library you require needs to be added as an explicit dependencies, unless it is provided by another starter already defined inside your project's POM.

Packaging and Running a Spring Boot Application

Spring Boot provides a Maven plug-in that packages the application and its dependencies in a *Fat JAR* archive. A Fat JAR is a JAR that embeds not only application classes, but all of the application dependencies. It can be executed by the **java -jar** command without configuring a classpath.

To configure a Maven project to use the Spring Boot plug-in, add the **org.springframework.boot:spring-boot-maven-plugin** plug-in to the project's POM file, as in the following example:

```

...
<properties>
...
    <spring-boot.version>1.5.12.RELEASE</spring-boot.version>
<properties>
...
<build>
    <plugins>
        ...
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
            <version>${spring-boot.version}</version>
            <executions>
                <execution>
                    <goals>
                        <goal>repackage</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    <plugins>
<build>
...

```

The developer must take care to specify the Spring Boot version that is compatible with the target Red Hat Fuse release. Mixing different Spring Boot libraries and plug-ins may yield unpredictable results.

The previous configuration binds the Spring Boot Maven Plug-in to the Maven **package** command generates a Fat JAR.

```
$ mvn package
```

And then you can execute the Spring Boot application as a standalone Java SE application:

```
$ java -jar target/my-app.jar
```



References

Spring framework web site

<https://spring.io/projects/spring-framework>

Spring Boot framework web site

<https://spring.io/projects/spring-boot>

Spring Cloud framework web site

<https://projects.spring.io/spring-cloud/>

JBoss Enterprise Maven Repository

<https://access.redhat.com/maven-repository>

For more information about using Red Hat Fuse with Spring Boot, refer to *Deploying into Spring Boot at*

https://access.redhat.com/documentation/en-us/red_hat_fuse/7.0/html-single/deploying_into_spring_boot/

► Guided Exercise

Guided Exercise: Deploying a Camel project with Spring Boot

In this exercise, you will run a Camel Java route and a Camel XML route inside a Spring Boot application.

Outcomes

You should be able to:

- Configure a Maven project's POM file to include Spring Boot starters for Camel.
- Configure a Maven project's POM file to include the Spring Boot Maven plug-in.
- Configure a Spring Boot application class to load Camel routes from an XML file.
- Run the Spring Boot application from its Fat JAR and verify that the Camel routes are active.

Before You Begin

A starter project is provided for you, which includes a **CamelContext** configured in a Spring configuration file, a **RouteBuilder** implementation, and a Spring beans configuration file.

Run the following command to download the starter project and data files used by this exercise:

```
[student@workstation ~]$ lab rest-springboot setup
```

1. Open JBoss Developer Studio and import the starter Maven project.
 - 1.1. Open JBoss Developer Studio (**Applications** → **Programming** → **JBoss Developer Studio**). JBoss Developer Studio requests a default workspace; keep the default configuration (**/home/student/workspace**) and click **Launch**.
 - 1.2. Import the Maven project by selecting JBoss Developer Studio main menu **File** → **Import**.
 - 1.3. From the **Import** dialog box, select **Maven** → **Existing Maven Projects** and click **Next**.
 - 1.4. Click **Browse** and choose **/home/student/JB421/labs/rest-springboot** and click **OK** to choose the Maven project. Click **Finish** to complete the import process in JBoss Developer Studio.
Dismiss the prompt about problems importing the project. You will fix these problems during this exercise.
A new project named **rest-springboot** will be listed in the **Project Explorer** view.
2. Complete the project's POM file.

- 2.1. Include the bill of materials (BOM) for Spring Boot-based Fuse applications.

In the **Project Explorer** view, expand the **rest-springboot** project, double-click the **pom.xml** file, and switch to the **pom.xml** tab.

Complete the **dependencyManagement** element to include the **org.jboss.redhat-fuse:fuse-springboot-bom** dependency:

```
...  
    <dependencyManagement>  
        <dependencies>  
            <dependency>  
                <groupId>org.jboss.redhat-fuse</groupId>  
                <!-- TODO add the correct BOM -->  
                <artifactId>fuse-springboot-bom</artifactId>  
                <version>${jboss.fuse.bom.version}</version>  
                <type>pom</type>  
                <scope>import</scope>  
            </dependency>  
        </dependencies>  
    </dependencyManagement>  
...
```

Save your changes to the project's POM file and wait until JBoss Developer Studio rebuilds the project.

This change to the **pom.xml** file fixes all of the errors in the project. In a few moments all error markers are gone.

Do not close the project's POM file because you will make further changes to it.

- 2.2. Add the Spring Boot Starter for the Camel Servlet.

At the end of the **dependencies** element, add the following:

```
...  
    <!-- TODO: add the camel-servlet-starter -->  
    <dependency>  
        <groupId>org.apache.camel</groupId>  
        <artifactId>camel-servlet-starter</artifactId>  
    </dependency>  
</dependencies>  
...
```

- 2.3. Add the Spring Boot Maven plug-in to the project's POM.

Inside the **plugins** element, add the **org.springframework.boot:spring-boot-maven-plugin** dependency and configure the plug-in to run as part of the **repackage** goal:

```
...  
    <build>  
        <plugins>  
            <!-- TODO: add the spring-boot-maven-plugin -->  
            <plugin>  
                <groupId>org.springframework.boot</groupId>  
                <artifactId>spring-boot-maven-plugin</artifactId>  
                <version>${spring-boot.version}</version>  
            </plugin>  
        </plugins>  
    </build>
```

```

<executions>
  <execution>
    <goals>
      <goal>repackage</goal>
    </goals>
  </execution>
</executions>
</plugin>
</plugins>
</build>
...

```

Save your changes to the project's POM file.

2.4. Update the project configuration.

JBoss Developer Studio adds a new error marker because of the last change to the POM, which changed the build plug-ins. To remove this error marker, in the **Project Explorer** view, right-click the **rest-springboot** project and click **Maven** → **Update Project**. Click **OK** and wait while the IDE rebuilds the project.

▶ 3. Inspect the Camel routes in the starter project.

3.1. Inspect the XML route in the starter project.

In the **Project Explorer** view, expand the **rest-springboot** project, then open the **src/main/resources/spring** folder. Double-click the **camel-context.xml** file. Switch to the **Source** tab.

The file defines a single XML route, that logs a message every second:

```

...
<route id="log-route">
  <from id="message-timer" uri="timer:foo?period=1s"/>
  <setBody id="set-message">
    <simple>Hello from Camel!</simple>
  </setBody>
  <log id="log-message" message=">>> ${body} : ${id}" />
</route>
...

```

3.2. Inspect the Java route in the starter project.

In the **Project Explorer** view, expand **src/main/java** → **com.redhat.training.jb421**. Double-click the **RestRouteBuilder.java** file.

The **RestRouteBuilder** class uses the Camel REST DSL to define the **/hello/{name}** resource URI and concatenates strings to produce a JSON response:

```

...
rest("/hello").get("{name}").produces("application/json").to("direct:sayHello");

from("direct:sayHello").routeId("HelloREST")
  .setBody().simple("{\n"
    + "  greeting: Hello, ${header.name}\n"

```

```
+ "    server: " + System.getenv("HOSTNAME") + "\n" + "}\n");  
...
```

Do not close the **RestRouteBuilder.java** file because you will make changes to it.

► 4. Complete the Spring Boot application.

- 4.1. Annotate the **RestRouteBuilder** class as a Spring managed bean.

Add the **org.springframework.stereotype.Component** annotation to the **RestRouteBuilder** class:

```
...  
import org.springframework.stereotype.Component;  
  
//TODO: annotate the RouteBuilder as a component  
@Component  
public class RestRouteBuilder extends RouteBuilder {  
...
```

Save your changes to the **RestRouteBuilder.java** file.

- 4.2. Annotate the Spring Boot application class to initialize XML routes.

In the **Project Explorer** view, expand **src/main/java** → **com.redhat.training.jb421**. Double-click the **Application.java** file.

Add the **org.springframework.context.annotation.ImportResource** to the **Application** class. Pass an array as an argument with one string: "**classpath:spring/camel-context.xml**".

```
...  
import org.springframework.context.annotation.ImportResource;  
  
@SpringBootApplication  
//TODO: add an annotation to process the spring/camel-context.xml file from the  
//classpath  
@ImportResource({"classpath:spring/camel-context.xml"})  
public class Application {  
...
```

Save your changes to the **Application.java** file.

► 5. Run the Spring Boot application and verify that all routes are active.

- 5.1. Open a terminal, enter the project folder, and invoke the **package** Maven goal to build the Spring Boot application's Fat JAR:

```
[student@workstation ~]$ cd ~/JB421/labs/rest-springboot  
[student@workstation rest-springboot]$ mvn package  
...  
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ rest-springboot ---  
[INFO] Building jar: /home/student/JB421/labs/rest-springboot/target/rest-  
springboot-1.0.jar  
...
```

Chapter 7 | Deploying Camel Routes

```
[INFO] --- spring-boot-maven-plugin:1.5.12.RELEASE:repackage (default) @ rest-springboot ---  
[INFO] -----  
[INFO] BUILD SUCCESS  
...
```

- 5.2. Verify that a Fat JAR exists in the **target** folder:

```
[student@workstation rest-springboot]$ ls -sh target/*jar*  
22M target/rest-springboot-1.0.jar 8.0K target/rest-springboot-1.0.jar.original
```

- 5.3. Run the Spring Boot application using the **java -jar** command. Leave the application running, and notice the log messages from XML route:

```
[student@workstation rest-springboot]$ java -jar target/rest-springboot-1.0.jar  
...  
10:56:19.319 [main] INFO o.s.b.w.s.ServletRegistrationBean - Mapping servlet: 'CamelServlet' to [/camel/*]  
...  
10:56:21.225 [main] INFO o.a.camel.spring.SpringCamelContext - Route: HelloREST started and consuming from: direct://sayHello  
10:56:21.229 [main] INFO o.a.camel.spring.SpringCamelContext - Route: route1 started and consuming from: servlet:/hello/%7Bname%7D?httpMethodRestrict=GET  
10:56:21.230 [main] INFO o.a.camel.spring.SpringCamelContext - Route: log-route started and consuming from: timer://foo?period=1s  
10:56:21.231 [main] INFO o.a.camel.spring.SpringCamelContext - Total 3 routes, of which 3 are started  
...  
10:56:21.315 [main] INFO o.s.b.c.e.u.UndertowEmbeddedServletContainer - Undertow started on port(s) 8080 (http)  
10:56:21.318 [main] INFO c.redhat.training.jb421.Application - Started Application in 5.253 seconds (JVM running for 5.766)  
10:56:22.241 [Camel (HelloREST) thread #1 - timer://foo] INFO log-route - >>> Hello from Camel! : ID-workstation-lab-example-com-1533772579406-0-2  
10:56:23.232 [Camel (HelloREST) thread #1 - timer://foo] INFO log-route - >>> Hello from Camel! : ID-workstation-lab-example-com-1533772579406-0-4  
...
```

- 5.4. Open another terminal and verify that the Spring Boot application responds to HTTP requests.

Invoke the **curl** command, using **localhost** as the host name with port 8080 and the **/camel/hello/Developer** resource URI:

```
[student@workstation rest-springboot]$ curl -si \  
http://localhost:8080/camel/hello/Developer  
HTTP/1.1 200 OK  
...  
{  
    greeting: Hello, Developer  
    server: workstation.lab.example.com  
}
```

- 5.5. Press **Ctrl+C** to stop the Spring Boot application.

► **6.** Close the project.

In the **Project Explorer** view, right-click **rest-springboot** and click **Close Project**.

This concludes the guided exercise.

▶ Lab

Deploying Camel Routes

Performance Checklist

In this lab, you will execute a REST service implemented using the Camel REST DSL and Spring Boot.

The REST service provides a single endpoint that retrieves vendor data from the bookstore database. The data resides inside a MySQL database server outside the OpenShift cluster.

The REST endpoint takes a single argument, the vendor ID, and returns JSON data. The resource URI is `/camel/vendor/{id}`.

Outcomes

You should be able to:

- Complete the project's POM file to dependencies required by Camel and Spring Boot.
- Package the application as a Fat JAR using the Spring Boot Maven plug-in.
- Test the application running standalone using the `curl` command.

Before You Begin

A skeleton project is provided as a starter, including all the code required to retrieve vendor data from the database, using the Spring Data JPA module. This code is encapsulated as the **VendorDAO** Java Bean invoked by a Camel Java route. The data source configuration is part of the Spring Boot's **application.properties** file.

The Camel Java route is also ready to run, as part of the **RestRouteBuilder** class. There are no other Camel routes in this application.

This application makes no changes to the database, which is already initialized in the classroom. If you need to initialize the database again, because it was cleaned by a previous exercise, run the **setup-data.sh** script inside the starter project.

Run the following command to download the starter project used by this lab:

```
[student@workstation ~] lab rest-database setup
```

1. Inspect the provided starter project.
Examine the Spring Boot configuration file **src/main/resources/application.properties** to see the data source configuration.
Examine the route builder **com.redhat.training.jb421.RestRouteBuilder** and the single Java route that defines a REST endpoint.
2. Update the project **pom.xml** file with the necessary Spring Boot starters:
 - **org.apache.camel:camel-servlet-starter**

- **org.apache.camel:camel-jackson-starter**
 - **org.springframework.boot:spring-boot-starter-data-jpa**
3. Package the Spring Boot application as a Fat JAR.
 4. Execute and test the application using the following URL:
http://localhost:8080/camel/vendor/1
 5. Grade your work. Execute the following command:
- ```
[student@workstation ~] lab rest-database grade
```
6. Clean up.
    - 6.1. Stop the **java** command where the Spring Boot application is running using **Ctrl+C**.
    - 6.2. In the **Project Explorer** view, right-click **rest-database** and click **Close Project**.

This concludes the lab.

## ► Solution

---

# Deploying Camel Routes

### Performance Checklist

In this lab, you will execute a REST service implemented using the Camel REST DSL and Spring Boot.

The REST service provides a single endpoint that retrieves vendor data from the bookstore database. The data resides inside a MySQL database server outside the OpenShift cluster.

The REST endpoint takes a single argument, the vendor ID, and returns JSON data. The resource URI is `/camel/vendor/{id}`.

### Outcomes

You should be able to:

- Complete the project's POM file to dependencies required by Camel and Spring Boot.
- Package the application as a Fat JAR using the Spring Boot Maven plug-in.
- Test the application running standalone using the `curl` command.

### Before You Begin

A skeleton project is provided as a starter, including all the code required to retrieve vendor data from the database, using the Spring Data JPA module. This code is encapsulated as the **VendorDAO** Java Bean invoked by a Camel Java route. The data source configuration is part of the Spring Boot's **application.properties** file.

The Camel Java route is also ready to run, as part of the **RestRouteBuilder** class. There are no other Camel routes in this application.

This application makes no changes to the database, which is already initialized in the classroom. If you need to initialize the database again, because it was cleaned by a previous exercise, run the **setup-data.sh** script inside the starter project.

Run the following command to download the starter project used by this lab:

```
[student@workstation ~] lab rest-database setup
```

1. Inspect the provided starter project.  
Examine the Spring Boot configuration file **src/main/resources/application.properties** to see the data source configuration.  
Examine the route builder **com.redhat.training.jb421.RestRouteBuilder** and the single Java route that defines a REST endpoint.
2. Update the project **pom.xml** file with the necessary Spring Boot starters:
  - **org.apache.camel:camel-servlet-starter**

- **org.apache.camel:camel-jackson-starter**
- **org.springframework.boot:spring-boot-starter-data-jpa**

Open the **pom.xml** and add the following dependencies:

```
<dependency>
 <groupId>org.apache.camel</groupId>
 <artifactId>camel-servlet-starter</artifactId>
</dependency>
<dependency>
 <groupId>org.apache.camel</groupId>
 <artifactId>camel-jackson-starter</artifactId>
</dependency>
<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
```

**3.** Package the Spring Boot application as a Fat JAR.

Enter the Maven project folder and use the **mvn** command and the **package** goal:

```
[student@workstation ~] cd ~/JB421/labs/rest-database
[student@workstation rest-database] mvn clean package
...
[INFO] BUILD SUCCESS
...
```

**4.** Execute and test the application using the following URL:

**http://localhost:8080/camel/vendor/1**

Use the **java -jar** command and wait until the Spring Boot application is fully started. Leave the **java** command running:

```
[student@workstation rest-database] java -jar target/rest-database-1.0.jar
...
10:49:59.004 [main] INFO c.redhat.training.jb421.Application - Started
Application in 7.351 seconds (JVM running for 7.847)
```

Open another terminal and use the **curl** command:

```
[student@workstation ~] curl -si http://localhost:8080/camel/vendor/1
HTTP/1.1 200 OK
...
{"id":1,"name":"Bookmart, Inc."}
```

**5.** Grade your work. Execute the following command:

```
[student@workstation ~] lab rest-database grade
```

**6.** Clean up.

6.1. Stop the **java** command where the Spring Boot application is running using **Ctrl+C**.

6.2. In the **Project Explorer** view, right-click **rest-database** and click **Close Project**.

This concludes the lab.

# Summary

---

In this chapter, you learned:

- Each Red Hat Fuse 7 supported runtime requires a different application package:
  - Apache Karaf requires OSGi bundles.
  - JBoss EAP requires JAR, WAR, or EAR archives.
  - Spring Boot requires executable JAR archives.
- For each Red Hat Fuse runtime, there is a bill of materials (BOM) that lists supported and tested dependencies.
- To package an application as an OSGi bundle, use the Apache Felix Maven plug-in.
- To deploy an OSGi bundle on Apache Karaf, use the Karaf management console.
- Red Hat recommends that applications targeting Apache Karaf use the XML configuration file from the Blueprint framework instead of the one from the Spring framework.
- Camel applications that target JBoss EAP can use XML routes without requiring Spring framework dependencies.
- Red Hat recommends that Camel applications that target Spring Boot use the starters from the Red Hat Fuse product.
- Sprint Boot provides a Maven plug-in that packages applications as an executable, self-contained, Fat JAR.



## Chapter 8

# Implementing Transactions

### Goal

Provide data integrity in route processing by implementing transactions.

### Objectives

- Implement transaction management in routes using the Spring Transaction Manager.
- Develop tests for transactional routes.

### Sections

- Developing Transactional Routes with Fuse on EAP (and Guided Exercise)
- Testing Transactional Routes (and Guided Exercise)

### Lab

Lab: Implementing Transactions

# Developing Transactional Routes with Fuse on EAP

## Objective

After completing this section, students should be able to implement transaction management in Camel routes using the Spring Transaction Manager.

## Developing Transactional Routes

When developing an integration system that accesses multiple systems with a route, problems such as data integrity inconsistencies or processing failures create business inconsistencies and unexpected results. For example, a route that processes bank transfers and withdrawals must require that each message is able to complete each endpoint without data loss. Using transactions during route processing alleviates many common integration issues by reverting either the withdrawal or the deposit operation and avoiding money loss from the financial institution.

In an integration system, a *transaction manager* restores the processing state immediately following a failure. This allows a *two-phase commit* (2PC) approach where each system is part of a transaction that is committed when the entire route execution is complete and all the external systems are ready to process the transaction. The transaction that spans over all the systems is called a *global transaction*.

Camel supports transaction management either inside or outside of a container, which gives enough flexibility to develop robust transactional applications.

## Supporting Transaction Management with Camel

When developing a Camel route with transactions, each component must individually be configured to support transactions and they must all use the same transaction manager. This allows the transaction manager to manage a single transaction throughout a route execution that uses multiple components.

To enable transactions in your Camel route, configure it with a Spring-managed transaction class (**org.springframework.transaction.jta.JtaTransactionManager**). This class is used by all transaction-supported components to start and execute the process using the same transaction.



### Note

Refer to the Camel component documentation to determine whether a component supports transactions.



### Note

Spring supports several transaction managers, such as Atomikos, JOTM, Narayana, and most of Java EE container transaction managers.

Traditional databases and message-oriented middleware solutions support transaction managers and provide a reliable environment to allow rollback transactions if problems occur during a

process. Camel combines the processes executed on all services that supports transactions into a single transaction, and uses the transaction manager to perform the commit operation. If the route raises any runtime exceptions, the transaction manager rolls back the operation.

## Transaction Propagation Policies

Occasionally, transactions are more complex than a typical single write to a database. For example, a database may need to create nested transactions, such as managing a money transfer between two bank accounts. In this scenario, the application must:

- Update the customer's account balance to reflect the new value.
- Add a record to the customer's checking account history that money was transferred to a destination account to another customer's account.
- Add a record to the final account destination checking account history that the money was deposited.
- Update the destination account balance to reflect the new balance.

In this case, there are two transactions: the one that updates both the origin and destination customer balances, and a transaction that records the checking account history. The second transaction is nested in the balance update transaction because that transaction cannot occur until the first transaction completes. In this case, you need to define a custom *transaction propagation policy* to allow this nested transaction and to propagate the existing transaction from the account balance updates to the history records.

Camel supports pre-defined and custom transaction propagation policies. These propagation policies are defined in the Spring XML configuration file:

### **PROPAGATION\_REQUIRED**

All Camel processors from a route must use the same transaction.

### **PROPAGATIONQUIRES\_NEW**

Each processor creates its own transaction.

### **PROPAGATIONNOT\_SUPPORTED**

Does not support a current transaction.

Camel supports these propagation policies using the Spring **org.apache.camel.spring.spi.SpringTransactionPolicy** class. A subclass must be created where the transaction manager mentioned later in the Transaction Propagation Policies must be injected.

## Customizing the camel-jpa Component to Support Transactions

Camel supports the **camel-jpa** component to consume or store information in a database using Java Persistence API. To configure **camel-jpa** to support transactions, you can use the Java API or Camel configuration file. In a configuration file, the transaction manager based on **JtaTransactionManager** class must be injected to the component:

```
<bean class="org.apache.camel.component.jpa.JpaComponent" id="jpa">
 <property name="entityManagerFactory" ref="emf"/>
 <property name="transactionManager" ref="txMgr"/>
</bean>
```

Alternatively, the same configuration can be obtained using Java in the **RouteBuilder** when creating a route:

```
JpaComponent jpaComponent = new JpaComponent();
jpaComponent.setEntityManagerFactory(entityManager.getEntityManagerFactory());
jpaComponent.setTransactionManager(transactionManager);
getContext().addComponent("jpa", jpaComponent);
```

When an exchange enters a transacted route, the transacted processor invokes the default transaction manager to *begin* a transaction (attaching it to the current thread), and when the exchange reaches the end of the route, the transacted processor invokes the transaction manager to *commit* the current transaction.

## Customizing the camel-jms Component to Support Transactions

Camel supports the camel-jms component to consume or store information in a message broker using the Java Messaging Service. To configure **camel-jms** to support transactions, you can use the Java API or Camel configuration file. In a configuration file, the transaction manager based on **JtaTransactionManager** class must be injected to the component:

```
<bean id="jmsConnectionFactory"
 class="org.apache.activemq.ActiveMQConnectionFactory">
 <property name="brokerURL" value="tcp://localhost:61616"/> ①
</bean>

<bean id="transactionManager"
 class="org.springframework.jms.connection.JmsTransactionManager"> ②
 <property name="connectionFactory" ref="jmsConnectionFactory"/>
</bean>

<bean name="activemq" class="org.apache.camel.component.jms.JmsComponent"> ③
 <property name="connectionFactory" ref="jmsConnectionFactory"/>
 <property name="transactionManager" ref="transactionManager"/>
 <property name="transacted" value="true"/> ④
</bean>
```

- ① Connect to the ActiveMQ broker.
- ② Enable the Spring transaction manager.
- ③ Inject the connection factory and transaction manager into **camel-jms**.
- ④ Enable local transactions (not XA).

## Implementing Transactions with the Camel transacted DSL method

In a route accessing multiple transaction-enabled components, any error requires a rollback operation from the route execution. To enable transaction management in a route, it must have a **transacted** DSL method:

```
from("direct:split")
.transacted()
...
```

## Managing Transaction Rollback with onException

If an exception is thrown while a route is executed, the transaction may not be rolled back. According to transaction management rules from the Java Transaction API (JTA), **RuntimeException** exceptions are automatically rolled back by a route processing operation. However, other exceptions are not subject to rollback. To roll back a transaction, it must be marked as **markRollbackOnly** as part of the exception management.

In the following snippet, if a **ConnectionException** is raised, then it must mark the exception as handled and mark the rollback

```
onException(ConnectionException.class)
 .handled(true)
 .to("jms:queue:DeadLetter")
 .markRollbackOnly();

from("jpa:com.redhat.training.Order")
 .transacted()
 ...
 .to("jms:queue:small");
```

## Customizing Transaction Management Policies

You can also create a transaction management policy other than the three default ones mentioned in Transaction Propagation Policies. For example, some transaction managers support the **PROPAGATION\_MANDATORY**, **PROPAGATION\_NOT\_SUPPORTED**, **PROPAGATION\_SUPPORTS**, **PROPAGATION\_NEVER**, and **PROPAGATION\_NESTED** transaction management policies. To enable them in Camel, create either:

- Multiple classes that extend the **SpringTransactionPolicy** class and name each one with the **@Named** annotation and refer to the CDI component name using the **transacted** DSL method.

```
@Named("myTransactionPolicy")
public class JmsPropagationRequiredPolicy extends SpringTransactionPolicy {
 @Inject
 public JmsPropagationRequiredPolicy(JmsTransactionManager cdiTransactionManager) {
 super(new TransactionTemplate(cdiTransactionManager,
 new
 DefaultTransactionDefinition(TransactionDefinition.PROPAGATION_NESTED)));
 }
}
```

In the route, the **transacted** method call must have the same argument **String** defined in the **@Named** argument.

```
.transacted("myTransactionPolicy")
```

- Multiple bean configurations that use the **SpringTransactionPolicy** class and use the transaction policy CDI component name on the **transacted** DSL method.

```
<bean class="org.apache.camel.spring.spi.SpringTransactionPolicy"
 id="myTransactionPolicy">
 <property name="transactionManager" ref="txMgr"/>
 <property name="propagationBehaviorName" value="myTransactionPolicy"/>
</bean>
```

Similar to the first example, the String defined in the **id** attribute from the bean must be used.

```
.transacted("myTransactionPolicy")
```

## Configuring Routes to Support Distributed Transactions in Camel

A database or a message queue can manage multiple instructions, such as data creation or deletion as a single unit of work. If something goes wrong with one of the commands, the whole operation is rolled back. This is managed by a local transaction manager embedded in the message queue or database runtime environment. Transactions that span through multiple services, however, are harder to manage because any operation must be coordinated by multiple resources, such as databases. If an error is raised during processing, the whole operation must be rolled back on each resource to avoid inconsistencies.

Furthermore, the orchestration must be configured to identify problems on each system participating in the transaction (such as system unavailability) and automatically alert the other systems participating in the transaction to undo their work. A transaction manager guarantees these capabilities in a distributed transaction.

In Camel, a transaction manager must be available to manage a distributed transaction. This can be accomplished either using a Spring XML configuration file or Java code.

When using Camel outside of a container, the following Spring XML configuration file can be used to configure the transaction manager:

```
<bean class="org.springframework.transaction.jta.JtaTransactionManager"
 id="txMgr">
 <property name="allowCustomIsolationLevels" value="true"/>
 <property name="transactionManager">
 <bean class="TransactionManagerClass" />
 </property>
 <property name="userTransaction">
 <bean class="UserTransactionManagerClass" />
 </property>
</bean>
```

Each transaction manager has its own **TransactionManager** and **UserTransactionManager** implementations.

For CDI-enabled environments, such as EAP, the following Java code uses the container transaction manager to create the **JtaTransactionManager** instance.

```
@Named("transactionManager")
public class JmsTransactionManager extends JtaTransactionManager {
```

```
@Resource(mappedName = "java:/TransactionManager")①
private TransactionManager transactionManager;

@Resource(mappedName="java:jboss/UserTransaction")②
private UserTransaction userTransaction;

@PostConstruct
public void initTransactionManager() {
 setTransactionManager(transactionManager);
 setUserTransaction(userTransaction);
}
}
```

- ① Injects the transaction manager instance provided by an application server such as EAP.
- ② Injects the user transaction initialized by the application server to manage a distributed transaction.



## References

### Transactional clients

<http://camel.apache.org/transactional-client.html>

*Camel in Action, Second Edition* - Chapter Enterprise Integration Patterns

*Camel in Action, Second Edition* - Chapter Transactions

## ► Guided Exercise

# Enhancing a Route with Transactions

In this exercise, you will add functionality to a route to support transactions and then deploy that route on EAP and test it.

## Outcomes

You should be able to support transactional routes in Camel running on EAP.

## Before You Begin

A skeleton project is provided as a starter project. The starter project includes a working Camel route you can improve to support transactions.

Run the following command to download the starter project and data files used by this exercise:

```
[student@workstation ~]$ lab enhancing-transaction setup
```

► 1. Open JBoss Developer Studio and import the starter Maven project.

- 1.1. Open JBoss Developer Studio (**Applications** → **Programming** → **JBoss Developer Studio**). JBoss Developer Studio requests a default workspace. Leave the default configuration (**/home/student/workspace**) and click **Launch**.
- 1.2. Import the Maven project by selecting the JBoss Developer Studio main menu **File** → **Import**.
- 1.3. From the **Import** dialog box, select **Maven** → **Existing Maven Projects** and click **Next >**.
- 1.4. Click **Browse** and choose **/home/student/JB421/labs/enhancing-transaction** and click **Ok** to choose the Maven project. Click **Finish** to complete the import process in JBoss Developer Studio.

A new project named **enhancing-transaction** will be listed in the **Project Explorer** view.

► 2. Inspect the starter project.

- 2.1. Inspect the **JmsRouteBuilder** class.

In the **Project Explorer** view, expand **enhancing-transaction** → **src/main/java** → **com.redhat.training.camel.routes**. Double-click **JmsRouteBuilder.java** file.

The file has two routes:

- The first route reads files from the **/home/student/JB421/labs/enhancing-transaction/contact** folder to a database. It transforms the XML body to a Java object and writes it to the database using the JPA component.

- The second route reads contents from the database with a JPA component and sends it to a JMS queue named **ContactsQueue**. It deletes the database contents. It also evaluates the contents and if the email is empty, then an **IllegalStateException** exception is thrown.

The class is also a CDI component annotated with the **@ApplicationScoped** annotation.

- Evaluate the transaction manager managed by Spring.

In the **Project Explorer** view, expand **enhancing-transaction** → **src/main/java** → **com.redhat.training.camel.tx**. Double-click **ContactTransactionManager.java**.

To enable transactional behavior in Camel, you need to configure a transaction manager. This is implemented by the **com.redhat.training.camel.tx.ContactTransactionManager** class. The class is annotated with **@Named("transactionManager")**, which is its CDI component name. It may be injected on every CDI component.

### ▶ 3. Enable the transaction manager on each component.

Camel components need to use the same transaction manager to guarantee that the transaction started on a component is the same as the other component.

- Inject the transaction manager into the route.

In the already opened **JmsRouteBuilder** class, add after the comment **//TODO inject TransactionManager**:

```
@Inject
private ContactTransactionManager transactionManager;
```

- Configure the JMS component with the transaction manager.

The JMS component is configured by the route to identify the queues used by the route. Add to the component instantiation after the **//TODO configure JMS component to support JMS transactions** comment:

```
// TODO configure JMS component to support JMS transactions.
JmsComponent jmsComponent =
JmsComponent.jmsComponentTransacted(connectionFactory, transactionManager);
getContext().addComponent("jms", jmsComponent);
```

The source code is provided in the **jms-transacted.java.txt** file.

- Configure the **jpa** component with the transaction manager.

The jpa component is configured by the route to identify the queues used by the route. After the **// TODO configure JPA component to support transactions**. comment, update the component instantiation as follows:

```
JpaComponent jpaComponent = new JpaComponent();
jpaComponent.setEntityManagerFactory(entityManager.getEntityManagerFactory());
jpaComponent.setTransactionManager(transactionManager);
getContext().addComponent("jpa", jpaComponent);
```

The source code is provided in the **jpa-transact.java.txt** file.

► 4. Enable the route to support transactions.

In the second route, add the following source code after the **//TODO Configure as transacted** comment:

```
//TODO Configure as transacted and set the type to "PROPAGATION_REQUIRE_NEW"
.transacted("PROPAGATION_REQUIRE_NEW")
```

This configures the route as a transacted one, and tells Camel to leverage the transaction manager for the JPA component to start a new transaction for each exchange produced by the JPA component.

4.1. Start EAP to configure queues and deploy the application.

JBoss EAP with Fuse is available at [/opt/jboss-eap-7.1](#). Invoke its **bin/standalone.sh** script with the **-c standalone-full.xml** option. Open a new terminal window and run the following commands:

```
[student@workstation ~]$ cd /opt/jboss-eap-7.1/bin
[student@workstation bin]$./standalone.sh -c standalone-full.xml
```

Wait until a message similar to the following is outputted from EAP:

```
08:09:21,215 INFO [org.jboss.as] (Controller Boot Thread) WFLYSRV0025: JBoss EAP
7.1.0.GA (WildFly Core 3.0.10.Final-redhat-1) started in 6851ms - Started 506 of
734 services (374 services are lazy, passive or on-demand)
```

4.2. Evaluate the data from the files.

Use the **setup-data.sh** script inside the project to copy the files from the **~/JB421/data/contact-transact** directory to **~/JB421/labs/contact** directory. Open a new terminal window and run the following commands:

```
[student@workstation ~]$ cd ~/JB421/labs/enhancing-transaction
[student@workstation enhancing-transaction]$./setup-data.sh
```

Select the project in JBoss Developer Studio, right-click the project and choose **Refresh**. Open the **contact** folder and open each XML file. There are three files, but one of the files fail (**contact-2.xml**). The file does not have an email, which is a mandatory field.

4.3. Count the number of messages sent for each queue.

Run the **monitor-dead.sh**, **monitor-queue.sh**, and **monitor-database.sh** scripts and take note of the number of contacts in each of them. In the same terminal window, run the following commands:

```
[student@workstation enhancing-transaction]$./monitor-dead.sh
[student@workstation enhancing-transaction]$./monitor-queue.sh
[student@workstation enhancing-transaction]$./monitor-database.sh
```

All scripts must return zero because the queues and the database are empty.

4.4. Deploy the application to EAP.

Use the Maven **wildfly:deploy** target and the **-DskipTests** option. In the same terminal window as the previous step, run the following command:

```
[student@workstation enhancing-transaction]$ mvn wildfly:deploy -DskipTests
```

The command should return a **build successful** message.

- 4.5. Verify that one of the contacts was not processed because it did not have the required email. The JBoss EAP log contains an **Invalid email** message:

```
09:42:34,115 INFO [route2] (Camel (camel-1) thread #1 - jpa://com.redhat.training.camel.model.Contact) Invalid email - rolling back transaction!
```

- 4.6. Evaluate the contents from the queue if the message returned to the queue.

In the same terminal window as the previous steps, run the **monitor-dead.sh**, **monitor-queue.sh**, and **monitor-database.sh** scripts and take note of the number of orders in each of them. The database is not empty because the contact was rolled back. But the **ContactsQueue** has one message.

```
[student@workstation enhancing-transaction]$./monitor-queue.sh
"message-count" => 1L,
"messages-added" => 1L,
[student@workstation enhancing-transaction]$./monitor-database.sh
+-----+
| COUNT(*) |
+-----+
| 2 |
+-----+
```

The first contact record was successfully processed by the route and has been delivered to **ContactQueue**. The second contact record, which is missing the **email** field, triggers an **IllegalStateException** to occur during the route processing, and causes the JPA transaction to rollback. This means that given the current route definition, the rollback actually causes our system to constantly retrieve, process and rollback the second contact record without ever reaching the third.

To improve this route, add an **onException** definition to handle the **IllegalStateException** and store the problem record in the **DeadLetter** queue.

- 4.7. Undeploy the application from EAP.

In the terminal window already opened, run the following commands:

```
[student@workstation ~]$ cd JB421/labs/enhancing-transaction
[student@workstation enhancing-transaction]$ mvn wildfly:undeploy
```

- 5. Update the route to handle the **IllegalStateException** and use the **DeadLetter** queue to store records that trigger the exception.

- 5.1. Return to editing the **JmsRouteBuilder** class and update the route definition to include the **onException** definition after the **//TODO add onException to handle IllegalStateException ...** comment:

```
//TODO add onException to handle IllegalStateException and send the problem record
to the DeadLetter queue
onException(IllegalStateException.class)
.handled(true)
.to("jms:queue:DeadLetter");
```

5.2. Press **Ctrl+S** to save your changes to the **JmsRouteBuilder** class.

► 6. Retest the route with the **onException** definition included.

6.1. Use the **setup-data.sh** script inside the project to re-copy the files from the **~/JB421/data/contact-transact** directory to **~/JB421/labs/contact** directory. Open a new terminal window and run the following commands:

```
[student@workstation ~]$ cd ~/JB421/labs/enhancing-transaction
[student@workstation enhancing-transaction]$./setup-data.sh
```

6.2. Count the number of messages sent for each queue and in the database.

Run the **monitor-dead.sh**, **monitor-queue.sh**, and **monitor-database.sh** scripts and take note of the number of contacts in each of them. In the same terminal window, run the following commands:

```
[student@workstation enhancing-transaction]$./monitor-dead.sh
"message-count" => 0L,
"messages-added" => 0L,
[student@workstation enhancing-transaction]$./monitor-queue.sh
"message-count" => 1L,
"messages-added" => 1L,
[student@workstation enhancing-transaction]$./monitor-database.sh
+-----+
| COUNT(*) |
+-----+
| 0 |
+-----+
```

The database is emptied by the **setup-data.sh** script so the data from the previous test is no longer stored there. The **DeadLetter** queue is empty and the **ContactQueue** contains a single record from the previous test.

6.3. Deploy the application to EAP.

Use the Maven **wildfly:deploy** target and the **-DskipTests** option. In the same terminal window as the previous step, run the following command:

```
[student@workstation enhancing-transaction]$ mvn wildfly:deploy -DskipTests
```

The command should return a **build successful** message.

The route should immediately process the files and this time successfully process the third contact record.

6.4. Verify the number of messages sent for each queue and in the database.

Run the **monitor-dead.sh**, **monitor-queue.sh**, and **monitor-database.sh** scripts and take note of the number of contacts in each of them. In the same terminal window, run the following commands:

```
[student@workstation enhancing-transaction]$./monitor-dead.sh
 "message-count" => 1L,
 "messages-added" => 1L,
[student@workstation enhancing-transaction]$./monitor-queue.sh
 "message-count" => 3L,
 "messages-added" => 3L,
[student@workstation enhancing-transaction]$./monitor-database.sh
+-----+
| COUNT(*) |
+-----+
| 0 |
+-----+
```

This time the **DeadLetter** queue contains the second record that is missing the **email** field. Notice this time the record was not rolled back to the database, because we marked the exception as handled in the **onException** definition. The database is therefore empty as all records have been successfully processed by the route or stored in the **DeadLetter** queue for manual intervention and reprocessing at a later time.

► 7. Clean up.

- 7.1. Close the project in JBoss Developer Studio to conserve memory.

In the **Project Explorer** view, right-click enhancing-transaction project and click **Close Project**.

- 7.2. Undeploy the application from EAP.

In the terminal window already opened, run the following commands:

```
[student@workstation ~]$ cd JB421/labs/enhancing-transaction
[student@workstation enhancing-transaction]$ mvn wildfly:undeploy
```

Wait until a **build success** message is outputted.

- 7.3. Remove all the queues created during this lab.

In the same terminal window, run the following commands:

```
[student@workstation enhancing-transaction]$./clear-environment.sh
```

- 7.4. Stop JBoss EAP by pressing **Ctrl+C** on the terminal window running EAP.

This concludes the guided exercise.

# Testing Transactional Routes

## Objectives

After completing this section, students should be able to:

- Develop tests for transactional routes.
- List the complexities of testing transacted routes.
- Implement tests to verify transacted routes with JUnit.

## Testing Transacted Routes

Transacted routes can be challenging to test because external systems, such as a database, message queues, or transaction managers, must be available while running integration tests. Further, test data must be processed by the route to check the integrity across multiple services. Finally, emulating services such as database can not always be easily accomplished by mock frameworks or stubs.

Camel supports a set of mechanisms to emulate these requirements, such as the Camel Test Kit (CTK) to mock external systems, CDI and Spring integration to run integration tests with live systems, or Spring's extension capabilities to plug external technologies to substitute heavyweight services with lighter ones.

## Developing JUnit Tests With Transaction Support

Developing a JUnit test with transaction support requires some additional configuration to activate transactional capabilities. With JUnit and Spring test component support simplifies the configuration needed to support transaction.

Heterogeneous systems such as a database and a message queue might also be part of the transaction. Under these circumstances, working with a Java EE container makes transaction management simpler, but for testing purposes, a fully fledged container is mandatory to test the integration behavior.

For testing purposes, a transaction manager must be configured and enabled during the tests to create a runtime environment as accurate as possible to a real world environment. Some external transaction managers are available to be used by Camel during testing, such as Atomikos, JOTM, or other alternatives supported by Java.

All of the external systems used by the route must be available to the Camel route during test execution. Even though some behaviors can be emulated using mocks, the actual transaction behavior needs all the services and systems used by a Camel route to be running during the test execution.

Traditional services, such as databases and message queues, can be configured and enabled to the runtime environment using a lightweight version of the service, such as an H2 database or ActiveMQ, respectively. These resources must be started by the JUnit test during the test initialization process, but it can require lots of extra coding when you are not using the Camel test component.

To simplify the process, the Spring beans configuration file starts these resources in an easily configurable manner. These resources can be shared with other elements managed by Spring, such as JPA persistence units or connection factories from message queues. In the following Spring bean configuration file, a JPA implementation (Hibernate), a dummy database (H2), a message broker (ActiveMQ), and a transaction manager (Atomikos) are configured to support transactions:

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

 <tx:annotation-driven mode="proxy"
 proxy-target-class="false"
 transaction-manager="JtaTransactionManager"/>
 <bean class="com.atomikos.icatch.config.UserTransactionServiceImp"
 destroy-method="shutdownForce"
 id="userTransactionService"
 init-method="init"/> 1
 <bean class="com.atomikos.icatch.jta.UserTransactionManager"
 depends-on="userTransactionService"
 destroy-method="close"
 id="AtomikosTransactionManager"
 init-method="init"/> 2
 <bean class="com.atomikos.icatch.jta.UserTransactionImp"
 depends-on="userTransactionService"
 id="AtomikosUserTransaction"> 3
 <!-- transaction manager configuration -->
 </bean>
 <bean class="org.springframework.transaction.jta.JtaTransactionManager"
 depends-on="userTransactionService"
 id="JtaTransactionManager"> 4
 <!-- jta configuration-->
 <property name="transactionManager"
 ref="AtomikosTransactionManager"/>
 <property name="userTransaction"
 ref="AtomikosUserTransaction"/>
 </bean>

 <bean class="org.apache.activemq.ActiveMQXAConnectionFactory"
 depends-on="amqBroker" id="xaFactory"> 5
 <!-- ActiveMQ broker information-->
 </bean>
 <bean class="com.atomikos.jms.AtomikosConnectionFactoryBean"
 depends-on="amqBroker"
 destroy-method="close"
 id="atomikosJmsConnectionFactory"
 init-method="init"> 6
 <property name="uniqueResourceName"
 value="ActiveMQXA"/>
 <property name="xaConnectionFactory"
```

```

 ref="xaFactory"/>
 <!-- customize ActiveMQ/Atomikos config -->
</bean>
<bean
 class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean"
 depends-on="JtaTransactionManager"
 id="emf">⑦
 <property name="jtaDataSource"
 ref="atomikosDataSource"/>
 <!-- JPA-specific configuration -->
</bean>
<bean class="com.atomikos.jdbc.AtomikosDataSourceBean"
 destroy-method="close"
 id="atomikosDataSource"
 init-method="init">⑧
 <property name="uniqueResourceName"
 value="AtomikosDataSource"/>
 <property name="xaDataSource"
 ref="xaReferent"/>
</bean>
<bean class="org.h2.jdbcx.JdbcDataSource"
 id="xaReferent">⑨
 <!--H2-specific configuration -->
</bean>
<!--Camel route definitions-->
</beans>
```

- ①** Configures Atomikos transaction manager service to start with Spring.
- ②** Accesses Atomikos transaction manager as a JTA-compliant transaction manager.
- ③** Access Atomikos user transaction as a JTA-compliant user transaction.
- ④** Configures Spring transaction manager API to user Atomikos transaction manager.
- ⑤** Accesses ActiveMQ connection factory using XA connections.
- ⑥** Configures ActiveMQ connection manager to be accessed using Atomikos transaction manager.
- ⑦** Configures JPA implementation to use Atomikos transaction manager.
- ⑧** Configures Atomikos to provide database connections that are managed by itself.
- ⑨** Accesses H2 database to use the JDBC driver compliant with XA.

To test whether a transaction fails during a route execution, the Camel Test Kit supports the capability to throw various route exceptions and evaluate the results of the error as well as the success or failure of any transaction rollbacks.

```

@Test
public void testRedelivery() throws Exception {
 context.getRouteDefinition("route-one").adviceWith(①(context, new
RouteBuilder() {
 public void configure() {
 interceptSendToEndpoint("jpa:*")②
 .throwException(new SQLException("Cannot connect to the database"));③
 }
});
 context.start();
```

```

 ...
 //Include assertions
}
```

- ➊ Updates the **route-one** route.
- ➋ Intercepts any message sent to any JPA endpoint.
- ➌ Throws an **SQLException** exception whenever a message is sent to the database.

This exception rolls back the transaction in progress, because it is a runtime exception. To check if a rollback was successful, the resources must be queried for changes. They should not be affected by route execution.

Alternatively, invalid content can be sent using the testing facilities to generate a transaction error. For example, text may overflow the size of a field from a database, and this may raise an error during route processing. Anything from this route must be rolled back to guarantee integrity of the route execution.

## Demonstration: Exploring a Method of Testing Transactions

**Use Case:** A process was developed to detect orders with a discount over US\$ 100.00. Add some tests to ensure that the transactions are working properly to avoid any data inconsistencies for your retailers.

1. Start JBoss Developer Studio and import the **explore-transaction** project from the `/home/student/JB421/labs/` directory.
2. Evaluate how the route is customized to manage transactions.

Open the **com.redhat.training.camel.routes.JmsRouteBuilder** class. Check that the provided route reads data from a directory, transforms them to Java objects, and stores them in the database.

```

onException(IllegalStateException.class)
 .maximumRedeliveries(1)
 .handled(true)
 .to("jms:queue:DeadLetter")
 .markRollbackOnly();

from("file:/home/student/JB421/labs/explore-transaction/orders")
 .unmarshal(jaxbDataFormat)
 .to("jpa:com.redhat.training.model.Order");

from("jpa:com.redhat.training.model.Order")
 .transacted()
 .to("jms:queue:OrdersQueue")
 .choice()
 .when(simple("${body.discount} > 100"))
 .log("Order discount is greater than 100 - rolling back transaction!")
 .throwException(new IllegalStateException())
 .otherwise()
 .log("Order processed successfully");
```

The **camel-context.xml** file is used only for testing purposes to mimic the transaction manager from EAP. Also, the route does not refer to anything specific to EAP or Spring.

**Chapter 8 |** Implementing Transactions

The object is sent to a message queue. If something goes wrong, the object is stored back to the database due to a transaction rollback.

3. Evaluate the **testDelivery** test method.

Open the **com.redhat.training.camel.routes.JmsRouteTest** test case. There are two test methods: The **testDelivery** method checks if the normal processing was successful. The **testRedelivery** checks if an order with a discount above US\$ 100.00 was rolled back.

4. Uncomment the snippet from the **testDelivery** test method.

The test sends a file to the database and checks the ordinary processing. To check if the order was processed successfully, the method connects to a database to evaluate the number of orders.

```
int rows = jdbc.queryForObject("select count(1) from order_", Integer.class);
Assert.assertEquals(0, rows);
```

Comment the **Assert.fail** method call.

```
Assert.fail("Test not implemented");
```

5. Run the test case.

There are two test methods but only one is marked successful.

6. Uncomment the **configure** method snippet from the **testRedelivery** test method.

```
interceptSendToEndpoint("jms:*")
 .choice()
 .when(header("JMSRedelivered").isEqualTo("false"))
 .throwException(new ConnectException("Cannot connect"+
 "to the database"));
```

It raises a **ConnectException** exception when the message is received for the first time on the message queue.

7. Uncomment the remaining code snippet from the **testRedelivery** test method.

This block evaluates if the database has rolled back the transaction. It confirms that an order was not processed because it has a discount greater than US\$ 100.

```
int rows = jdbc.queryForObject("select count(1) from order_", Integer.class);
Assert.assertEquals(1, rows);
```

Comment the **Assert.fail** method call.

```
Assert.fail("Test not implemented");
```

8. Run the test case.

Check the output from the demo. Look for the following message: **Discount is greater than 100 - rolling back transaction!**.

The test method has a green check mark, and the test case passes.

9. Clean up the environment.

Close the project in JBoss Developer Studio.

This concludes this demonstration.



## References

### Transactional clients

<http://camel.apache.org/transactional-client.html>

*Camel in Action, Second Edition* - Chapter Testing

*Camel in Action, Second Edition* - Chapter Transactions

## ► Guided Exercise

# Testing a Transactional Route

In this exercise, you will test the functionality of a specific route to support transactions.

### Outcomes

You should be able to develop JUnit tests to verify transactional routes in Camel.

### Before You Begin

Run the following command to download the starter project and data files used by this exercise. Click **Applications** → **Favorites** → **Terminal** in the desktop menu and type:

```
[student@workstation ~]$ lab test-transaction setup
```

► 1. Import the starter Maven project.

- 1.1. Open JBoss Developer Studio (**Applications** → **Programming** → **JBoss Developer Studio**). JBoss Developer Studio requests a default workspace; keep the default configuration (**/home/student/workspace**) and click **Launch**.
- 1.2. Import the Maven project by selecting JBoss Developer Studio main menu **File** → **Import**.
- 1.3. From the **Import** dialog box, select **Maven** → **Existing Maven Projects** and click **Next >**.
- 1.4. Click **Browse** and choose **/home/student/JB421/labs/test-transaction**, and then click **OK** to choose the Maven project. Click **Finish** to complete the import process in JBoss Developer Studio.

A new project named **test-transaction** is listed in the **Project Explorer** view.

► 2. Inspect the starter project.

- 2.1. Inspect the **JmsRouteBuilder** class.

In the **Project Explorer** view, expand **test-transaction** → **src/main/java** → **com.redhat.training.camel.routes**. Double-click **JmsRouteBuilder.java**.

The route is identical to the previous guided exercise, but this time it needs an infrastructure to run an integration test to check that the transaction for the route is rolled back if an exception is raised.

- 2.2. Inspect the transaction manager configuration managed by Spring.

In the **Project Explorer** view, expand **test-transaction** → **src/test/resources** → **META-INF/spring**. Double-click **camel-context.xml**. Select the **Source** tab at the bottom of the editor.

To enable transactional behavior in Camel, you need to configure a transaction manager. This is implemented by Atomikos, a transaction manager that has the same behavior as the JBoss EAP transactional manager.

The **UserTransaction** and **TransactionManager** implementations are defined as two beans (whose IDs are **userTransactionService** and **AtomikosTransactionManager**, respectively), and they are managed by a Spring Transaction Manager bean identified as **JtaTransactionManager**.

```
<bean class="com.atomikos.icatch.config.UserTransactionServiceImp"
 destroy-method="shutdownForce" id="userTransactionService" init-
 method="init"/>
...
<bean class="com.atomikos.icatch.jta.UserTransactionImp"
 depends-on="userTransactionService" id="AtomikosUserTransaction">
 <property name="transactionTimeout" value="300"/>
</bean>
```

► **3.** Evaluate the test case.

In the **Project Explorer** view, expand **test-transaction** → **src/test/java** → **com.redhat.training.camel.routes**. Double-click **JmsRouteTest.java**.

There are two test methods: **testDelivery** and **testRedelivery**, but neither of them are implemented.

The **testDelivery** method sends a **Contact** object with an email to the route and detects whether the database is empty, which means the exchange was sent to the queue.

The **testRedelivery** method sends a **Contact** object without an email to the route and checks if the database is populated.

► **4.** Complete the **testDelivery** test method.

In the test method, an XML file with a **Contact** object is sent to the route and the database is queried to check the number of **Contact** objects stored. It must be zero because the second route sends the message to the message queue.

4.1. Send a valid XML file to the route.

Immediately after the **//TODO Send the VALID\_CONTACT as the body** comment, add the following code:

```
fileTemplate.sendBody(VALID_CONTACT);
```

4.2. Review the number of **Contact** objects stored in the database.

Immediately after the **//TODO Check the number of contacts is zero in the database.** comment, add the following assert:

```
Assert.assertEquals(0, rows);
```

4.3. Comment the failed assert.

Comment the code immediately after the **//TODO Comment the following line** comment.

```
//Assert.fail("Test not implemented yet!");
```

4.4. Run the test.

Open the **src/test/java** → **com.redhat.training.routes**. Right-click the **JmsRouteTest** class and select **Run As** → **JUnit test**. The method **testDelivery** has a green check mark, but the other test does not pass because it will be completed in the next step of the lab.

► 5. Implement the **testRedelivery** test method.

In the test method, an XML file with a contact is sent to the route and the database is queried to check the number of contacts stored. The result must be **1** because the second route sends the message to the message queue that is not accepting messages and it rolls back the transaction.

5.1. Send an invalid XML file to the route.

Immediately after the **//TODO Send the INVALID\_CONTACT as the body** comment, add the following code:

```
fileTemplate.sendBody(INVALID_CONTACT);
```

5.2. Review the number of contacts stored in the database.

Immediately after the **//TODO Check the message was rolled back to the database**. comment, add the following assert:

```
Assert.assertEquals(1, rows);
```

5.3. Intercept the message and throw an exception to the route.

To emulate that a message was stored in the message queue due to a message broker problem, you must throw a **ConnectException** exception to the route.

In the **RouteBuilder** anonymous class, inspect the **JMSRedelivered** header in the message and determine whether the value is **false**, meaning the message was not already processed by the message broker. Then throw a **ConnectException** exception to the route to force the transaction rollback.

```
//TODO intercept jms endpoints calls.
interceptSendToEndpoint("jms:*")
 .choice()
//TODO evaluate if the JMSRedelivered is false
 .when(header("JMSRedelivered").isEqualTo("false"))
//TODO throw a ConnectException
 .throwException(new ConnectException("Cannot connect to the message queue));
```

The source code is available as a text file (**intercept.java.txt**) in the project root directory.

5.4. Comment the failed assert.

Comment the code immediately after the **//TODO Comment the following line** comment.

```
//Assert.fail("Test not implemented yet!");
```

5.5. Run the test.

Expand **src/test/java** → **com.redhat.training.routes**. Right-click the **JmsRouteTest** class and select **Run As** → **JUnit test**. The test passes.

► **6.** Clean up.

Close the project in JBoss Developer Studio to conserve memory.

In the **Project Explorer** view, right-click **test-transaction** and click **Close Project**.

This concludes the guided exercise.

## ► Lab

# Implementing Transactions

### Performance Checklist

In this lab, you will implement a transactional route that stores a text message in the database and forwards it to a message queue. If the message queue is down, the transaction must rollback and the message must be reverted back to the database.

### Outcomes

You should be able to manage errors raised during a route execution and roll back any transaction executed by the route.

### Before You Begin

A skeleton project is provided as a starter project, which includes annotated (JPA annotations included) model classes and a previously configured Camel context, including the necessary data source configuration, a message queue, and a stubbed route builder. The starter project includes a unit test to test the route.

- Run the following command to download the starter project used by this lab:

```
[student@workstation ~]$ lab implement-transaction setup
```

- Import the starter project into JBoss Developer Studio from **/home/student/JB421/labs/implement-transaction** directory.

- Inspect the provided starter project.

Examine the Spring configuration file **src/main/resources/META-INF/spring/bundle-camel-context.xml** and the components it defines, including a data source named **atomikosDataSource**.

Also examine the model classes, located in the **com.redhat.training.camel.tx.model** package, specifically the **Message** class, which you will use to write messages to the database.

- Update the route processing from a database to a message queue to support transactions.
- Roll back the message whenever the message queue is not available. In ActiveMQ broker, the **ConnectionFailedException** exception is raised whenever the queue or the broker is not available.

Add exception handling to the route, which rolls back the transaction if a **ConnectionFailedException** exception is thrown.

- Implement the test method responsible for checking whether the transaction rolls back when you throw an **org.apache.activemq.ConnectionFailedException** exception. In order to accomplish that objective, override the **configure** method from the **AdviceWithRouteBuilder** anonymous class implemented in the **testCreateMessageRollback** method from **com.redhat.training.camel.MessageRouteTest** test case.

5. Grade your work. Execute the following command:

```
[student@workstation implement-transaction]$ lab implement-transaction grade
```

If you do not get a **PASS** grade, review your work. The **/home/student/JB421/labs/**  
**implement-transaction/grade.log** file contains the Maven messages produced by  
the grading process. This may be helpful in debugging any issues.

6. Clean up.

Close the implement-transaction project in JBoss Developer Studio to conserve memory.

This concludes the lab.

## ► Solution

# Implementing Transactions

### Performance Checklist

In this lab, you will implement a transactional route that stores a text message in the database and forwards it to a message queue. If the message queue is down, the transaction must rollback and the message must be reverted back to the database.

### Outcomes

You should be able to manage errors raised during a route execution and roll back any transaction executed by the route.

### Before You Begin

A skeleton project is provided as a starter project, which includes annotated (JPA annotations included) model classes and a previously configured Camel context, including the necessary data source configuration, a message queue, and a stubbed route builder. The starter project includes a unit test to test the route.

- Run the following command to download the starter project used by this lab:

```
[student@workstation ~]$ lab implement-transaction setup
```

- Import the starter project into JBoss Developer Studio from **/home/student/JB421/labs/implement-transaction** directory.

- Inspect the provided starter project.

Examine the Spring configuration file **src/main/resources/META-INF/spring/bundle-camel-context.xml** and the components it defines, including a data source named **atomikosDataSource**.

Also examine the model classes, located in the **com.redhat.training.camel.tx.model** package, specifically the **Message** class, which you will use to write messages to the database.

- Update the route processing from a database to a message queue to support transactions.

In the **com.redhat.training.camel.tx.MessageRouteBuilder** class add the following Java DSL to the route definition in **MessageRouteBuilder** in place of the **// TODO Add transaction** comment.

```
.transacted()
```

- Roll back the message whenever the message queue is not available. In ActiveMQ broker, the **ConnectionFailedException** exception is raised whenever the queue or the broker is not available.

Add exception handling to the route, which rolls back the transaction if a **ConnectionFailedException** exception is thrown.

In the **MessageRouteBuilder** class, add the **onException** DSL method to roll back the exception. Add the following code immediately after the **TODO develop transaction rollback** comment:

```
onException(ConnectionFailedException.class)
 .handled(true)
 .markRollbackOnly();
```

4. Implement the test method responsible for checking whether the transaction rolls back when you throw an **org.apache.activemq.ConnectionFailedException** exception. In order to accomplish that objective, override the **configure** method from the **AdviceWithRouteBuilder** anonymous class implemented in the **testCreateMessageRollback** method from **com.redhat.training.camel.tx.MessageRouteTest** test case.

Expand **src/test/java** → **com.redhat.training.camel.tx** and double-click **MessageRouteTest.java**. Add the following code to the **configure** method from the **testCreateMessageRollback** test method:

```
@Override
public void configure() throws Exception {
 //TODO implement exception
 interceptSendToEndpoint("jms:queue:messages")
 .throwException(new ConnectionFailedException());
}
```

5. Grade your work. Execute the following command:

```
[student@workstation implement-transaction]$ lab implement-transaction grade
```

If you do not get a **PASS** grade, review your work. The **/home/student/JB421/labs/implement-transaction/grade.log** file contains the Maven messages produced by the grading process. This may be helpful in debugging any issues.

6. Clean up.

Close the implement-transaction project in JBoss Developer Studio to conserve memory. This concludes the lab.

# Summary

---

In this chapter, you learned:

- Any Camel route that requires a transactional route must have access to a transaction manager in order to manage transactions.
- To declare a route as transactional, it must have a transacted DSL method call.
- A component must also be configured as transactional by passing the transaction manager to the component as a parameter during its instantiation.
- Even though a component may support transaction, the underlying system may not provide all the infrastructure required to manage transactions (such as a database without two-phase commit support or XA JDBC drivers.)
- A transaction can be defined to start and end in specific routes using a different transaction strategy by using the `org.apache.camel.spring.spi.SpringTransactionPolicy` class with a `propagationBehaviorName` defined.
- Testing transactions in a route may require an `adviceWith` method to make changes to the route prior to testing.

## Chapter 9

# Implementing Parallel Processing

### Goal

Improve the throughput of route processing using Camel parallel processing mechanisms.

### Objectives

- Implement concurrency in routes to positively impact performance of an application.
- Implement parallel processing in EIPs.

### Sections

- Increasing Throughput with Parallel Processing (and Guided Exercise)
- Implementing Parallel Processing in EIP Routes (and Guided Exercise)

### Lab

Implementing Parallel Processing

# Increasing Throughput with Parallel Processing

---

## Objectives

After completing this section, students should be able to:

- Implement concurrency in routes to positively impact performance of an application.
- Leverage concurrency to improve performance when splitting messages.
- Describe the **threads** DSL method.
- Differentiate between the **seda** and **vm** Camel components.

## Introducing Concurrency into Camel Routes

The goal of concurrency is to run several tasks or several parts of a computer program in parallel in order to reduce the total time of processing time. With the prevalence of multi-core CPUs parallel execution is quite easy as you may run each thread on its own core without splitting the processing time among multiple threads.

Using concurrency can greatly reduce the amount of time required to complete a large set of tasks since more than one task can be completed simultaneously. This is particularly important when creating integration solutions as you want to avoid creating unnecessary bottlenecks when integrating disparate systems.

In Java as well as Camel, threads are typically used to provide concurrency for performance improvement. This means that Camel processes multiple messages concurrently, reducing the total time required to process a large batch of messages. For example, an online store that needs to process multiple orders at once cannot wait for sequential processing, but should instead create a separate thread for each order to prevent users' shopping experience from slowing down.

All of the threading capabilities provided in Camel are based on leveraging the JDK **concurrency** API. This includes threads and thread pools, which can be leveraged in the following parts of Camel:

- Several of the EIP patterns, including splitter and aggregator
- The SEDA component
- The **threads** DSL method
- Several of the Camel components including JMS and Jetty

## Improving Performance when Splitting Messages

A common problem when working with large data sets in Camel is that the data need to be split and processed in small "chunks". While each chunk may only take a few milliseconds to process, this can become slow if the chunks are all processed sequentially.

The following picture shows a route that reads data from a file, splits the file content on a line by line basis using the splitter EIP, and converts the line from CSV format to an internal object model. After this, the model object is sent to another route that updates the database.

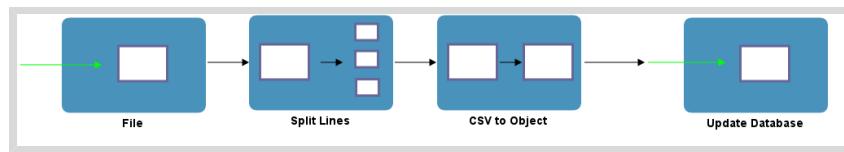


Figure 9.1: Split route example

This route can be created with the following code:

```

public class JavaRouteBuilder extends RouteBuilder {

 @Override
 public void configure() throws Exception {
 from("file:in/orders")
 .log("Starting to process file")
 .split(body().tokenize("\n")).streaming()
 .bean(Service.class, "convertCSVToObject")
 .to("direct:update").end()
 .log("Done processing file");
 from("direct:update").bean(Service.class, "updateDatabase");
 }
}

```

Processing a big file takes a long time because only one process, or thread, is used to convert CSV to object. The following scenario shows how to use concurrency to improve performance:

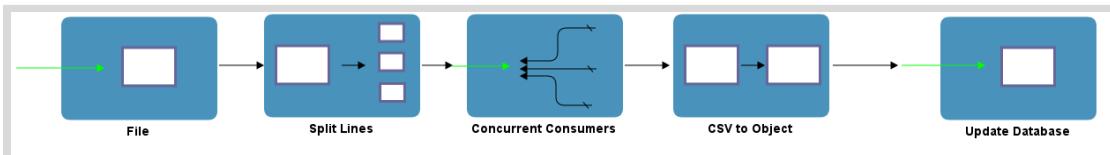


Figure 9.2: Split route with concurrency

The basic difference in this scenario is that concurrency is used after the lines have been split. This means that each split line is processed through the remaining route, converting from CSV to object and the updating of the database, happen concurrently. The splitter EIP provides two solutions for working with threads:

- **parallelProcessing** DSL method
- Custom thread pool

## ParallelProcessing

The splitter EIP offers the **parallelProcess** DSL method to enable parallel processing:

```

.split(body().tokenize("\n")).streaming().parallelProcessing()
.bean(Service.class, "convertCSVToObject")
.to("direct:update").end()

```

To enable parallel processing in XML, use the following option:

```
<split streaming="true" parallelProcessing="true">
 <tokenize token="\n"/>
 <bean beanType="com.redhat.training.Service"
 method="convertCSVToObject"/>
 <to uri="direct:update"/>
</split>
```

When you use **parallelProcessing** DSL method, the splitter EIP uses a thread pool to process messages concurrently. The size of the thread pool indicates how many available threads can be used concurrently. For example, if the thread pool size is 10 and there are 11 requests for processes made, one of the requests has to wait until a thread becomes available again.

By default, the thread pool is configured to use 10 threads to process the messages. This means that this route performance improves because the conversion and the database updates happen with up to 10 parallel processes running concurrently.

## Custom Thread Pool

The splitter EIP also allows you to use a custom thread pool using the **ExecutorService** class from the Java concurrency API, as shown in the following example:

```
@Override
public void configure() throws Exception {
 ExecutorService threadPool = Executors.newFixedThreadPool(20); ①
 from("file:rider/inventory")
 .log("Starting to process file")
 .split(body().tokenize("\n")).streaming().executorService(threadPool) ②
 .bean(Service.class, "convertCSVToObject")
 .to("direct:update").end()
 .log("Done processing file");
 from("direct:update").bean(Service.class, "updateDatabase");
}
```

- ① Create a fixed size thread pool setting the number of threads to 20
- ② Set the splitter EIP to use the created thread pool

The following is the same example using the Spring XML:

```
<bean id="threadPool" class="java.util.concurrent.Executors"
 factory-method="newFixedThreadPool">
 <constructor-arg index="0" value="20"/>
</bean>

<split streaming="true" executorServiceRef="threadPool">
 <tokenize token="\n"/>
 <bean beanType="com.redhat.training.Service"
 method="convertCSVToObject"/>
 <to uri="direct:update"/>
</split>
```

**Important**

Do not use threads in cases where the order of messages is important. Camel does not guarantee that messages are processed in order when concurrency is used.

## Executing Multiple Threads using Custom Thread Pools

A number of Camel components provide the **concurrentConsumers** option. This allows you to specify how many threads you want processing exchanges received by that consumer simultaneously. For example, the following Java DSL creates 5 threads to read from the JMS queue:

```
from("jms:MyQueue?concurrentConsumers=5").bean(SomeBean.class);
```

In Spring DSL:

```
<route>
 <from uri="jms:MyQueue?concurrentConsumers=5"/>
 <to uri="bean:someBean"/>
</route>
```

Most Camel components which support concurrent consumers also support using a custom thread pool. When creating a custom thread pool, any implementation of the **ExecutorService** class can be used. If you wish to use a fixed number of threads you must specify the number of threads when you create the pool as shown in the following code snippet:

```
ExecutorService threadPool = Executors.newFixedThreadPool(20);
```

With the fixed thread pool approach threads are only created and destroyed once, running constantly while the application is running. However, a common option is the cached thread pool which automatically grows and shrinks on demand. The following code shows an example of creating a cached thread pool:

```
ExecutorService threadPool = Executors.newCachedThreadPool();
```

To configure this with Spring XML, use the following code:

```
<bean id="threadPool" class="java.util.concurrent.Executors"
 factory-method="newCachedThreadPool"/>
```

This approach works well with limited hardware, but there can be unexpected side effects in an enterprise system where a high number of created threads may impact applications in other areas.

**Note**

Try to choose a type of thread pool that makes sense with the type of application you are building. This can also include adjusting configuration specific to the thread pool type you choose to optimize the pool's behavior.

## Leveraging Camel's threads DSL Method

The **threads** DSL method leverages the JDK concurrency framework for multithreading inside a Camel route. The inclusion of the **threads** DSL method is used to turn a synchronous route into an asynchronous route. Any actions in the route that occur after the **threads** declaration are routed asynchronously in a new thread. An example of using **threads** DSL method is shown in the following route definition:

```
// listen on the queue for new orders
from("activemq:queue:order")
 // do some validation
 .to("bean:checkOrder")
 // use multi threading with a pool size of 20
 .threads(20)
 // do some CPU intensive processing of the message
 .unmarshal(mySecureDataFormat).to("bean:handleOrder");
```

This approach can be useful when you want to let more process intensive tasks run concurrently, but also need synchronous logic inside the route prior to splitting into multiple threads.

## Demonstration: Improving Performance with Concurrency in Camel Routes

1. Start JBoss Developer Studio and import the **introduce-concurrency** project from the `/home/student/JB421/labs/` directory.
2. Evaluate the existing route definition in **ConcurrentRouteBuilder** class.

This class pulls orders from the database deleting them as they are consumed, and splits them by order item. Each order item is processed by the **InventoryRequestProcessor** processor class which converts the items into inventory requests to be sent to vendors. Inventory requests are then marshaled into CSV records and written to a file.

Note that in its current state, the route has no concurrency optimizations.

3. Evaluate the **InventoryRequestProcessor** class.

Note that this class uses a **Thread.sleep** method invocation to simulate a processing delay. This delay could be caused by communication with a remote system or a complex data transformation. For this example, each item request takes 30 ms to be processed.

4. Create test data using the provided script.

Run the **setup-data.sh** to create the sample order data. This creates 5 orders, each with 300 order items.

```
[student@workstation introduce-concurrency]$./setup-data.sh
Creating a batch of 5 test orders with 300 order items each
Order 1 was created!
Order 2 was created!
Order 3 was created!
Order 4 was created!
Order 5 was created!
```

5. Review the **TimerBean** class.

Note that it keeps track of how many order items have been processed, and outputs the total time each time an order is completed.

- Run the route in its current state, without streaming or parallel processing, and observe the timing.

Open a new terminal window and start the route using the Maven Camel plug-in. The data starts processing immediately, and you see output similar to the following, which has been trimmed for brevity:

```
INFO Order 1 Processing complete! Time elapsed: 11 seconds
...
INFO Order 5 Processing complete! Time elapsed: 55 seconds
```

- Stop the route and recreate test data using the provided script.

Stop the route execution with **Ctrl+C** and then in the same terminal window, run the **setup-data.sh** to recreate the sample order data.

- Update the route definition to include parallel processing.

Return to JBoss Developer Studio and editing the **ConcurrentRouteBuilder** class. Update the **split** method to include the parallel processing option enabled as shown in the following Java DSL snippet:

```
.split(simple("${body.getOrderItems()}")).parallelProcessing()
```

Press **Ctrl+S** to save your changes.

- Run the route again with parallel processing and observe the timing.

Return to the terminal window, run the **setup-data.sh** to recreate the sample order data, and start the route using the Maven Camel plug-in. The data starts processing immediately and you see output similar to the following, which has been trimmed for brevity:

```
... INFO Order 1 Processing complete! Time elapsed: 1 seconds
...
... INFO Order 5 Processing complete! Time elapsed: 7 seconds
```

Note that the logging now includes the thread names and that the performance was drastically improved.

- Stop the route and recreate test data using the provided script.

Stop the route execution with **Ctrl+C** and then in the same terminal window, run the **setup-data.sh** script to recreate the sample order data.

- Update the route definition to use a SEDA endpoint for all processing after the order items are split.

Return to editing the **ConcurrentRouteBuilder** class in JBoss Developer Studio. Uncomment the **seda** consumer route. Update the DSL after the **split** method by removing all of the existing method calls after the **split** method and uncomment the **seda** producer. Your route definition should match the following:

```

from("jpa:com.redhat.training.jb421.model.Order?persistenceUnit=mysql"
 + "&consumer.namedQuery=getUndeliveredOrders")
 .bean("timerBean", "start")
 .split(simple("${body.getOrderItems()}")).parallelProcessing()
 .to("seda:inventoryProcessor")
 .end();

from("seda:inventoryProcessor")
 .process(new InventoryRequestProcessor())
 .marshal().bindy(BindyType.Csv, InventoryRequest.class)
 .to("file:"+OUTPUT_FOLDER+"?fileName=${header.publisherId}/"
 + "inventory-reservation-${date:now:yyyy-MM-dd}.csv&fileExist=Append")
 .bean("timerBean", "stop");

```

- Run the route again with SEDA endpoint and observe the timing.

Return to the terminal window and start the route using the Maven Camel plug-in. The data starts processing immediately and you see output similar to the following, which has been trimmed for brevity:

```

...://inventoryProcessor] TimerBean INFO Order 1 Processing complete! Time
elapsed: 11 seconds
...
...://inventoryProcessor] TimerBean INFO Order 5 Processing complete! Time
elapsed: 55 seconds

```

As you can see, by default **seda** only uses a single thread, and therefore the performance shows no improvement from concurrency.

- Recreate test data using the provided script.

Stop the route execution with **Ctrl+C** and then in the same terminal window, run the **setup-data.sh** script to recreate the sample order data.

- Update the SEDA endpoint to include 20 concurrent consumers.

Return to JBoss Developer Studio and editing the **ConcurrentRouteBuilder** class. Append a URI option to set the **concurrentConsumers** option on the **seda** consumer to 20 as shown in the following Java DSL:

```
from("seda:inventoryProcessor?concurrentConsumers=20")
```

- Run the route again with more SEDA consumers and observe the timing.

Return to the terminal window and start the route using the Maven Camel plug-in. The data starts processing immediately and the **TimerBean** prints out each order as it is processed and the total time elapsed, similar to following output which has been trimmed for brevity:

```

...://inventoryProcessor] TimerBean INFO Order 1 Processing complete! Time
elapsed: 1 seconds
...
...://inventoryProcessor] TimerBean INFO Order 5 Processing complete! Time
elapsed: 7 seconds

```

16. Clean up.

Stop the execution of the Camel Maven plug-in using **Ctrl+C**, and close the **introduce-concurrency** project in JBoss Developer Studio.

This concludes the demonstration.

## Comparing the seda and vm Components

*Stage event driven architecture* or SEDA is an approach to software development for event-driven applications, which breaks the functionality of the application into a set of stages connected by queues. This approach avoids the high overhead associated with thread-based concurrency and decouples event and threading logic from the actual application logic without needing to rely on JMS or other messaging technologies. Instead the Camel SEDA and **vm** components rely on in-memory queues.

Camel provides support for SEDA with both the **sed**a and **vm** components. Both of these components create in memory queues to hold exchanges between stages, but neither of these components offers any persistence. If the JVM terminates while exchanges are not yet processed, those exchanges are lost. If you need persistence or distributed SEDA, you can use JMS or ActiveMQ.

### SEDA Component

The **sed**a component provides asynchronous SEDA behavior, so that messages are exchanged on a **BlockingQueue** instance and consumers are invoked in a separate thread from the producer. SEDA endpoints are only visible within a single Camel context and therefore cannot be used to communicate across applications. However, if you only require messages to be route within the same Camel context, then a **sed**a component is a good replacement for JMS as it does not require as much overhead.

The end point URI format is: `sed :<queueName>[?options]`.

#### SEDA Options

Option name	Default value	Description
<b>concurrentConsumers</b>	1	Number of concurrent threads processing exchanges.
<b>size</b>	unbounded	The maximum capacity of the SEDA queue. By default, queue size is unbounded.
<b>multipleConsumers</b>	false	Specifies whether multiple consumers are allowed. If enabled, you can use SEDA for Pub/Sub messaging, or you can send a message to a SEDA queue and have each consumer receive a copy of the message. This option needs to be specified on all consumer endpoints that share a queue.

The following is an example of a **sed**a component with a content-based router that determines the file type of the message and queues the message with an in-memory queue based on that file type:

```
from("jms:orders")
 .choice()
 .when(header("CamelFileName").endsWith(".csv"))
 .to("seda:csvOrders")
 .when(header("CamelFileName").endsWith(".json"))
 .to("seda:jsonOrders")
 .otherwise()
 .to("seda:errorQueue")
```

## VM Component

The VM component also offers the same asynchronous SEDA behavior, but differs from the **seda** component in that **vm** supports communication across **CamelContext** instances. You can use this mechanism to communicate across web applications in the same container, provided they are using the same **camel-core** Maven dependency. This is useful if a route needs to interact with another **war** file deployed onto the same server.

The endpoint URI format is: `vm:<queueName>[?options]`. The **vm** component supports all the same options as the **seda** component.

The following is an example of sending a message onto a queue asynchronously using the **vm** component:

```
from("direct:orders")
 .to("vm:processing");
```

In a separate deployment, this queue is referenced using the following route:

```
from("vm:processing")
 .to("jms:toProcess");
```



### References

#### SEDA component

<http://camel.apache.org/seda.html>

#### VM component

<http://camel.apache.org/vm.html>

#### Camel Threading Model

<http://camel.apache.org/threading-model.html>

## ► Guided Exercise

# Testing the Performance of Memory Queues with SEDA

In this exercise, you will monitor the execution and test the performance of using SEDA endpoints to implement concurrency in a route.

## Outcomes

You should be able to improve the performance of the provided route using SEDA endpoints to implement concurrency.

## Before You Begin

A starter project is provided for you. It includes a **CamelContext** configured in a Spring configuration file, a **RouteBuilder** subclass, and sample data for you to use to verify your work.

Run the following command to download the starter project and data files used by this exercise:

```
[student@workstation ~]$ lab introduce-seda setup
```

## Steps

- ▶ 1. Open JBoss Developer Studio and import the starter Maven project.
  - 1.1. Open JBoss Developer Studio (**Applications** → **Programming** → **JBoss Developer Studio**). JBoss Developer Studio requests a default workspace; keep the default configuration (**/home/student/workspace**) and click **Launch**.
  - 1.2. Import the Maven project by selecting JBoss Developer Studio main menu **File** → **Import**.
  - 1.3. From the **Import** dialog box, select **Maven** → **Existing Maven Projects** and click **Next**.
  - 1.4. Click **Browse** and choose **/home/student/JB421/labs/introduce-seda** and click **OK** to choose the Maven project. Click **Finish** to complete the import process in JBoss Developer Studio.  
A new project named **introduce-seda** will be listed in the **Project Explorer** view.
- ▶ 2. Review the Spring and Camel configurations.  
In the **Project Explorer** view, expand **introduce-seda** → **src/main/resources** → **META-INF/spring**, and double-click **bundle-camel-context.xml** to review the Spring configuration for the starter project.  
Notice the following components are defined:
  - Camel context is defined with a single route builder.

- The data source defined as **mysqlDataSource**, this is the data source you need to connect to using the **jpa** component. The database for this exercise is already running and populated with the necessary tables on the **services** VM.
- The entity manager used by JPA to connect to a database.
- A **JPATransactionManager** class that manages transactions executed within JPA.

► 3. Review the route definition.

In the **Project Explorer** view, expand **src/main/java** → **com.redhat.training.jb421**. Double-click the **ConcurrentRouteBuilder.java** file.

```
from("jpa:com.redhat.training.jb421.model.Order?persistenceUnit=mysql"
 + "&consumer.namedQuery=getUndeliveredOrders")
 .bean("timerBean", "start")
 .split(simple("${body.getOrderItems()}"))
 .process(new InventoryRequestProcessor())
 .marshal().bindy(BindyType.Csv, InventoryRequest.class)
 .to("file:"+OUTPUT_FOLDER+"?fileName=${header.publisherId}))"
 + "inventory-reservation-${date:now:yyyy-MM-dd}.csv&fileExist=Append"
 .bean("timerBean", "stop")
// .to("seda:inventoryProcessor")
 .end();
```

In its current state, the route has no concurrency optimizations.

► 4. Set up a new Maven run configuration in JBoss Developer Studio to test the route.

4.1. Open Run configurations dialog.

Find the **Run** icon in the JBoss Developer Studio toolbar. It is shown in the following screen shot.



Figure 9.3: Run button in JBoss Developer Studio

Click the drop-down, and choose **Run Configurations....**

4.2. Create a new Maven run configuration.

Once the **Run Configurations** dialog is active, scroll down the left pane until you see **Maven Build**, select this, and then click the **New launch configuration** icon at the top of the dialog window as shown in the following screenshot.

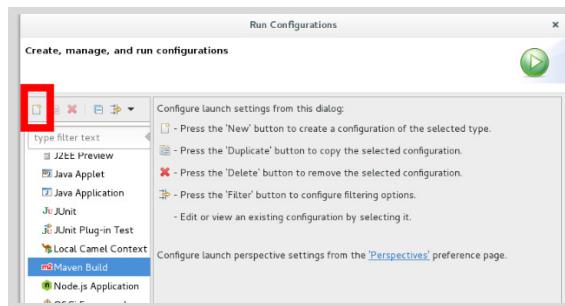


Figure 9.4: Maven Build run configuration

#### 4.3. Configure the new Maven run configuration.

Set the **Name** on the new configuration to **Camel Run**. Select the **Base directory** using the **Workspace...** button and choose the **introduce-seda** project. For the **Goals** specify **camel:run**. Once you are finished, click **Apply** to finalize and save the run configuration. Make sure your configuration matches the following screen shot.

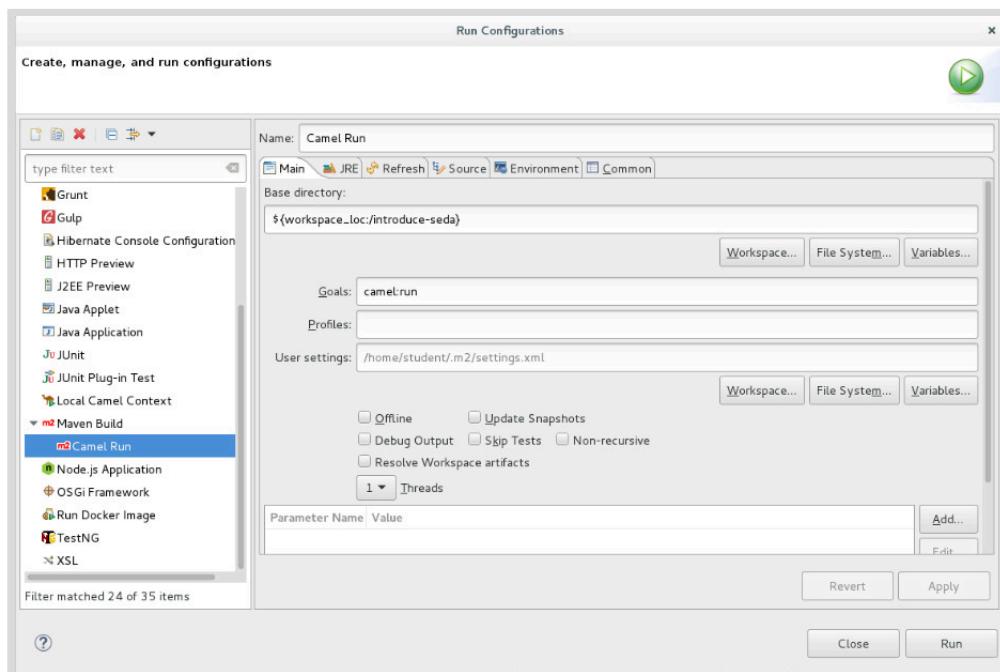


Figure 9.5: Camel run configuration

#### 4.4. Test your new run configuration.

Once you have finished creating the run configuration and have saved it by clicking **Apply** in the dialog window, click **Run** to test it out.

You have not loaded any test data yet, so just make sure the route starts cleanly. You should see the following output in the log if everything was successful:

```
... INFO Total 1 routes, of which 1 are started.
... INFO Apache Camel 2.21.0.fuse-000077-redhat-1 (CamelContext: jb421Context)
started in 0.340 seconds
```

#### 4.5. Stop the execution of the run configuration.

Click **Terminate**, shown in the following screen capture, to stop the execution.

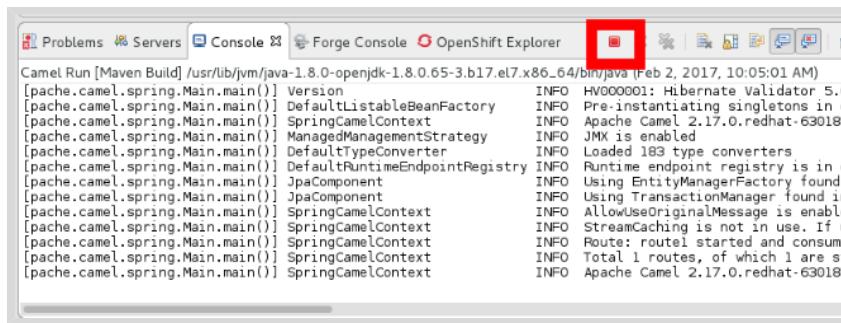


Figure 9.6: Stop button in JBoss Developer Studio

► 5. Create test data using the provided script.

To execute the script to add data, open a new terminal window, navigate to **/home/student/JB421/labs/introduce-seda**, and run **./setup-data.sh** using the following commands:

```
[student@workstation ~]$ cd JB421/labs/introduce-seda
[student@workstation introduce-seda]$./setup-data.sh
Creating a batch of 5 test orders with 300 order items each
Order 1 was created!
Order 2 was created!
Order 3 was created!
Order 4 was created!
Order 5 was created!
```

► 6. Start the route for baseline testing using debug mode.

- 6.1. Switch to the **Debug** perspective using the button in the top right of the JBoss Developer Studio window.

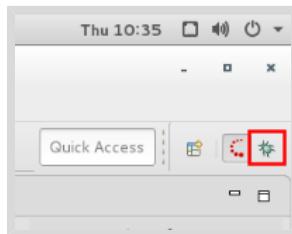


Figure 9.7: Debug button



#### Note

If you do not see the **Debug** perspective icon, this may be because you have not opened this perspective before. Use the **Open Perspective** icon to choose **Debug** from the list.

- 6.2. Execute the route in debug mode monitoring the threads shown in the **Debug** window shown in the following screen capture.



**Figure 9.8: Debug threads shown in the debug perspective**

Using the **Debug** drop-down menu, choose your newly created run configuration.

The route starts processing orders immediately, wait until all five orders have been processed and check the timing. You should see output in the console similar to the following:

```
...jb421.model.Order] TimerBean INFO Order 1 Processing complete! Time elapsed: 11
seconds
...jb421.model.Order] TimerBean INFO Order 2 Processing complete! Time elapsed: 22
seconds
...jb421.model.Order] TimerBean INFO Order 3 Processing complete! Time elapsed: 33
seconds
...jb421.model.Order] TimerBean INFO Order 4 Processing complete! Time elapsed: 44
seconds
...jb421.model.Order] TimerBean INFO Order 5 Processing complete! Time elapsed: 55
seconds
```

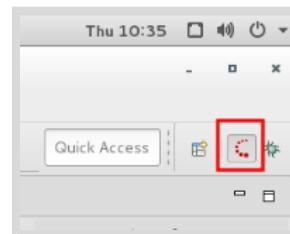
There was only one thread labeled as **jb421Context** throughout the entire execution of the route. This is because there is no concurrency built into the route yet.

- 6.3. Stop the execution of the route.

Click **Terminate** in the console window to stop the route.

► **7.** Update the route to use SEDA with 20 concurrent consumers.

- 7.1. To edit the route, switch back to the **JBoss** perspective using the icon in the top right of the window.



**Figure 9.9: JBoss perspective button**

- 7.2. Open and edit the route definition in **ConcurrentRouteBuilder** class.

Uncomment the **seda** consumer route and remove all existing actions after the **split** method and uncomment the **seda** producer. Make sure your route definition matches the following DSL snippet:

```

from("jpa:com.redhat.training.jb421.model.Order?persistenceUnit=mysql"
 + "&consumer.namedQuery=getUndeliveredOrders")
 .bean("timerBean", "start")
 .split(simple("${body.getOrderItems()}"))
 .to("seda:inventoryProcessor")
 .end();

from("seda:inventoryProcessor?concurrentConsumers=20")
 .process(new InventoryRequestProcessor())
 .marshal().bindy(BindyType.Csv, InventoryRequest.class)
 .to("file:"+OUTPUT_FOLDER+"?fileName=${header.publisherId}/"
 + "inventory-reservation-${date:now:yyyy-MM-dd}.csv&fileExist=Append")
 .bean("timerBean", "stop");

```

Save your changes using **Ctrl+S**.

► 8. Recreate the test data.

Return to the terminal window you opened, and again run **./setup-data.sh** using the following commands:

```

[student@workstation introduce-seda]$./setup-data.sh
Creating a batch of 5 test orders with 300 order items each
Order 1 was created!
Order 2 was created!
Order 3 was created!
Order 4 was created!
Order 5 was created!

```

► 9. Retest the route in debug mode, again monitoring the timing and number of threads shown in the debug window.

9.1. Switch back to the **Debug** perspective using the icon in the top right of the JBoss Developer Studio window.

9.2. Use the **Debug** drop-down, and choose your run configuration.

The route starts processing orders immediately, wait until all five orders have been processed and check the timing. You will see output in the console similar to the following:

```

[#1 - ... TimerBean INFO Order 1 Processing complete! Time elapsed: 2 seconds
[#2 - ... TimerBean INFO Order 2 Processing complete! Time elapsed: 4 seconds
[#4 - ... TimerBean INFO Order 3 Processing complete! Time elapsed: 7 seconds
[#3 - ... TimerBean INFO Order 4 Processing complete! Time elapsed: 9 seconds
[#1 - ... TimerBean INFO Order 5 Processing complete! Time elapsed: 11 seconds

```

There are extra threads in the **Debug** perspective, one for each SEDA consumer. Adding concurrency greatly reduced the total time required to process the test orders.

9.3. Stop the execution of the route.

Click **Terminate** in the console window to stop the route.

- ▶ 10. Update the SEDA endpoint to have 100 concurrent consumers and retest the route performance.

- 10.1. Open and edit the route definition in **ConcurrentRouteBuilder** class.

Adjust the **concurrentConsumers** option setting on the **seda** component from 20 to 100 concurrent consumers, as shown in the following DSL snippet:

```
from("seda:inventoryProcessor?concurrentConsumers=100")
```

- 10.2. Recreate the test data.

Return to the terminal window you opened, and again run **./setup-data.sh** using the following commands:

```
[student@workstation introduce-seda]$./setup-data.sh
Creating a batch of 5 test orders with 300 order items each
Order 1 was created!
Order 2 was created!
Order 3 was created!
Order 4 was created!
Order 5 was created!
```

- 10.3. Run the route again using debug mode to monitor the threads and check the results.

Use the **Debug** drop-down to choose your run configuration.

The route starts processing orders immediately, wait until all five orders have been processed and check the timing. You see output in the console similar to the following:

```
[#1 - ... TimerBean INFO Order 1 Processing complete! Time elapsed: 1 seconds
[#2 - ... TimerBean INFO Order 2 Processing complete! Time elapsed: 3 seconds
[#4 - ... TimerBean INFO Order 3 Processing complete! Time elapsed: 4 seconds
[#3 - ... TimerBean INFO Order 4 Processing complete! Time elapsed: 5 seconds
[#1 - ... TimerBean INFO Order 5 Processing complete! Time elapsed: 7 seconds
```

As you can see, adding more threads for concurrency again reduced the total time required to process the orders, but not as dramatically as before. You may have also noticed even more extra threads in the **Debug** perspective, one for each SEDA consumer.

- ▶ 11. Close the project.

In the **Project Explorer** view, right-click **introduce-seda** and click **Close Project**.

This concludes the guided exercise.

# Implementing Parallel Processing with Camel Enterprise Integration Patterns

## Objectives

After completing this section, students should be able to:

- Implement concurrency in a Camel route that uses an EIP.
- Describe and avoid the common pitfalls of concurrency.
- Leverage **multicast** and **wireTap** to run tasks in parallel.
- Use the **threads** DSL to enable concurrent in a route that uses a single-threaded consumer.

## Describing the Caveats of Using Parallel Processes

When using concurrency while developing Camel routes, there is an increased risk of bugs that manifest in unexpected ways and can be difficult to track down and fix. There are a few common problems that can arise while attempting to run routes and tasks concurrently:

- **Deadlocks:** Deadlocks occur when concurrent tasks attempt to modify the same data in a database. If a thread hits a deadlock, it must wait for the lock to be removed before it can continue, causing poor performance. There are strategies to avoid deadlocks, such as optimistic locking that allows for concurrent access to a database.
- **Resource starvation:** When the number of tasks running concurrently is too large, the task scheduler cannot keep up resulting in resource starvation. This forces tasks to wait for resources, drastically reducing performance. It is important to do proper performance testing when implementing concurrency to avoid resource starvation while still benefiting from concurrency. To find this balance, you must have a good understanding of the load required for each of your concurrent tasks and how many of those tasks your resources can handle.
- **Messages delivered out of order:** Any time concurrency is used in a route, the processing ordering of the messages is no longer sequential. This means that the order of the messages arrival is no longer guaranteed or reliable. For this reason, it is best to avoid any dependency on message order when possible. Alternatively, the **resequence** element corrects message order after concurrent processing.

## Enterprise Integration Patterns Supporting Parallel Processing

### Aggregator

In the aggregator pattern, multiple messages are combined into a single message to be processed by a consumer. If the **parallelProcessing** option is enabled on the aggregator, then Camel uses a thread pool and concurrency instead of a single thread to process each of the aggregated messages.

### Multicast

The multicast pattern sends a message to multiple endpoint. Multicast supports two concurrency options: **parallelProcessing** and **parallelAggregate**. Both are disabled by default. The **parallelProcessing** option enables multicast to send exchanges to each destination

concurrently. The **parallelAggregate** option allows the aggregate method from the **AggregationStrategy** implementation to be called concurrently, but that method must be implemented as thread-safe.

## Splitter

The splitter component breaks a single message into multiple messages that are processed separately. When the **parallelProcessing** option is enabled, messages are processed concurrently. This allows Camel to use a thread pool and concurrency instead of a single thread to process each of the split messages.

## Wire tap

The wire tap pattern spawns a copy of the exchange and creates a new thread to process it. Because of this, the wire tap pattern inherently allows for concurrency of tasks. Wire tap always uses a thread pool to execute any spawned exchanges.

## Threads

The **threads** pattern allows the use of concurrency by creating a thread pool for any set of tasks inside a route contained within the **threads** element. Any exchanges entering a section of a Camel route that is wrapped by **threads** is processed by a thread pool concurrently until the **threads** portion of the route is completed.

## Describing the Multicast Pattern

As mentioned previously, the multicast pattern allows a route to send the same message to multiple endpoints. This allows for the same message to be processed by multiple endpoints. If necessary, the route aggregates the results from each endpoint back into a single exchange.

To use the multicast pattern, wrap your producer endpoints in a **multicast** element as shown in the following code:

```
from("direct:a")
 .multicast()
 .to("direct:x")
 .to("direct:y")
 .to("direct:z")
 .end();
```

By default, **multicast** sends the message to each endpoint sequentially and waits until a reply is received before sending the message on to the next endpoint.

To enable concurrency with multicast to prevent this sequential waiting, use the **parallelProcessing** option as shown in the following example:

```
from("direct:a")
 .multicast().parallelProcessing()
 .to("direct:x")
 .to("direct:y")
 .to("direct:z")
 .end();
```

Replies from the endpoints are aggregated using an **AggregationStrategy**, similar to the **aggregate** element. An example of using multicast with aggregation is shown in the following route definition:

```
from("direct:a")
 .multicast(new MyAggregationStrategy()).parallelProcessing()
 .to("direct:x")
 .to("direct:y")
 .to("direct:z")
 .end();
```

## Demonstration: Comparing Parallel Processing Methods That Use Enterprise Integration Patterns

1. Start JBoss Developer Studio and import the **introduce-parallel** project from the `/home/student/JB421/labs/` directory.
2. Evaluate the existing route definition in **ConcurrentRouteBuilder** class.

This class pulls inventory updates from the file system, processes the files, and delete them as they are consumed. Each inventory update is sent to the **InventoryUpdateProcessor** processor, which extracts the vendor ID from the file name and communicates with an external system to process each update. Additionally, the **PartnerInventoryUpdateProcessor** processor is used to communicate the updates to partners. Inventory updates are then marshaled into CSV records and written to a file, which corresponds to their vendor ID.

In its current state, the route has no concurrency optimizations.

3. Evaluate the **InventoryUpdateProcessor** and **PartnerInventoryUpdateProcessor** processors.

Note that both processors use a **Thread.sleep** method to simulate a processing delay. This delay could be caused by communication with a remote system or a complex data transformation. For this example, each item request takes 20 ms to be processed by each of the two processors.

4. Create test data using the provided script.

Run the **setup-data.sh** to create the sample inventory data. 1500 inventory updates are created.

```
[student@workstation introduce-parallel]$./setup-data.sh
Creating a batch of 1500 test inventory updates
```

5. Run the route in its current state, without any streaming or parallel processing, and observe the timing.

In your existing terminal window, start the route using the Maven Camel plug-in. The data starts processing immediately and you will see output similar to the following which has been trimmed for brevity:

```
...INFO Item Number 100 Processing complete! Time elapsed: 4 seconds
...
...INFO Item Number 1500 Processing complete! Time elapsed: 76 seconds
```

Keep the timings for this execution for comparison purposes with the following steps.

6. Stop the route and recreate test data using the provided script.

Stop the route execution with **Ctrl+C** and then in the same terminal window, run the **setup-data.sh** to recreate the same inventory data.

```
[student@workstation introduce-parallel]$./setup-data.sh
Creating a batch of 1500 test inventory updates
```

7. Update the route definition to include parallel processing using **threads** method.

Return to JBoss Developer Studio and editing the **ConcurrentRouteBuilder** class. Update the route definition to include threading with five concurrent threads processing updates as shown in the following Java DSL snippet:

```
from("file://" + INPUT_FOLDER + "?delete=true")
 .bean("timerBean", "start")
 .threads(5)
 .marshal().bindy(BindyType.Csv, InventoryUpdate.class)
 .process(new InventoryUpdateProcessor())
 .process(new PartnerInventoryUpdateProcessor())
 .to("file:" + OUTPUT_FOLDER + "?fileName=${header.publisherId}/"
 + "inventory-reservation-${date:now:yyyy-MM-dd}.csv&fileExist=Append")
 .bean("timerBean", "stop")
 .end();
```

Save your changes using **Ctrl+S**.

8. Run the route again with parallel processing and observe the timing.

Return to the terminal window and start the route using the Maven Camel plug-in. The data starts processing immediately and you see output similar to the following which has been trimmed for brevity:

```
...INFO Item Number 100 Processing complete! Time elapsed: 0 seconds
...
...INFO Item Number 1500 Processing complete! Time elapsed: 7 seconds
```

Note that the logging now includes the thread names and that the performance was drastically improved.

9. Stop the route and recreate test data using the provided script.

Stop the route execution with **Ctrl+C** and then in the same terminal window, run the **setup-data.sh** to recreate the inventory update data.

```
[student@workstation introduce-parallel]$./setup-data.sh
Creating a batch of 1500 test inventory updates
```

10. Update the route definition to use a custom thread pool for all concurrency in the route.

Return to editing the **ConcurrentRouteBuilder** class in JBoss Developer Studio. Remove the **5** from the DSL **threads** method and add a reference to the **ExecutorService** instance, which is instantiated at the top of the route builder. Make sure your route definition matches the following:

```
ExecutorService threadPool = Executors.newCachedThreadPool();

from("file://" + INPUT_FOLDER +"?delete=true")
 .bean("timerBean", "start")
 .threads().executorService(threadPool)
 .marshal().bindy(BindyType.Csv,InventoryUpdate.class)
 .process(new InventoryUpdateProcessor())
 .process(new PartnerInventoryUpdateProcessor())
 .to("file:"+OUTPUT_FOLDER+"?fileName=${header.publisherId}"+
 + "inventory-reservation-${date:now:yyyy-MM-dd}.csv&fileExist=Append")
 .bean("timerBean", "stop")
 .end();
```

11. Re-run route with the custom thread pool and check timing.

Return to the terminal window and start the route using the Maven Camel plug-in. The data starts processing immediately and the **TimerBean** class prints out each order as it is processed and the total time elapsed, similar to following output:

```
...INFO Item Number 100 Processing complete! Time elapsed: 0 seconds
...
...INFO Item Number 1500 Processing complete! Time elapsed: 5 seconds
```

The cached thread pool performs slightly better than the fixed pool with five threads. Other customization options could be set on the **ExecutorService** to further tweak the concurrency behavior.

12. Stop the route and recreate test data using the provided script.

Stop the route execution with **Ctrl+C**. In the same terminal window, run the **setup-data.sh** to recreate the inventory update data.

```
[student@workstation introduce-parallel]$./setup-data.sh
Creating a batch of 1500 test inventory updates
```

13. Update the route definition to use **multicast** method to execute the two processors and the file consumer in parallel.

Return to editing the **ConcurrentRouteBuilder** class in JBoss Developer Studio. Remove the existing multi-threading and update the existing route to use **multicast** DSL method with the **parallelProcessing** option enabled to execute the processors in parallel. Make sure your route definition matches the following:

```
from("file://" + INPUT_FOLDER + "?delete=true")
 .bean("timerBean", "start")
 .marshal().bindy(BindyType.Csv, InventoryUpdate.class)
 .multicast().parallelProcessing()
 .process(new InventoryUpdateProcessor())
 .process(new PartnerInventoryUpdateProcessor())
 .end()
 .to("file:" + OUTPUT_FOLDER + "?fileName=${header.publisherId}))"
 + "inventory-reservation-${date:now:yyyy-MM-dd}.csv&fileExist=Append")
 .bean("timerBean", "stop")
```

14. Re-run route using multicast for concurrency and check timing.

Return to the terminal window and start the route using the Maven Camel plug-in. The data starts processing immediately and you will see output similar to the following which has been trimmed for brevity:

```
.../incoming] TimerBean INFO Item Number 100 Processing complete! Time elapsed:
3 seconds
...
.../incoming] TimerBean INFO Item Number 1500 Processing complete! Time elapsed:
58 seconds
```

Now only two processors are running in parallel. This is different than before, when everything after the file consumer ran in parallel. The performance is better than when you used no concurrency, but not as good as using **threads** to make more steps of the route run in parallel.

15. Clean up.

Stop the execution of the Maven plug-in using **Ctrl+C**, and close the **introduce-parallel** project in JBoss Developer Studio.

This concludes the demonstration.

## Implementing Parallel Processing with the Wire Tap Pattern

Anytime a wire tap is used, a thread pool is automatically created to process any messages concurrently. You can customize the thread pool for the wire tap to use, but if no pool is specified, wire tap uses Camel's default thread pool profile. An example of customizing the thread pool when using a wire tap is show in the following example route definition:

```
public void configure() throws Exception {
 ExecutorService fixedPool = Executors.newFixedThreadPool(20);

 from("direct:start")
 .wireTap("direct:log", fixedPool)
 .to("mock:result");

 from("direct:log")
```

```
.log("Message: ${body}")
.to("mock:service");
}
```

## Implementing Parallel Processing with Threads and Files

Using the **threads** element, you can create concurrency in a route even when using components that do not support parallel processing. An example of a commonly used single-threaded component is the **file** component.

The **file** component must only use a single thread to avoid issues with concurrent modification of files. By using the **threads** element, you can parallelize the processing of files consumed by the component. An example of this approach is shown in the following route definition:

```
from("file://" + INPUT_FOLDER +"?delete=true")
.threads()
 .marshal().bindy(BindyType.Csv, InventoryUpdate.class)
 .process(new InventoryUpdateProcessor())
 .process(new PartnerInventoryUpdateProcessor())
 .to("activemq:orderQueue")
.end();
```

In this example, every part of the route that is contained inside the **threads** element will run concurrently using a thread pool.



### References

#### Multicast pattern

<http://camel.apache.org/multicast.html>

#### Parallel processing

<http://camel.apache.org/parallel-processing-and-ordering.html>

## ► Guided Exercise

# Increasing Throughput When Reading Files

In this exercise, you use a number of different techniques to implement concurrency in a Camel route that reads a large number of files, and then monitor the performance of each.

## Outcomes

You should be able to improve the performance of the provided route using concurrency.

## Before You Begin

A starter project is provided for you. It includes a **CamelContext** XML definition file and a **RouteBuilder** implementation.

Run the following command to download the starter project and data files used by this exercise:

```
[student@workstation ~]$ lab throughput-files setup
```

- ▶ 1. Open JBoss Developer Studio and import the starter Maven project.
  - 1.1. Open JBoss Developer Studio (**Applications** → **Programming** → **JBoss Developer Studio**). JBoss Developer Studio requests a default workspace; keep the default configuration (**/home/student/workspace**) and click **Launch**.
  - 1.2. Import the Maven project by selecting JBoss Developer Studio main menu **File** → **Import**.
  - 1.3. From the **Import** dialog box, select **Maven** → **Existing Maven Projects** and click **Next**.
  - 1.4. Click **Browse** and choose **/home/student/JB421/labs/throughput-files** and click **OK** to choose the Maven project. Click **Finish** to complete the import process in JBoss Developer Studio.  
A new project named throughput-files is listed in the **Project Explorer** view.
- ▶ 2. Review the route definition.  
In the **Project Explorer** view, expand **src/main/java** → **com.redhat.training.jb421**. Double-click the **ConcurrentRouteBuilder.java** file.

```
from("file://" + INPUT_FOLDER + "?delete=true")
 .bean("timerBean", "start")
 .unmarshal().bindy(BindyType.Csv, InventoryUpdate.class)
 .process(new InventoryUpdateProcessor())
 .process(new PartnerInventoryUpdateProcessor())
 .to("file:" + OUTPUT_FOLDER + "?fileName=${header.publisherId}/"
 + "inventory-reservation-${date:now:yyyy-MM-dd}.csv&fileExist=Append")
 .bean("timerBean", "stop");
```

Notice that in its current state the route has no concurrency optimizations.

- 3. Create test data using the provided script and run a baseline test without any concurrency using the Camel Maven plug-in.

- 3.1. Create the test data.

To execute the script to add the data, open a new terminal window, navigate to the **/home/student/JB421/labs/throughput-files** folder and run **./setup-data.sh** using the following commands:

```
[student@workstation ~]$ cd ~/JB421/labs/throughput-files/
[student@workstation throughput-files]$./setup-data.sh
Creating a batch of 1500 test inventory updates
```

- 3.2. Start the route and monitor the output.

In the same terminal window you used to create the test data, run the following command to start the Maven Camel plug-in.

```
[student@workstation throughput-files]$ mvn camel:run
```

Make sure you see output similar to the following, which has been trimmed for brevity.

```
...incoming] TimerBean INFO Item Number 100 Processing complete! Time elapsed: 7
seconds
...
...incoming] TimerBean INFO Item Number 1500 Processing complete! Time elapsed:
109 seconds
```

Running without any concurrency performs poorly, as each file must be completely processed before the next file can begin processing.

- 3.3. Stop the route.

Press **Ctrl+C** to terminate the route execution.

- 4. Update the route to use the **threads** DSL method with a custom fixed thread pool to improve performance.

Return to JBoss Developer Studio and open **ConcurrentRouteBuilder**. Edit the route definition to use multiple threads from a fixed thread pool.

Uncomment the **ExecutorService** instantiation as follows:

```
ExecutorService threadPool = Executors.newFixedThreadPool(10);
```

Update the route definition to match the following:

```
from("file://" + INPUT_FOLDER + "?delete=true")
 .bean("timerBean", "start")
 .threads().executorService(threadPool)
 .marshal().bindy(BindyType.Csv, InventoryUpdate.class)
 .process(new InventoryUpdateProcessor())
 .process(new PartnerInventoryUpdateProcessor())
 .to("file:" + OUTPUT_FOLDER + "?fileName=${header.publisherId}/"
 + "inventory-reservation-${date:now:yyyy-MM-dd}.csv&fileExist=Append")
 .bean("timerBean", "stop")
 .end();
```

Press **Ctrl+S** to save your changes.

► **5.** Recreate the test data and retest the route using the Camel Maven plug-in.

5.1. Recreate the test data.

Return to the terminal window and run **./setup-data.sh** using the following commands:

```
[student@workstation throughput-files]$./setup-data.sh
Creating a batch of 1500 test inventory updates
```

5.2. Run the route again using the Camel Maven plug-in. Monitor the output.

```
[student@workstation throughput-files]$ mvn camel:run
```

You should see output similar to the following which has been trimmed for brevity:

```
....thread-7] TimerBean INFO Item Number 100 Processing complete! Time elapsed: 0
seconds
...
....thread-6] TimerBean INFO Item Number 1500 Processing complete! Time elapsed:
11 seconds
```

5.3. Stop the route.

Press **Ctrl+C** to terminate the route execution.

► **6.** Further improve performance by moving one of the processors behind a wire tap.

Return to JBoss Developer Studio, and open **ConcurrentRouteBuilder** class and edit the route definition to use a **wireTap** method to move the execution of one of the processors into a separate thread, thereby removing its execution time from the time required to process each file.

Uncomment the direct route at the bottom of the route.

```
from("direct:partnerUpdate")
 .process(new PartnerInventoryUpdateProcessor());
```

Replace the existing **process** method, which uses the **PartnerInventoryUpdateProcessor**, with a **wireTap** method that sends to the direct route

Your route definition must match the following:

```
from("file://" + INPUT_FOLDER + "?delete=true")
 .bean("timerBean", "start")
 .threads().executorService(threadPool)
 .marshal().bindy(BindyType.Csv, InventoryUpdate.class)
 .process(new InventoryUpdateProcessor())
 .wireTap("direct:partnerUpdate")
 .to("file:" + OUTPUT_FOLDER + "?fileName=${header.publisherId}))"
 + "inventory-reservation-${date:now:yyyy-MM-dd}.csv&fileExist=Append")
 .bean("timerBean", "stop")
.end();

from("direct:partnerUpdate")
 .process(new PartnerInventoryUpdateProcessor());
```

► 7. Test the route performance with the addition of the wire tap EIP.

- 7.1. Recreate the test data.

Return to the terminal window you opened, and again run `./setup-data.sh` using the following commands:

```
[student@workstation throughput-files]$./setup-data.sh
Creating a batch of 1500 test inventory updates
```

- 7.2. Run the route again using the Camel Maven plug-in and monitor the output.

```
[student@workstation throughput-files]$ mvn camel:run
```

You should see output similar to the following, which has been trimmed for brevity:

```
...-thread-7] TimerBean INFO Item Number 100 Processing complete! Time elapsed: 0
seconds
...
...-thread-2] TimerBean INFO Item Number 1500 Processing complete! Time elapsed:
7 seconds
```

Performance is further improved now that the latency of the second processor is not included in the route execution but offloaded to a separate route.

- 7.3. Stop the route.

Press **Ctrl+C** to terminate the route execution.

► 8. Close the project.

In the **Project Explorer** view, right-click `throughput-files` and click **Close Project**.

This concludes the guided exercise.

## ► Lab

# Implementing Parallel Processing

## Performance Checklist

In this lab, you will be given an integration project that includes a Camel route that retrieves undelivered orders from the database, splits that order data into separate order line items, and then enriches those line items with customer and vendor information before writing each item to an XML file. Frequently, the number of order items needed to process is quite large. You will implement concurrency to improve the performance of the route.

## Outcomes

You should be able to improve route performance using a variety of techniques to implement concurrency.

## Before You Begin

A starter project is provided, including annotated (with JAXB and JPA) model classes and a preconfigured Camel context. The project includes the necessary data source configuration and a stubbed route builder.

- Run the following command to download the starter project used by this lab:

```
[student@workstation ~]$ lab implement-parallel setup
```

- Import the starter project, located at `/home/student/JB421/labs/implement-parallel`, into JBoss Developer Studio.

## Steps

- Inspect the provided starter project.

Examine the Spring configuration file `src/main/resources/META-INF/spring/bundle-camel-context.xml` and the components it defines, including a data source named `mysqlDataSource`.

Inspect the two processors `VendorLookupProcessor` and `CustomerLookupProcessor` from `com.redhat.training.jb421` package. They are used in the enrich routes to formulate SQL queries for the `jdbc` component.

Review the `com.redhat.training.jb421.DBLookupAggregationStrategy` class, which is used by the `enrich` DSL method to combine the enriched data from the JDBC lookups with the original message. This aggregation strategy is built to support both results from the vendor lookup and the customer lookup.

Open and review the existing route, without concurrency modifications, in the `com.redhat.training.jb421.EnrichRouteBuilder` class. Notice the `delay` option used in some of the routes. This allows concurrency optimizations to have a greater effect during the lab.

- Run a baseline test to get the current performance of the route without concurrency optimizations.

Open a new terminal window and run the **setup-data.sh** to create the sample order data. This creates 5 orders, each with 300 order items.

Execute the route to process the order data.

Stop the route execution.

3. Update the route definition in the **com.redhat.training.jb421.EnrichRouteBuilder** class. Modify the **split** DSL method to process the order items in parallel.
4. Enable parallel processing of the two **enrich** DSL methods using **multicast** DSL method and parallel processing.



#### Note

Hint: Because both **enrich** DSL methods use the same aggregation strategy, this strategy can also be used by multicast when aggregating the results from the two database lookups.

5. Run another test to get the current performance of the route with concurrency optimizations.  
Open a new terminal window and run the **setup-data.sh** script to create the sample order data. Five orders are created, each with 300 order items.  
Execute the route to process the order data.  
Stop the route execution.
6. Grade your work. Execute the following command:

```
[student@workstation implement-parallel]$ lab implement-parallel grade
```

If you do not get a **PASS** grade, review your work. The **~/JB421/labs/implement-parallel/grade.log** file contains the Maven messages produced by the grading process. This may be helpful in debugging any issues.

7. Clean up.  
Close the **implement-parallel** project in JBoss Developer Studio to conserve memory.  
This concludes the lab.

## ► Solution

# Implementing Parallel Processing

### Performance Checklist

In this lab, you will be given an integration project that includes a Camel route that retrieves undelivered orders from the database, splits that order data into separate order line items, and then enriches those line items with customer and vendor information before writing each item to an XML file. Frequently, the number of order items needed to process is quite large. You will implement concurrency to improve the performance of the route.

### Outcomes

You should be able to improve route performance using a variety of techniques to implement concurrency.

### Before You Begin

A starter project is provided, including annotated (with JAXB and JPA) model classes and a preconfigured Camel context. The project includes the necessary data source configuration and a stubbed route builder.

- Run the following command to download the starter project used by this lab:

```
[student@workstation ~]$ lab implement-parallel setup
```

- Import the starter project, located at **/home/student/JB421/labs/implement-parallel**, into JBoss Developer Studio.

### Steps

- Inspect the provided starter project.

Examine the Spring configuration file **src/main/resources/META-INF/spring/bundle-camel-context.xml** and the components it defines, including a data source named **mysqlDataSource**.

Inspect the two processors **VendorLookupProcessor** and **CustomerLookupProcessor** from **com.redhat.training.jb421** package. They are used in the enrich routes to formulate SQL queries for the **jdbc** component.

Review the **com.redhat.training.jb421.DBLookupAggregationStrategy** class, which is used by the **enrich** DSL method to combine the enriched data from the JDBC lookups with the original message. This aggregation strategy is built to support both results from the vendor lookup and the customer lookup.

Open and review the existing route, without concurrency modifications, in the **com.redhat.training.jb421.EnrichRouteBuilder** class. Notice the **delay** option used in some of the routes. This allows concurrency optimizations to have a greater effect during the lab.

- Run a baseline test to get the current performance of the route without concurrency optimizations.

Open a new terminal window and run the **setup-data.sh** to create the sample order data. This creates 5 orders, each with 300 order items.

```
[student@workstation ~]$ cd JB421/labs/implement-parallel/
[student@workstation implement-parallel]$./setup-data.sh
Creating a batch of 5 test orders with 300 order items each
Order 1 was created!
Order 2 was created!
Order 3 was created!
Order 4 was created!
Order 5 was created!
```

Execute the route to process the order data.

Start the route using the Maven Camel plug-in.

```
[student@workstation implement-parallel]$ mvn camel:run -DskipTests
```

The data should start processing immediately, and you see output similar to the following, which has been trimmed for brevity:

```
...jb421.model.Order] TimerBean INFO Order 1 Processing complete! Time elapsed: 19
seconds
...
...jb421.model.Order] TimerBean INFO Order 5 Processing complete! Time elapsed: 86
seconds
```

Stop the route execution.

In the same terminal window where you ran the Maven Camel plug-in, press **Ctrl+C** to stop the route.

3. Update the route definition in the **com.redhat.training.jb421.EnrichRouteBuilder** class. Modify the **split** DSL method to process the order items in parallel.

Add the **parallelProcessing** option to the **split** to allow the order items to process in parallel. Your route definition should match the following:

```
...
.wireTap("direct:updateOrder")
//TODO use parallel processing in the splitter
.split(simple("${body.getOrderItems()}")).parallelProcessing()
//TODO use multicast to replace two serial enrich calls
.enrich("direct:vendorLookupJDBC", new DBLookupAggregationStrategy())
...
```

4. Enable parallel processing of the two **enrich** DSL methods using **multicast** DSL method and parallel processing.

**Note**

Hint: Because both **enrich** DSL methods use the same aggregation strategy, this strategy can also be used by multicast when aggregating the results from the two database lookups.

Remove the two **enrich** DSL method invocations and replace them with a **multicast** DSL method which has parallel processing enabled and uses the **DBLookupAggregationStrategy** class for aggregation. Make sure your route definition matches the following:

```
from("jpa:com.redhat.training.jb421.model.Order?persistenceUnit=mysql"
 + "&consumeLockEntity=false&consumer.delay=10000"
 + "&consumer.namedQuery=getUndeliveredOrders&consumeDelete=false")
 .bean("timerBean", "start")
 .setHeader("order_id", simple("${body.getId()}"))
 .wireTap("direct:updateOrder")
 .split(simple("${body.getOrderItems()}")).parallelProcessing()
 .multicast(new DBLookupAggregationStrategy()).parallelProcessing()
 .to("direct:vendorLookupJDBC")
 .to("direct:customerLookupJDBC")
 .end()
 .setHeader("vendor_id", simple("${body.getVendorId()}"))
 .marshal().jaxb()
 .to("file://" + OUTPUT_FOLDER + "?fileName=item-${in.header.order_id}-
 + "${in.header.vendor_id}-${date:now:yyyy_MM_dd_hh_mm_ss_SS}.xml")
 .bean("timerBean", "stop")
.end();
```

5. Run another test to get the current performance of the route with concurrency optimizations.

Open a new terminal window and run the **setup-data.sh** script to create the sample order data. Five orders are created, each with 300 order items.

```
[student@workstation implement-parallel]$./setup-data.sh
Creating a batch of 5 test orders with 300 order items each
Order 1 was created!
Order 2 was created!
Order 3 was created!
Order 4 was created!
Order 5 was created!
```

Execute the route to process the order data.

Start the route using the Maven Camel plug-in.

```
[student@workstation implement-parallel]$ mvn camel:run -DskipTests
```

The data starts processing immediately. Watch for output similar to the following (trimmed for brevity):

```
... thread #2 - Split] TimerBean INFO Order 1 Processing complete! Time elapsed:
2 seconds
... thread #6 - Split] TimerBean INFO Order 2 Processing complete! Time elapsed:
4 seconds
... thread #8 - Split] TimerBean INFO Order 3 Processing complete! Time elapsed:
7 seconds
... thread #6 - Split] TimerBean INFO Order 4 Processing complete! Time elapsed:
9 seconds
... thread #11 - Split] TimerBean INFO Order 5 Processing complete! Time elapsed:
12 seconds
```

Stop the route execution.

In the same terminal window where you ran the Camel Maven plug-in, press **Ctrl+C** to stop the route.

6. Grade your work. Execute the following command:

```
[student@workstation implement-parallel]$ lab implement-parallel grade
```

If you do not get a **PASS** grade, review your work. The **~/JB421/labs/implement-parallel/grade.log** file contains the Maven messages produced by the grading process. This may be helpful in debugging any issues.

7. Clean up.

Close the **implement-parallel** project in JBoss Developer Studio to conserve memory.

This concludes the lab.

# Summary

---

In this chapter, you learned:

- Threads and thread pools can be leveraged in the following parts of Camel:
  - Several of the EIP patterns, including splitter and aggregator
  - The SEDA component
  - The **threads** DSL method
  - Several Camel components, including JMS and Jetty
- The **threads** DSL method is used to turn a synchronous route into an asynchronous route.
- The splitter EIP provides two solutions for working with threads:
  - **parallelProcessing** DSL method
  - Custom thread pool
- Stage event-driven architecture (SEDA) is an approach to software development for event-driven applications, which breaks the functionality of an application into a set of stages connected by queues.
- The **vm** component supports communication across **CamelContext** instances.
- Deadlocks occur when concurrent tasks attempt to modify the same data in a database.
- The wire tap pattern spawns a copy of the exchange and creates a new thread to process it.



## Chapter 10

# Creating Microservices with Red Hat Fuse

### Goal

Create microservices from Camel routes.

### Objectives

- Describe microservices and how they can be built and packaged using Red Hat Fuse.
- Develop microservices by defining Camel routes.
- Design a microservice so that it can handle potential failures in its interactions with other services.

### Sections

- Describing Red Hat Fuse Microservices (and Quiz)
- Developing Microservices with Camel Routes (and Guided Exercise)
- Designing Microservices for Failures (and Guided Exercise)

### Lab

Creating Microservices with Red Hat Fuse

# Describing Red Hat Fuse Microservices

## Objectives

After completing this section, students should be able to describe microservices and how they can be built and packaged using Red Hat Fuse.

## Defining Microservice Architecture

Microservice architecture is a method of dividing a traditional monolithic or single deployment enterprise application into a set of small, modular services. Each service is built around a specific business domain, such as customer information or inventory pricing. Microservices can be written in different programming languages and they can even be managed and deployed using completely different tools. For example, a developer builds a microservice for a specific business domain or problem, such as inventory tracking, and then develops a web application that calls the microservice to display information and provide user actions relevant to that business domain.

Microservices can be independently managed and deployed using automation. They run in unique processes and communicate using a lightweight mechanism, usually HTTP calls to a REST API. This modularity means that microservices are good for running within a cloud environment, and fit naturally into containerized deployment platforms like OpenShift Container Platform.

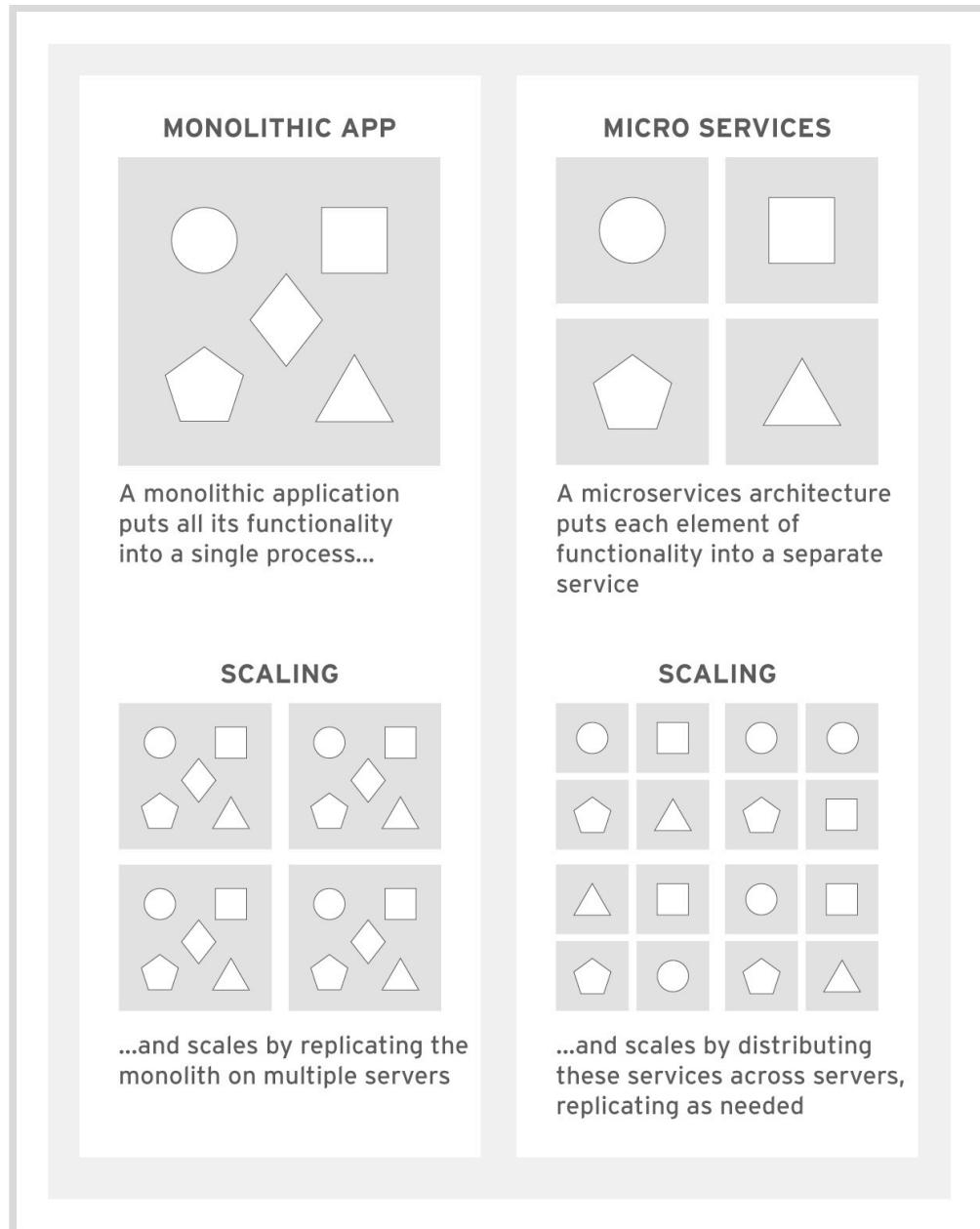
Additionally, a microservice generally has the following characteristics:

- Modeled around a single business problem or domain
- Implements its own business logic, persistent storage, and external collaboration
- Has an individually published contract, also known as an API
- Capable of running in isolation
- Independent and loosely-coupled from other services
- Easily replaced or upgraded
- Scaled and deployed independently of other services

## Comparing Microservices and Monolithic Applications

Monolithic applications are large enterprise applications developed and deployed as a single project. These applications are written in a single programming language, managed by a single team of developers who test, deploy, and release as a single artifact. They are complex to build, test, and migrate into production. Even though monolithic applications are divided into tiers or layers, the applications are almost always packaged into a single WAR or EAR file.

Because monolithic applications are deployed as a single app, all of the memory, and other resources, are limited by the hardware. Monolithic applications must scale by replicating the entire application on multiple servers. Additionally, these applications tend to be more complex and tightly coupled, making them more difficult to maintain and update.

**Figure 10.1: Comparing monolithic applications and microservices**

Compared to monolithic applications, microservices are better organized, smaller, more loosely-coupled, and they are independently developed, tested, and deployed. Because microservices can be released independently, the time required to fix bugs or to add new features is much shorter, and changes can be deployed to production more efficiently. Additionally, because microservices are small and stateless, they can scale much easier.

## Reviewing the Principles for Successful Adoption

As more companies attempt to move towards using a microservice architecture, a few important principles have become critical to the success of microservice-based development. These principles include:

- **Reorganizing to DevOps:** Microservices teams are responsible for the entire lifecycle of their microservice, from development to operation. DevOps teams can work on their individual

product at their own pace. The principle of "you build it, you own it" provides autonomy and speed for delivery teams.

- **Packaging the service as a container:** A container is a set of isolated processes that provides all of the necessary dependencies to run an application. By using tools like Docker or LXC, the deployment platform for the microservice can be packaged and deployed as a container. Using containers allows developers to have control and confidence over the exact runtime environment the service uses.
- **Using an elastic infrastructure:** Provide elastic infrastructure to meet the on-demand requirements of the microservices. OpenShift Container Platform automates the provisioning, management, and scaling of containerized applications and provides on-demand scaling by providing automatic horizontal pod scaling. As load increases on certain microservices, OpenShift automatically increases available resources for those microservices.
- **Automating Processes:** Use scripting or other tools to automate everything related to the provisioning of the service infrastructure and the deployment of the service. Manage these automation scripts or other resources in your version control system just like any other application code. *Infrastructure As Code* (IAC) is an approach that uses descriptive language to code versatile and adaptive provisioning and deployment processes using tools such as Ansible, Puppet, or Jenkins.
- **Continuous integration and delivery pipeline:** Continuous integration and delivery (CI/CD) is the practice of building, testing, and delivering software as the code is developed. A key to continuous delivery is to automate the entire pipeline, including code check-in, builds, tests, and deployments across multiple environments. Make sure software is always deployable. When the build is broken, make sure that fixing the broken code takes priority over other tasks. Tools like Jenkins can be used for CI/CD.

## Understanding the Complexities of Microservice Architecture

Designing and building a large scale distributed application with many microservices is occasionally challenging and complex. These challenges tend to arise when planning and managing a large number of small components that must interact with each other. Testing an individual microservice is simple, but when multiple services are dependent upon each other, they must be tested together. Furthermore, introducing dependencies between microservices can cause the system to communicate in an unpredictable ways when one of the services does not behave as expected.

Additionally, maintaining data consistency may be difficult in a large distributed system. Every microservice can have its own persistent storage, so shared entity data formats cannot be changed without changing all of the microservices. For example, if two microservices are both persisting instances of the entity **Employee** and the data model for **Employee** changes, both services need to update the entity.

Tracing and monitoring is also more complex for microservice-based applications. Instead of monitoring a single application server, microservices are typically running on multiple servers and writing to multiple log files. Collecting all the log information and collecting it in one place to analyze and visualize efficiently is a complex process.

Finally, securing microservice applications can be a challenging and complex task. Users authenticate with the UI layer just like a monolithic application, but individual services need to be protected by authentication and authorization as well. The added communication between services in a microservice-based application creates more opportunity for security risks, and requires more points of authentication.

# Understanding the Principles of a Resilient Microservice Architecture

## Deploy Independently on a Lightweight Runtime

Deploying services independently is one of the most important principles in microservices architecture. Microservices are designed to be stateless, and state is maintained outside the application in databases or data grids. This is done so that a microservice can scale easily and deploy independently of other services.

Microservices are good candidates for lightweight, embedded runtimes over full-fledged application servers. Some of the desired runtimes for microservices are Eclipse Vert.x, Wildfly Swarm, and Sprint Boot. Additionally, JBoss Enterprise Application Platform (EAP) is considered lightweight as most of EAP's application server components are started on-demand.

Microservices are packaged and delivered as containers that include the runtime and its dependencies. To support continuous integration and delivery of the microservices, fully-automated build and deployment pipelines are required. Microservices provide Blue-Green deployment for quick rollback to the last working version. *Blue-Green deployment* is a technique used to reduce downtime by running two identical production environments called Blue and Green. Only one environment is active at a time.

## Design for Failure

Distributed applications can fail due to application, hardware, or network failures. Applications need to be designed to withstand such failure. Various design patterns can be incorporated to make microservices applications fault-tolerant.

- High availability must be provided for individual microservices for failure management.
- Use the *circuit breaker* design pattern to prevent a service or network failure from cascading to other services. The fault tolerance section of this chapter covers this pattern is covered in more detail.
- Use the *retry* design pattern to recover from transient issues and provide a more resilient service. The fault tolerance section of this chapter covers this pattern is covered in more detail.

Applications require extensive testing to assess the effects of a variety of different types of system failure on the end user experience. Real-time monitoring is required to detect these failures quickly and alert developers. Additionally, use a self-healing infrastructure like Kubernetes and OpenShift Container Platform to leverage readiness and liveness checks which monitor the state of running services, and allow the deployment platform to act on failures automatically. These approaches are discussed in detail later in the course.

## Highly configurable

A common practice of quality software development is to never hard code environment-specific configuration, such as host names, user names, or passwords into your application's source code. Instead, best practice is to keep this configuration outside of your application in external configuration files or environment variables. This gives your application portability across many different environments without the need to recompile, or repackage the source code. As container platforms like OpenShift become more popular this approach is now a necessity in order to safely move your container image that contains your applications between environments (test, staging, pre-prod, production) by dynamically applying environment specific configuration to the container image.

## Packaging and Running Camel-based Microservices

Camel supports a number of microservice-ready runtimes. Camel is just a library you include with your application code in your JVM runtime, it can run basically anywhere that provides a JVM. This includes, but is not limited to, the following standalone microservice deployment options supported by Red Hat Fuse:

### Fat JAR deployment on Spring Boot

Spring Boot allows running Camel with a lightweight embedded application server. To deploy to this runtime, package your application as a JAR file and run it directly inside the Java Virtual Machine (JVM).

### Java EE Web Archive (WAR) deployment on JBoss EAP

You can deploy Fuse applications on Red Hat JBoss Enterprise Application Platform (JBoss EAP), after installing the Fuse on EAP package into the JBoss EAP container.

### OSGI Deployment on Apache Karaf

Apache Karaf is an OSGi-based runtime container for deploying and managing bundles.

Apache Karaf also provides native operating system integration, and can be integrated into the operating system as a service so that the life cycle is bound to the operating system.

In addition, Red Hat Fuse on OpenShift allows you to take each of the standalone microservice deployment options listed above and run containerize them and run them in your OpenShift cluster. An application deployment with Fuse on OpenShift consists of an application and all required runtime components packaged inside a container image. Applications are not deployed to a runtime as with Fuse Standalone, the application image itself is a complete runtime environment deployed and managed through OpenShift. This deployment option will be covered in more detail in the next chapter.



### References

#### What are Microservices?

<http://microservices.io/>

## ► Quiz

# Quiz: Describing Microservices

Choose the correct answers to the following questions:

► 1. **Which of the following statements about monolithic applications is true?**

- a. A monolithic application is always packaged into a **.tar.gz** file.
- b. A monolithic application is a lightweight application that is divided into a multiple services and a client tier.
- c. Monolithic applications are often complex to build and can be difficult to maintain.
- d. Individual pieces of a monolithic application are typically developed by individual small autonomous teams.

► 2. **Which two of the following statements about microservices are true? (Choose two.)**

- a. Microservices are managed centrally, and typically share one common database.
- b. Microservices are typically deployed on full-fledged application servers that manage their lifecycle.
- c. Microservices are self-contained, independent services.
- d. Microservices within a single organization can be built using different programming languages.

► 3. **A company is considering using microservice architecture for its new grocery delivery application. They have two small teams to manage IT development and operations in 7 different states. Which of the following two statements support their decision in favor of microservices? (Choose two.)**

- a. All order operations must be centralized in a single data center.
- b. Microservices can be built for each business operation.
- c. Microservices must be deployed manually to ensure there are no errors.
- d. Individual services in a microservice-based application can go live quickly by using continuous integration and delivery.

► 4. **Which of the two following statements about microservices architecture are correct? (Choose two.)**

- a. Microservice applications are required to be deployed on the same physical host.
- b. Microservices architecture supports high availability of individual microservices.
- c. Microservices cannot be used if a company is embracing DevOps.
- d. Microservices are designed using a bounded context that can communicate with other bounded contexts.

► **5. Which of the following runtimes can you use to deploy Camel-based microservices?**

(Choose all that apply.)

- a. Red Hat Fuse containerized deployment on OpenShift
- b. OSGI bundle deployment on Apache Karaf
- c. Python script deployment on RHEL
- d. Fat JAR deployment on WildFly Swarm or Spring Boot
- e. Swift deployment on iOS devices

## ► Solution

# Quiz: Describing Microservices

Choose the correct answers to the following questions:

► 1. **Which of the following statements about monolithic applications is true?**

- a. A monolithic application is always packaged into a `.tar.gz` file.
- b. A monolithic application is a lightweight application that is divided into a multiple services and a client tier.
- c. Monolithic applications are often complex to build and can be difficult to maintain.
- d. Individual pieces of a monolithic application are typically developed by individual small autonomous teams.

► 2. **Which two of the following statements about microservices are true? (Choose two.)**

- a. Microservices are managed centrally, and typically share one common database.
- b. Microservices are typically deployed on full-fledged application servers that manage their lifecycle.
- c. Microservices are self-contained, independent services.
- d. Microservices within a single organization can be built using different programming languages.

► 3. **A company is considering using microservice architecture for its new grocery delivery application. They have two small teams to manage IT development and operations in 7 different states. Which of the following two statements support their decision in favor of microservices? (Choose two.)**

- a. All order operations must be centralized in a single data center.
- b. Microservices can be built for each business operation.
- c. Microservices must be deployed manually to ensure there are no errors.
- d. Individual services in a microservice-based application can go live quickly by using continuous integration and delivery.

► 4. **Which of the two following statements about microservices architecture are correct? (Choose two.)**

- a. Microservice applications are required to be deployed on the same physical host.
- b. Microservices architecture supports high availability of individual microservices.
- c. Microservices cannot be used if a company is embracing DevOps.
- d. Microservices are designed using a bounded context that can communicate with other bounded contexts.

► **5. Which of the following runtimes can you use to deploy Camel-based microservices?**

(Choose all that apply.)

- a. Red Hat Fuse containerized deployment on OpenShift
- b. OSGI bundle deployment on Apache Karaf
- c. Python script deployment on RHEL
- d. Fat JAR deployment on WildFly Swarm or Spring Boot
- e. Swift deployment on iOS devices

# Developing Microservices with Camel Routes

---

## Objectives

After completing this section, students should be able to develop microservices by defining Camel routes.

## Building a Microservice using Camel's REST DSL

Using the Camel REST DSL to build lightweight REST services is a simple solution for developing your microservices. The REST DSL topic is covered in-depth in a previous chapter. As a review, examine the following code sample which includes a basic microservice defined using Camel's REST DSL and deployed inside a Spring Boot application:

```
public class CartRoute extends RouteBuilder {

 public void configure() throws Exception {
 restConfiguration("jetty") 1
 .port("{{port}}") 2
 .contextPath("api") 3
 .bindingMode(RestBindingMode.json) 4
 .dataFormatProperty("disableFeatures", "FAIL_ON_EMPTY_BEANS") 5
 .apiContextPath("api-doc") 6
 .enableCORS(true); 7

 rest("/cart") 8
 .consumes("application/json") 9
 .produces("application/json") 10
 .get() 11
 .outType(CartDto[].class) 12
 .description("Returns the items currently in the shopping cart") 13
 .to("bean:cart?method=getItems") 14
 .post() 15
 .type(CartDto.class)
 .description("Adds the item to the shopping cart")
 .to("bean:cart?method=addItem")
 .delete()
 .description("Removes the item from the shopping cart")
 .param().name("itemId").description("Id of item to remove").endParam() 16
 .to("bean:cart?method=removeItem");
 }
}
```

- 1** Configure all REST DSL endpoints for a given **RouteBuilder** instance. This element allows you to specify which server implementation you want to use for the REST services and to customize the behavior of the embedded web server created by the REST DSL.
- 2** Set the port where the web server listens for incoming requests.
- 3** Set the root context path for all REST services created by this **RouteBuilder**.

- ④ Specify how the server attempts to bind incoming and outgoing data, in this case all the REST services must communicate using JSON, and the web server will automatically.
- ⑤ Set a custom property specific to the data format, in this case JSON, that these services will use when attempting to marshal incoming data from JSON to Java objects, and vice versa.
- ⑥ Specify the context path where the built-in Swagger API documentation is hosted on the web server.
- ⑦ Enable *Cross-Origin Resource Sharing* (CORS) to allow the web server to make requests to external servers.
- ⑧ Create a REST service endpoint at the context path **/cart**.
- ⑨ Specify that the **/cart** endpoint consumes JSON data for the Swagger documentation.
- ⑩ Specify that the **/cart** endpoint provides JSON data for the Swagger documentation.
- ⑪ Map HTTP GET requests to the **/cart** endpoint to execute this route.
- ⑫ Specify the return type of the **/cart** service as an instance of **CartDTO[]** in the Swagger documentation.
- ⑬ Specify a description of the **/cart** service to be displayed in the Swagger documentation.
- ⑭ Call a bean method from the REST DSL route to complete the route execution.
- ⑮ Map HTTP POST requests to the **/cart** endpoint to execute this route.
- ⑯ Document the **itemId** parameter with a description to be displayed in the Swagger documentation.

By using Camel to develop your microservice endpoints, you have access to the wealth of components provided by Camel to implement the necessary business logic and integrations in your microservices as well as an easy way to create a RESTful interface for your clients to use.

Additionally, Camel's portability and compatibility with multiple run times provides an advantage when choosing the stack on top of which you wish to build your microservices. Camel's support for lightweight runtimes such as Spring Boot that are easily containerized provides a simple path to scalable, highly-available microservices deployments using a container deployment platform such as OpenShift.

## Implementing a REST Client to Communicate with Other Microservices

When adopting a microservices architecture, each service is responsible for providing functionality to other collaborators. This distributed design typically includes lots of intra-service communication. When developing a microservice that needs to communicate with other REST services over HTTP, you need a mechanism to create an HTTP-based REST client.

There are two main approaches to take when creating a REST client in your Camel microservice. You can use the **camel-**http4**** component to program the client directly in your Camel route using Java DSL, or you can leverage Camel's bean functionality and put the REST client code inside a bean that you call from your Camel route.

## Creating a REST Client in a Camel Route Using the Camel HTTP4 component

The **http4** component provides HTTP based endpoints for calling external HTTP resources (as a client to call external servers using HTTP). To use the **http4** component in your Camel route, add the following dependency to your **pom.xml** for this component:

```
<dependency>
 <groupId>org.apache.camel</groupId>
 <artifactId>camel-http4</artifactId>
</dependency>
```

The URI format for the **camel-`http4`** component is similar to the HTTP URI you would use to connect to the service directly:

```
http4:hostname[:port][/path][?options]
```

By default, if no port is specified, then port 80 is used for HTTP and port 443 for HTTPS URLs. You can also append options to the URL using the format `?option=value&option2=value2&...`.

If you are also using the REST DSL as the consumer endpoint before you use the **camel-`http4`** component as a producer to call an external service, it is important to remember some of the shared Exchange headers that are used by both components. You may need to override or update these header values as they will contain the values from the incoming HTTP request and may not match what you want to send externally. Some of the common header values you should watch for are:

- **Exchange.HTTP\_URI**
- **Exchange.HTTP\_PATH**
- **Exchange.HTTP\_QUERY**

## Creating a REST Client in a Bean Using a RestTemplate

If you prefer, you can put your REST client code inside a bean, and map the different external endpoints you need to communicate with to different methods on that bean. By using a bean, you give yourself access to all the Java APIs you are used to, and you are no longer limited to using Camel components exclusively.

The **RestTemplate** provided by Spring is a simple to use REST client that makes it simple to communicate with external services over HTTP using REST. Using a **RestTemplate** is very simple, the following example shows a basic HTTP GET request:

```
RestTemplate restTemplate = new RestTemplate();
String resourceUrl = "http://localhost:8080/spring-rest/endpoint";
ResponseEntity<String> response = restTemplate.getForEntity(resourceUrl + "/1",
 String.class);
if(response.getStatusCode() != 200){
 //TODO handle error
} else{
 return response;
```

Notice that this provides you with full access to the HTTP response, allowing you to check the status code to make sure the operation was actually successful, or work with the actual body of the response:

```
ObjectMapper mapper = new ObjectMapper();
JsonNode root = mapper.readTree(response.getBody());
JsonNode name = root.path("name");
```

## Using a Health Endpoint to Monitor a Microservice

Another consequence of moving to a distributed microservices architecture is that monitoring your services becomes harder as there are many more running deployments to monitor. Therefore it is important to consider simple health checks so that you can tell when one of your microservices

is not functioning properly and act accordingly. Health check endpoints are also critical to deploying on a container management platform like OpenShift as they are leveraged by the platform to determine whether or not a given container is healthy. OpenShift can automatically restart unhealthy containers as a potential resolution to intermittent issues.

Spring Boot offers a simple solution for creating health endpoints in your microservice applications. This section covers this approach.

## Creating Health Endpoints using the Spring Boot Actuator Starter

You can add the **spring-boot-actuator** module to an existing Spring Boot application using the following dependency:

```
<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

Actuator creates several HTTP endpoints to let you monitor and interact with your application. Some examples of endpoints included by the Actuator:

- a **/health** endpoint that provides basic information about the application's health
- a **/metrics** endpoint that includes several useful metrics information like JVM memory used, system CPU usage, open files, and more
- a **/loggers** endpoint that shows application's logs and also lets you change the log level at runtime

Note that, every actuator endpoint can be explicitly enabled and disabled. Moreover, the endpoints also need to be exposed over HTTP or JMX to make them remotely accessible. By default, only the **health** and **info** endpoints are exposed over HTTP. You can enable or disable an actuator endpoint by setting the property **management.endpoint.<id>.enabled** to **true** or **false** (where **id** is the identifier for the endpoint).

The **health** endpoint only shows a simple **UP** or **DOWN** status. To get the complete details including the status of every health indicator that was checked as part of the health checkup process, add the following property in the **application.properties** file:

```
endpoint.health.show-details=always
```

You can also create a custom health indicator by implementing the **HealthIndicator** interface:

```
@Component ①
public class DatabaseHealthCheck implements HealthIndicator ② {

 @Autowired
 private EntityManagerFactory entityManagerFactory;

 @Override
 public Health health() ③ {
 try {
 EntityManager entityManager = entityManagerFactory.createEntityManager();
```

```
Query q = entityManager.createNativeQuery("select 1");
q.getFirstResult();
//TODO return status of UP
return null;
}catch(Exception e) {
 //TODO return status of DOWN
 return null;
}

}
```

- ➊ This class is annotated with **@Component** so that Spring automatically discovers and instantiates it at run time.
- ➋ This class implements the **HealthIndicator** interface, which the Spring Boot Actuator starter provides.
- ➌ The **HealthIndicator** interface requires a single method named **health()**.



## References

### Camel HTTP4 component

<https://github.com/apache/camel/blob/master/components/camel-http4/src/main/docs/http4-component.adoc>

### Spring Boot Actuator

<https://spring.io/guides/gs/actuator-service/>

## ► Guided Exercise

# Developing Microservices with Camel Routes

In this exercise, you will implement and execute two Camel-based microservices.

## Outcomes

You should be able to:

- Implement a REST client using the **http** Camel component to allow one microservice to call the other
- Configure the Spring Boot Actuator starter to provide a **/health** endpoint to both of the microservices
- Develop a custom health check to verify connectivity to the database

## Before You Begin

A pair of starter projects are provided for you. They both include a **CamelContext** configured in a Spring configuration file, and a **RouteBuilder** subclass.

Run the following command to download the starter project and data files used by this exercise:

```
[student@workstation ~]$ lab camel-microservices setup
```

## Steps

- 1. Open JBoss Developer Studio and import the starter Maven projects.
- 1.1. Open JBoss Developer Studio (**Applications** → **Programming** → **JBoss Developer Studio**). JBoss Developer Studio requests a default workspace. Leave the default configuration (**/home/student/workspace**) and click **Launch**.
  - 1.2. Import the Maven project by selecting the JBoss Developer Studio main menu **File** → **Import**.
  - 1.3. From the **Import** dialog box, select **Maven** → **Existing Maven Projects** and click **Next >**.
  - 1.4. Click **Browse** and choose **/home/student/JB421/labs/camel-microservices/hola-service**. Click **OK** to choose the Maven project. Click **Finish** to complete the import process in JBoss Developer Studio.
- A new project named **hola-service** is listed in the **Project Explorer** view.

**Note**

This project imports with errors, you will resolve these errors in the subsequent steps of the exercise.

- 1.5. Import the Maven project by selecting the JBoss Developer Studio main menu **File** → **Import**.
  - 1.6. From the **Import** dialog box, select **Maven** → **Existing Maven Projects** and click **Next >**.
  - 1.7. Click **Browse** and choose **/home/student/JB421/labs/camel-microservices/aloha-service** and click **OK** to choose the Maven project. Click **Finish** to complete the import process in JBoss Developer Studio.  
A new project named aloha-service is listed in the **Project Explorer** view.
- 2. Inspect the existing code for the aloha-service microservice.
- 2.1. Review the Spring Boot application definition.  
In the **Project Explorer** view, expand the **aloha-service** → **src/main/java/** → **com/redhat/training/jb421** package, and double click the **Application.java** file to open it for editing:
- ```
@SpringBootApplication
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```
- A simple Spring Boot application is defined, with no additional configuration.
- 2.2. Review the Spring Boot application properties.
In the **Project Explorer** view, expand the **aloha-service** → **src/main/resources/** directory, and double click the **application.properties** file to open it for editing:

```
logging.config=classpath:logback.xml

# the options from org.apache.camel.spring.boot.CamelConfigurationProperties can
# be configured here
camel.springboot.name=AlohaService

# lets listen on all ports to ensure we can be invoked from the pod IP
server.address=0.0.0.0 ①
management.address=0.0.0.0 ②

# lets use a different management port
server.port=8081 ③
management.port=8181 ④
```

```
# disable all management endpoints except health
endpoints.enabled = false ⑤
endpoints.health.enabled = true ⑥
```

- ① Configure the Spring Boot server to bind to the **0.0.0.0** local address.
- ② Configure the Spring Boot management server to bind to the **0.0.0.0** local address.
- ③ Configure the Spring Boot server to bind to the port 8081
- ④ Configure the Spring Boot management server to bind to the port 8181
- ⑤ Disable all management endpoints.
- ⑥ Enable the **/health** endpoint at the management address.

- 2.3. Review the Camel route that defines the REST endpoints for the aloha-service using REST DSL.

In the **Project Explorer** view, expand the **aloha-service → src/main/java/ → com/redhat/training/jb421** package, and double click the **RestRouteBuilder.java** file to open it for editing:

```
package com.redhat.training.jb421;

import org.springframework.stereotype.Component;
import org.apache.camel.builder.RouteBuilder;

@Component
public class RestRouteBuilder extends RouteBuilder {

    @Override
    public void configure() throws Exception {

        rest("/aloha")
            .get("{name}")
            .produces("application/json")
            .to("direct:sayHello");

        from("direct:sayHello").routeId("HelloREST")
            .setBody().simple("{\n                + " greeting: Aloha, ${header.name}\n                + " server: " + System.getenv("HOSTNAME") + "\n                + "}\n            ");
    }
}
```

When you call the aloha-service using an HTTP GET request, this service replies:

```
{
  greeting: Aloha, Developer,
  server: workstation.lab.example.com
}
```

- 3. Inspect the existing code for the hola-service microservice.

- 3.1. Review the Spring Boot application definition.

In the **Project Explorer** view, expand the **hola-service** → **src/main/java/** → **com/redhat/training/jb421** package, and double click the **Application.java** file to open it for editing:

```
@SpringBootApplication  
@ImportResource({"classpath:spring/camel-context.xml"})  
public class Application {  
  
    public static void main(String[] args) {  
        SpringApplication.run(Application.class, args);  
    }  
  
}
```

A simple Spring Boot application is defined, with some additional configuration in the **spring/camel-context.xml** file that defines the datasource used by this service.

3.2. Review the Spring Boot application properties.

In the **Project Explorer** view, expand the **hola-service** → **src/main/resources/** directory, and double click the **application.properties** file to open it for editing:

```
logging.config=classpath:logback.xml  
  
# the options from org.apache.camel.spring.boot.CamelConfigurationProperties can  
# be configured here  
camel.springboot.name=HolaService  
  
# lets listen on all ports to ensure we can be invoked from the pod IP  
server.address=0.0.0.0 ①  
management.address=0.0.0.0 ②  
  
# lets use a different management port  
server.port=8082 ③  
management.port=8182 ④  
  
# disable all management endpoints except health  
endpoints.enabled = false ⑤  
endpoints.health.enabled = true ⑥  
  
alohaHost = localhost ⑦  
alohaPort = 8081 ⑧
```

- ① Configure the Spring Boot server to bind to the **0.0.0.0** local address.
- ② Configure the Spring Boot management server to bind to the **0.0.0.0** local address.
- ③ Configure the Spring Boot server to bind to the port 8082
- ④ Configure the Spring Boot management server to bind to the port 8182
- ⑤ Disable all management endpoints.
- ⑥ Enable the **/health** endpoint at the management address.
- ⑦ Configure the host where the aloha-service is deployed.
- ⑧ Configure the port the aloha-service is listening on.

3.3. Review the Camel route that defines the REST endpoints for the aloha-service using REST DSL.

In the **Project Explorer** view, expand the **hola-service** → **src/main/java/** → **com/redhat/training/jb421** package, and double-click the **RestRouteBuilder.java** file to open it for editing:

```

@Component
public class RestRouteBuilder extends RouteBuilder {

    //TODO Inject value from configuration

    private String alohaHost;

    //TODO Inject value from configuration

    private String alohaPort;

    @Override
    public void configure() throws Exception {

        rest("/hola")
            .get("{name}")
            .produces("application/json")
            .to("direct:sayHello");

        rest("/hola-chained")
            .get("{name}")
            .produces("application/json")
            .to("direct:callAloha");

        from("direct:callAloha")
            //TODO remove header Exchange.HTTP_URI

            //TODO set header Exchange.HTTP_PATH to the ${header.name} value

            .to("mock:"+alohaHost +":"+alohaPort+"/camel/aloha");

        from("direct:sayHello").routeId("HelloREST")
            .setBody().simple("\n"
                + " greeting: Hola, ${header.name}\n"
                + " server: " + System.getenv("HOSTNAME") + "\n"
                + "}\n");
    }
}

```

Ignore the **TODO** comments and the **hola-chained** endpoint for now. You will implement them in a subsequent step.

► 4. Configure the Spring Boot actuator starter in both of the projects **pom.xml** files.

- 4.1. In the **Project Explorer** view, expand the **aloha-service** project, double-click the **pom.xml** file, and switch to the **pom.xml** tab.
- 4.2. Locate the comment **TODO Add Spring Boot Actuator Starter** and insert the following dependency definition after the comment:

```
<!--TODO Add Spring Boot Actuator Starter -->
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

The version for this dependency is inherited by the **fuse-springboot-bom** defined higher in the **pom.xml** file.

- 4.3. Press **Ctrl+S** to save your changes to the aloha-service's **pom.xml** file.
- 4.4. In the **Project Explorer** view, expand the hola-service project, double-click the **pom.xml** file, and switch to the **pom.xml** tab.
- 4.5. Locate the comment **TODO Add Spring Boot Actuator Starter** and insert the following dependency definition after the comment:

```
<!--TODO Add Spring Boot Actuator Starter -->
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

The version for this dependency is inherited by the **fuse-springboot-bom** defined higher in the **pom.xml** file.

- 4.6. Press **Ctrl+S** to save your changes to the hola-service's **pom.xml** file.
 5. Add a new endpoint to the hola-service at the path **/camel/hola-chained/Name**. This endpoint must call the aloha-service when it is invoked and return the response.
- 5.1. Use the **camel-http4** component as a REST client to call the aloha-service. To use this component, add a dependency on the **camel-http4** component in the hola-service's **pom.xml** file.
- In the **Project Explorer** view, expand the hola-service project, double-click the **pom.xml** file, and switch to the **pom.xml** tab.
- Locate the comment **TODO Add camel-http4 component** and add the following dependency definition:

```
<!--TODO Add camel-http4 component-->
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-http4</artifactId>
</dependency>
```

- 5.2. Press **Ctrl+S** to save your changes to the hola-service's **pom.xml** file.
- 5.3. Return to editing the **hola-service/src/main/java/com/redhat/training/jb421/RestRouteBuilder.java** file.
- 5.4. Inject the values for the hostname and port for the aloha-service using the **@Value** annotation:

```
//TODO Inject value from configuration
@Value("${alohaHost}")
private String alohaHost;

//TODO Inject value from configuration
@Value("${alohaPort}")
private String alohaPort;
```

5.5. Finish the **direct:callAloha** route definition.

The **hola-chained** Camel route originally starts with a REST DSL endpoint. Because of this, it is necessary to modify some of the exchange headers that are set on the exchange by the REST DSL. The **camel-http4** component shares a few of the headers set by the REST DSL, so we need to make sure their values are set properly for the outgoing call to the aloha-service instead of being set for the incoming REST call.

Unset the header value **Exchange . HTTP _URI**

```
from("direct:callAloha")
//TODO remove header Exchange.HTTP_URI
.removeHeader(Exchange.HTTP_URI)
```

Set the header value **Exchange . HTTP _PATH** using the **header . name** value that was passed into the **hola-chained** endpoint originally to pass the name onto the aloha-service.

```
from("direct:callAloha")
//TODO remove header Exchange.HTTP_URI
.removeHeader(Exchange.HTTP_URI)
//TODO set header Exchange.HTTP_PATH to the ${header.name} value
.setHeader(Exchange.HTTP_PATH,simple("${header.name}))
```

Finally, update the component from **mock** to **http4** in the route's producer:

```
from("direct:callAloha")
//TODO remove header Exchange.HTTP_URI
.removeHeader(Exchange.HTTP_URI)
//TODO set header Exchange.HTTP_PATH to the ${header.name} value
.setHeader(Exchange.HTTP_PATH,simple("${header.name}))
```

5.6. Press **Ctrl+S** to save your changes to the **RestRouteBuilder.java** file.

- ▶ 6. Add a custom health check to the hola-service which verifies the connection to the database.

- 6.1. In the **Project Explorer** view, expand the **hola-service → src/main/java/ → com/redhat/training/jb421** package, and double click the **DatabaseHealthCheck.java** file to open it for editing:

```

package com.redhat.training.jb421;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Query;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.actuate.health.Health;
import org.springframework.boot.actuate.health.HealthIndicator;
import org.springframework.stereotype.Component;

@Component
public class DatabaseHealthCheck implements HealthIndicator ①{

    @Autowired
    private EntityManagerFactory entityManagerFactory;

    @Override
    public Health health() ② {

        try {
            EntityManager entityManager = entityManagerFactory.createEntityManager();
            Query q = entityManager.createNativeQuery("select 1");
            q.getFirstResult();
            //TODO return status of UP
            return null;
        }catch(Exception e) {
            //TODO return status of DOWN
            return null;
        }
    }

}

```

- ① This class implements the **HealthIndicator** interface, which the Spring Boot Actuator starter provides.
- ② The **HealthIndicator** interface requires a single method named **health()**.

- 6.2. Update the return statements to include a status of **UP** or **DOWN** depending on whether or not an exception occurred connecting to the database:

```

try {
    EntityManager entityManager = entityManagerFactory.createEntityManager();
    Query q = entityManager.createNativeQuery("select 1");
    q.getFirstResult();
    //TODO return status of UP
    return Health.up().build();
}catch(Exception e) {
    //TODO return status of DOWN
    return Health.down(e).build();
}

```

6.3. Press **Ctrl+S** to save your changes to the **DatabaseHealthCheck.java** file.

- 7. Run the aloha-service Spring Boot application and verify that all expected endpoints are active.

- 7.1. Open a terminal, enter the project folder, and invoke the **package** Maven goal to build the aloha-service Spring Boot application's Fat JAR:

```
[student@workstation ~]$ cd ~/JB421/labs/camel-microservices/aloha-service
[student@workstation aloha-service]$ mvn package
...
[INFO] Scanning for projects...
[INFO]
[INFO] -----
[INFO] Building GE: Developing Microservices with Camel Routes - Aloha Service 1.0
[INFO] -----
[INFO] -----
[INFO] Building jar: /home/student/JB421/labs/camel-microservices/aloha-service/
target/aloha-service-1.0.jar
[INFO]
[INFO] --- spring-boot-maven-plugin:1.5.12.RELEASE:repackage (default) @ aloha-
service ---
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 6.648 s
...
```

- 7.2. Verify that a aloha-service Fat JAR exists in the **target** folder:

```
[student@workstation aloha-service]$ ls -sh target/*jar*
22M target/aloha-service-1.0.jar
```

- 7.3. Run the Spring Boot application using the **java -jar** command. Leave the application running, and notice the log messages from XML route:

```
[student@workstation aloha-service]$ java -jar target/aloha-service-1.0.jar
...
   .__ 
  / \ \ \ ' _ _ _ _ _ ( _ ) _ _ _ _ _ \ \ \ \ \
  ( ( ) \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \
  \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \
  \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \
  \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \
  :: Spring Boot ::      (v1.5.13.RELEASE)

...
16:52:57.384 [main] INFO  o.s.b.c.e.u.UndertowEmbeddedServletContainer - Undertow
started on port(s) 8081 (http)
```

```
16:52:57.388 [main] INFO c.redhat.training.jb421.Application - Started  
Application in 8.235 seconds (JVM running for 8.934)  
...
```

- 7.4. Open another terminal and verify that the Spring Boot application responds to HTTP requests.

Invoke the **curl** command, using **localhost** as the host name with port 8081 and the **/camel/aloha/Developer** resource URI:

```
[student@workstation aloha-service]$ curl -si \  
http://localhost:8081/camel/aloha/Developer  
HTTP/1.1 200 OK  
...  
{  
    greeting: Aloha, Developer  
    server: workstation.lab.example.com  
}
```

- 7.5. Verify that the **health** endpoint is running.

Invoke the **curl** command, using **localhost** as the host name with port 8181 and the **/health** resource URI:

```
[student@workstation aloha-service]$ curl -si http://localhost:8181/health  
HTTP/1.1 200 OK  
...  
{"status":"UP"}
```

- ▶ 8. Run the hola-service Spring Boot application and verify that all expected endpoints are active and functional.

- 8.1. Open a terminal, enter the project folder, and invoke the **package** Maven goal to build the hola-service Spring Boot application's Fat JAR:

```
[student@workstation ~]$ cd ~/JB421/labs/camel-microservices/hola-service  
[student@workstation hola-service]$ mvn package  
...  
[INFO] Scanning for projects...  
[INFO]  
[INFO] -----  
[INFO] Building GE: Developing Microservices with Camel Routes - Aloha Service 1.0  
[INFO] -----  
...  
[INFO] Building jar: /home/student/JB421/labs/camel-microservices/hola-service/  
target/hola-service-1.0.jar  
[INFO]  
[INFO] --- spring-boot-maven-plugin:1.5.12.RELEASE:repackage (default) @ hola-  
service ---  
[INFO] -----  
[INFO] BUILD SUCCESS
```

```
[INFO] -----  
[INFO] Total time: 6.648 s  
...  
...
```

- 8.2. Verify that a hola-service Fat JAR exists in the **target** folder:

```
[student@workstation hola-service]$ ls -sh target/*jar*
22M target/hola-service-1.0.jar
```

- 8.3. Run the Spring Boot application using the **java -jar** command. Leave the application running, and notice the log messages from XML route:

- 8.4. Open another terminal and verify that the hola-service Spring Boot application responds to HTTP requests.

Invoke the `curl` command, using `localhost` as the host name with port 8082 and the `/camel/hola/Developer` resource URI:

```
[student@workstation hola-service]$ curl -si \
    http://localhost:8082/camel/hola/Developer
HTTP/1.1 200 OK
...
{
  greeting: Hola, Developer
  server: workstation.lab.example.com
}
```

- 8.5. Verify that the **hola-chained** endpoint is working properly and able to call the alpha-service.

Invoke the **curl** command, using **localhost** as the host name with port 8082 and the **/camel/hola-chained/Developer** resource URI:

```
[student@workstation aloha-service]$ curl -si http://localhost:8082/camel/hola-chained/Developer
HTTP/1.1 200 OK
...
{
  greeting: Aloha, Developer
  server: workstation.lab.example.com
}
```

- 8.6. Verify that the **health** endpoint is running.

Invoke the **curl** command, using **localhost** as the host name with port 8182 and the **/health** resource URI:

```
[student@workstation hola-service]$ curl -si http://localhost:8182/health
HTTP/1.1 200 OK
...
>{"status":"UP"}
```

- 9. Temporarily remove connectivity to the database and test that the health check is working properly and reports a status of **DOWN**.

- 9.1. Run the provided **block-db.sh** script to disable communication from the **workstation** host to the database.

```
[student@workstation hola-service]$ ./block-db.sh
Database connectivity blocked!
```

- 9.2. Re-test the **health** endpoint.

Invoke the **curl** command, using **localhost** as the host name with port 8182 and the **/health** resource URI:



Important

Wait for the **curl** command to complete. The connection to the database must timeout, which takes a few minutes.

```
[student@workstation hola-service]$ curl -si http://localhost:8182/health
HTTP/1.1 503 Service Unavailable
...
>{"status":"DOWN"}
```

The status is **DOWN** because the database cannot be reached due to a network connectivity problem.

- 9.3. Unblock communication with the database by running the provided **unblock-db.sh** script:

```
[student@workstation hola-service]$ ./unblock-db.sh
Database connectivity un-blocked!
```

**Important**

Failure to execute this step can cause issues in later exercises.

- 9.4. Re-verify that the **/health** endpoint is back to **UP**.

Invoke the **curl** command, using **localhost** as the host name with port 8182 and the **/health** resource URL:

```
[student@workstation hola-service]$ curl -si http://localhost:8182/health
HTTP/1.1 200 OK
...
>{"status":"UP"}
```

▶ **10.** Clean up.

- 10.1. Stop the aloha-service Spring Boot application.

Return to the terminal window where the aloha-service is running, and press **Ctrl+C** to stop the microservice.

- 10.2. Stop the hola-service Spring Boot application.

Return to the terminal window where the hola-service is running, and press **Ctrl+C** to stop the microservice.

- 10.3. Close the projects.

In the **Project Explorer** view, right-click **aloha-service** and click **Close Project**.

In the **Project Explorer** view, right-click **hola-service** and click **Close Project**.

This concludes the guided exercise.

Designing Microservices for Failures

Objectives

After completing this section, students should be able to develop microservices by defining Camel routes.

Building Resilient Microservices to Handle Potential Failures

In a distributed system such as a microservice architecture, you often encounter unexpected failures of various components. You can spend a lot of time preventing these errors from happening, but you can't predict every problem in your microservices that could fail in a distributed system. Therefore, you must design your microservices as resilient and *fault-tolerant*. Fault tolerance is the ability of your services to experience unexpected outages of the services or infrastructure on which they rely. By introducing fault tolerance capabilities, an application is able to operate at a certain degree of satisfaction despite any failures that appear.

This section covers two common fault tolerance techniques: the *retry* pattern and the *circuit breaker* pattern.

Implementing the Retry Pattern Using the Camel Error Handler

The retry pattern is intended for handling transient or temporary failures, such as a brief network outage or server deployment, by retrying the operation with the expectation that it'll then succeed. The Camel error handler implements this pattern as its primary functionality.

To handle failures when calling another microservice, you can configure how Camel's error handler retries the route execution:

```
public void configure() throws Exception {  
  
    errorHandler(defaultErrorHandler()  
        .maximumRedeliveries(5)①  
        .redeliveryDelay(2000));②  
  
    from("direct:inventory")  
        .to("http4:"+inventoryHost +":"+inventoryPort+"/inventory/addItem");"  
        .unmarshal().jaxb("com.example");  
}
```

① Configure the route to retry up to five attempts.

② Configure a 2000 millisecond or 2 second delay between each retry attempt.

Not all failures are candidates for the retry pattern. For example, network calls such as HTTP calls or database operations can often be retried successfully. An HTTP server may not respond temporarily, and retrying the operation after a pause might succeed. Similarly, a database

operation can fail because of a locking error due to concurrent access to a table, which can then succeed on the next retry.

However, if a database call fails because the wrong credentials were used to login, then retrying the operation will keep failing, and therefore it is not an ideal candidate for the retry pattern. Remember that Camel's error handler supports configuring different strategies for different exceptions. Therefore, you could configure your route to retry only network issues, and not exceptions caused by invalid credentials.

Another factor to consider when using the retry pattern is whether your call to the external service is *idempotent*. Idempotency is the property of certain operations whereby they can be applied multiple times without changing the result beyond the initial application. A call to a service is idempotent if calling the same service again and again with the same input provides the same result. A classic example is a shopping cart service: calling the endpoint of the service that retrieves the current cart is idempotent, whereas calling the add to cart service is not idempotent.

Distributed systems are inherently more complex. Because of remote networks, a request may have been processed by the remote service but the client might not have received a response, the client then assumes the operation failed, and retries the same operation, which then may cause unexpected side effects. For this reason, it might be worth designing all your microservices with duplicate requests in mind as another form of fault tolerance.

Describing the Circuit Breaker Pattern

The previous section covered the retry pattern and the ways it can overcome transient errors, such as unavailable networks, or a temporarily locked database, by retrying the same operation. However, sometimes failures are not transient. For instance, a downstream service may be completely overloaded, or may have a bug in its code, causing application-level exceptions. In any of those problem situations, having a client retrying the same operation and putting additional load on an already struggling service provide no value.

As a client, if you don't deal with these problems gracefully, you risk degrading your own service, holding up threads, locks, and other resources, and contributing to cascading failures that can degrade the overall system, and even take down the entire distributed system. Instead, client applications must proactively detect that the downstream service is having issues and deal with those issues in a graceful matter until the service is back and healthy again. This means the client need a way to detect failures and fail fast in order to avoid calling the remote service until it's recovered. The solution to this problem is known as the circuit breaker pattern.

The circuit breaker pattern is inspired by the real-world electrical circuit breaker, which can detect excessive current draw and fail fast to protect electrical equipment. The software-based circuit breaker works in a similar fashion, by encapsulating an operation and monitoring it for failures. When everything is normal and the operation is succeeding, the circuit breaker is in the closed state. When the number of failures (an exception or timeout during the call) reaches a preconfigured threshold, the circuit breaker trips open. When the circuit breaker is open, no calls are made to the dependent service, but a fallback response is returned. After a configurable amount of time, the circuit breaker moves to a half-open state. In the half-open state, the circuit breaker executes the service calls periodically to check the health of the dependent service. If the service is healthy again, and the test calls are successful, the circuit state switches back to closed.

The circuit breaker life cycle is shown in the following diagram:

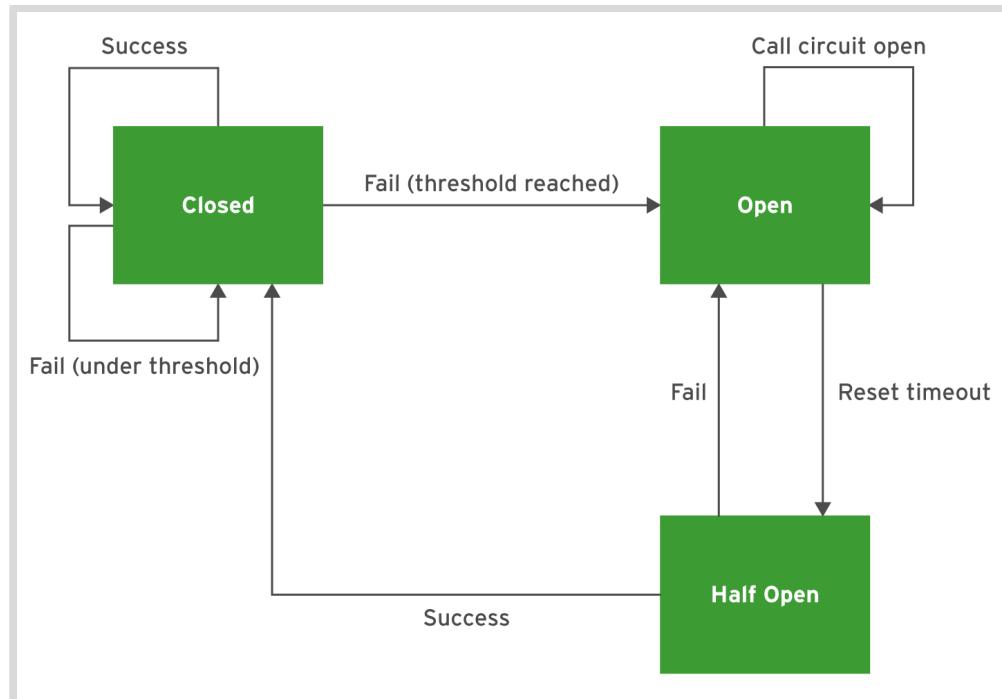


Figure 10.2: Circuit breaker states

Using the Hystrix Enterprise Integration Pattern in a Camel Route

Hystrix is a latency and fault tolerance library designed to isolate points of access to remote systems, services and 3rd party libraries, stop cascading failure and enable resilience in complex distributed systems where failure is inevitable. Camel integrates directly with Hystrix to provide a native Hystrix EIP pattern in Camel. This EIP provides a circuit breaker implementation directly inside a Camel route for any actions included inside the Hystrix block.

To use the Hystrix EIP in your Camel route, you need to add the following dependency:

```

<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-hystrix</artifactId>
</dependency>

```

You can protect any of the calls in your Camel route with Hystrix by wrapping them inside a Hystrix protection block. To denote the beginning of the Hystrix protection use the **hystrix()** DSL element, and be sure to include a corresponding **end()** DSL element to close the Hystrix protection block. The following example shows an example Hystrix protection block:

```

public void configure() throws Exception {
    from("direct:start")
        .hystrix()
        .to("bean:counter")
        .end()
        .log("After calling counter service: ${body}");
}

```

You can put multiple service calls, or even other EIPs inside the Hystrix block. Any failure that occurs inside the Hystrix block will be monitored and acted on by the circuit breaker.

To fine tune the behavior of the circuit breaker itself, use the **hystrixConfiguration** DSL element as shown in the following example:

```
.hystrix()
  .hystrixConfiguration()
    .circuitBreakerRequestVolumeThreshold(10)
      .metricsRollingPercentileWindowInMilliseconds(5000)
    .end()
    .to("bean:counter")
  .end()
```

Be sure to include the matching **end()** DSL element to close the **hystrixConfiguration** block. The following table lists the most commonly used options provided by the **hystrixConfiguration** DSL element:

Commonly Used Hystrix Options

Option	Description
circuitBreakerRequestVolumeThreshold	Sets the minimum number of requests in a rolling window that will trip the circuit. If below this number, the circuit won't trip, regardless of error percentage. The default value is 20 .
circuitBreakerErrorThresholdPercentage	Indicates the error percentage threshold (as a whole number, such as 50), at which point the circuit breaker will trip open and reject requests. It will stay tripped for the duration defined in the circuitBreakerSleepWindowInMilliseconds option. The default value is 50 .
circuitBreakerSleepWindowInMilliseconds	Sets the time in milliseconds after a circuit breaker trips open that it should wait before trying requests again. The default value is 5000 .

Option	Description
metricsRollingStatisticalWindowInMilliseconds	Indicates the duration of the statistical rolling window in milliseconds. This is how long metrics are kept for the successes and failures. The default value is 10000 .
executionTimeoutInMilliseconds	Indicates the time in milliseconds at which point the command will time out and halt execution. The default value is 1000 .

Finally, you can define a *fallback* to be executed when a failure occurs inside your Hystrix block, or when the circuit is open. To include this, use the **onFallback** DSL element inside your Hystrix block as shown in the following example:

```
from("direct:start")
    .hystrix()
        .to("bean:counter")
        .onFallback()
            .transform(constant("No Counter"))
    .end()
```

In this example, you transformed the body of the exchange to a constant value of "**No Counter**". A fallback is a nice way to provide some type of response to your clients even when you cannot complete their request as intended.



References

Camel Error Handler

<http://camel.apache.org/error-handler.html>

Camel Hystrix Component

<https://github.com/apache/camel/blob/master/components/camel-hystrix/src/main/docs/hystrix.adoc>

► Guided Exercise

Designing Microservices for Failures

In this exercise, you will implement and test fault tolerance strategies for communication between two Camel-based microservices.

Outcomes

You should be able to:

- Implement retry logic for a external service call using Camel's built-in error handler
- Add additional fault tolerance using a circuit breaker using Camel's Hystrix integration
- Provide a fallback implementation for the circuit breaker

Before You Begin

A pair of starter projects are provided for you. They both include a **CamelContext** configured in a Spring configuration file, and a **RouteBuilder** subclass.

Run the following command to download the starter project and data files used by this exercise:

```
[student@workstation ~]$ lab camel-ft setup
```

Steps

- 1. Open JBoss Developer Studio and import the starter Maven projects.
 - 1.1. Open JBoss Developer Studio (**Applications** → **Programming** → **JBoss Developer Studio**). JBoss Developer Studio requests a default workspace. Leave the default configuration (**/home/student/workspace**) and click **Launch**.
 - 1.2. Import the Maven project by selecting the JBoss Developer Studio main menu **File** → **Import**.
 - 1.3. From the **Import** dialog box, select **Maven** → **Existing Maven Projects** and click **Next >**.
 - 1.4. Click **Browse** and choose **/home/student/JB421/labs/camel-ft/hola-ft** and click **OK** to choose the Maven project. Click **Finish** to complete the import process in JBoss Developer Studio.
A new project named hola-ft is listed in the **Project Explorer** view.
 - 1.5. Import the Maven project by selecting the JBoss Developer Studio main menu **File** → **Import**.
 - 1.6. From the **Import** dialog box, select **Maven** → **Existing Maven Projects** and click **Next >**.

- 1.7. Click **Browse** and choose **/home/student/JB421/labs/camel-ft/aloha-ft** and click **OK** to choose the Maven project. Click **Finish** to complete the import process in JBoss Developer Studio.

A new project named aloha-ft is listed in the **Project Explorer** view.

- 2. Inspect the existing code for the aloha-service microservice. A new bean class named **DelayBean** has been added so that a configurable amount of delay time can be added to the reply.
- 2.1. Review the Camel route that defines the REST endpoints for the aloha-service using REST DSL.

In the **Project Explorer** view, expand the **aloha-ft** → **src/main/java/** → **com/redhat/training/jb421** package, and double click the **RestRouteBuilder.java** file to open it for editing:

```
package com.redhat.training.jb421;

import org.springframework.stereotype.Component;
import org.apache.camel.builder.RouteBuilder;

@Component
public class RestRouteBuilder extends RouteBuilder {

    @Override
    public void configure() throws Exception {

        rest("/aloha")
            .get("{name}")
            .produces("application/json")
            .to("direct:sayHello");

        from("direct:sayHello").routeId("HelloREST")
            .bean(DelayBean.class, "waitDelay")
            .setBody().simple("{\n"
                + "    greeting: Aloha, ${header.name}\n"
                + "    server: " + System.getenv("HOSTNAME") + "\n"
                + "}\n");
    }
}
```

When you call the aloha-service using an HTTP GET request, this service replies after waiting a configurable amount of time:

```
{
greeting: Aloha, Developer,
server: workstation.lab.example.com
}
```

- 2.2. Review the **DelayBean** class.

In the **Project Explorer** view, expand the **aloha-ft** → **src/main/java/** → **com/redhat/training/jb421** package, and double click the **DelayBean.java** file to open it for editing:

```

@Component
public class DelayBean {

    @Value("${replyDelay}") ①
    private Integer replyDelay;

    public void waitDelay() throws InterruptedException {
        Thread.sleep(replyDelay); ②
    }

}

```

- ➊ Load the delay value from external configuration. A default value of 0 milliseconds is defined in the **application.properties** file packaged with the aloha-ft application.
 - ➋ Sleep the thread for the configured amount of time before continuing with the reply.
- ▶ 3. Inspect the existing code for the hola-ft microservice and implement retry logic using the default error handler.
- 3.1. Review the Camel route that defines the REST endpoints for the aloha-ft using REST DSL.
- In the **Project Explorer** view, expand the **hola-ft → src/main/java/ → com/redhat/training/jb421** package, and double-click the **RestRouteBuilder.java** file to open it for editing:

```

@Component
public class RestRouteBuilder extends RouteBuilder {

    @Value("${alohaHost}")
    private String alohaHost;

    @Value("${alohaPort}")
    private String alohaPort;

    @Override
    public void configure() throws Exception {

        //TODO configure defaultErrorHandler to retry 5 times and wait 2000 ms between
        retries

        rest("/hola")
            .get("{name}")
            .produces("application/json")
            .to("direct:sayHello");

        rest("/hola-chained")
            .get("{name}")
            .produces("application/json")
            .to("direct:callAloha");
    }
}

```

```
from("direct:callAloha")
    .removeHeader(Exchange.HTTP_URI)
    .setHeader(Exchange.HTTP_PATH, simple("${header.name}"))
//TODO add hystrix EIP
    .to("http4:"+alohaHost +":"+alohaPort+"/camel/aloha");

from("direct:sayHello").routeId("HelloREST")
    .setBody().simple("{\n"
        + "  greeting: Hola, ${header.name}\n"
        + "  server: " + System.getenv("HOSTNAME") + "\n"
        + "}\n");
}
```

Ignore the **TODO** comments for now, you will implement them in a subsequent step.

When you call the hola-ft using an HTTP GET request and the URI **/camel/hola**, this service replies:

```
{
greeting: Hola, Developer,
server: workstation.lab.example.com
}
```

- 3.2. Update the route definition to include retry logic using the default error handler.

Find the comment **//TODO configure defaultErrorHandler to retry 5 times and wait 2000 ms between retries**, and add the following code after it:

```
@Override
public void configure() throws Exception {

    //TODO configure defaultErrorHandler to retry 5 times and wait 2000 ms between
    retries
    errorHandler(defaultErrorHandler())❶
        .maximumRedeliveries(5)❷
        .redeliveryDelay(2000));❸
}
```

- ❶ Configure the error handler for the entire route to be an instance of the **defaultErrorHandler** implementation
- ❷ Customize this instance of **defaultErrorHandler** implementation to retry each failure five times
- ❸ Customize this instance of the **defaultErrorHandler** implementation to wait 2000 ms between each retry attempt.

- ▶ 4. Run the aloha-ft Spring Boot application and verify that all expected endpoints are active.

- 4.1. Open a terminal, enter the project folder, and invoke the **package** Maven goal to build the aloha-ft Spring Boot application's Fat JAR:

```
[student@workstation ~]$ cd ~/JB421/labs/camel-ft/aloha-ft
[student@workstation aloha-ft]$ mvn package
...
[student@workstation aloha-ft]$ mvn clean package
[INFO] Scanning for projects...
```

```
[INFO]
[INFO] -----
[INFO] Building GE: Designing Microservices for Failures - Aloha Service 1.0
[INFO] -----
...
[INFO] Building jar: /home/student/JB421/labs/camel-ft/aloha-ft/target/aloha-
ft-1.0.jar
[INFO]
[INFO] --- spring-boot-maven-plugin:1.5.12.RELEASE:repackage (default) @ aloha-ft
---
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 6.648 s
...
...
```

- 4.2. Verify that a aloha-ft Fat JAR exists in the **target** folder:

```
[student@workstation aloha-ft]$ ls -sh target/*jar*
22M target/aloha-ft-1.0.jar
```

- 4.3. Run the Spring Boot application using the **java -jar** command. Leave the application running:

```
[student@workstation aloha-ft]$ java -jar target/aloha-ft-1.0.jar
...
```
 _/ _
 ()\|_
 \\\|_)|_|
 ___|_|_____
 :: Spring Boot :: (v1.5.13.RELEASE)
```
...
16:52:57.384 [main] INFO o.s.b.c.e.u.UndertowEmbeddedServletContainer - Undertow
started on port(s) 8081 (http)
16:52:57.388 [main] INFO c.redhat.training.jb421.Application - Started
Application in 8.235 seconds (JVM running for 8.934)
...
```

- 4.4. Open another terminal and verify that the Spring Boot application responds to HTTP requests.

Invoke the **curl** command, using **localhost** as the host name with port 8081 and the **/camel/aloha/Developer** resource URL:

```
[student@workstation aloha-ft]$ curl -si \
http://localhost:8081/camel/aloha/Developer
HTTP/1.1 200 OK
...
{
  greeting: Aloha, Developer
  server: workstation.lab.example.com
}
```

- ▶ 5. Run the hola-ft Spring Boot application and verify that all expected endpoints are active and functional.

- 5.1. Open a terminal, enter the project folder, and invoke the **package** Maven goal to build the hola-ft Spring Boot application's Fat JAR:

```
[student@workstation ~]$ cd ~/JB421/labs/camel-ft/hola-ft
[student@workstation hola-ft]$ mvn package
...
[INFO] Scanning for projects...
[INFO]
[INFO] -----
[INFO] Building GE: Designing Microservices for Failures - Hola Service 1.0
[INFO] -----
...
[INFO] Building jar: /home/student/JB421/labs/camel-ft/hola-ft/target/hola-
ft-1.0.jar
[INFO]
[INFO] --- spring-boot-maven-plugin:1.5.12.RELEASE:repackage (default) @ hola-ft
---
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 6.648 s
...
```

- 5.2. Verify that a hola-ft Fat JAR exists in the **target** folder:

```
[student@workstation hola-ft]$ ls -sh target/*jar*
22M target/hola-ft-1.0.jar
```

- 5.3. Run the Spring Boot application using the **java -jar** command. Leave the application running:

```
[student@workstation hola-ft]$ java -jar target/hola-ft-1.0.jar
...
```
 _\ / _` - - - (_) _ _ - \ \ \ \
(() \ _ _ | _ | ' _ | ' _ \ ` | \ \ \ \
 ``\ \) | (_) | | | | | (_ |))))
 ' | _ | . _ | _ | _ | _ , | / / / /
=====|_|=====|_|/_/ /_/
```
```

```
:: Spring Boot ::      (v1.5.13.RELEASE)

...
16:52:57.384 [main] INFO o.s.b.c.e.u.UndertowEmbeddedServletContainer - Undertow
started on port(s) 8082 (http)
16:52:57.388 [main] INFO c.redhat.training.jb421.Application - Started
Application in 6.354 seconds (JVM running for 7.538)

...
```

- 5.4. Open another terminal and verify that the hola-ft Spring Boot application responds to HTTP requests.

Invoke the **curl** command, using **localhost** as the host name with port 8082 and the **/camel/hola/Developer** resource URI:

```
[student@workstation hola-ft]$ curl -si \
  http://localhost:8082/camel/hola/Developer
HTTP/1.1 200 OK
...
{
  greeting: Hola, Developer
  server: workstation.lab.example.com
}
```

- 5.5. Verify the **hola-chained** endpoint is working properly and able to call the aloha-ft.

Invoke the **curl** command, using **localhost** as the host name with port 8082 and the **/camel/hola-chained/Developer** resource URI:

```
[student@workstation aloha-ft]$ curl -si http://localhost:8082/camel/hola-chained/
Developer
HTTP/1.1 200 OK
...
{
  greeting: Aloha, Developer
  server: workstation.lab.example.com
}
```

► 6. Stop the aloha-ft microservice and test the retry logic you added to the hola-ft application.

- 6.1. Stop the aloha-ft microservice.

Return to the terminal window running the aloha-ft microservice and press **Ctrl+C** to stop the Spring Boot application.

- 6.2. Invoke the **curl** command, using **localhost** as the host name with port 8082 and the **/camel/hola-chained/Developer** resource URI:

```
[student@workstation aloha-ft]$ curl -si http://localhost:8082/camel/hola-chained/
Developer
...
Caused by: java.net.ConnectException: Connection refused (Connection
refused)
...
```

```

Caused by: java.net.ConnectException: Connection refused (Connection
refused)
...
Caused by: java.net.ConnectException: Connection refused (Connection
refused)
...
Caused by: java.net.ConnectException: Connection refused (Connection refused)
...
Caused by: java.net.ConnectException: Connection refused (Connection refused)
...
Caused by: java.net.ConnectException: Connection refused (Connection refused)
...

```

You should see six **java.net.ConnectException** exceptions, one by the original attempt to call the aloha-ft service, and then one for each of the five retries that were attempted by Camel.

6.3. Stop the hola-ft Spring Boot application.

Return to the terminal window running the hola-ft, and press **Ctrl+C** to stop the microservice.

► 7. Replace the retry logic with an implementation of the circuit breaker pattern using the Hystrix EIP provided by the **camel-hystrix** component.

7.1. Add the **camel-hystrix** dependency to the project's **pom.xml** file.

In the **Project Explorer** view, expand the hola-ft project, double-click the **pom.xml** file, and switch to the **pom.xml** tab.

Locate the comment **<!-- TODO add camel-hystrix dependency -->** and add the following dependency definition:

```

<!-- TODO add camel-hystrix dependency -->
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-hystrix</artifactId>
</dependency>

```

7.2. Press **Ctrl+S** to save your changes to the hola-ft's **pom.xml** file.

7.3. Remove the retry logic you added previously using the **defaultErrorHandler**.



Note

As of Camel version 2.21, the Camel Hystrix integration currently does not inherit the retry handler, and therefore does not support using Camel's internal retry mechanism in conjunction with the hystrix pattern.

Comment out the following lines where you defined the retry logic:

```

@Override
public void configure() throws Exception {

    //TODO configure defaultErrorHandler to retry 5 times and wait 2000 ms between
    retries
    // errorHandler(defaultErrorHandler()
    //     .maximumRedeliveries(5)
    //     .redeliveryDelay(2000));
}

```

7.4. Update the route definition to include the hystrix EIP.

Find the comment `//TODO add hystrix EIP` and add the following DSL to implement the Hystrix EIP to create a circuit breaker in the `direct:callAloha` route:

```

from("direct:callAloha")
    .removeHeader(Exchange.HTTP_URI)
    .setHeader(Exchange.HTTP_PATH, simple("${header.name}"))
    //TODO add hystrix EIP
    .hystrix()①
        .hystrixConfiguration()②
            .executionTimeoutInMilliseconds(4000)③
            .circuitBreakerRequestVolumeThreshold(2)④
            .metricsRollingPercentileWindowInMilliseconds(60000)⑤
            .circuitBreakerSleepWindowInMilliseconds(15000)⑥
            .circuitBreakerErrorThresholdPercentage(50)⑦
        .end()⑧
        .to("http4:"+alohaHost +":"+alohaPort+"/camel/aloha")
    .endHystrix();⑨

```

- ① Start the definition of the hystrix block. Any actions inside the block are subjected to the circuit breaker behavior and monitored for failures.
- ② Start the definition of the hystrix config block, allowing you to configure the behavior of the circuit breaker.
- ③ Specify an execution timeout for actions inside the hystrix block. In this example the timeout is set to 4000 ms or 4 seconds.
- ④ Specify the circuit breaker request volume threshold. This is the minimum number of requests that must be received a minimum before the circuit can be opened due to failures. In this example the threshold is set to two requests.
- ⑤ Specify the duration of percentile rolling window in milliseconds. This is period of time that is analyzed to determine if the circuit should be opened due to failures. Only the outcomes of requests received inside this window of time are considered when determining the circuit breaker behavior. In this example the window is set to 60000 milliseconds or 60 seconds.
- ⑥ Specify the time in milliseconds after the circuit breaker trips open that it should wait before trying requests again. In this example it is set to wait 15000 milliseconds or 15 seconds.
- ⑦ Specify the error percentage threshold (as whole number such as 50) at which point the circuit breaker will trip open and reject requests. It will stay tripped for the duration defined in `circuitBreakerSleepWindowInMilliseconds`.
- ⑧ End the definition of the hystrix config block.
- ⑨ End the definition of the hystrix block.

- 7.5. Add a fallback behavior for the circuit breaker to use when the circuit is open or a failure occurs. Make the fallback return the text **Aloha Fallback**.

Include the following DSL in the route definition to include fallback behavior:

```
from("direct:callAloha")
    .removeHeader(Exchange.HTTP_URI)
    .setHeader(Exchange.HTTP_PATH, simple("${header.name}"))
    .hystrix()
        .hystrixConfiguration()
            .executionTimeoutInMilliseconds(4000)
            .circuitBreakerRequestVolumeThreshold(2)
            .metricsRollingPercentileWindowInMilliseconds(60000)
            .circuitBreakerSleepWindowInMilliseconds(15000)
            .circuitBreakerErrorThresholdPercentage(50)
        .end()
    .to("http4:"+alohaHost +":"+alohaPort+"/camel/aloha");
.onFallback()
    .transform(constant("Aloha Fallback"))
.endHystrix();
```

- 7.6. Press **Ctrl+S** to save your changes to the route definition.

► 8. Test the circuit breaker and fallback behavior.

- 8.1. Start the aloha-ft microservice with a 5000 ms delay.

Open a terminal, enter the aloha-ft project folder, and invoke the **package** Maven goal to build the aloha-ft Spring Boot application's Fat JAR:

```
[student@workstation ~]$ cd ~/JB421/labs/camel-ft/aloha-ft
[student@workstation aloha-ft]$ mvn package
[INFO] Scanning for projects...
[INFO]
[INFO] -----
[INFO] Building GE: Designing Microservices for Failures - Aloha Service 1.0
[INFO] -----
...
[INFO] Building jar: /home/student/JB421/labs/camel-ft/aloha-ft/target/aloha-
ft-1.0.jar
[INFO]
[INFO] --- spring-boot-maven-plugin:1.5.12.RELEASE:repackage (default) @ aloha-ft
---
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 6.648 s
...
```

- 8.2. Verify that a aloha-ft Fat JAR exists in the **target** folder:

```
[student@workstation aloha-ft]$ ls -sh target/*jar*
22M target/aloha-ft-1.0.jar
```

- 8.3. Run the Spring Boot application using the **java -jar** command with **-DreplyDelay=5000** option specified. Leave the application running:

```
[student@workstation aloha-ft]$ java -DreplyDelay=5000 -jar target/aloha-ft-1.0.jar
...
```
 .\` / _`_ _ _ _(_)_ _ _ _ _\` ``\`
(()\`_ | '_| '_| '_\` _` | ` \| \|
\` \` _`|_|_|_|_|_|_|(_|_|_))))
 ' |__| .__|_|_|_|_|_\`_, | / / /
=====|_|=====|_|/_=/_/_/
:: Spring Boot :: (v1.5.13.RELEASE)

...
16:52:57.384 [main] INFO o.s.b.c.e.u.UndertowEmbeddedServletContainer - Undertow
started on port(s) 8081 (http)
16:52:57.388 [main] INFO c.redhat.training.jb421.Application - Started
Application in 8.235 seconds (JVM running for 8.934)

...```

```

- 8.4. Open a terminal, enter the hola-ft project folder, and invoke the **package** Maven goal to build the hola-ft Spring Boot application's Fat JAR:

```
[student@workstation ~]$ cd ~/JB421/labs/camel-ft/hola-ft
[student@workstation hola-ft]$ mvn package
...
[INFO] Scanning for projects...
[INFO]
[INFO] -----
[INFO] Building GE: Designing Microservices for Failures - Hola Service 1.0
[INFO] -----
...
[INFO] Building jar: /home/student/JB421/labs/camel-ft/hola-ft/target/hola-ft-1.0.jar
[INFO]
[INFO] --- spring-boot-maven-plugin:1.5.12.RELEASE:repackage (default) @ hola-ft

[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 6.648 s
...```

```

- 8.5. Verify that a hola-ft Fat JAR exists in the **target** folder:

```
[student@workstation hola-ft]$ ls -sh target/*jar*
22M target/hola-ft-1.0.jar```

```

- 8.6. Run the Spring Boot application using the **java -jar** command. Leave the application running:

```
[student@workstation hola-ft]$ java -jar target/hola-ft-1.0.jar
...
```
  .\` / _` - - - -(_)_ _ _ _ - \` \` \
( ( )\` _ | ' _ | '_ | ' V ` | \` \` \
\` \` | |_)| | | | | | | ( | | ) ) ) )
' | _ | . _ | | _ | | \_, | / / /
=====|_|=====|_|=/_/_/
:: Spring Boot ::          (v1.5.13.RELEASE)

...
16:52:57.384 [main] INFO  o.s.b.c.e.u.UndertowEmbeddedServletContainer - Undertow
started on port(s) 8082 (http)
16:52:57.388 [main] INFO  c.redhat.training.jb421.Application - Started
Application in 6.354 seconds (JVM running for 7.538)
...
```

```

- 8.7. Verify the **hola-chained** endpoint is timing out properly and able to use the circuit breaker fallback method during the call to the aloha-ft.

Invoke the **curl** command, using **localhost** as the host name with port 8082 and the **/camel/hola-chained/Developer** resource URI:

```
[student@workstation aloha-ft]$ curl -si http://localhost:8082/camel/hola-chained/
Developer
Aloha Fallback
```

After four seconds have elapsed you will see the message **Aloha Fallback**. The delay in reply from the aloha-ft service caused the circuit breaker to time out and the fallback method is invoked.

- 8.8. Invoke the **hola-chained** endpoint again using the same **curl** command until the circuit opens and you see the fallback response immediately.

Press the **Up Arrow** in the terminal window to retrieve the previous command.

```
[student@workstation aloha-ft]$ curl -si http://localhost:8082/camel/hola-chained/
Developer
Aloha Fallback
```

When the reply from the **hola-chained** endpoint is immediate you know that the circuit is open and the fallback is invoked immediately.

- 9. Restart the aloha-ft service without the delay and ensure that the circuit closes again.
- 9.1. Stop the aloha-ft Spring Boot application.  
Return to the terminal window running the aloha-ft and press **Ctrl+C** to stop the microservice.
  - 9.2. Start the aloha-ft Spring Boot application, this time without the **retryDelay** option.

```
[student@workstation aloha-ft]$ java -jar target/aloha-ft-1.0.jar
```

- 9.3. Retry the call to the **hola-chained** endpoint on the aloha-ft service and ensure that the circuit closes again.

Return to the terminal window where you invoked the **curl** command previously.

Press the **Up Arrow** in the terminal window to retrieve the previous command.

```
[student@workstation aloha-ft]$ curl -si http://localhost:8082/camel/hola-chained/
Developer
HTTP/1.1 200 OK
...
{
 greeting: Aloha, Developer
 server: workstation.lab.example.com
}
```

When the reply from the **hola-chained** endpoint is the actual reply from the aloha-ft service instead of the **Aloha Fallback** reply, you know that the circuit is closed.

You may need to invoke the command a few times until the rolling 60 second window expires.

## ► 10. Clean up.

- 10.1. Stop the aloha-ft Spring Boot application.

Return to the terminal window running the aloha-ft and press **Ctrl+C** to stop the microservice.

- 10.2. Stop the hola-ft Spring Boot application.

Return to the terminal window running the hola-ft and press **Ctrl+C** to stop the microservice.

- 10.3. Close the projects.

In the **Project Explorer** view, right-click **aloha-ft** and click **Close Project**.

In the **Project Explorer** view, right-click **hola-ft** and click **Close Project**.

This concludes the guided exercise.

## ► Lab

# Deploying Camel Routes

### Performance Checklist

In this lab, you will execute two REST services implemented using the Camel REST DSL and Spring Boot.

The customer-service REST service provides a single endpoint that retrieves customer data from the bookstore database by ID. The data resides inside a MySQL database server located on the **services** machine.

The customer-service REST endpoint takes a single argument, the customer ID, and returns JSON data. The resource URI is **/camel/customer/{id}**.

The order-service REST service provides a single endpoint that creates a new order in the bookstore database. The order-service REST endpoint takes a two arguments, the customer ID, and an order ID for the new order. The resource URI is **/camel/order/createOrder/{customerId}/{orderId}**.

### Outcomes

You should be able to:

- Complete the customer-service project's POM file to dependencies required by Camel and Spring Boot.
- Update the order-service route definition to include a circuit breaker for the call to the customer-service using the Hystrix EIP provided by Camel.
- Package both the customer-service and order-service applications as a Fat JAR using the Spring Boot Maven plug-in.
- Test the applications running standalone using the **curl** command and verify the circuit breaker and fallback behavior.

### Before You Begin

Two starter projects are provided for you, a customer-service project, and a order-service project. Both of these are intended to be Spring Boot based microservices that use the Camel REST DSL. Use the customer-service to retrieve customer information for the order-service to use when creating new orders.

Run the following command to download the starter project used by this lab:

```
[student@workstation ~] lab microservices-review setup
```

1. Open JBoss Developer Studio and import the starter Maven projects.
  - 1.1. Open JBoss Developer Studio (**Applications → Programming → JBoss Developer Studio**). JBoss Developer Studio requests a default workspace. Leave the default configuration (**/home/student/workspace**) and click **Launch**.

- 1.2. Import the Maven project by selecting the JBoss Developer Studio main menu **File** → **Import**.
  - 1.3. From the **Import** dialog box, select **Maven** → **Existing Maven Projects** and click **Next >**.
  - 1.4. Click **Browse** and choose **/home/student/JB421/labs/microservices-review/order-service**. Click **OK** to choose the Maven project. Click **Finish** to complete the import process in JBoss Developer Studio.  
A new project named **order-service** is listed in the **Project Explorer** view.
  - 1.5. Import the Maven project by selecting the JBoss Developer Studio main menu **File** → **Import**.
  - 1.6. From the **Import** dialog box, select **Maven** → **Existing Maven Projects** and click **Next >**.
  - 1.7. Click **Browse** and choose **/home/student/JB421/labs/microservices-review/customer-service**. Click **OK** to choose the Maven project. Click **Finish** to complete the import process in JBoss Developer Studio.  
A new project named **customer-service** is listed in the **Project Explorer** view.
2. Update the customer-service project **pom.xml** file with the necessary Spring Boot starters:
    - **org.apache.camel:camel-spring-boot-starter**
    - **org.apache.camel:camel-servlet-starter**
  3. Create the sample data for the lab.  
Run the provided **setup-data.sh** script at the root of the projects to create the sample data for this lab.

```
[student@workstation ~]$ cd ~/JB421/labs/microservices-review/
[student@workstation microservices-review]$./setup-data.sh
Lab setup complete!
```
  4. Run the customer-service Spring Boot application and verify that all expected endpoints are active. Use the customer ID of 100 when calling the endpoints.
  5. Update the order-service to call the customer-service using the Camel HTTP4 component and the provided **customerHost** and **customerPort** properties.  
Be sure to ensure the necessary camel components are defined as Maven dependencies and that all the appropriate Exchange headers used by the HTTP4 component are set correctly for the call to the customer-service.
  6. Add the circuit breaker pattern to the call from the **order-service** to the customer-service. Include the following configuration for the circuit breaker:
    - Any execution over three seconds must time out.
    - The circuit breaker must receive a minimum of two requests before the circuit can open.
    - The circuit breaker must only monitor a rolling window of the last 60 seconds of requests when deciding whether to open.
    - When opened, the circuit breaker must wait 20 seconds before attempting to close again.
    - The circuit breaker must open if greater than 50 percent of the requests are failing.

- If there is a failure, or the circuit is open, the message **Customer Not Found!** must be used as a fallback.
7. Test the call to the customer-service.

- 7.1. Start the order-service microservice.

Open a terminal and enter the order-service project folder. Invoke the **package** Maven goal to build the order-service Spring Boot application's Fat JAR:

```
[student@workstation ~]$ cd ~/JB421/labs/microservice-review/order-service
[student@workstation order-service]$ mvn package
...
[INFO] BUILD SUCCESS
...
```

- 7.2. Verify that a order-service Fat JAR exists in the **target** folder:

```
[student@workstation order-service]$ ls -sh target/*jar*
22M target/order-service-1.0.jar
```

- 7.3. Run the Spring Boot application using the **java -jar** command. Leave the application running.

```
[student@workstation order-service]$ java -jar target/order-service-1.0.jar
...
 .
 \\\ / ____'_ -- _(_)_ __ _ _ _ \ \\ \\ \\\
 (()__| '_| '_| '_ \ _` | \ \ \ \ \\\
 \\\ / ____| |_)| | | | | || (_| |))))
 ' |____| .__|_|_|_|_|__, | / / / /
 ======|_|=====|_|=/_/_/_/
 :: Spring Boot :: (v1.5.13.RELEASE)
 ...
16:52:57.384 [main] INFO o.s.b.c.e.u.UndertowEmbeddedServletContainer - Undertow
started on port(s) 8081 (http)
16:52:57.388 [main] INFO c.redhat.training.jb421.Application - Started
Application in 8.235 seconds (JVM running for 8.934)
...
```

- 7.4. Verify the **createOrder** endpoint is functioning properly when the call to customer-service succeeds.

Invoke the **curl** command, using **localhost** as the host name with port 8082 and the **/camel/createOrder/{customerId}/{orderId}** resource URI, with a customer ID of 100 and an order ID of 1:

```
[student@workstation customer-service]$ curl -X PUT -si http://localhost:8082/
camel/order/createOrder/100/1
{ "id":1,"orderDate":1541008933000,"discount":0.00,"delivered":false,"customer":null,"orderItems":
[]}
```

**Note**

Each time you successfully create an order you need to use a different orderID value for subsequent requests to avoid issues with the primary key constraints on the database table.

8. Stop and restart the customer-service application with a 5 second delay and then verify the circuit breaker and fallback behavior in the order-service application.
  - 8.1. Return to the terminal window where the customer-service application is running, and press **Ctrl+C** to stop the service.
  - 8.2. Run the Spring Boot application using the **java -jar** command with **-DreplyDelay=5000** option specified. Leave the application running:

```
[student@workstation customer-service]$ java -DreplyDelay=5000 -jar target/customer-service-1.0.jar
...
.
.
.
(()__ | '_| '|| '_ \ _ | \ \ \
_\ _)(_|_)| | | | | | | (_| |)))
' |__| .|_|_|_|_|_|__, | / / /
=====|_|=====|_|/_=//_/_/
:: Spring Boot :: (v1.5.13.RELEASE)
...
16:52:57.384 [main] INFO o.s.b.c.e.u.UndertowEmbeddedServletContainer - Undertow
started on port(s) 8081 (http)
16:52:57.388 [main] INFO c.redhat.training.jb421.Application - Started
Application in 8.235 seconds (JVM running for 8.934)
...
```

9. Switch back to the terminal window where you ran the **curl** command and invoke the **createOrder** endpoint again using the same **curl** command with the same customer ID but different order ID values until the circuit opens and you see the fallback response immediately.

Press the **Up Arrow** in the terminal window to retrieve the previous command.

```
[student@workstation customer-service]$ curl -X PUT -si http://localhost:8082/
camel/order/createOrder/100/2
ERROR Locating Customer
```

When the reply from the **createOrder** endpoint is immediate you know that the circuit is open and the fallback is invoked immediately.

10. Restart the customer-service application without the delay and ensure that the circuit closes again.
  - 10.1. Stop the customer-service Spring Boot application.  
Return to the terminal window where the customer-service is running, and press **Ctrl+C** to stop the microservice.
  - 10.2. Start the customer-service Spring Boot application, this time without the **retryDelay** option.

```
[student@workstation customer-service]$ java -jar target/customer-service-1.0.jar
```

- 10.3. Retry the call to the **createOrder** endpoint on the order-service and ensure the circuit closes again.

Return to the terminal window where you invoked the **curl** command previously.

Press the **Up Arrow** in the terminal window to retrieve the previous command.

```
[student@workstation order-service]$ curl -X PUT -si http://localhost:8082/camel/order/createOrder/100/2
HTTP/1.1 200 OK
...
```

When the reply from the **createOrder** endpoint is the actual JSON reply of a created order instead of the **ERROR Locating Customer!** reply, you know that the circuit is closed.

You may need to invoke the command a few times until the rolling 60 second window expires.

11. Leave the two microservices running, be sure the delay is set on the customer-service

Grade your work. Execute the following command:

```
[student@workstation ~] lab microservice-review grade
```

12. Clean up.

- 12.1. Stop the customer-service Spring Boot application.

Return to the terminal window where the customer-service is running, and press **Ctrl+C** to stop the microservice.

- 12.2. Stop the order-service Spring Boot application.

Return to the terminal window where the order-service is running, and press **Ctrl+C** to stop the microservice.

- 12.3. Close the projects.

In the **Project Explorer** view, right-click **order-service** and click **Close Project**.

In the **Project Explorer** view, right-click **customer-service** and click **Close Project**.

This concludes the lab.

## ► Solution

# Deploying Camel Routes

### Performance Checklist

In this lab, you will execute two REST services implemented using the Camel REST DSL and Spring Boot.

The customer-service REST service provides a single endpoint that retrieves customer data from the bookstore database by ID. The data resides inside a MySQL database server located on the **services** machine.

The customer-service REST endpoint takes a single argument, the customer ID, and returns JSON data. The resource URI is **/camel/customer/{id}**.

The order-service REST service provides a single endpoint that creates a new order in the bookstore database. The order-service REST endpoint takes a two arguments, the customer ID, and an order ID for the new order. The resource URI is **/camel/order/createOrder/{customerID}/{orderID}**..

### Outcomes

You should be able to:

- Complete the customer-service project's POM file to dependencies required by Camel and Spring Boot.
- Update the order-service route definition to include a circuit breaker for the call to the customer-service using the Hystrix EIP provided by Camel.
- Package both the customer-service and order-service applications as a Fat JAR using the Spring Boot Maven plug-in.
- Test the applications running standalone using the **curl** command and verify the circuit breaker and fallback behavior.

### Before You Begin

Two starter projects are provided for you, a customer-service project, and a order-service project. Both of these are intended to be Spring Boot based microservices that use the Camel REST DSL. Use the customer-service to retrieve customer information for the order-service to use when creating new orders.

Run the following command to download the starter project used by this lab:

```
[student@workstation ~] lab microservices-review setup
```

1. Open JBoss Developer Studio and import the starter Maven projects.
  - 1.1. Open JBoss Developer Studio (**Applications** → **Programming** → **JBoss Developer Studio**). JBoss Developer Studio requests a default workspace. Leave the default configuration (**/home/student/workspace**) and click **Launch**.

- 1.2. Import the Maven project by selecting the JBoss Developer Studio main menu **File** → **Import**.
- 1.3. From the **Import** dialog box, select **Maven** → **Existing Maven Projects** and click **Next >**.
- 1.4. Click **Browse** and choose **/home/student/JB421/labs/microservices-review/order-service**. Click **OK** to choose the Maven project. Click **Finish** to complete the import process in JBoss Developer Studio.  
A new project named order-service is listed in the **Project Explorer** view.
- 1.5. Import the Maven project by selecting the JBoss Developer Studio main menu **File** → **Import**.
- 1.6. From the **Import** dialog box, select **Maven** → **Existing Maven Projects** and click **Next >**.
- 1.7. Click **Browse** and choose **/home/student/JB421/labs/microservices-review/customer-service**. Click **OK** to choose the Maven project. Click **Finish** to complete the import process in JBoss Developer Studio.  
A new project named customer-service is listed in the **Project Explorer** view.

2. Update the customer-service project **pom.xml** file with the necessary Spring Boot starters:

- **org.apache.camel:camel-spring-boot-starter**
- **org.apache.camel:camel-servlet-starter**

Open the **pom.xml** and add the following dependencies:

```
<dependency>
 <groupId>org.apache.camel</groupId>
 <artifactId>camel-spring-boot-starter</artifactId>
</dependency>
<dependency>
 <groupId>org.apache.camel</groupId>
 <artifactId>camel-servlet-starter</artifactId>
</dependency>
```

3. Create the sample data for the lab.

Run the provided **setup-data.sh** script at the root of the projects to create the sample data for this lab.

```
[student@workstation ~]$ cd ~/JB421/labs/microservices-review/
[student@workstation microservices-review]$./setup-data.sh
Lab setup complete!
```

4. Run the customer-service Spring Boot application and verify that all expected endpoints are active. Use the customer ID of 100 when calling the endpoints.
  - 4.1. Open a terminal, enter the project folder, and invoke the **package** Maven goal to build the customer-service Spring Boot application's Fat JAR:

```
[student@workstation ~]$ cd ~/JB421/labs/microservices-review/customer-service
[student@workstation customer-service]$ mvn package
...
[INFO] BUILD SUCCESS
...
```

- 4.2. Verify that a customer-service Fat JAR exists in the **target** folder:

```
[student@workstation customer-service]$ ls -sh target/*jar*
22M target/customer-service-1.0.jar
```

- 4.3. Run the Spring Boot application using the **java -jar** command. Leave the application running.

```
[student@workstation customer-service]$ java -jar target/customer-service-1.0.jar
...
```
  _/ \_ ' - _ - _ ( _ ) _ _ _ \ \ \ \
 ( ( ) \ \ \ | ' \ | ' \ \ \ \ \ \ \ \
 ``\ \ \ \ ) | ( ) | | | | | | ( ( ) ) ) )
   ' | \ \ \ | . \ \ \ | | \ \ \ \ \ \ \ \ \
 ======|_|=====|____/_=/\_/_/_
 :: Spring Boot ::      (v1.5.13.RELEASE)

...
16:52:57.384 [main] INFO o.s.b.c.e.u.UndertowEmbeddedServletContainer - Undertow
started on port(s) 8081 (http)
16:52:57.388 [main] INFO c.redhat.training.jb421.Application - Started
Application in 8.235 seconds (JVM running for 8.934)
...
```

```

- 4.4. Open another terminal and verify that the Spring Boot application responds to HTTP requests.

Invoke the **curl** command, using **localhost** as the host name with port 8081 and the **/camel/customer/100** resource URL:

```
[student@workstation customer-service]$ curl -si \
 http://localhost:8081/camel/customer/100
HTTP/1.1 200 OK
...
{"id":null,"firstName":"User","lastName":"Name","username":"user01","password":"changeme","email":"user01@...
```

5. Update the order-service to call the customer-service using the Camel HTTP4 component and the provided **customerHost** and **customerPort** properties.

Be sure to ensure the necessary camel components are defined as Maven dependencies and that all the appropriate Exchange headers used by the HTTP4 component are set correctly for the call to the customer-service.

- 5.1. Use the **camel-http4** component as a REST client to call the customer-service.

To use this component, add a dependency on the **camel-http4** component in the order-service's **pom.xml** file.

In the **Project Explorer** view, expand the order-service project, double-click the **pom.xml** file, and switch to the **pom.xml** tab.

Locate the comment **TODO Add camel-http4 component** and add the following dependency definition:

```
<!--TODO Add camel-http4 component-->
<dependency>
 <groupId>org.apache.camel</groupId>
 <artifactId>camel-http4</artifactId>
</dependency>
```

- 5.2. Press **Ctrl+S** to save your changes to the order-service's **pom.xml** file.
- 5.3. Return to editing the **order-service/src/main/java/com/redhat/training/jb421/RestRouteBuilder.java** file.
- 5.4. Finish the **direct:getCustomer** route definition.

The **createOrder** Camel route originally starts with a REST DSL endpoint. Because of this, it is necessary to modify some of the exchange headers that are set on the exchange by the REST DSL. The **camel-http4** component shares a few of the headers set by the REST DSL, so we need to make sure their values are set properly for the outgoing call to the customer-service instead of being set for the incoming REST call.

Unset the header value **Exchange.HTTP\_URI**

```
from("direct:getCustomer")
//TODO remove header Exchange.HTTP_URI
.removeHeader(Exchange.HTTP_URI)
```

Set the header value **Exchange.HTTP\_PATH** using the **header.customerId** value that was passed into the **createOrder** endpoint originally to pass the customer ID onto the customer-service.

```
from("direct:getCustomer")
//TODO remove header Exchange.HTTP_URI
.removeHeader(Exchange.HTTP_URI)
//TODO set header Exchange.HTTP_PATH to the ${header.customerId} value
.setHeader(Exchange.HTTP_PATH,simple("${header.customerId}"))
```

Set the header value **Exchange.HTTP\_METHOD** to **GET**. This is required because the incoming REST DSL endpoint uses the HTTP method of **PUT**, but the customer-service endpoint only responds to HTTP **GET** requests.

```
from("direct:getCustomer")
//TODO remove header Exchange.HTTP_URI
.removeHeader(Exchange.HTTP_URI)
//TODO set header Exchange.HTTP_PATH to the ${header.customerId} value
.setHeader(Exchange.HTTP_PATH,simple("${header.customerId}"))
//TODO set header Exchange.HTTP_METHOD to GET
.setHeader(Exchange.HTTP_METHOD, simple("GET"))
```

Finally, update the component from **mock** to **http4** in the route's producer:

```
from("direct:getCustomer")
 //TODO remove header Exchange.HTTP_URI
 .removeHeader(Exchange.HTTP_URI)
 //TODO set header Exchange.HTTP_PATH to the ${header.customerId} value
 .setHeader(Exchange.HTTP_PATH, simple("${header.customerId}"))
 //TODO set header Exchange.HTTP_METHOD to GET
 .setHeader(Exchange.HTTP_METHOD, simple("GET"))
 //TODO use the http4 component instead of mock
 .to("http4:"+ customerHost +": "+customerPort+ "/camel/customer");
```

5.5. Press **Ctrl+S** to save your changes to the **RestRouteBuilder.java** file.

6. Add the circuit breaker pattern to the call from the **order-service** to the customer-service. Include the following configuration for the circuit breaker:

- Any execution over three seconds must time out.
- The circuit breaker must receive a minimum of two requests before the circuit can open.
- The circuit breaker must only monitor a rolling window of the last 60 seconds of requests when deciding whether to open.
- When opened, the circuit breaker must wait 20 seconds before attempting to close again.
- The circuit breaker must open if greater than 50 percent of the requests are failing.
- If there is a failure, or the circuit is open, the message **Customer Not Found!** must be used as a fallback.

- 6.1. Update the route definition to include the Hystrix EIP.

Add the following DSL to implement the Hystrix EIP to create a circuit breaker in the **direct:getCustomer** route:

```
from("direct:getCustomer")
 .removeHeader(Exchange.HTTP_URI)
 .setHeader(Exchange.HTTP_PATH, simple("${header.customerId}"))
 .setHeader(Exchange.HTTP_METHOD, simple("GET"))
 .hystrix()
 .hystrixConfiguration()
 .executionTimeoutInMilliseconds(3000)
 .circuitBreakerRequestVolumeThreshold(2)
 .metricsRollingPercentileWindowInMilliseconds(60000)
 .circuitBreakerSleepWindowInMilliseconds(20000)
 .circuitBreakerErrorThresholdPercentage(50)
 .end()
 .to("http4:"+customerHost +": "+customerPort+ "/camel/customer")
 .endHystrix();
```

- 6.2. Add a fallback behavior for the circuit breaker to use when the circuit is open or a failure occurs. Make the fallback return the text **Customer Not Found!** using the provided **CUSTOMER\_ERROR\_MSG** value.

Include the following DSL in the route definition to include fallback behavior:

```

from("direct:getCustomer")
 .removeHeader(Exchange.HTTP_URI)
 .setHeader(Exchange.HTTP_PATH, simple("${header.customerId}"))
 .setHeader(Exchange.HTTP_METHOD, simple("GET"))
 .hystrix()
 .hystrixConfiguration()
 .executionTimeoutInMilliseconds(3000)
 .circuitBreakerRequestVolumeThreshold(2)
 .metricsRollingPercentileWindowInMilliseconds(60000)
 .circuitBreakerSleepWindowInMilliseconds(20000)
 .circuitBreakerErrorThresholdPercentage(50)
 .end()
 .to("http4:"+customerHost +":"+customerPort+"/camel/customer")
.onFallback()
 .transform(constant(CUSTOMER_ERROR_MSG))
.endHystrix();

```

6.3. Press **Ctrl+S** to save your changes to the route definition.

## 7. Test the call to the customer-service.

### 7.1. Start the order-service microservice.

Open a terminal and enter the order-service project folder. Invoke the **package** Maven goal to build the order-service Spring Boot application's Fat JAR:

```

[student@workstation ~]$ cd ~/JB421/labs/microservice-review/order-service
[student@workstation order-service]$ mvn package
...
[INFO] BUILD SUCCESS
...

```

### 7.2. Verify that a order-service Fat JAR exists in the **target** folder:

```

[student@workstation order-service]$ ls -sh target/*jar*
22M target/order-service-1.0.jar

```

### 7.3. Run the Spring Boot application using the **java -jar** command. Leave the application running.

```

[student@workstation order-service]$ java -jar target/order-service-1.0.jar
...
```
  _   _ 
  \  \ / 
   \  / 
    )  ( 
     \  \ 
      )  ) 
     /  / 
    ==  == 
  :: Spring Boot ::    (v1.5.13.RELEASE)
...

```

```
16:52:57.384 [main] INFO o.s.b.c.e.u.UndertowEmbeddedServletContainer - Undertow started on port(s) 8081 (http)
16:52:57.388 [main] INFO c.redhat.training.jb421.Application - Started Application in 8.235 seconds (JVM running for 8.934)
...

```

- 7.4. Verify the **createOrder** endpoint is functioning properly when the call to customer-service succeeds.

Invoke the `curl` command, using `localhost` as the host name with port 8082 and the `/camel/createOrder/{customerId}/{orderId}` resource URI, with a customer ID of 100 and an order ID of 1:

```
[student@workstation customer-service]$ curl -X PUT -si http://localhost:8082/camel/order/createOrder/100/1
{"id":1,"orderDate":1541008933000,"discount":0.0,"delivered":false,"customer":null,"orderItems":[]}
```



Note

Each time you successfully create an order you need to use a different orderID value for subsequent requests to avoid issues with the primary key constraints on the database table.

- 8.** Stop and restart the customer-service application with a 5 second delay and then verify the circuit breaker and fallback behavior in the order-service application.
 - 8.1.** Return to the terminal window where the customer-service application is running, and press **Ctrl+C** to stop the service.
 - 8.2.** Run the Spring Boot application using the **java -jar** command with **-DreplyDelay=5000** option specified. Leave the application running:

```
[student@workstation customer-service]$ java -DreplyDelay=5000 -jar target/customer-service-1.0.jar
...
.
.
.
\ \ / _ _ _ _ _(_)_ _ _ _ _ \ \ \
( ( )\__ | '_ | ' | | ' \ \_ ` | \ \ \ \
\ \ / _ _ | |_)| | | | | | ( | | ) ) ) )
' | _ _ | . _ | _ | _ | _ \_, | / / / /
=====|_|=====|__/_=/_/_/_/
:: Spring Boot ::          (v1.5.13.RELEASE)
...
16:52:57.384 [main] INFO o.s.b.c.e.u.UndertowEmbeddedServletContainer - Undertow
started on port(s) 8081 (http)
16:52:57.388 [main] INFO c.redhat.training.jb421.Application - Started
Application in 8.235 seconds (JVM running for 8.934)
```

- Switch back to the terminal window where you ran the `curl` command and invoke the `createOrder` endpoint again using the same `curl` command with the same customer ID but different order ID values until the circuit opens and you see the fallback response immediately.

Press the **Up Arrow** in the terminal window to retrieve the previous command.

```
[student@workstation customer-service]$ curl -X PUT -si http://localhost:8082/camel/order/createOrder/100/2
ERROR Locating Customer
```

When the reply from the **createOrder** endpoint is immediate you know that the circuit is open and the fallback is invoked immediately.

10. Restart the customer-service application without the delay and ensure that the circuit closes again.

- 10.1. Stop the customer-service Spring Boot application.

Return to the terminal window where the customer-service is running, and press **Ctrl+C** to stop the microservice.

- 10.2. Start the customer-service Spring Boot application, this time without the **retryDelay** option.

```
[student@workstation customer-service]$ java -jar target/customer-service-1.0.jar
```

- 10.3. Retry the call to the **createOrder** endpoint on the order-service and ensure the circuit closes again.

Return to the terminal window where you invoked the **curl** command previously.

Press the **Up Arrow** in the terminal window to retrieve the previous command.

```
[student@workstation order-service]$ curl -X PUT -si http://localhost:8082/camel/order/createOrder/100/2
HTTP/1.1 200 OK
...
```

When the reply from the **createOrder** endpoint is the actual JSON reply of a created order instead of the **ERROR Locating Customer!** reply, you know that the circuit is closed.

You may need to invoke the command a few times until the rolling 60 second window expires.

11. Leave the two microservices running, be sure the delay is set on the customer-service Grade your work. Execute the following command:

```
[student@workstation ~] lab microservice-review grade
```

12. Clean up.

- 12.1. Stop the customer-service Spring Boot application.

Return to the terminal window where the customer-service is running, and press **Ctrl+C** to stop the microservice.

- 12.2. Stop the order-service Spring Boot application.

Return to the terminal window where the order-service is running, and press **Ctrl+C** to stop the microservice.

- 12.3. Close the projects.

In the **Project Explorer** view, right-click **order-service** and click **Close Project**.

In the **Project Explorer** view, right-click **customer-service** and click **Close Project**.

This concludes the lab.

Summary

In this chapter, you learned:

- Microservices are small, self-contained, loosely-coupled, and independently deployable services. These are decentralized and can be developed in different programming languages, run in their own process, and communicate using a lightweight mechanism.
- Spring Boot both provides an ideal deployment platform for building Red Hat Fuse microservices to be deployed in a containerized environment.
- Often in a distributed system, one service must communicate with another. Building this communication with Red Hat Fuse is simple using either the Camel HTTP4 component as a REST client, or a framework like Spring Boot's **RestTemplate**.
- Monitoring is extremely important as you increase the number of independent components in your system by moving to a microservices architecture. You can use the Spring Boot Actuator starter to provide easy to use health check endpoints.
- The circuit breaker and retry patterns provide fault tolerance in microservices that call dependent services.
- Camel's error handler implements the retry pattern for you, which can also be customized to provide different retry behavior based on the type of failure that occurs.
- Camel provides integration with Hystrix that includes a **hystrix()** DSL element to wrap your Camel routes in Hystrix blocks, which creates a circuit breaker that protects the execution of the route inside it.

Chapter 11

Deploying Microservices with Fuse on OpenShift

Goal

Deploy microservices based on Camel Routes to an OpenShift cluster using Fuse on OpenShift.

Objectives

- Describe the architecture and features of Red Hat Fuse on OpenShift.
- Deploy a microservice on an OpenShift cluster using the Fabric8 Maven plug-in.

Sections

- Describing Red Hat Fuse on OpenShift (and Quiz)
- Deploying a Microservice to an OpenShift Cluster (and Guided Exercise)

Lab

Lab: Deploying Camel Routes

Describing Red Hat Fuse on OpenShift

Objective

After completing this section, students should be able to describe the architecture and features of Red Hat Fuse on OpenShift.

OpenShift Terminology

Red Hat OpenShift Container Platform (OCP) is a set of modular components and services built on top of Red Hat Enterprise Linux and Docker. OCP adds *Platform as a Service* (PaaS) capabilities such as remote management, multitenancy, increased security, monitoring and auditing, application life cycle management, and self-service interfaces for developers. OCP leverages open source container technology and Kubernetes container orchestration and management to provide a robust application deployment solution.

Throughout this course, the terms OCP and OpenShift are used to refer to the Red Hat OpenShift Container Platform.

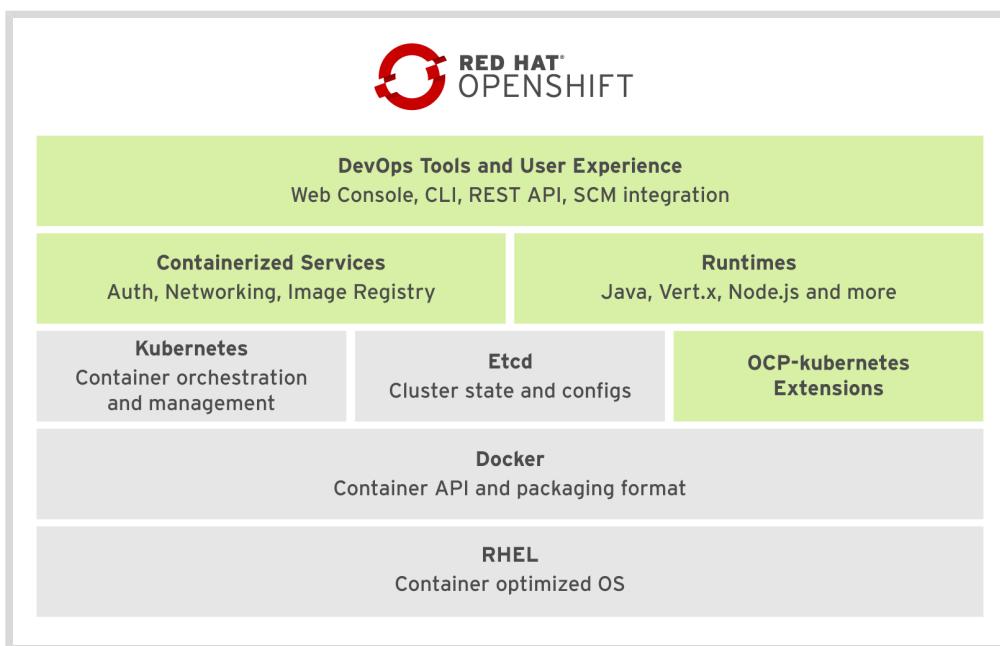


Figure 11.1: OpenShift architecture

The following list defines the various components found in the previous diagram:

Red Hat Enterprise Linux (RHEL)

Provides the base operating system on top of which OCP runs, optimized for containers.

Docker containers

Defines the container management API and container image file format that OCP uses.

Kubernetes

Manages a cluster of hosts, physical or virtual, that run containers. It uses the concept of resources, which describes multi-container applications composed of multiple resources, and

how they interconnect. While Docker containers provide the underlying model for application deployment, Kubernetes is focused on orchestrating and monitoring the deployment and lifecycle of those containers.

Etcdb

Implements a distributed key-value store that Kubernetes uses to store configuration and state information about containers and other resources inside the Kubernetes cluster.

On top of these foundational technologies, OpenShift adds the capabilities required to provide a production PaaS platform to the Docker and Kubernetes container infrastructure. Continuing up the diagram:

OCP-Kubernetes extensions

Provides an additional set of resource types stored in Etcdb and managed by Kubernetes. These additional resource types form the OCP internal state and configuration.

Containerized services

Fulfils many PaaS infrastructure functions, such as networking and authorization. OCP leverages the basic container infrastructure from Docker and Kubernetes for most internal functions. That is, most OCP internal services run as containers orchestrated by Kubernetes.

Runtimes

Includes a set of base container images ready for use by developers, each preconfigured with a particular runtime language or database. The xPaaS offering is a set of base images for JBoss middleware products, such as JBoss EAP and ActiveMQ.

DevOps tools and user experience

Includes Web and CLI management tools for managing user applications and OCP services. The OpenShift Web and CLI tools are built from REST APIs that can be used by external tools such as IDEs and CI platforms.

To deploy your containerized applications OpenShift provides a Kubernetes cluster, which is a set of *node* servers that run containers and are centrally managed by a set of *master* servers. A single host can act as both a master and a node, but those roles are usually segregated for increased stability.

Kubernetes Terminology

| Term | Definition |
|--------|--|
| Master | A server that manages the workload and communications in a Kubernetes cluster. |
| Node | A server that hosts applications in a Kubernetes cluster. |
| Label | A key-value pair that can be assigned to any Kubernetes resource. A selector uses labels to filter eligible resources for scheduling and other operations. |

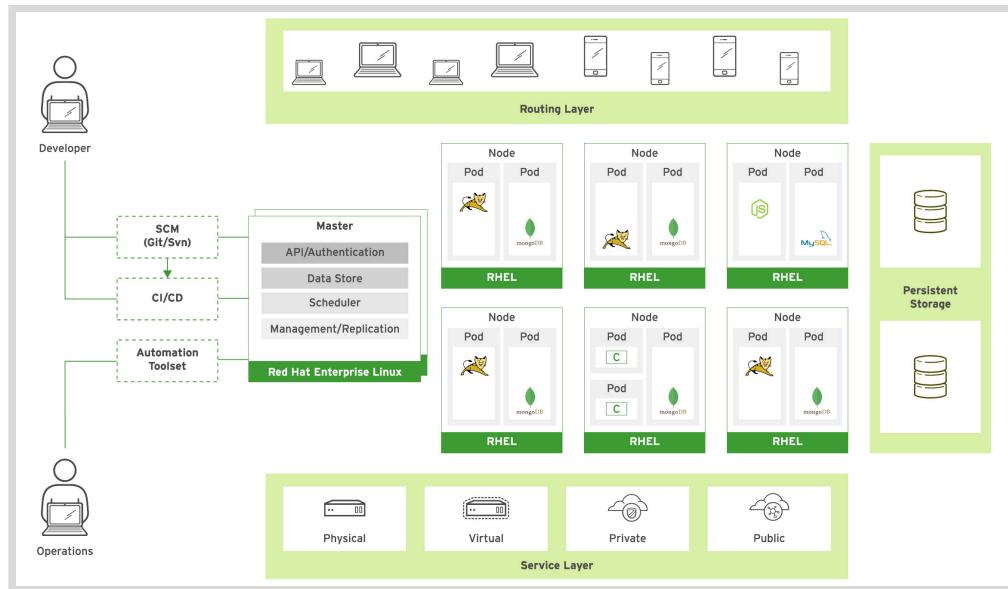


Figure 11.2: OpenShift and Kubernetes architecture

An OpenShift cluster is a Kubernetes cluster that can be managed the same way, but using the management tools provided by OpenShift, such as the command-line interface or the web console. This allows for more productive workflows and makes common tasks much easier.

Kubernetes Resource Types

Kubernetes has five main resource types that can be created and configured using a YAML or a JSON file, or using OpenShift management tools:

Pods

A collection of containers that share resources, such as IP addresses and persistent storage volumes. It is the basic unit of work for Kubernetes.

Services

A single IP/port combination that provides access to a pool of pods. By default, services connect clients to pods in a round-robin fashion.

Replication Controllers

A framework for defining pods that are meant to be horizontally scaled. A replication controller includes a pod definition that is to be replicated, and the pods created from it can be scheduled to different nodes.

Persistent Volumes (PV)

A persistent networked storage resource for pods that can be mounted inside a container to store data.

Persistent Volume Claims (PVC)

A request for storage by a pod that can be fulfilled by a Persistent Volume.



Note

For the purpose of this course, the PVs are provisioned on local storage, not on networked storage. This is a valid approach for development purposes, but it is not a recommended approach for a production environment.

Although Kubernetes pods can be created manually, they are usually created by high-level resources such as replication controllers.

OpenShift Resource Types

As previously described, OpenShift Container Platform provides additional features to Kubernetes. As a result, OCP has several additional resource types that facilitate application development and deployment. The main resource types added by OpenShift Container Platform to Kubernetes are as follows:

Deployment Configurations (dc)

Represent a set of pods created from the same container image, managing workflows such as rolling updates. A **dc** also provides a basic but extensible continuous delivery workflow.

Build Configurations (bc)

Used by the OpenShift Source-to-Image (S2I) feature to build a container image from application source code stored in a Git server. A **bc** works together with a **dc** to provide a basic but extensible continuous integration and continuous delivery workflows.

Routes

Represent a DNS host name recognized by the OpenShift router as an entry point for applications and microservices. Routes expose Kubernetes services to make the IP address from the service externally accessible.

Although Kubernetes replication controllers can be created manually in OpenShift, they are usually created by high level resources such as deployment controllers.

Introducing Fuse on OpenShift

The Fuse on OpenShift offering is a set of container images that provide everything you need to deploy your Red Hat Fuse applications to the OpenShift Container Platform. There are some important differences between using Fuse standalone and Fuse on OpenShift.

- An application deployment with Fuse on OpenShift consists of an application and all required runtime components packaged inside a Docker image. Applications are not deployed to a runtime as with Fuse Standalone, the application image itself is a complete runtime environment deployed and managed through OpenShift.
- Messaging services needed by your Fuse applications are created and managed using the AMQ for OpenShift image and are not included directly within a Karaf container. Fuse on OpenShift provides an enhanced version of the **camel-amq** component to allow for connectivity to messaging services in OpenShift through Kubernetes.
- You must not update a running Karaf instance using the Karaf shell because live updates are not preserved if an application container is restarted or scaled up.
- Patching in an OpenShift environment is different from Fuse Standalone, as each application image is a complete runtime environment. To apply a patch, the application image is rebuilt and redeployed within OpenShift. Core OpenShift management capabilities allow for rolling upgrades and side by side deployment to maintain availability of your application during an upgrade.
- Maven dependencies directly linked to Red Hat Fuse components are supported by Red Hat. Third-party Maven dependencies introduced by users are not supported.

Prepare the OpenShift Server to Deploy Fuse Applications

By default, OpenShift is not configured for Fuse on OpenShift deployments. To prepare OpenShift for Fuse deployments, you need to be an OpenShift administrator and complete the following the steps:

1. Log in to the OpenShift master using the CLI tool as an administrator to the necessary access to configure the OpenShift server.
2. Install the Fuse on OpenShift image streams so that the necessary images are available for the container builders of your Fuse applications.
3. Install the Fuse quickstart templates to provide some example Fuse application templates as a starting point for new Fuse applications created on the OpenShift server.
4. Install the Fuse Console templates in order to have access to the Hawtio console and Prometheus monitoring solutions provided to monitor and interact with Red Hat Fuse applications.

See the references section to get details about the specific commands you need to use to prepare the OpenShift server for deploying Fuse applications.

S2I Images

Four container images are available to deploy Fuse applications on Openshift:

Java

Use this container image to deploy *fat-JAR* applications, or those that include the application server runtime inside an executable JAR file such as Spring Boot or WildFly Swarm (Thorntail).

Karaf

Use this container image to build and deploy Karaf custom assembly based applications.

EAP

Use this container image to deploy Java EE applications that require a JBoss EAP server.



References

Docker documentation website

<https://docs.docker.com/>

Kubernetes documentation website

<https://kubernetes.io/docs/>

OpenShift documentation website

<https://docs.openshift.com/>

For more information, refer to the *Get Started For Administrators* chapter in the *Fuse on OpenShift Guide* at

https://access.redhat.com/documentation/en-us/red_hat_fuse/7.0/html/fuse_on_openshift_guide/get-started-admin

► Quiz

Quiz: Describing Red Hat Fuse on OpenShift

Choose the correct answers to the following questions:

► 1. Which three sentences are correct regarding Kubernetes architecture? (Choose three.)

- a. Kubernetes nodes can be managed without a master.
- b. A Kubernetes master manages pod scaling.
- c. A Kubernetes master schedules pods to specific nodes.
- d. A pod is a set of containers managed by Kubernetes as a single unit.
- e. Kubernetes tools cannot be used to manage resources in an OpenShift cluster.

► 2. Which two sentences are correct regarding Kubernetes and OpenShift resource types? (Choose two.)

- a. A pod is responsible for provisioning its own persistent storage.
- b. All pods generated from the same replication controller have to run in the same node.
- c. A route is responsible for providing IP addresses for external access to pods.
- d. A replication controller is responsible for monitoring and maintaining the number of pods for a particular application.
- e. Containers created from Kubernetes pods cannot be managed using standard Docker tools.

► 3. Which statement is correct regarding persistent storage in OpenShift and Kubernetes?

- a. A PVC represents a storage area that a pod can use to store data and is provisioned by the application developer.
- b. PVC represents a storage area that can be requested by a pod to store data but is provisioned by the cluster administrator.
- c. A PVC represents the amount of memory that can be allocated on a node, so that a developer can state how much memory is required for an application to run.
- d. A PVC represents the number of CPU processing units that can be allocated on a node, subject to a limit managed by the cluster administrator.

► 4. Which statement is correct regarding an application deployment with Fuse on OpenShift?

- a. You must choose a runtime to deploy your application.
- b. The required runtime components are packaged inside a Docker image.
- c. You can choose a runtime to deploy your application.

► **5. Which S2I container image is NOT available to deploy Fuse applications on Openshift?**

- a. Java
- b. Karaf
- c. NodeJS
- d. EAP
- e. Python

► Solution

Quiz: Describing Red Hat Fuse on OpenShift

Choose the correct answers to the following questions:

► 1. Which three sentences are correct regarding Kubernetes architecture? (Choose three.)

- a. Kubernetes nodes can be managed without a master.
- b. A Kubernetes master manages pod scaling.
- c. A Kubernetes master schedules pods to specific nodes.
- d. A pod is a set of containers managed by Kubernetes as a single unit.
- e. Kubernetes tools cannot be used to manage resources in an OpenShift cluster.

► 2. Which two sentences are correct regarding Kubernetes and OpenShift resource types? (Choose two.)

- a. A pod is responsible for provisioning its own persistent storage.
- b. All pods generated from the same replication controller have to run in the same node.
- c. A route is responsible for providing IP addresses for external access to pods.
- d. A replication controller is responsible for monitoring and maintaining the number of pods for a particular application.
- e. Containers created from Kubernetes pods cannot be managed using standard Docker tools.

► 3. Which statement is correct regarding persistent storage in OpenShift and Kubernetes?

- a. A PVC represents a storage area that a pod can use to store data and is provisioned by the application developer.
- b. PVC represents a storage area that can be requested by a pod to store data but is provisioned by the cluster administrator.
- c. A PVC represents the amount of memory that can be allocated on a node, so that a developer can state how much memory is required for an application to run.
- d. A PVC represents the number of CPU processing units that can be allocated on a node, subject to a limit managed by the cluster administrator.

► 4. Which statement is correct regarding an application deployment with Fuse on OpenShift?

- a. You must choose a runtime to deploy your application.
- b. The required runtime components are packaged inside a Docker image.
- c. You can choose a runtime to deploy your application.

► **5. Which S2I container image is NOT available to deploy Fuse applications on Openshift?**

- a. Java
- b. Karaf
- c. NodeJS
- d. EAP
- e. Python

Deploying a Microservice to an OpenShift Cluster

Objectives

After completing this section, students should be able to deploy a microservice on an OpenShift cluster using the Fabric8 Maven plug-in.

Introducing the OpenShift CLI

OpenShift Container Platform (OCP) ships with a command-line tool that enables system administrators and developers to work with an OpenShift cluster. The **oc** command-line tool provides the ability to modify and manage resources throughout the delivery life cycle of a software development project. Common operations with this tool include deploying applications, scaling applications, checking the status of projects, and similar tasks.

Installing the oc Command-line Tool

During OpenShift installation, the **oc** command-line tool is installed on all master and node machines. You can install the **oc** client on systems that are not part of the OpenShift cluster, such as developer machines. Once installed, you can issue commands after authenticating against any master node with a user name and password.

There are several different methods available for installing the **oc** command-line tool, depending on the platform being used:

- On Red Hat Enterprise Linux (RHEL) systems with valid subscriptions, the tool is available directly as an RPM and can be installed using the **yum install** command.

```
[user@host ~]$ sudo yum install -y atomic-openshift-clients
```

- For alternative Linux distributions and other operating systems, such as Windows and MacOS, native clients are available for download from the Red Hat Customer Portal. This also requires an active OpenShift subscription. These downloads are statically compiled to reduce incompatibility issues.

Once the **oc** CLI tool is installed, the **oc help** command displays help information. There are **oc** subcommands for tasks such as:

- Logging in and out of an OpenShift cluster.
- Creating, changing, and deleting projects.
- Creating applications inside a project. For example, creating a deployment configuration from a container image, or a build configuration from application source, and all associated resources.
- Creating, deleting, inspecting, editing, and exporting individual resources such as pods, services, and routes inside a project.
- Scaling applications.
- Starting new deployments and builds.

- Checking logs from application pods, deployments, and build operations.

Core CLI Commands

You can use the **oc login** command to log in interactively, which prompts you for a server name, a user name, and a password, or you can include the required information on the command line.

```
[student@workstation ~]$ oc login https://master.lab.example.com \
-u developer -p redhat
```



Note

Note that the backslash character (\) in the previous command is a command continuation character and should only be used if you are not entering the command as a single line.

After successful authentication from a client, OpenShift saves an authorization token in the user's home folder. This token is used for subsequent requests, negating the need to re-enter credentials or the full master URL.

To check your current credentials, run the **oc whoami** command:

```
[student@workstation ~]$ oc whoami
```

This command outputs the user name that you used when logging in.

```
developer
```

To create a new project, use the **oc new-project** command:

```
[student@workstation ~]$ oc new-project working
```

Run the **oc status** command to verify the status of the project:

```
[student@workstation ~]$ oc status
```

Initially, the output from the status command reads:

```
In project working on server https://master.lab.example.com:443

You have no services, deployment configs, or build configs.
Run 'oc new-app' to create an application.
```

The output of the above command changes as you create new projects, and resources like **Services**, **DeploymentConfigs**, or **BuildConfigs** are added throughout this course.

To delete a project, use the **oc delete project** command:

```
[student@workstation ~]$ oc delete project working
```

To log out of the OpenShift cluster, use the **oc logout** command:

```
[student@workstation ~]$ oc logout  
Logged "developer" out on "https://master.lab.example.com:443"
```

To verify the current project, use the **oc project** command:

```
[student@workstation ~]$ oc project  
TODO
```

To list all pods from the project, use the **oc get pods** command:

```
[student@workstation ~]$ oc get pod  
NAME          READY   STATUS    RESTARTS   AGE  
rest-openshift-1-6n85d   1/1     Running   0          3m  
rest-openshift-s2i-1-build   0/1     Completed   0          5m
```

Accessing the OpenShift Web Console

The OpenShift web console allows a user to execute many of the same tasks as the OpenShift command line. Projects can be created, applications can be created within those projects, and application resources can be examined and manipulated as needed. The OpenShift web console runs as a pod on the master.

Accessing the Web Console

The web console runs in a web browser. The URL is of the format `https://{{hostname of OpenShift master}}/console`. The console requires authentication. By default, OpenShift generates a self-signed certificate for the web console. The user must trust this certificate in order to gain access.

Managing Projects

Upon successful login, the user may select, edit, delete, and create projects on the home page. Once a project is selected, the user is taken to the **Overview** page which shows all of the applications created within that project space.

Application Overview Page

The application overview page is the main component of the web console.

The screenshot shows the OpenShift Application Overview page for the 'php-helloworld' application. At the top, it displays the application name 'php-helloworld' and its route URL 'http://php-helloworld-console.apps.lab.example.com'. Below this, under 'DEPLOYMENT CONFIG', there is a section for 'php-helloworld, #1' showing the image and build status. Under 'CONTAINERS', the 'php-helloworld' container is listed with its image, build status (highlighted with a red box), source, and ports. A 'Scale tool' button with up and down arrows is shown next to a circular icon indicating 1 pod. In the 'NETWORKING' section, it shows a 'Service - Internal Traffic' named 'php-helloworld' (highlighted with a red box) and its port mapping. To the right, it shows 'External Traffic' routes, including the route URL and its target port. The entire interface has a light gray background with red highlights on specific links and boxes.

Figure 11.3: Application overview page

From this page, users can view the route, build, service, and deployment information. The scale tool (arrows) can be used to increase and decrease the number of replicas of the application that are running in the cluster. All of the hyperlinks lead to detailed information about that particular application resource including the ability to manipulate that resource. For example, clicking on the link for the build allows the user to start a new build.

Creating New Applications

The user can select the **Add to Project** link to create a new application. The user can create an application using a template (Source-to-Image), deploy an existing image, and define a new application by importing YAML or JSON formatted resources. Once an application has been created with one of these three methods, it can be managed on the overview page.

Other Web Console Features

The web console allows the user to:

- Manage resources such as project quotas, user membership, secrets, and other advanced resources.
- Create persistent volume claims.
- Monitor builds, deployments, pods, and system events.
- Create continuous integration and deployment pipelines with Jenkins.

Introducing the Fabric8 Maven Plug-in

To deploy a microservice to an OpenShift cluster, you must first wrap the microservice application in a container image that is properly customized to run the application. You also need to create a **Deployment** object for the microservice container to deploy the container image. Then, to expose the microservice endpoints to the other pods running in the cluster, you must create a **Service** object. Finally, you must create a **Route** object to make the microservice endpoints reachable from outside of the cluster.

The fabric8 Maven plug-in simplifies the container image build process, because it uses the OpenShift S2I build process, introduced in the previous section, to produce a container image from the application. The plug-in also generates the resource descriptor artifacts which can be used to create the objects that OpenShift needs to deploy the microservice.

The fabric8 Maven plug-in still needs to build the actual Java application and all the dependencies, typically using the regular Maven **package** goal. However, when that build is complete, the plug-in starts an S2I build on the OpenShift cluster to generate a container image to run the Java application. It then produces the other resource descriptors that are necessary to deploy the microservice on OpenShift. The plug-in can also automatically trigger a deployment of the microservice application to the OpenShift cluster using the container image and resource descriptors that it generates.

Configuring the Plug-in in the Maven POM File

To enable the fabric8 Maven plug-in for the project, add the following configuration to the **pom.xml** file:

```
<plugin>
  <groupId>io.fabric8</groupId>
  <artifactId>fabric8-maven-plugin</artifactId>
  <version>${version.fabric8-maven-plugin}</version>
</plugin>
```

In this example, the `${version.fabric8-maven-plugin}` property is populated using a Maven variable.

Configuration Options

The fabric8 Maven plug-in can be configured using one or more of the following methods: *zero-config*, using the *XML plug-in configuration* in the project's **pom.xml** file, or using *OpenShift resource fragments*.

Zero-config

The zero-config option for fabric8 is the simplest approach and requires no additional XML configuration in the **pom.xml** file beyond defining the plug-in itself.

This approach makes a lot of assumptions about your application, and may not work for more customized needs. Without any further configuration, the fabric8 Maven plug-in defaults to the following values:

- The base image `fabric8/java-jboss-openjdk8-jdk` [<https://github.com/fabric8io-images/java/tree/master/images/jboss/openjdk8/jdk>] is chosen as the source container image where the application runs.
- An OpenShift **Build Config**, **Deployment Config**, **ImageStream** and a **Service** are created as resource objects.
- Port 8080 is exposed as the application service port (as well as ports 8778 and 9779 for Jolokia and `jmx_exporter` access, respectively).



Note

Jolokia is remote JMX with JSON accessible over HTTP. `jmx_exporter` is intended to be run as a Java Agent, exposing an HTTP server and serving metrics of the local JVM. See the **References** at the end of the section for more information.

XML Plug-in Configuration

The zero-config approach offers a convenient starting point for developers, and can be marginally customized. However, in many applications, more flexibility and control is required. To support this, you can use an XML-based plug-in configuration in the **pom.xml** file directly inline with the Maven plug-in configuration.

The fabric8 Maven plug-in XML configuration in the **pom.xml** file can be roughly divided into the following sections:

An **<images>** section specifies the container images to build and how to build them.

```
<configuration>
  ...
  <images>
    <image>
      <name>xml-config-demo:1.0.0</name>①
      <!-- "alias" is used to correlate to the containers in the pod spec -->
      <alias>camel-app</alias>
      <build>
        <from>fabric8/java</from>②
        <assembly>③
          <basedir>/deployments</basedir>
          <descriptorRef>artifact-with-dependencies</descriptorRef>
        </assembly>
        <env>④
          <JAVA_LIB_DIR>/deployments</JAVA_LIB_DIR>
          <JAVA_MAIN_CLASS>org.apache.camel.cdi.Main</JAVA_MAIN_CLASS>
        </env>
      </build>
    </image>
  </images>
  ...
</configuration>
```

- ①** Specifies the name and version of the container that the plug-in builds. In this case, the name is **xml-config-demo** and the version is **1.0.0**.
- ②** Specifies the base image to use during the docker build. In this case, the plug-in uses the **fabric8/java** as the base image.
- ③** Specifies how build artifacts and other files can enter the container image.
- ④** Sets one or more environment variables that are present in the container while it is built.



Note

Refer to the plug-in documentation for more information regarding the specifics of building container images, which is not covered in detail in this class.

A **<resources>** section defines the resource descriptors for deploying on an OpenShift.

```
<configuration>
  ...
  <resources>
    <labels>
      <all>
```

```

<group>quickstarts</group>
</all>
</labels>
<deployment>
  <name>${project.artifactId}</name>
  <replicas>1</replicas>
  <containers>
    <container>
      <alias>camel-app</alias>
      <ports>
        <port>8778</port>
      </ports>
    </container>
  </containers>
</deployment>
<services>
  <service>
    <name>camel-service</name>
  </service>
</services>
</resources>
...
</configuration>

```

A **<generator>** section configures generators that are responsible for creating images. Generators are used as an alternative to a dedicated **<images>** section. A generator is a Java component that provides an auto-detection mechanism for certain build types, such as WildFly Swarm, Spring Boot, or plain Java builds.

```

<configuration>
...
<generator>
  <includes>❶
    <include>spring-boot</include>❷
  </includes>
</generator>
...
</configuration>

```

- ❶ Contains one or more **<include>** elements with generator names that should be included. If present, only this list of generators is included, and in the given order. Order is important because, by default, only the first matching generator is applied.
- ❷ Specifies the name of the generator to include. The supported default values include:
 - **java-exec**: Generic generator for flat class path and fat-jar Java applications
 - **spring-boot**: Spring Boot specific generator
 - **wildfly-swarm**: Generator for WildFly Swarm applications
 - **thorntail-v2**: Generator for Thorntail V2 applications
 - **vertx**: Generator for Vert.x applications
 - **karaf**: Generator for Karaf applications

When a generator detects that it is applicable, it is called with the list of images configured in the **pom.xml** file. A generator typically only creates a new image configuration dynamically if the list is empty. A generator can also add new images to an existing list or even change the current image list. Each generator also supports a set of customization options that can be used to tweak the default configurations.

OpenShift Resource Fragments

Uses an external configuration in the form of YAML resource descriptors, which are located in the project's **src/main/fabric8** directory. Each resource (service, deployment, route) can be defined in its own file, which contains a skeleton of a resource description. The fabric8 Maven plug-in picks up these resource fragments, enriches them, and then combines them into two versions of single YAML file, one specific to OCP (**openshift.yml**) and one for Kubernetes (**kubernetes.yml**).

Reviewing the Maven Fabric8 Plug-in Goals

fabric8:resource

This goal creates the resource descriptors needed to deploy the application to an OpenShift cluster. To use it, run the **mvn fabric8:resource** command in the same directory as the project **pom.xml** file:

```
[user@demo project]$ mvn fabric8:resource
```

The **fabric8:resource** goal automatically creates a YAML resource descriptor for a deployment configuration, along with any service or route you define. The goal uses the descriptor fragments you provide in the **src/main/fabric8** directory of your Maven project to produce enriched resource descriptors. For example, the following YAML fragment is defined in a **route.yml** file:

```
spec:
  port:
    targetPort: 8080
  to:
    kind: Service
    name: ${project.artifactId}
```

There is no **metadata** section as expected for each OpenShift resource object. The fabric8 Maven plug-in creates this section automatically. The plug-in also extracts the object's **kind**, if not specified, from the filename. In this case it is a **Route** because the file is called **route.yml**.

The resulting route definition after enrichment is:

```
- apiVersion: v1
kind: Route
metadata:
  labels:
    app: inventory-service
    provider: fabric8
    version: 1.2.0-SNAPSHOT
    group: com.redhat.coolstore
    name: inventory-service
spec:
```

```

port:
  targetPort: 8080
to:
  kind: Service
  name: inventory-service

```

The plug-in places the final configuration files in the `projectname/target/classes/META-INF/fabric8/openshift` directory.

fabric8:build

The **fabric8:build** goal builds a container image that wraps the Java application. The plug-in supports two different ways to build the image. This is set using the **fabric8.mode** environment property. The property supports the following values:

- **kubernetes**: Builds plain Docker container images and Kubernetes resource descriptors.
- **openshift**: Builds the images compatible with the OpenShift deployment model using an S2I build.
- **auto** (default): Checks whether an OpenShift cluster is accessible. If that is true, then the **openshift** value is used.

To pass the **fabric8.mode** environment variable explicitly, use the following command:

```
[demo@demo project]$ mvn package fabric8:build \
-Dfabric8.mode=openshift
```

OpenShift Build

Whenever the **fabric8.mode** variable is set to **openshift**, the **fabric8.build.strategy** environment variable can be defined to set up how the container must be built. The plug-in supports the following OpenShift binary source builds:

- **s2i**: Uses a binary deployment model. The application is built locally with Maven, and the resulting binary is pushed to the OpenShift cluster and then injected into the builder image. The resulting image is pushed to the OpenShift registry.
- **docker**: Similar to a regular Docker container image build process except that it is done by the OpenShift cluster. This build pushes the generated image to the OpenShift internal registry to make it accessible to the whole cluster. This course does not cover this option.

To pass the **fabric8.build.strategy** environment variable explicitly, use the following command:

```
[user@demo project]$ mvn package fabric8:build \
-Dfabric8.mode=openshift -Dfabric8.build.strategy=s2i
```

fabric8:deploy

The **fabric8:deploy** goal builds the container image, generates the OpenShift resources, and deploys them to the cluster.

```
[demo@demo project]$ mvn fabric8:deploy
```

The deploy goal is designed to run after the **fabric8:build** and **fabric8:resource** goals have been run and generated their respective outputs.

To ensure this is the case, you can bind the **resource** and **build** goals to the standard Maven life cycle so that they are called with normal goals, such as **package** or **install**. For example, to always include the building of the OpenShift resource files and the container images, add the following goals to the **execution** section of the plug-in.

```
<plugin>
  <groupId>io.fabric8</groupId>
  <artifactId>fabric8-maven-plugin</artifactId>
  <version>${version.fabric8-maven-plugin}</version>
  <code><executions></code>
    <code><execution></code>
      <code><id>fmp</id></code>
      <code><goals></code>
        <code><goal>resource</goal></code>
        <code><goal>build</goal></code>
      <code></goals></code>
    <code></execution></code>
  <code></executions></code>
</plugin>
```

fabric8:undeploy

The **fabric8:undeploy** goal deletes the OpenShift resources defined by **fabric8:resource** goal and deployed by the **fabric8:deploy** goal on the cluster.

```
[student@workstation project] $ mvn fabric8:undeploy
```

Reviewing OpenShift Liveness and Readiness Probes

In containerized microservice environments, it is common for individual components to become unhealthy due to issues such as temporary connectivity loss, configuration errors, or problems with external dependencies. OpenShift Container Platform provides a number of options to detect and handle unhealthy containers. The primary resource used by OpenShift to monitor container health is called a *probe*.

A probe is a diagnostic process that uses some action to query the health of individual containers, typically on a configurable schedule. There are two main types of probes that OpenShift leverages: *liveness probes* and *readiness probes*.

Liveness Probes

A liveness probe checks if the container in which it is configured is still running. If the liveness probe fails, then OpenShift kills the container, which is subjected to its restart policy. After a pod is successfully deployed, its liveness probes are run continually on a schedule monitoring the health of the pod.

Readiness Probes

A readiness probe determines whether a container is ready to service requests. Readiness probes are run during the deployment of the pod to determine if the pod is finished deploying. If the readiness probe fails for a container, the endpoints controller built into OpenShift ensures the container has its IP address removed from the endpoints of all attached services.

OpenShift also uses readiness probes to signal to the endpoints controller that even though a container is running, it should not receive any traffic from a proxy.

When you design a health check, it is important to consider whether it will be used as a liveness probe or a readiness probe. The distinction is important, as the readiness probe health check must indicate whether the container is up and running **and** ready to serve requests. A failed readiness probe can simply indicate that the pod needs more time to finish starting up. A liveness probe health check, however, can be much simpler, and only needs to indicate the current status, either up or down, of the container. A failed liveness probe indicates that the pod needs to be restarted immediately.

Both liveness and readiness probes support some common options for controlling when they are to be executed by OpenShift and how they react to failures. These common options include:

initialDelaySeconds

The time in seconds that the probe must wait after the container finishes starting.

timeoutSeconds

The time in seconds that OpenShift must wait for the probe to finish, before considering the probe a failure because no response was received.

Additionally, both liveness and readiness probes are configured by leveraging one of the three possible approaches for defining probes. These approaches include:

HTTP Checks

OpenShift sends an HTTP GET request to a configurable URL to determine the healthiness of the pod. The check is deemed successful if the HTTP response is received before the timeout and the response code is between **200** and **399**. The following is an example of a readiness probe using the **httpGet** method for probing a pod:

```
...  
readinessProbe:  
  httpGet:  
    path: /health  
    port: 8080  
  initialDelaySeconds: 15  
  timeoutSeconds: 1  
...
```

Container Execution Checks

OpenShift executes a command inside the container. Exiting the check with status 0 is considered a success. The following is an example of a liveness probe using the **exec** method for probing a pod:

```
...  
livenessProbe:  
  exec:  
    command:  
      - cat  
      - /tmp/health  
  initialDelaySeconds: 15  
  timeoutSeconds: 1  
...
```

TCP Socket Checks

OpenShift attempts to open a socket to the container. The container is only considered healthy if the check can establish a connection.

```
...
livenessProbe:
  tcpSocket:
    port: 8080
  initialDelaySeconds: 15
  timeoutSeconds: 1
...
```

Using the HTTP check works very well with the health checks endpoints provided by the WildFly Swarm **health** fraction or the Spring Boot Actuator starter, because they return an HTTP status of 200 if the health check succeeds and an HTTP status of 503 if it fails. Both container execution checks and TCP socket checks are useful for probing containers where this type of HTTP-based health check endpoint is not available.

Defining Health Check Resources With the fabric8 Maven Plug-in

The fabric8 Maven plug-in offers a simple approach to automatically creating application health checks for your microservice deployed on OpenShift Container Platform. To do this, include YAML definitions for whatever probes you want in a **deployment.yml** OpenShift resource fragment. Place this YAML file in the **src/main/fabric8** directory of your project. The following is an example of a **deployment.yml** file that defines a liveness and a readiness probe for its microservice:

```
spec:
  template:
    spec:
      containers:
        - readinessProbe:
            httpGet:
              path: /health
              port: 8080
              scheme: HTTP
            initialDelaySeconds: 15
            timeoutSeconds: 5
        livenessProbe:
          httpGet:
            path: /health
            port: 8080
            scheme: HTTP
          initialDelaySeconds: 15
          timeoutSeconds: 5
      ports:
        - containerPort: 8080
          name: http
          protocol: TCP
```

Generate a Stub Project Using Maven Archetypes

You can use a Maven archetype to generate a new project.

```
[user@host ~]$ mvn org.apache.maven.plugins:maven-archetype-plugin:2.4:generate \
-DarchetypeCatalog=https://maven.repository.redhat.com/earlyaccess/all/io/
fabric8/archetypes/archetypes-catalog/2.2.0.fuse-000075-redhat-3/archetypes-
catalog-2.2.0.fuse-000075-redhat-3-archetype-catalog.xml \
-DarchetypeGroupId=org.jboss.fuse.fis.archetypes \
-DarchetypeArtifactId=spring-boot-camel-xml-archetype \
-DarchetypeVersion=2.2.0.fuse-000075-redhat-3
```

The archetype plug-in switches to interactive mode to prompt you for the remaining fields such as **groupId**, **artifactId**, **version** and **package**.

This artifact generates a new Spring Boot Camel project that you can deploy on OpenShift using the Maven Fabric8 plugin.

Demonstration: Configuring and Executing the Fabric8 Maven Plug-in

1. Import the deploy-demo project into JBoss Developer Studio.

2. Inspect the **pom.xml** file from the starter project.

In JBDS **Project Explorer** view, double-click **pom.xml**.

Click the **Source** tab to see the Fabric8 maven plug-in configuration.

3. Review the **RestRouteBuilder** class that has been provided.

This class provides a rest route to check if the application is health.

4. Inspect the **deployment.yml** resource fragment file.

Review the liveness and readiness probes.

5. Inspect the service resource fragment file.

Note that the service fragment exposes TCP port **8080**.

6. Inspect the route resource fragment file.

Note the defined host.

7. Deploy the application on a OpenShift cluster.

- a. Log in to the OpenShift cluster at **master.lab.example.com** as the **developer**

user with the password **redhat**.

- b. Create the **demo-deploy** project on OpenShift.

- c. Use the Fabric8 Maven Plug-in to build a container image for the application, create the required resources on OpenShift and start the deployment.

```
[student@workstation ~]$ cd ~/JB421/labs/deploy-demo
[student@workstation deploy-demo]$ mvn -Popenshift fabric8:deploy
```

8. Test the application on OpenShift.

- Wait for the application pod to be ready and running.

Run the **oc get pods** command until the output resembles the following:

```
[student@workstation deploy-demo]$ oc get pods
NAME           READY   STATUS    RESTARTS   AGE
deploy-demo-1-jt8sc   1/1     Running   0          34s
deploy-demo-s2i-1-build   0/1     Completed  0          1m
```

- Verify that the liveness and readiness probe are working.

Run the **oc logs** command to check the log information.

```
[student@workstation deploy-demo]$ oc logs deploy-demo-1-jt8sc
...
10:56:31.355 [XNIO-3 task-1] INFO  HealthREST - Health endpoint invoked
10:56:31.583 [XNIO-3 task-2] INFO  HealthREST - Health endpoint invoked
...
```

9. Clean up. Delete the OpenShift project and close the project in JBoss Developer Studio to conserve memory.



References

fabric8 Maven Plug-in Homepage

<https://maven.fabric8.io>



References

For more information, refer to the *Developer CLI Operations* chapter in the *CLI REFERENCE Guide* at

https://access.redhat.com/documentation/en-us/openshift_container_platform/3.9/html/cli_reference/

► Guided Exercise

Deploying a Microservice to an OpenShift Cluster

In this exercise, you will deploy a Camel Java route inside a Spring Boot application on an OpenShift cluster.

Outcomes

You should be able to:

- Complete the Fabric8 Maven Plug-in configuration to use the supported Fuse Spring Boot container image.
- Complete the Fabric8 resource fragment file that defines a route to use a short host name.
- Deploy and test the route on an OpenShift cluster.

Before You Begin

A starter project is provided for you, including a **CamelContext** configured in a Spring configuration file, a **RouteBuilder** implementation, and resource fragment files for the Fabric8 Maven Plug-in.

Run the following command to download the starter project and data files used by this exercise:

```
[student@workstation ~]$ lab rest-openshift setup
```

- 1. Open JBoss Developer Studio and import the starter Maven project.
 - 1.1. Open JBoss Developer Studio (**Applications** → **Programming** → **JBoss Developer Studio**). JBoss Developer Studio requests a default workspace; keep the default configuration (**/home/student/workspace**) and click **Launch**.
 - 1.2. Import the Maven project by selecting JBoss Developer Studio main menu **File** → **Import**.
 - 1.3. From the **Import** dialog box, select **Maven** → **Existing Maven Projects** and click **Next**.
 - 1.4. Click **Browse** and choose **/home/student/JB421/labs/rest-openshift** and click **OK** to choose the Maven project. Click **Finish** to complete the import process in JBoss Developer Studio.
A new project named **rest-openshift** is listed in the **Project Explorer** view.
- 2. Review the Spring and Camel configurations.
 - 2.1. Inspect the project's Spring beans configuration file.

In the **Project Explorer** view, expand **rest-openshift → src/main/resources/spring**. Double-click the **camel-context.xml** file. Switch to the **Source** tab.

The Spring beans configuration is empty. It exists only as a placeholder; later you can add XML routes to the Spring Boot application.

- 2.2. Inspect the Spring Boot configuration class.

In the **Project Explorer** view, expand **src/main/java → com.redhat.training.jb421**. Double-click the **Application.java** file.

Note the use of the **@SpringBootApplication** and **@ImportResource** annotations.

- 2.3. Inspect the Java route.

In the **Project Explorer** view, expand **src/main/java → com.redhat.training.jb421**. Double-click the **RestRouteBuilder.java** file.

Note the use of the Camel REST DSL to define the `/hello/{name}` resource URI and the JSON response produced by concatenating strings.

▶ 3. Complete the Fabric8 Maven Plug-in configuration.

- 3.1. Inspect the project's POM file.

In the **Project Explorer** view, Double-click the **pom.xml** file and switch to the **pom.xml** tab.

Note the reference to the **fuse-springboot-bom** BOM and the set of Spring Boot starters declared as dependencies. Note also the **spring-boot-maven-plugin** and **fabric8-maven-plugin** Maven plug-ins.

- 3.2. Complete the Fabric8 Maven Plug-in configuration to use the supported Fuse container image for Spring Boot. Use the **fuse7-java-openshift:1.1** image stream from the **openshift** project.

Use the following listing as a reference for the changes to make to the POM file:

```
...
<plugins>
  <plugin>
    <groupId>io.fabric8</groupId>
    <artifactId>fabric8-maven-plugin</artifactId>
    <version>${fabric8.maven.plugin.version}</version>
    <configuration>
      <generator>
        <config>
          <spring-boot>
            <!-- TODO: configure the image stream name -->
            <fromMode>istag</fromMode>
            <from>openshift/fuse7-java-openshift:1.1</from>
          </spring-boot>
        </config>
      </generator>
    </configuration>
    <executions>
      <execution>
        <goals>
          <goal>resource</goal>
        </goals>
      </execution>
    </executions>
  </plugin>
</plugins>
```

```
<goal>build</goal>
</goals>
</execution>
</executions>
...
```

▶ 4. Complete the Fabric8 Maven Plug-in resource fragments.

4.1. Configure the readiness probe.

In the **Project Explorer** view, expand **rest-openshift** → **src/main/fabric8**.

Double-click the **deployment.yml** file.

Change the **path** attribute to refer to the **/camel/readiness** endpoint.

Use the following listing as a reference for the changes to make to the route resource fragment file:

```
livenessProbe:
failureThreshold: 3
httpGet:
#TODO change the readiness probe path
path: /camel/readiness
port: 8080
scheme: HTTP
```

4.2. Configure the liveness probe.

Change the **path** attribute to refer to the **/camel/liveness** endpoint.

Use the following listing as a reference for the changes to make to the route resource fragment file:

```
livenessProbe:
failureThreshold: 3
httpGet:
#TODO change the liveness probe path
path: /camel/liveness
port: 8080
scheme: HTTP
```

4.3. Inspect the service resource fragment file.

In the **Project Explorer** view, expand **rest-openshift** → **src/main/fabric8**.

Double-click the **service.yml** file.

Note that the service fragment exposes TCP port **8080**.

4.4. Complete the route resource fragment file.

In the **Project Explorer** view, expand the **rest-openshift** → **src/main/fabric8**.

Double-click the **route.yml** file.

Change the **host** attribute to refer to the **hello-rest.apps.lab.example.com** host name.

Use the following listing as a reference for the changes to make to the route resource fragment file:

```
spec:
  port:
    targetPort: 8080
  to:
    kind: Service
    name: ${project.artifactId}
#TODO change the host name of the OpenShift route
host: hello-rest.apps.lab.example.com
```

► 5. Deploy the Spring Boot application on an OpenShift cluster.

- 5.1. Log in to the OpenShift cluster at **master.lab.example.com** as the **developer** user with the password **redhat**.

Open a terminal window and run the **oc login** command. If the **oc login** command prompts you about using insecure connections, answer **y**:

```
[student@workstation ~]$ oc login -u developer -p redhat \
https://master.lab.example.com
...
Use insecure connections? (y/n): y
Login successful.
...
```

- 5.2. Create the **rest-openshift** project on OpenShift.

Run the **oc new-project** command:

```
[student@workstation ~]$ oc new-project rest-openshift
Now using project "rest-openshift" on server "https://master.lab.example.com:443".
...
```

- 5.3. Use the Fabric8 Maven Plug-in to build a container image for the Spring Boot application and create the required resources on OpenShift.

Enter the **~/JB421/labs/rest-openshift** folder and invoke the **fabric8:deploy** Maven goal, from the **openshift** Maven profile:

```
[student@workstation ~]$ cd ~/JB421/labs/rest-openshift
[student@workstation rest-openshift]$ mvn -Popenshift fabric8:deploy
...
[INFO] --- fabric8-maven-plugin:3.5.33.fuse-710023-redhat-00002:resource (default)
@ rest-openshift ---
[INFO] F8: Running in OpenShift mode
[INFO] F8: Using docker image name of namespace: rest-openshift
[INFO] F8: Running generator spring-boot
[INFO] F8: spring-boot: Using ImageStreamTag 'jboss-fuse70-java-openshift:1.0'
from namespace 'openshift' as builder image
[INFO] F8: using resource templates from /home/student/JB421/labs/rest-openshift/
src/main/fabric8
...
[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) @ rest-openshift
---
```

```
[...]
[INFO] --- spring-boot-maven-plugin:1.5.12.RELEASE:repackage (default) @ rest-
openshift ---
[INFO]
[INFO] --- fabric8-maven-plugin:3.5.33.fuse-710023-redhat-00002:build (default) @
rest-openshift ---
[INFO] F8: Using OpenShift build with strategy S2I
[INFO] F8: Running generator spring-boot
[INFO] F8: spring-boot: Using ImageStreamTag 'jboss-fuse70-java-openshift:1.0'
from namespace 'openshift' as builder image
...
[INFO] F8: Creating BuildServiceConfig rest-openshift-s2i for Source build
[INFO] F8: Creating ImageStream rest-openshift
[INFO] F8: Starting Build rest-openshift-s2i
Trying internal type for name:Pod
[INFO] F8: Waiting for build rest-openshift-s2i-1 to complete...
[INFO] F8: =====
[INFO] F8: Starting S2I Java Build .....
[INFO] F8: Starting S2I Java Build .....
[INFO] F8: S2I binary build from fabric8-maven-plugin detected
[INFO] F8: Copying binaries from /tmp/src/maven to /deployments ...
Trying internal type for name:Build
[INFO] F8: Checking for fat jar archive...
[INFO] F8: Found rest-openshift-1.0.jar...
[INFO] F8: ... done
[INFO] F8:
[INFO] F8: Pushing image docker-registry.default.svc:5000/rest-openshift/rest-
openshift:1.0 ...
[INFO] F8: Pushed 0/6 layers, 11% complete
[INFO] F8: Pushed 1/6 layers, 33% complete
[INFO] F8: Pushed 2/6 layers, 34% complete
[INFO] F8: Pushed 3/6 layers, 56% complete
[INFO] F8: Pushed 4/6 layers, 77% complete
[INFO] F8: Pushed 5/6 layers, 92% complete
[INFO] F8: Pushed 6/6 layers, 100% complete
[INFO] F8: Push successful
[INFO] F8: Build rest-openshift-s2i-1 in status Complete
...
[INFO] --- fabric8-maven-plugin:3.5.33.fuse-710023-redhat-00002:deploy (default-
cli) @ rest-openshift ---
...
[INFO] OpenShift platform detected
[INFO] Using project: rest-openshift
[INFO] Creating a Service from openshift.yml namespace rest-openshift name rest-
openshift
[INFO] Created Service: target/fabric8/applyJson/rest-openshift/service-rest-
openshift.json
[INFO] Using project: rest-openshift
[INFO] Creating a DeploymentConfig from openshift.yml namespace rest-openshift
name rest-openshift
[INFO] Created DeploymentConfig: target/fabric8/applyJson/rest-openshift/
deploymentconfig-rest-openshift.json
[INFO] Creating Route rest-openshift:rest-openshift host: hello-
rest.apps.lab.example.com
```

```
[INFO] F8: HINT: Use the command `oc get pods -w` to watch your pods start up  
[INFO] -----  
[INFO] BUILD SUCCESS  
...
```

You can safely ignore warnings and errors similar to the following from the **mvn** command. Notice that the image build step is successful despite the error that follows it.

```
...  
[INFO] F8: Starting S2I Java Build .....  
...  
[INFO] F8: {"kind": "Status", "apiVersion": "v1", "metadata": {}, "status": "Failure", "message": "container \"rest Openshift-s2i-1\" in pod \"rest-database-s2i-1-build\" is waiting to start: PodInitializing", "reason": "BadRequest", "code": 400}  
[INFO] Current reconnect backoff is 1000 milliseconds (T0)  
[INFO] Current reconnect backoff is 2000 milliseconds (T1)  
[INFO] Current reconnect backoff is 4000 milliseconds (T2)  
[INFO] Current reconnect backoff is 8000 milliseconds (T3)  
[INFO] Current reconnect backoff is 16000 milliseconds (T4)  
[INFO] Current reconnect backoff is 32000 milliseconds (T5)  
[INFO] F8: Build rest-openshift-s2i-1 Complete  
[INFO] Current reconnect backoff is 32000 milliseconds (T5)  
[ERROR] Exception in reconnect  
java.util.concurrent.RejectedExecutionException: Task  
    java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutureTask@5f123860  
    rejected from java.util.concurrent.ScheduledThreadPoolExecutor@1a10ade6[Shutting  
    down, pool size = 1, active threads = 1, queued tasks = 0, completed tasks = 12]  
...  
[INFO] BUILD SUCCESS  
...
```

5.4. Run the **mvn** command from the previous step again if you see this error during the Maven build:

```
...  
[ERROR] Exception in reconnect  
java.util.concurrent.RejectedExecutionException: ...  
...  
[ERROR] F8: Failed to execute the build [Unable to build the image using the  
OpenShift build service]  
...
```

► 6. Test the Spring Boot application on OpenShift.

6.1. Wait for the application pod to be ready and running.

Run the **oc get pod** command until the output resembles the following:

```
[student@workstation rest-openshift]$ oc get pod  
NAME                  READY   STATUS    RESTARTS   AGE  
rest-openshift-1-6n85d  1/1     Running   0          3m  
rest-openshift-s2i-1-build 0/1     Completed  0          5m
```

- 6.2. Verify that the readiness and liveness probes are working.

Run the **oc logs** command:

```
[student@workstation rest-openshift]$ oc logs rest-openshift-1-6n85d
...
11:27:44.133 [XNIO-3 task-2] INFO  LivenessREST - Liveness endpoint invoked
11:27:52.486 [XNIO-3 task-1] INFO  ReadinessREST - Readiness endpoint invoked
11:27:54.132 [XNIO-3 task-2] INFO  LivenessREST - Liveness endpoint invoked
11:28:02.486 [XNIO-3 task-1] INFO  ReadinessREST - Readiness endpoint invoked
...
```

- 6.3. Verify that the OpenShift route has the correct host name.

Run the **oc get route** command:

NAME	HOST/PORT	PATH	SERVICES
rest-openshift	hello-rest.apps.lab.example.com		rest-openshift
8080	...		

- 6.4. Verify that the Spring Boot application runs the Camel REST route.

Invoke the **curl** command, using the host name from the previous step, and the **/camel/hello/Developer** resource URI:

```
[student@workstation rest-openshift]$ curl -si \
  http://hello-rest.apps.lab.example.com/camel/hello/Developer
HTTP/1.1 200 OK
...
{
  greeting: Hello, Developer
  server: rest-openshift-1-6n85d
}
```

► 7. Clean up.

- 7.1. Delete the OpenShift project.

Run the **oc delete** command on the project resource:

```
[student@workstation rest-openshift]$ oc delete project rest-openshift
project "rest-openshift" deleted
```

- 7.2. Close the project.

In the **Project Explorer** view, right-click rest-openshift project and click **Close Project**.

This concludes the guided exercise.

► Lab

Deploying Camel Routes

Performance Checklist

In this lab, you will use Camel to implement a REST service inside a Spring Boot application and deploy on an OpenShift cluster.

The REST service you will build needs two REST endpoints that retrieve:

- The shipping address from an order
- The application health check

Both endpoints must output JSON data

Outcomes

After completing the lab, you should be able to deploy a Camel REST service on an OpenShift cluster.

Before You Begin

A skeleton project is provided as a starter, including annotated (JAXB and JPA annotations included) model classes, and a preconfigured Camel context, which includes the necessary data source configuration and a stubbed route builder.

The route builder stub includes one route, which implement the necessary logic to retrieve the data from the database given a header containing the order ID, named **id**.

- Run the following command to download the starter project used by this lab:

```
[student@workstation ~]$ lab openshift-lab setup
```

- Import the starter project into JBoss Developer Studio, located at **/home/student/JB421/labs/openshift-lab**.

Steps

1. Inspect the provided starter project.

Examine the Spring configuration file **src/main/resources/spring/camel-context.xml** and the components it defines, including a data source named **mysqlDataSource**.

2. Review the REST service definition.

Open the **OrderRouteBuilder** class by expanding the **openshift-lab** item in the **Project Explorer** tab in the left pane of JBoss Developer Studio, then click **openshift-lab → src/main/java → com.redhat.training.jb421** to expand it. Double-click the **OrderRouteBuilder.java** file.

The REST service uses the URI `/orders/shipAddress/id` and `/orders/bookTitles/id` with the HTTP GET method for the services. It is very important to include the path parameter, which must be stored in a header named **id**.

Also, the URI `/health` is available to check if the application is healthy.

3. Complete the Fabric8 Maven Plug-in configuration to use the supported Fuse container image for Spring Boot. Use the **openshift/fuse7-java-openshift:1.1** image stream from the **openshift** project.
4. Complete the Fabric8 Maven Plug-in resource fragments:
 - Use the `/camel/health` endpoint to configure the readiness probe.
 - Use the `/camel/health` endpoint to configure the liveness probe.
 - Use the **openshift-lab.apps.lab.example.com** host name to access the application.
5. Populate the orders table with a test order with an ID of **100**, using the **setup-data.sh** script provided.
6. Create the **lab-openshift** project on an OpenShift cluster and deploy the Spring Boot application.
7. Test the Spring Boot application on OpenShift.

- 7.1. Wait for the application pod to be ready and running.

Run the **oc get pod** command until the output resembles the following:

```
[student@workstation openshift-lab]$ oc get pod
NAME                  READY   STATUS    RESTARTS   AGE
openshift-lab-1-hdvgv   1/1     Running   0          38s
openshift-lab-s2i-1-build   0/1     Completed   0          1m
```

- 7.2. Verify that the readiness and liveness probes are working.

Run the **oc logs** command:

```
[student@workstation openshift-lab]$ oc logs openshift-lab-1-hdvgv
...
12:21:30.439 [XNIO-3 task-1] INFO  HealthREST - Health endpoint invoked
12:21:39.528 [XNIO-3 task-2] INFO  HealthREST - Health endpoint invoked
...
```

- 7.3. Verify that the OpenShift route has the correct host name.

Run the **oc get route** command:

```
[student@workstation openshift-lab]$ oc get route
NAME            HOST/PORT           PATH      SERVICES
PORT ...
openshift-lab   openshift-lab.apps.lab.example.com   openshift-lab
8080 ...
```

- 7.4. Verify that the Spring Boot application runs the Camel REST route.

Test the **/camel/orders/shipAddress** REST endpoint using the Rest Client plug-in for Firefox. Use to the service to retrieve information for order with an ID of **100**.



Note

The RESTClient plug-in file is located at **/home/student/restclient.xpi**. You can open this file in Firefox to install the plug-in if needed.

8. Grade your work. Execute the following command:

```
[student@workstation openshift-lab]$ lab openshift-lab grade
```

If you do not get a **PASS** grade, review your work. The **~/JB421/labs/openshift-lab/grade.log** file contains the Maven messages produced by the grading process. This may be helpful in debugging any issues.

9. Clean up.

- 9.1. Delete the project from the OpenShift cluster to clean up the resources.

Run the **oc delete** command on the project resource:

```
[student@workstation openshift-lab]$ oc delete project openshift-lab  
project "openshift-lab" deleted
```

- 9.2. Close the **openshift-lab** project in JBoss Developer Studio to conserve memory.

This concludes the lab.

► Solution

Deploying Camel Routes

Performance Checklist

In this lab, you will use Camel to implement a REST service inside a Spring Boot application and deploy on an OpenShift cluster.

The REST service you will build needs two REST endpoints that retrieve:

- The shipping address from an order
- The application health check

Both endpoints must output JSON data

Outcomes

After completing the lab, you should be able to deploy a Camel REST service on an OpenShift cluster.

Before You Begin

A skeleton project is provided as a starter, including annotated (JAXB and JPA annotations included) model classes, and a preconfigured Camel context, which includes the necessary data source configuration and a stubbed route builder.

The route builder stub includes one route, which implement the necessary logic to retrieve the data from the database given a header containing the order ID, named **id**.

- Run the following command to download the starter project used by this lab:

```
[student@workstation ~]$ lab openshift-lab setup
```

- Import the starter project into JBoss Developer Studio, located at **/home/student/JB421/labs/openshift-lab**.

Steps

1. Inspect the provided starter project.

Examine the Spring configuration file **src/main/resources/spring/camel-context.xml** and the components it defines, including a data source named **mysqlDataSource**.

2. Review the REST service definition.

Open the **OrderRouteBuilder** class by expanding the **openshift-lab** item in the **Project Explorer** tab in the left pane of JBoss Developer Studio, then click **openshift-lab** → **src/main/java** → **com.redhat.training.jb421** to expand it. Double-click the **OrderRouteBuilder.java** file.

The REST service uses the URI `/orders/shipAddress/id` and `/orders/bookTitles/id` with the HTTP GET method for the services. It is very important to include the path parameter, which must be stored in a header named **id**.

Also, the URI `/health` is available to check if the application is healthy.

3. Complete the Fabric8 Maven Plug-in configuration to use the supported Fuse container image for Spring Boot. Use the `openshift/fuse7-java-openshift:1.1` image stream from the **openshift** project.

Use the following listing as a reference for the changes to make to the POM file:

```
...
<plugins>
  <plugin>
    <groupId>io.fabric8</groupId>
    <artifactId>fabric8-maven-plugin</artifactId>
    <version>${fabric8.maven.plugin.version}</version>
    <configuration>
      <generator>
        <config>
          <spring-boot>
            <!-- TODO: configure the image stream name -->
            <fromMode>istag</fromMode>
            <from>openshift/fuse7-java-openshift:1.1</from>
          </spring-boot>
        </config>
      </generator>
    </configuration>
    <executions>
      ...
    </executions>
  </plugin>
</plugins>
```

4. Complete the Fabric8 Maven Plug-in resource fragments:

- Use the `/camel/health` endpoint to configure the readiness probe.
- Use the `/camel/health` endpoint to configure the liveness probe.
- Use the `openshift-lab.apps.lab.example.com` host name to access the application.

4.1. Configure the readiness probe.

In the **Project Explorer** view, expand `openshift-lab` → `src/main/fabric8`. Double-click the `deployment.yml` file.

Change the `path` attribute to refer to the `/camel/health` endpoint.

Use the following listing as a reference for the changes to make to the route resource fragment file:

```
livenessProbe:
  failureThreshold: 3
  httpGet:
    #TODO change the readiness probe path
    path: /camel/health
    port: 8080
    scheme: HTTP
```

4.2. Configure the liveness probe.

Change the **path** attribute to refer to the **/camel/health** endpoint.

Use the following listing as a reference for the changes to make to the route resource fragment file:

```
livenessProbe:
  failureThreshold: 3
  httpGet:
    #TODO change the liveness probe path
    path: /camel/health
    port: 8080
    scheme: HTTP
```

4.3. Complete the route resource fragment file.

In the **Project Explorer** view, expand the **openshift-lab** → **src/main/fabric8**.

Double-click the **route.yml** file.

Change the **host** attribute to refer to the **openshift-lab.apps.lab.example.com** host name.

Use the following listing as a reference for the changes to make to the route resource fragment file:

```
spec:
  port:
    targetPort: 8080
  to:
    kind: Service
    name: ${project.artifactId}
#TODO change the host name of the OpenShift route
  host: openshift-lab.apps.lab.example.com
```

5. Populate the orders table with a test order with an ID of **100**, using the **setup-data.sh** script provided.

Use the provided script to populate the data:

```
[student@workstation ~]$ cd ~/JB421/labs/openshift-lab
[student@workstation openshift-lab]$ ./setup-data.sh
```

6. Create the **lab-openshift** project on an OpenShift cluster and deploy the Spring Boot application.

6.1. Log in to the OpenShift cluster at **master.lab.example.com** as the **developer** user with the password **redhat**.

Open a terminal window and run the **oc login** command. If the **oc login** command prompts you about using insecure connections, answer **y**:

```
[student@workstation ~]$ oc login -u developer -p redhat \
  https://master.lab.example.com
...
Use insecure connections? (y/n): y
Login successful.
...
```

6.2. Create the **lab-openshift** project on OpenShift.

Run the **oc new-project** command:

```
[student@workstation ~]$ oc new-project lab-openshift
Now using project "lab-openshift" on server "https://master.lab.example.com:443".
...
```

6.3. Use the Fabric8 Maven Plug-in to build a container image for the Spring Boot application and create the required resources on OpenShift.

Enter the **~/JB421/labs/openshift** folder and invoke the **fabric8:deploy** Maven goal, from the **openshift** Maven profile:

```
[student@workstation ~]$ cd ~/JB421/labs/openshift-lab
[student@workstation openshift-lab]$ mvn -Popenshift fabric8:deploy
...
[INFO] --- fabric8-maven-plugin:3.5.33.fuse-710023-redhat-00002:resource (default)
@ openshift-lab ---
[INFO] F8: Running in OpenShift mode
[INFO] F8: Using docker image name of namespace: lab-openshift
[INFO] F8: Running generator spring-boot
[INFO] F8: spring-boot: Using ImageStreamTag 'jboss-fuse70-java-openshift:1.0'
from namespace 'openshift' as builder image
[INFO] F8: using resource templates from /home/student/JB421/labs/openshift-lab/
src/main/fabric8
...
[INFO] --- maven-compiler-plugin:3.1:compile (default-compile) @ openshift-lab ---
...
[INFO] --- spring-boot-maven-plugin:1.5.12.RELEASE:repackage (default) @
openshift-lab ---
[INFO]
[INFO] --- fabric8-maven-plugin:3.5.33.fuse-710023-redhat-00002:build (default) @
openshift-lab ---
[INFO] F8: Using OpenShift build with strategy S2I
[INFO] F8: Running generator spring-boot
[INFO] F8: spring-boot: Using ImageStreamTag 'jboss-fuse70-java-openshift:1.0'
from namespace 'openshift' as builder image
...
[INFO] F8: Creating BuildServiceConfig openshift-lab-s2i for Source build
[INFO] F8: Creating ImageStream openshift-lab
[INFO] F8: Starting Build openshift-lab-s2i
Trying internal type for name:Pod
[INFO] F8: Waiting for build openshift-lab-s2i-1 to complete...
[INFO] F8: =====
[INFO] F8: Starting S2I Java Build .....
```

```
[INFO] F8: Starting S2I Java Build ....  
[INFO] F8: S2I binary build from fabric8-maven-plugin detected  
[INFO] F8: Copying binaries from /tmp/src/maven to /deployments ...  
Trying internal type for name:Build  
[INFO] F8: Checking for fat jar archive...  
[INFO] F8: Found openshift-lab-1.0.jar...  
[INFO] F8: ... done  
[INFO] F8:  
[INFO] F8: Pushing image docker-registry.default.svc:5000/openshift-lab/openshift-lab:1.0 ...  
[INFO] F8: Pushed 0/6 layers, 11% complete  
[INFO] F8: Pushed 1/6 layers, 33% complete  
[INFO] F8: Pushed 2/6 layers, 34% complete  
[INFO] F8: Pushed 3/6 layers, 56% complete  
[INFO] F8: Pushed 4/6 layers, 77% complete  
[INFO] F8: Pushed 5/6 layers, 92% complete  
[INFO] F8: Pushed 6/6 layers, 100% complete  
[INFO] F8: Push successful  
[INFO] F8: Build openshift-lab-s2i-1 in status Complete  
...  
[INFO] --- fabric8-maven-plugin:3.5.33.fuse-710023-redhat-00002:deploy (default-cli) @ openshift-lab ---  
...  
[INFO] OpenShift platform detected  
[INFO] Using project: openshift-lab  
[INFO] Creating a Service from openshift.yml namespace openshift-lab name openshift-lab  
[INFO] Created Service: target/fabric8/applyJson/openshift-lab/service-openshift-lab.json  
[INFO] Using project: openshift-lab  
[INFO] Creating a DeploymentConfig from openshift.yml namespace openshift-lab name openshift-lab  
[INFO] Created DeploymentConfig: target/fabric8/applyJson/openshift-lab/deploymentconfig-openshift-lab.json  
[INFO] Creating Route openshift-lab:openshift-lab host: openshift-lab.apps.lab.example.com  
[INFO] F8: HINT: Use the command `oc get pods -w` to watch your pods start up  
[INFO] -----  
[INFO] BUILD SUCCESS  
...
```

You can safely ignore warnings and errors similar to the following from the **mvn** command. Notice that the image build step is successful despite the error that follows it.

```
...  
[INFO] F8: Starting S2I Java Build ....  
...  
[INFO] F8: {"kind":"Status","apiVersion":"v1","metadata":{},"status":"Failure","message":"container \"openshift-lab-s2i-1\" in pod \"rest-database-s2i-1-build\" is waiting to start: PodInitializing","reason":"BadRequest","code":400}  
[INFO] Current reconnect backoff is 1000 milliseconds (T0)  
[INFO] Current reconnect backoff is 2000 milliseconds (T1)  
[INFO] Current reconnect backoff is 4000 milliseconds (T2)
```

```
[INFO] Current reconnect backoff is 8000 milliseconds (T3)
[INFO] Current reconnect backoff is 16000 milliseconds (T4)
[INFO] Current reconnect backoff is 32000 milliseconds (T5)
[INFO] F8: Build openshift-lab-s2i-1 Complete
[INFO] Current reconnect backoff is 32000 milliseconds (T5)
[ERROR] Exception in reconnect
java.util.concurrent.RejectedExecutionException: Task
java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutureTask@5f123860
rejected from java.util.concurrent.ScheduledThreadPoolExecutor@1a10ade6[Shutting
down, pool size = 1, active threads = 1, queued tasks = 0, completed tasks = 12]
...
[INFO] BUILD SUCCESS
...
```

- 6.4. Run the **mvn** command from the previous step again if you see this error during the Maven build:

```
...
[ERROR] Exception in reconnect
java.util.concurrent.RejectedExecutionException: ...
...
[ERROR] F8: Failed to execute the build [Unable to build the image using the
OpenShift build service]
...
```

7. Test the Spring Boot application on OpenShift.

- 7.1. Wait for the application pod to be ready and running.

Run the **oc get pod** command until the output resembles the following:

```
[student@workstation openshift-lab]$ oc get pod
NAME             READY   STATUS    RESTARTS   AGE
openshift-lab-1-hdvgv   1/1     Running   0          38s
openshift-lab-s2i-1-build   0/1     Completed   0          1m
```

- 7.2. Verify that the readiness and liveness probes are working.

Run the **oc logs** command:

```
[student@workstation openshift-lab]$ oc logs openshift-lab-1-hdvgv
...
12:21:30.439 [XNIO-3 task-1] INFO  HealthREST - Health endpoint invoked
12:21:39.528 [XNIO-3 task-2] INFO  HealthREST - Health endpoint invoked
...
```

- 7.3. Verify that the OpenShift route has the correct host name.

Run the **oc get route** command:

```
[student@workstation openshift-lab]$ oc get route
NAME          HOST/PORT           PATH      SERVICES
PORT ...
openshift-lab  openshift-lab.apps.lab.example.com   openshift-lab
8080 ...
```

- 7.4. Verify that the Spring Boot application runs the Camel REST route.

Test the **/camel/orders/shipAddress** REST endpoint using the Rest Client plug-in for Firefox. Use the service to retrieve information for order with an ID of **100**.

In the RESTClient window, under the **Request** section, set the method to **GET** and the URL to `http://openshift-lab.apps.lab.example.com/camel/orders/shipAddress/100`.



Note

The RESTClient plug-in file is located at **/home/student/restclient.xpi**. You can open this file in Firefox to install the plug-in if needed.

Click **SEND** to send the request to service. If successful, you will see an HTTP status code **200** in the **Response** section.

8. Grade your work. Execute the following command:

```
[student@workstation openshift-lab]$ lab openshift-lab grade
```

If you do not get a **PASS** grade, review your work. The **~/JB421/labs/openshift-lab/grade.log** file contains the Maven messages produced by the grading process. This may be helpful in debugging any issues.

9. Clean up.

- 9.1. Delete the project from the OpenShift cluster to clean up the resources.

Run the **oc delete** command on the project resource:

```
[student@workstation openshift-lab]$ oc delete project openshift-lab
project "openshift-lab" deleted
```

- 9.2. Close the **openshift-lab** project in JBoss Developer Studio to conserve memory.

This concludes the lab.

Summary

In this chapter, you learned:

- Fuse on OpenShift provides supported container images and quickstart application templates for each of the runtimes supported by Red Hat Fuse.
- Kubernetes manages load balancing, high availability, and persistent storage for containerized applications.
- OpenShift adds to Kubernetes multitenancy, security, ease of use, and continuous integration and continuous development features.
- The OpenShift Container Platform organizes entities in the OpenShift cluster as objects stored on the master node. These are collectively known as **resources**. The most common ones are:
 - Pod
 - Service
 - Route
 - Project
- The Fuse on OpenShift offering includes base images that support the following runtimes:
 - Java
 - Spring Boot
 - Karaf
 - JBoss EAP
- To deploy a microservice to an OpenShift cluster, you must first wrap the microservice application in a container image that is properly customized to run the application.
- The fabric8 Maven plug-in simplifies the container image build process, because it uses the OpenShift S2I build process to produce a container image from the application.

Chapter 12

Comprehensive Review

Goal

Review tasks from *Camel Integration and Development with Red Hat Fuse on OpenShift*

Objectives

- Review tasks from *Camel Integration and Development with Red Hat Fuse on OpenShift*

Sections

- Implementing Routes with Camel
- Implementing REST-based Routes
- Managing Transacted Routes with Camel
- Deploying Microservices with Fuse on OpenShift

Lab

- Lab: Implementing Routes with Camel
- Lab: Implementing REST-based Routes
- Lab: Managing Transacted Routes with Camel
- Lab: Deploying Microservices with Fuse on OpenShift

Comprehensive Review

Objectives

After completing this section, students should have reviewed and refreshed the knowledge and skills learned in *Camel Integration and Development with Red Hat Fuse on OpenShift*.

Reviewing Camel Integration and Development with Red Hat Fuse on OpenShift

Before beginning the comprehensive review for this course, students should be comfortable with the topics covered in each chapter.

Students can refer to earlier sections in the textbook for extra study.

Chapter 1, *Introducing Fuse and Camel*

Describe how Fuse and Camel are used to integrate applications.

- Discuss integration concepts with Red Hat Fuse and Camel.
- Describe Enterprise Integration Patterns with Camel.
- Describe the basics of Camel.

Chapter 2, *Creating Routes*

Develop simple Camel routes.

- Create a route that reads and writes data to the file system.
- Create a route that reads from an FTP server.
- Develop a route that filters messages.
- Implement a Camel route with a Content Based Router and provide basic error handling.
- Manipulate exchange headers in routes.

Chapter 3, *Transforming Data*

Convert messages between data formats using implicit and explicit transformation.

- Invoke data transformation automatically and explicitly using a variety of different techniques.
- Transform messages using additional data formats, custom type converters, and the Message Translator pattern.
- Merge multiple messages using the Aggregator pattern and customizing the output to a traditional data format.
- Access a database with JDBC and JPA.

Chapter 4, Creating Tests for Routes and Error Handling with Camel

Develop reliable routes by developing route tests and handling errors.

- Develop tests for Camel routes with Camel Test Kit.
- Create realistic test cases with mock components.
- Create reliable routes that handle errors gracefully.

Chapter 5, Routing with Java Beans

Create dynamic routes in Camel using Java Beans

- Create Java Beans for use in routing and transforming messages.
- Implement routes that include dynamic routing with CDI.
- Execute bean methods within DSL predicates.

Chapter 6, Implementing REST Services

Enabling REST support in Camel with the Java REST DSL

- Create a route that hosts a REST Service using the REST DSL.
- Consume HTTP resources to implement the content enricher pattern.
- Document a REST service using the REST DSL integration with Swagger.

Chapter 7, Deploying Camel Routes

Package and deploy Camel applications for deployment with Red Hat Fuse

- Deploy Camel integration projects to Karaf as an OSGi bundle.
- Deploy Camel integration projects to EAP as a Java EE archive.
- Deploy Camel integration projects to Spring Boot.

Chapter 8, Implementing Transactions

Provide data integrity in route processing by implementing transactions.

- Implement transaction management in routes using the Spring Transaction Manager.
- Develop tests for transactional routes.

Chapter 9, Implementing Parallel Processing

Improve the throughput of route processing using Camel parallel processing mechanisms.

- Implement concurrency in routes to positively impact performance of an application.
- Implement parallel processing in EIPs.

Chapter 10, Creating Microservices with Red Hat Fuse

Create microservices from Camel routes.

- Describe microservices and how they can be built and packaged using Red Hat Fuse.
- Develop microservices by defining Camel routes.
- Design a microservice so that it can handle potential failures in its interactions with other services.

Chapter 11, Deploying Microservices with Fuse on OpenShift

Deploy microservices based on Camel Routes to an OpenShift cluster using Fuse on OpenShift.

- Describe the architecture and features of Red Hat Fuse on OpenShift.
- Deploy a microservice on an OpenShift cluster using the Fabric8 Maven plug-in.

► Lab

Implementing Routes with Camel

In this review, you will create a route that reads orders from two directories and groups them together as a batch order that gets sent to the manufacturers.

Outcomes

You should be able to:

- Implement aggregator EIP to create batch orders.
- Log auditing information using wire tap EIP.
- Convert CSV files to Java objects and XML files.
- Create processors and beans to customize the exchange contents transported during the route processing.

Before You Begin

Set up your environment for this exercise by logging into **workstation** as **student**, and run the following command:

```
[student@workstation ~]$ lab review1 setup
```

Instructions

In this comprehensive review, you need to create a Camel route that processes a set of files representing incoming orders from two directories on the file system. Split the incoming files into separate orders, and batch the orders together every two seconds. Calculate the total value of the orders, and transform the batch into an XML file. Each batch of orders may have a total value either over or under \$100. Store the batch XML file in a different directory depending on the total value of the orders in the batch.

A starter project is provided with all the source code needed for this laboratory. It is available in the **/home/student/JB421/labs/review1** directory.

- Read multiple CSV files representing an order from the file system using regular files that match a naming convention. The file name starts with **orders-v1** and has the suffix **.csv**.
- Two different directories store the files (**/tmp/data/orders1** and **/tmp/data/orders2**) and both must be monitored.
- The route that monitors the **/tmp/data/orders1** ID is **OrderRoute1**
- The route that monitors the **/tmp/data/orders2** ID is **OrderRoute2**.
- Improve the processing of the files using parallel processing.
- For each CSV file that the route consumes, assign a unique, sequential batch number.
- Log each file that the route processes and its unique batch number to the console.
The expected log output format is **Processing file: fileName in Batch batchNumber**.

- Convert each CSV file to multiple orders.
- Split the orders obtained during the CSV file processing. Consider using memory improvement techniques because the number of orders in each file may be large.
- Update the state field to be masked with asterisks.
- Calculate the total value of all orders in a batch generated every two seconds. For this lab purpose, the **Batch.addOrder** method is provided.
- Write all batches over \$100 to a file in the **/tmp/audit/large** directory, otherwise write the batch file to **/tmp/audit/small** in XML format to improve performance.
- The format of the file name should be **order-audit-yyMMddHHmmssBnn.xml**, where "nn" is the batch number (it does not need to be padded to two digits).
- Convert each order to XML.

1. Implement a route in the **com.redhat.training.routes.OrderRouteBuilder** route builder that reads files from the **/tmp/data/orders1** directory and sends them to an endpoint named **direct:process**.
The route must read comma-separated value (CSV) files with a file pattern **orders-v1-* .csv**. Furthermore, use a thread pool to allow multiple threads to process files simultaneously. The route ID must be **OrderRoute1** to allow test cases to test the route.
2. Create a new route in the **com.redhat.training.routes.OrderRouteBuilder** route builder that reads from the **/tmp/data/orders2** directory, with the same customization as the previous route, but identified as **OrderRoute2**.
3. Check that the implementation is correct so far.
Open the **src/test/java** folder. Right-click the **com.redhat.training.routes.OrderRouteTest** test case and select **Run As → JUnit test**. The test methods **testFileProcessed** and **testAcceptFile** must pass. If any of these method tests fails, review your work so far.
4. Add the batch number as part of the route processing and log it to the console.
The batch number is provided by the **com.redhat.training.beans.AssignBatchNumber** class. Update this class to use the number returned from it as the value of a header named **batchNumber**.
Include a log statement to output the batch number and the name of the file that was processed. Use the following text as your expected log output: **Processing file: \${header.CamelFileName} in Batch \${header.batchNumber}**.
5. Convert each CSV order to an **Order** object, using the camel-bindy component.
Map each field from the **com.redhat.training.model.Order** class using the bindy annotations. The file uses the UNIX end of line character (**\n**) and the date format used by the file uses the **MM/dd/yy** format. For additional information, refer to the **/tmp/orders1** directory and open any of the **orders-v1-* .csv** files.
Add the transformation from a CSV file to an **Order** object to the route whose URI starting point is **direct:process**.
6. Split the body to transform each object into a single order. To minimize memory usage, read it as a stream.
7. Update the order's state to ****** and add to the order the file name from which the order was obtained.

You may invoke the **OrderBean** class that is already mapped in the Spring XML configuration file. Also, update the **OrderBean** class to set the file name using the **CamelFileName** header value and the state with **.

8. Implement the wire tap EIP to send the exchange to an alternate route using the SEDA component named **audit**. This new route creates an audit trail from the original route.
Add to the route with a URI starting point of **direct:process**:
Observe that the **sedat:audit** route is already created for you, but requires customization to support requirements of this lab.
9. Marshal the output of the route with a starting URI of **direct:process** to XML format.
The model classes are already mapped for you, using JAXB.
To marshal or unmarshal the object, you may use the **com.redhat.training.converters.OrderConverter** class and annotate it with the **@Converter** annotation and configure the class among Camel's converters.
10. Output the XML data to a file in the **/tmp/orders** folder using the bean named **order**, the **generateRandomFileName** method to create a file name for each XML file, in the route with a URI starting point of **direct:process**.
11. Verify that the routes up to this point are correct.
Open the **src/test/java** folder, **com.redhat.training.routes.OrderRouteTest** test case, right-click it, and select **Run As → JUnit test**. The methods starting with **testAudit** do not pass because they are verifying the remaining steps of the lab. If any of the other tests fails, then review the implementation so far.
12. Implement the audit feature needed by the route with a URI starting point of **sedat:audit**.
Aggregate orders using the **batchNumber** as an aggregator that was assigned by the **assignBatch** bean and use the **BatchOrderAggregationStrategy** as the implementation to aggregate. Set the completion timeout to 2 seconds.
13. Change the file name generated from processing. In the route with a URI starting from **sedat:audit**, add a call to the **batch** bean that updates the message header to use a custom file name.
14. Update the route with a URI starting from **sedat:audit** to send the files to the correct destination directory.
From the route with a URI starting point of **direct:process**, the CBR EIP is implemented with a **choice** method call. Send large orders (over \$100) to the **/tmp/audit/large** folder and to send smaller orders to **/tmp/audit/small**.
15. Validate that the tests all pass successfully.
In the **src/test/java** folder, right-click on the **com.redhat.training.routes.OrderRouteTest** test case, and select **Run As → JUnit test**. Now all the tests pass, and you see a green bar.
16. Close the project in JBoss Developer Studio to conserve memory.
In the **Project Explorer** view, right-click **review1** and click **Close Project**.

Evaluation

Run the **lab review1** script with the **grade** argument to confirm success on this exercise. Correct any reported failures and run the script until successful.

```
[student@workstation ~]$ lab review1 grade
```

This concludes the comprehensive review lab.

► Solution

Implementing Routes with Camel

In this review, you will create a route that reads orders from two directories and groups them together as a batch order that gets sent to the manufacturers.

Outcomes

You should be able to:

- Implement aggregator EIP to create batch orders.
- Log auditing information using wire tap EIP.
- Convert CSV files to Java objects and XML files.
- Create processors and beans to customize the exchange contents transported during the route processing.

Before You Begin

Set up your environment for this exercise by logging into **workstation** as **student**, and run the following command:

```
[student@workstation ~]$ lab review1 setup
```

Instructions

In this comprehensive review, you need to create a Camel route that processes a set of files representing incoming orders from two directories on the file system. Split the incoming files into separate orders, and batch the orders together every two seconds. Calculate the total value of the orders, and transform the batch into an XML file. Each batch of orders may have a total value either over or under \$100. Store the batch XML file in a different directory depending on the total value of the orders in the batch.

A starter project is provided with all the source code needed for this laboratory. It is available in the **/home/student/JB421/labs/review1** directory.

- Read multiple CSV files representing an order from the file system using regular files that match a naming convention. The file name starts with **orders-v1** and has the suffix **.csv**.
- Two different directories store the files (**/tmp/data/orders1** and **/tmp/data/orders2**) and both must be monitored.
- The route that monitors the **/tmp/data/orders1** ID is **OrderRoute1**
- The route that monitors the **/tmp/data/orders2** ID is **OrderRoute2**.
- Improve the processing of the files using parallel processing.
- For each CSV file that the route consumes, assign a unique, sequential batch number.
- Log each file that the route processes and its unique batch number to the console.
The expected log output format is **Processing file: fileName in Batch batchNumber**.

- Convert each CSV file to multiple orders.
- Split the orders obtained during the CSV file processing. Consider using memory improvement techniques because the number of orders in each file may be large.
- Update the state field to be masked with asterisks.
- Calculate the total value of all orders in a batch generated every two seconds. For this lab purpose, the **Batch.addOrder** method is provided.
- Write all batches over \$100 to a file in the **/tmp/audit/large** directory, otherwise write the batch file to **/tmp/audit/small** in XML format to improve performance.
- The format of the file name should be **order-audit-yyMMddHHmmssBnn.xml**, where "nn" is the batch number (it does not need to be padded to two digits).
- Convert each order to XML.

- Implement a route in the **com.redhat.training.routes.OrderRouteBuilder** route builder that reads files from the **/tmp/data/orders1** directory and sends them to an endpoint named **direct:process**.

The route must read comma-separated value (CSV) files with a file pattern **orders-v1-.*.csv**. Furthermore, use a thread pool to allow multiple threads to process files simultaneously. The route ID must be **OrderRoute1** to allow test cases to test the route.

- Double-click the **com.redhat.training.routes.OrderRouteBuilder** class and add the following route in the beginning of the **configure** method declaration:

```
//TODO Read only orders with the following file name: orders-v1-.*.csv"
from("file:/tmp/data/orders1?include=orders-v1-.*.csv")
.to("direct:process");
```

- Identify the route as **OrderRoute1**. Add the following code to the created route:

```
//TODO Read only orders with the following file name: orders-v1-.*.csv"
from("file:/tmp/data/orders1?include=orders-v1-.*.csv")
.routeId("OrderRoute1")
.to("direct:process");
```

- Add a thread pool of five threads to the route. Add a **threads** method call to the created route:

```
//TODO Read only orders with the following file name: orders-v1-.*.csv"
from("file:/tmp/data/orders1?include=orders-v1-.*.csv")
.routeId("OrderRoute1")
.threads(5)
.to("direct:process");
```

- Create a new route in the **com.redhat.training.routes.OrderRouteBuilder** route builder that reads from the **/tmp/data/orders2** directory, with the same customization as the previous route, but identified as **OrderRoute2**.

- 2.1. Add the following code after the existing route:

```
//TODO Read only orders with the following file name: orders-v1-*.csv"  
from("file:/tmp/data/orders2?include=orders-v1-*.csv")  
.routeId("OrderRoute2")  
.threads(5)  
.to("direct:process");
```

3. Check that the implementation is correct so far.

Open the **src/test/java** folder. Right-click the **com.redhat.training.routes.OrderRouteTest** test case and select **Run As → JUnit test**. The test methods **testFileProcessed** and **testAcceptFile** must pass. If any of these method tests fails, review your work so far.

4. Add the batch number as part of the route processing and log it to the console.

The batch number is provided by the **com.redhat.training.beans.AssignBatchNumber** class. Update this class to use the number returned from it as the value of a header named **batchNumber**.

Include a log statement to output the batch number and the name of the file that was processed. Use the following text as your expected log output: **Processing file: \${header.CamelFileName} in Batch \${header.batchNumber}**.

- 4.1. Create the batch number and add it to the exchange as a header.

The calculation is provided by the **com.redhat.training.beans.AssignBatchNumber** class. Use the number returned from it as the value of a header named **batchNumber**.

```
//TODO set a header attribute named batchNumber and use the getBatchNumber method  
to calculate the value  
exchange.getIn().setHeader("batchNumber", getBatchNumber());
```

The processor is already configured as a bean in the **camel-context.xml** file.

- 4.2. In the route starting from **direct:process**, add the processor responsible for creating and assigning the batch number.

```
//TODO Call the processor named assignBatch configured in camel-context.xml  
.process("assignBatch")
```

- 4.3. After the **process** method call, include a log statement to output the batch number and the name of the file that was processed.

Add to the route:

```
//TODO log each batch number to the console  
.log("Processing file: ${header.CamelFileName} in Batch ${header.batchNumber}")
```

5. Convert each CSV order to an **Order** object, using the camel-bindy component.

Map each field from the **com.redhat.training.model.Order** class using the bindy annotations. The file uses the UNIX end of line character (**\n**) and the date format used by the file uses the **MM/dd/yy** format. For additional information, refer to the **/tmp/orders1** directory and open any of the **orders-v1-* .csv** files.

Add the transformation from a CSV file to an **Order** object to the route whose URI starting point is **direct:process**.

- 5.1. Edit the **com.redhat.training.model.Order** class to use bindy annotations using the bindy-camel component. The expected outcome is:

```
//imports and package declaration omitted

@CsvRecord(separator = ",", crlf = "UNIX")
@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class Order implements Serializable {

    private static final long serialVersionUID = 1L;

    @DataField(pos = 1)
    @XmlAttribute
    private String name;
    @DataField(pos = 2, pattern = "MM/dd/yy")
    @XmlAttribute
    private Date orderDate;
    @DataField(pos = 3)
    @XmlAttribute
    private String street;
    @DataField(pos = 4)
    @XmlAttribute
    private String city;
    @DataField(pos = 5)
    @XmlAttribute
    private String state;
    @DataField(pos = 6)
    @XmlAttribute
    private String book;
    @DataField(pos = 7)
    @XmlAttribute
    private int quantity;
    @DataField(pos = 8, precision = 2)
    @XmlAttribute
    private BigDecimal extendedAmount;
    @XmlAttribute
    private String filename;

    // toString method omitted

    // getters and setters omitted
}
```

- 5.2. Add the transformation from a CSV file to an **Order** object to the route whose starting URI is **direct:process**.

```
.log("Processing file: ${header.CamelFileName} in Batch ${header.batchNumber}")
//TODO Convert each exchange to a Order object using Bindy
.unmarshal().bindy(BindyType.Csv, Order.class)
```

6. Split the body to transform each object into a single order. To minimize memory usage, read it as a stream.

In the route declaration, add the following method call to the route with a URI starting point of **direct:process**:

```
.unmarshal().bindy(BindyType.Csv, Order.class)
//TODO split the list of orders to a single order
.split(body()).streaming()
```

7. Update the order's state to ** and add to the order the file name from which the order was obtained.

You may invoke the **OrderBean** class that is already mapped in the Spring XML configuration file. Also, update the **OrderBean** class to set the file name using the **CamelFileName** header value and the state with **.

- 7.1. In the route with a URI starting point of **direct:process**, add a **bean** method call to the **OrderBean** class.

The class is already mapped to the Spring XML configuration file and identified as order:

```
//TODO Call order bean to update the order state and add the origin file name
.bean("order")
```

- 7.2. Update the **OrderBean** class.

Open the **com.redhat.training.beans.OrderBean** class. Add logic to set the file name and order state to ** in the **processOrder** method:

```
//TODO Set filename associated with this order.
order.setFilename(exchange.getIn().getHeader("CamelFileName").toString());
//TODO Set order state to **
order.setState(**);
```

8. Implement the wire tap EIP to send the exchange to an alternate route using the SEDA component named **audit**. This new route creates an audit trail from the original route.

Add to the route with a URI starting point of **direct:process**:

```
//TODO use a wiretap for auditing purposes
.wireTap("seda:audit")
```

Observe that the **seda:audit** route is already created for you, but requires customization to support requirements of this lab.

9. Marshal the output of the route with a starting URL of **direct:process** to XML format.

The model classes are already mapped for you, using JAXB.

To marshal or unmarshal the object, you may use the **com.redhat.training.converters.OrderConverter** class and annotate it with the **@Converter** annotation and configure the class among Camel's converters.

- 9.1. Instantiate the **org.apache.camel.converter.jaxb.JaxbDataFormat** class to allow JAXB parsing.

In the beginning of the **configure** method, add the following code to instantiate the **JaxbDataFormat** object:

```
//TODO read classes from the com.redhat.training.model package for JAXB
JaxbDataFormat df = new JaxbDataFormat("com.redhat.training.model");
```

- 9.2. In the route with a URI starting point of **direct:process** route, add the following marshal call:

```
//TODO Convert to XML using JAXB
.marshall(df)
```

- 9.3. Use a converter to allow JAXB processing to output a file. By default it outputs to an in-memory Java object. Add the **@Converter** annotations to the **com.redhat.training.converters.OrderConverter** class and its methods.

```
//TODO Annotate as a converter
@Converter
public class OrderConverter {
    @Converter
    public static String orderToString(Order order, Exchange exchange) {
        return order.toString();
    }

    @Converter
    public static InputStream orderToIStream(Order order, Exchange exchange) {
        ByteArrayInputStream bais = new
        ByteArrayInputStream(order.toString().getBytes());
        return bais;
    }
}
```

- 9.4. To register the converter with Camel so that it is automatically used at runtime, add the necessary configuration file to the project.

Right-click **src/main/resources** and select **New → Folder** and create **META-INF/services/org/apache/camel**. Right-click the **src/main/resources/META-INF/services/org/apache/camel/** folder and select **New → File**.

Name the new file **TypeConverter**. Add the following content to the file:

```
com.redhat.training.converters.OrderConverter
```

Save the file by using **Ctrl+S**.

10. Output the XML data to a file in the **/tmp/orders** folder using the bean named **order**, the **generateRandomFileName** method to create a file name for each XML file, in the route with a URI starting point of **direct:process**.

Change the following DSL to the route with a URI starting point of **direct:process** as the final endpoint:

```
.to("file:/tmp/orders?fileName=order-${bean:order.generateRandomFileName}.xml")
```

11. Verify that the routes up to this point are correct.

Open the **src/test/java** folder, **com.redhat.training.routes.OrderRouteTest** test case, right-click it, and select **Run As → JUnit test**. The methods starting with

testAudit do not pass because they are verifying the remaining steps of the lab. If any of the other tests fails, then review the implementation so far.

12. Implement the audit feature needed by the route with a URI starting point of **sed:audit**.

Aggregate orders using the **batchNumber** as an aggregator that was assigned by the **assignBatch** bean and use the **BatchOrderAggregationStrategy** as the implementation to aggregate. Set the completion timeout to 2 seconds.

- 12.1. Aggregate orders using the batch number assigned by the **assignBatch** bean. Use the **BatchOrderAggregationStrategy** method as the aggregation implementation. Set the completion timeout to 2 seconds.

In the SEDA route, add the following code:

```
//TODO Aggregate orders based on the batchNumber
//provided by the assignBatch processor
.aggregate(header("batchNumber"), new BatchOrderAggregationStrategy())
//TODO aggregate all the order after 2 seconds.
.completionTimeout(2000)
```

Note that the **BatchOrderAggregationStrategy** class is not implemented yet.

- 12.2. Implement the **com.redhat.training.beans.BatchOrderAggregationStrategy** to process the order correctly.

Recall that the aggregator EIP implementation in Camel requires that you verify that the first exchange was not processed (it is **null**), and return the new exchange only with the batch object and the order in it.

Create the batch object during the first exchange processing and recover it from the old exchange. The batch number was added previously to the exchange as a message header. The new exchange contains only an order and it must be added to the batch object.

```
public class BatchOrderAggregationStrategy implements AggregationStrategy {

    @Override
    public Exchange aggregate(Exchange oldExchange, Exchange newExchange) {
        //TODO if this is the first aggregate execution, then
        if (oldExchange == null) {
            //TODO create a new batch instance with the value
            // obtained from batchNumber (Header attribute)
            Batch batch = new Batch((int) newExchange.getIn().getHeader("batchNumber"));
            //TODO add the order obtained from the body
            batch.addOrder(newExchange.getIn().getBody(Order.class));
            //TODO Add the order to the batch
            newExchange.getIn().setBody(batch);
            return newExchange;
        }
        //TODO otherwise get the batch stored in the oldExchange
        Batch batch = oldExchange.getIn().getBody(Batch.class);
        //TODO get the order from the newExchange
        Order order = newExchange.getIn().getBody(Order.class);
        //TODO add the order to the Batch instance
        batch.addOrder(order);
    }
}
```

```
//TODO store the batch to the oldExchange body  
oldExchange.getIn().setBody(batch);  
//TODO return the oldExchange  
return oldExchange;  
}  
}
```

13. Change the file name generated from processing. In the route with a URI starting from **seda:audit**, add a call to the **batch** bean that updates the message header to use a custom file name.

- 13.1. In the route with a URI starting from **seda:audit**, add a call to the **batch** bean that updates the message header to use a custom file name.

```
//TODO update the file name that will be generated by this aggregation  
.bean("batch")
```

- 13.2. The **batch** bean is already configured in the Spring XML configuration and it is available for completion as the **com.redhat.training.beans.BatchBean** class.

Add the following code to update the header to define a custom file name. The name is defined by the **CamelFileName** header.

```
//TODO set the CamelFileName attribute from  
// the header to "order-audit-" +  
// batchDate.format(formatter) + "B" + batchNumber + ".xml"  
exchange.getIn().setHeader("CamelFileName", "order-audit-" +  
batchDate.format(formatter) + "B" + batchNumber + ".xml");
```

14. Update the route with a URI starting from **seda:audit** to send the files to the correct destination directory.

From the to the route with a URI starting point of **direct:process**, the CBR EIP is implemented with a **choice** method call. Send large orders (over \$100) to the **/tmp/audit/large** folder and to send smaller orders to **/tmp/audit/small**.

Add the following code:

```
.choice()  
.when(xpath("number(/batch/@total) > 100"))  
//TODO if the total batch is over  
//US$ 100 than save to /tmp/audit/large  
.to("file:/tmp/audit/large")  
.otherwise()  
//TODO if the total batch is under  
// US$ 100 than save to /tmp/audit/small  
.to("file:/tmp/audit/small")
```

15. Validate that the tests all pass successfully.

In the **src/test/java** folder, right-click on the **com.redhat.training.routes.OrderRouteTest** test case, and select **Run As → JUnit test**. Now all the tests pass, and you see a green bar.

16. Close the project in JBoss Developer Studio to conserve memory.

In the **Project Explorer** view, right-click **review1** and click **Close Project**.

Evaluation

Run the **lab review1** script with the **grade** argument to confirm success on this exercise. Correct any reported failures and run the script until successful.

```
[student@workstation ~]$ lab review1 grade
```

This concludes the comprehensive review lab.

► Lab

Implementing REST-based Routes

In this review, you will create a RESTful API using Camel. The route will use REST as a client to connect to the bookstore application and review information about various publishers, including all books published by a publisher.

Outcomes

After completing this exercise, you should be able to:

- Develop a RESTful API using Camel REST DSL.
- Enrich data using Camel to integrate with an external system.

Before You Begin

Log in to **workstation** as **student**, and run the following command:

```
[student@workstation ~]$ lab review2 setup
```

Instructions

Provide a REST service to an external provider with a list of books from a publisher ID. A starter project is provided for you at </home/student/JB421/labs/review2>.

- The service must look up a Publisher ID and obtain a list of books available in the store based on the publisher's name.
- The REST service must be available on port **8081**.
- Develop a response to a GET HTTP method sent to **/publisher/name/<publisherName>**.
- The output must be a **text/xml** MIME type
- Look up publishers by name and return a full set of information for a given publisher.
- Make the REST API is available at **localhost:8080/bookstore/rest/pub/name/<publisherName>**.
- Return a 404 HTTP response code if the 204 HTTP response code (the method returns **null**) is returned from the bookstore REST API, and set the message to **Publisher not found**.
- A book list may be retrieved from the bookstore REST API at **http://localhost:8080/bookstore/rest/pub/id/<publisherID>/books**.
- Combine the publisher information and the book list.
- In order to avoid conflicts with internal requests, remove all **CamelHttp*** values from Camel's message headers.

- At any time, the `mvn camel:run` script may be executed and the web browser window may be used to send HTTP requests to `http://localhost/publisher/name/Books Incorporated`. If something fails, stop the execution and run the command again.

Steps

- Configure the REST DSL from Camel to use the `spark-rest` API and to make the REST service available on port `8081` in the `com.redhat.training.routes.WebServiceRouteBuilder` class.
- Implement an endpoint that listens for HTTP GET requests at the `http://localhost:8081/publisher/name/<publisherName>` URL. The endpoint must be able to send responses using `text/xml`. The business logic must be implemented in a different route with a consumer endpoint located at `direct:get`.
- Test the route.
- To avoid conflicts with the new route execution, remove the `CamelHttp*` header values.
- Obtain the publisher information by calling the bookstore API at `http://localhost:8080/bookstore/rest/pub/name/<publisherName>` with the HTTP GET method.
- Manage the output from the REST API call. If the HTTP response code is `204`, send the response to the endpoint named `direct:error`. If the response code is `200`, then send the response to the endpoint named `direct:enrich`.
- Test the route.
 - To verify that the REST API is working, start the route with the Camel Maven plug-in by running the following commands:

```
[student@workstation ~]$ cd JB421/labs/review2  
[student@workstation review2]$ mvn camel:run -DskipTests
```

Wait until the Camel route is started. You will see the following output:

```
INFO Started ServerConnector@358c2cd5{HTTP/1.1}{0.0.0.0:8081}  
INFO Started @4046ms
```

- Open Firefox and access the following URL to verify that the API can be accessed, using the URL `http://localhost:8081/publisher/name/a`. A `NullPointerException` exception is generated in the console window, which means that the route is not working. Evaluate the output. The message mentions that the connection was refused.

```
java.net.ConnectException: Connection refused at  
java.net.PlainSocketImpl.socketConnect(Native Method).
```

That happened because the bookstore REST API was not started.

- Start EAP to enable the REST API by running the following commands in a new terminal window:

```
[student@workstation ~]$ cd $JBOSS_HOME/bin
[student@workstation bin]$ ./standalone.sh -c standalone-full.xml
```

Wait until EAP is started.

- 7.4. Deploy the bookstore application on EAP. The application is available at **/home/student/JB421/labs/bookstore** folder. In a new terminal window, run the following commands:

```
[student@workstation ~]$ cd ~/JB421/labs/bookstore
[student@workstation bookstore]$ mvn clean wildfly:deploy -DskipTests
```

- 7.5. Open Firefox and access the following URL to verify that the API can be accessed:
<http://localhost:8081/publisher/name/will>.

The route takes a long time to process and it returns an empty response. An exception is generated in the console window running the Camel route, indicating that the route is not working. The output from the terminal window where Camel is running includes a message indicating that no route was found.

```
org.apache.camel.component.direct.DirectConsumerNotAvailableException: No
consumers available on endpoint: Endpoint[direct://error]. Exchange[ID-
workstation-lab-example-com-59455-1484224644800-0-5]
```

That is because the route is incomplete.

Stop the Camel route.

Use **Ctrl+C** in the terminal window running Camel.

8. Enrich the publisher information obtained from the bookstore response with a list of books published by the publisher.

Call a second REST API from the bookstore application. The request requires that the original response must be parsed using XPath to capture the publisher ID and send it to the bookstore application.

The HTTP GET method to get all the books from a publisher is: **http://localhost:8080/bookstore/rest/pub/id/\${publisherID}/books**.

9. Test the route implementation.

10. Return an HTTP response code 404 to the route if there are no publishers with the name provided.

11. Manually test the implementation.

- 11.1. From another terminal window, run the following commands to start the Camel route.

```
[student@workstation ~]$ cd JB421/labs/review2
[student@workstation review2]$ mvn camel:run -DskipTests
```

Wait until the route is started.

- 11.2. Open a Firefox window and, using the RESTClient plug-in, execute the following requests:

Set a custom header with the name of **Content-Type** and the value of **text/xml**.

Send an HTTP POST to `http://localhost:8081/publisher/name/Example`. This returns a 404 HTTP response code, which is expected because there is no publisher named **Example**.

Send an HTTP POST to `http://localhost:8081/publisher/name/Books Incorporated`. This returns a 200 HTTP response code. Look at the **Response** body tab and verify that all the books from **Books Incorporated** were returned.



Note

The RESTClient plug-in file is located at `/home/student/restclient.xpi`. You can open this file in Firefox to install the plug-in if needed.

12. Stop JBoss EAP.
Go back to the terminal running EAP and press **Ctrl+C**.
13. Stop the route.
Go back to the terminal running Camel and press **Ctrl+C**.
14. Close the project in JBoss Developer Studio to conserve memory.
In the **Project Explorer** view, right-click **review2** and click **Close Project**.

Evaluation

As the **student** user on **workstation**, run the **lab review2** script with the **grade** argument to confirm success on this exercise. Correct any reported failures and run the script until successful.

```
[student@workstation ~]$ lab review2 grade
```

This concludes the comprehensive review lab.

► Solution

Implementing REST-based Routes

In this review, you will create a RESTful API using Camel. The route will use REST as a client to connect to the bookstore application and review information about various publishers, including all books published by a publisher.

Outcomes

After completing this exercise, you should be able to:

- Develop a RESTful API using Camel REST DSL.
- Enrich data using Camel to integrate with an external system.

Before You Begin

Log in to **workstation** as **student**, and run the following command:

```
[student@workstation ~]$ lab review2 setup
```

Instructions

Provide a REST service to an external provider with a list of books from a publisher ID. A starter project is provided for you at </home/student/JB421/labs/review2>.

- The service must look up a Publisher ID and obtain a list of books available in the store based on the publisher's name.
- The REST service must be available on port **8081**.
- Develop a response to a GET HTTP method sent to **/publisher/name/<publisherName>**.
- The output must be a **text/xml** MIME type
- Look up publishers by name and return a full set of information for a given publisher.
- Make the REST API is available at **localhost:8080/bookstore/rest/pub/name/<publisherName>**.
- Return a 404 HTTP response code if the 204 HTTP response code (the method returns **null**) is returned from the bookstore REST API, and set the message to **Publisher not found**.
- A book list may be retrieved from the bookstore REST API at **http://localhost:8080/bookstore/rest/pub/id/<publisherID>/books**.
- Combine the publisher information and the book list.
- In order to avoid conflicts with internal requests, remove all **CamelHttp*** values from Camel's message headers.

- At any time, the **mvn camel:run** script may be executed and the web browser window may be used to send HTTP requests to `http://localhost/publisher/name/Books Incorporated`. If something fails, stop the execution and run the command again.

Steps

- Configure the REST DSL from Camel to use the **spark-rest** API and to make the REST service available on port **8081** in the **com.redhat.training.routes.WebServiceRouteBuilder** class.

Open the **com.redhat.training.routes.WebServiceRouteBuilder** class from within the **src/main/java** folder.

In the beginning of the **configure** method, configure the REST DSL by customizing it to work with the **spark-rest** component and listen to the port **8081**.

```
//TODO Configure the restComponent  
restConfiguration()  
//TODO Configure the use the spark-rest component  
    .component("spark-rest")  
//TODO Use port 8081  
    .port(8081);
```

- Implement an endpoint that listens for HTTP GET requests at the `http://localhost:8081/publisher/name/<publisherName>` URL. The endpoint must be able to send responses using **text/xml**. The business logic must be implemented in a different route with a consumer endpoint located at **direct:get**.

In the same **configure** method, but after the **restConfiguration** call, enable the endpoint.

```
//TODO Activate a REST endpoint that will receive requests  
// from /publisher/name  
rest("/publisher/name")  
//TODO Response to an HTTP GET request. Read the parameter  
  
// and generate an XML response  
    .get("{name}").produces("text/xml")  
//TODO send to a second route  
    .to("direct:get");
```

- Test the route.

- To verify that the REST API is working, start Camel by running the following commands:

```
[student@workstation ~]$ cd JB421/labs/review2  
[student@workstation review2]$ mvn camel:run -DskipTests
```

Wait until the Camel route is started. You see the following output:

```
Started ServerConnector@358c2cd5{HTTP/1.1}{0.0.0.0:8081}  
INFO org.eclipse.jetty.server.Server - Started @4046ms
```

- Open a web browser and verify that the endpoint is working.

Open Firefox and access the following URL: `http://localhost:8081`. A **Not Found** error is raised because there is no REST API available. To verify that the API can be accessed, use the following URL: `http://localhost:8081/publisher/name/will`. A stack trace will be listed, which means that the route is not implemented yet.

- 3.3. Stop the Camel route.

Use **Ctrl+C** in the terminal window running Camel.

4. To avoid conflicts with the new route execution, remove the **CamelHttp*** header values.

```
// TODO Eliminate all the headers starting with
// CamelHttp to avoid confusion
// with the bookstore app
.removeHeaders("CamelHttp*")
```

5. Obtain the publisher information by calling the bookstore API at `http://localhost:8080/bookstore/rest/pub/name/<publisherName>` with the HTTP GET method.

- 5.1. In the same route, set the header to call the HTTP GET method.

```
//TODO send header to get information
//from the bookstore REST API
.setHeader(Exchange.HTTP_METHOD, constant("GET"))
```

- 5.2. Send the request to the API `http://localhost:8080/bookstore/rest/pub/name/<publisherName>`. Recall that the name was stored in the header in the route execution.

```
// TODO Send a request to the bookstore
// application /rest/pub/name/
// and send the header field named name
.toD("http://localhost:8080/bookstore/rest/pub/name/${header.name}")
```

6. Manage the output from the REST API call. If the HTTP response code is **204**, send the response to the endpoint named **direct:error**. If the response code is **200**, then send the response to the endpoint named **direct:enrich**.

```
.choice()
// TODO Evaluate the response If the response is 204
.when(header("CamelHttpResponseCode").isEqualTo("204"))
//TODO manage with the route direct:error
.to("direct:error")
//TODO If the response is 200 (success)
.when(header("CamelHttpResponseCode").isEqualTo("200"))
//TODO Enrich the data
.to("direct:enrich")
.end();
```

7. Test the route.

- 7.1. To verify that the REST API is working, start the route with the Camel Maven plug-in by running the following commands:

```
[student@workstation ~]$ cd JB421/labs/review2  
[student@workstation review2]$ mvn camel:run -DskipTests
```

Wait until the Camel route is started. You will see the following output:

```
INFO Started ServerConnector@358c2cd5{HTTP/1.1}{0.0.0.0:8081}  
INFO Started @4046ms
```

- 7.2. Open Firefox and access the following URL to verify that the API can be accessed, using the URL `http://localhost:8081/publisher/name/a`. A **NullPointerException** exception is generated in the console window, which means that the route is not working. Evaluate the output. The message mentions that the connection was refused.

```
java.net.ConnectException: Connection refused at  
java.net.PlainSocketImpl.socketConnect(Native Method).
```

That happened because the bookstore REST API was not started.

- 7.3. Start EAP to enable the REST API by running the following commands in a new terminal window:

```
[student@workstation ~]$ cd $JBOSS_HOME/bin  
[student@workstation bin]$ ./standalone.sh -c standalone-full.xml
```

Wait until EAP is started.

- 7.4. Deploy the bookstore application on EAP. The application is available at `/home/student/JB421/labs/bookstore` folder. In a new terminal window, run the following commands:

```
[student@workstation ~]$ cd ~/JB421/labs/bookstore  
[student@workstation bookstore]$ mvn clean wildfly:deploy -DskipTests
```

- 7.5. Open Firefox and access the following URL to verify that the API can be accessed: `http://localhost:8081/publisher/name/will`.

The route takes a long time to process and it returns an empty response. An exception is generated in the console window running the Camel route, indicating that the route is not working. The output from the terminal window where Camel is running includes a message indicating that no route was found.

```
org.apache.camel.component.direct.DirectConsumerNotAvailableException: No  
consumers available on endpoint: Endpoint[direct://error]. Exchange[ID-  
workstation-lab-example-com-59455-1484224644800-0-5]
```

That is because the route is incomplete.

Stop the Camel route.

Use **Ctrl+C** in the terminal window running Camel.

8. Enrich the publisher information obtained from the bookstore response with a list of books published by the publisher.

Call a second REST API from the bookstore application. The request requires that the original response must be parsed using XPath to capture the publisher ID and send it to the bookstore application.

The HTTP GET method to get all the books from a publisher is: **http://localhost:8080/bookstore/rest/pub/id/\${publisherID}/books**.

- 8.1. Remove the headers named **CamelHttp*** to avoid collisions with later requests from Camel in the **direct:enrich** route.

```
//TODO Remove any existing header with HTTP protocol information  
.removeHeaders("CamelHttp")
```

- 8.2. Configure the route to send the HTTP GET method to the bookstore REST API:

```
//TODO configure the request to GET the remaining information.  
.setHeader(Exchange.HTTP_METHOD, constant("GET"))
```

- 8.3. Enrich the data obtained from **http://localhost:8080/bookstore/rest/pub/id/\${publisherID}/books**.

```
.enrich().simple("http://localhost:8080/bookstore/rest/pub/id/" +  
    "${header.publisher_id}/books")
```

- 8.4. Aggregate the result from the first request to the second request by calling the aggregator EIP using the provided **PublisherAggregate**.

```
.enrich()  
.simple("http://localhost:8080/bookstore/rest/pub/id/" +  
    "${header.publisher_id}/books")  
.aggregationStrategy(new PublisherAggregate());
```

9. Test the route implementation.

- 9.1. To verify that the REST API is working, start Camel for testing purposes. From a new terminal window, run the following commands:

```
[student@workstation ~]$ cd JB421/labs/review2  
[student@workstation review2]$ mvn camel:run -DskipTests
```

Wait until the Camel route is started. Once started, you see the following output:

```
INFO - Started ServerConnector@358c2cd5{HTTP/1.1}{0.0.0.0:8081}  
INFO - Started @4046ms
```

Open Firefox and access the following URL to verify that the API can be accessed, using the URL **http://localhost:8081/publisher/name/Books Incorporated**. An XML output is listed instead, meaning that the main route is working. However, accessing a different address generates a long response time and an empty response with an route processing error, similar to the previous ones observed.

Stop the Camel route.

Use **Ctrl+C** in the terminal window that Camel was started.

10. Return an HTTP response code 404 to the route if there are no publishers with the name provided.

10.1. Uncomment the following code to enable the error route:

```
from("direct:error")
    .routeId("ProcessError");
```

10.2. In the route with the **direct:error** end point, add the following code:

```
.setBody(constant("Publisher not found."))
.setHeader(Exchange.HTTP_RESPONSE_CODE, constant("404"))
```

11. Manually test the implementation.

11.1. From another terminal window, run the following commands to start the Camel route.

```
[student@workstation ~]$ cd JB421/labs/review2
[student@workstation review2]$ mvn camel:run -DskipTests
```

Wait until the route is started.

- 11.2. Open a Firefox window and, using the RESTClient plug-in, execute the following requests:

Set a custom header with the name of **Content-Type** and the value of **text/xml**.

Send an HTTP POST to `http://localhost:8081/publisher/name/Example`. This returns a 404 HTTP response code, which is expected because there is no publisher named **Example**.

Send an HTTP POST to `http://localhost:8081/publisher/name/Books Incorporated`. This returns a 200 HTTP response code. Look at the **Response** body tab and verify that all the books from **Books Incorporated** were returned.



Note

The RESTClient plug-in file is located at `/home/student/restclient.xpi`. You can open this file in Firefox to install the plug-in if needed.

12. Stop JBoss EAP.

Go back to the terminal running EAP and press **Ctrl+C**.

13. Stop the route.

Go back to the terminal running Camel and press **Ctrl+C**.

14. Close the project in JBoss Developer Studio to conserve memory.

In the **Project Explorer** view, right-click **review2** and click **Close Project**.

Evaluation

As the **student** user on **workstation**, run the **lab review2** script with the **grade** argument to confirm success on this exercise. Correct any reported failures and run the script until successful.

```
[student@workstation ~]$ lab review2 grade
```

This concludes the comprehensive review lab.

► Lab

Managing Transacted Routes with Camel

In this review, you process orders received from a message queue, convert the order to a compatible Java model class, and store them in a relational database. To avoid integrity problems, the route can roll back the route execution if anything from the database is corrupted. Finally, each order is sent to a message queue destination based on the order value.

Outcomes

After completing this exercise, you should be able to:

- Read **OrderReceived** objects from an ActiveMQ queue.
- Convert the **OrderReceived** object into an **Order** object compatible with the database.
- Using a custom type processor, convert the **OrderReceived** object to the **Order** object needed by the JPA component.
- Write the order to the database using JPA component.
- Transact the route.
- Roll back if a database exception is thrown, and attempt up to the default number of deliveries of the message before it fails.
- Read orders from the database in a separate route using the JPA consumer component with a named query.
- Convert the database record to XML.
- Send orders greater than \$100 to a queue named **largeOrders** and any smaller orders to one named **smallOrders**.
- Force the type of message to **Text** in the JMS URI. Use the recipient list EIP to do this with a header containing the URI.

Before You Begin

Log in to **workstation** as **student**, and run the following command:

```
[student@workstation ~]$ lab review3 setup
```

Instructions

A company processes orders from a message queue and needs to convert the data to be compatible with its database. The order must be transformed to an XML file that can be exported. If something goes wrong, the order must be sent back to the queue. A starter project is provided at </home/student/JB421/labs/review3>.

- Read data from the message queue named **orders**.
- Transform the order received in the message queue to be compatible with the database model.

- Convert the data to an XML format and send it to the appropriate destination according to the following rule:
 - Send orders greater than \$100 to a queue named **largeOrders**.
 - Send any smaller orders to one named **smallOrders**.

Steps

- In the first route, configure the destination to read data from the **orders** message queue and use the ActiveMQ option **transacted=true**.
- Enable transaction management in the route with a starting URI of **activemq:queue:orders**.

```
//TODO define the route as transacted
.transacted()
```

- Transform the data obtained from the queue to become compatible with the data stored in the database. Configure the **com.redhat.training.OrderConverter** class as a Camel converter so you can use it in the Camel route. Then update the route definition to include a call to the **convertBodyTo** method to convert the **OrderReceived** object sent by the ActiveMQ queue to an **Order** object.
- In the route with a starting URI of **activemq:queue:orders**, define the destination endpoint as the JPA endpoint to store the orders in the database.
- Convert the output from the database to XML using JAXB.
- In the route with a starting URI of **jpa:com.redhat.training.model.Order**, in the **TransactedRouteBuilder** class, configure the destination URI using the recipient list EIP to define the correct queue.
The **com.redhat.training.beans.DestinationBean** class identifies the order amount and defines the destination. The URI must be added to an exchange header with the name **recipients**.
- Implement the error handling for the **java.net.ConnectException** exception, log the message **Connection to database failed. Will re-attempt.**, and roll back the transaction if the transaction fails.

Evaluation

As the **student** user on **workstation**, run the **lab review3** script with the **grade** argument, to confirm success on this exercise. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab review3 grade
```

This concludes the comprehensive review lab.

► Solution

Managing Transacted Routes with Camel

In this review, you process orders received from a message queue, convert the order to a compatible Java model class, and store them in a relational database. To avoid integrity problems, the route can roll back the route execution if anything from the database is corrupted. Finally, each order is sent to a message queue destination based on the order value.

Outcomes

After completing this exercise, you should be able to:

- Read **OrderReceived** objects from an ActiveMQ queue.
- Convert the **OrderReceived** object into an **Order** object compatible with the database.
- Using a custom type processor, convert the **OrderReceived** object to the **Order** object needed by the JPA component.
- Write the order to the database using JPA component.
- Transact the route.
- Roll back if a database exception is thrown, and attempt up to the default number of deliveries of the message before it fails.
- Read orders from the database in a separate route using the JPA consumer component with a named query.
- Convert the database record to XML.
- Send orders greater than \$100 to a queue named **largeOrders** and any smaller orders to one named **smallOrders**.
- Force the type of message to **Text** in the JMS URI. Use the recipient list EIP to do this with a header containing the URI.

Before You Begin

Log in to **workstation** as **student**, and run the following command:

```
[student@workstation ~]$ lab review3 setup
```

Instructions

A company processes orders from a message queue and needs to convert the data to be compatible with its database. The order must be transformed to an XML file that can be exported. If something goes wrong, the order must be sent back to the queue. A starter project is provided at [/home/student/JB421/labs/review3](#).

- Read data from the message queue named **orders**.
- Transform the order received in the message queue to be compatible with the database model.

- Convert the data to an XML format and send it to the appropriate destination according to the following rule:
 - Send orders greater than \$100 to a queue named **largeOrders**.
 - Send any smaller orders to one named **smallOrders**.

Steps

- In the first route, configure the destination to read data from the **orders** message queue and use the ActiveMQ option **transacted=true**.

Update the first route from the configure method to read information from an ActiveMQ queue named **orders** with the **transacted=true** option.

```
//route from the order
from("activemq:queue:orders?transacted=true")
```

- Enable transaction management in the route with a starting URI of **activemq:queue:orders**.

Add the following method call:

```
//TODO define the route as transacted
.transacted()
```

- Transform the data obtained from the queue to become compatible with the data stored in the database. Configure the **com.redhat.training.OrderConverter** class as a Camel converter so you can use it in the Camel route. Then update the route definition to include a call to the **convertBodyTo** method to convert the **OrderReceived** object sent by the ActiveMQ queue to an **Order** object.

- 3.1. The transformation of an **OrderReceived** class to an **Order** class is made by the **com.redhat.training.converters.OrderConverter** class, but it is not configured. Add **@Converter** annotations to the **com.redhat.training.converters.OrderConverter** class.

```
//TODO Annotate as a converter
@Converter
public class OrderConverter {
    @Converter
    public static OrderReceived orderToOrderReceived(Order order, Exchange exchange)
    {
        return new OrderReceived(order.getId(), order.getName(), order.getItem(),
order.getQuantity(),
            order.getExtendedAmount().divide(new BigDecimal(order.getQuantity())));
    }

    @Converter
    public static Order OrderReceivedToOrder(OrderReceived order, Exchange exchange)
    {
        return new Order(order.getId(), order.getCustomerName(), order.getItemName(),
order.getQuantity(),
```

```

        order.getPrice().multiply(new BigDecimal(order.getQuantity())));
    }
}

```

- 3.2. Right-click **src/main/resources** and select **New → Folder** to create the **META-INF/services/org/apache/camel** directory. Right-click the **src/main/resources/META-INF/services/org/apache/camel/** folder and select **New → File**." Name the new file **TypeConverter**. Open it and add the following content:

```
com.redhat.training.converters.OrderConverter
```

Save the file by using **Ctrl+S**.

- 3.3. Add a call to the new converter in the first route.

Add the following call to convert the **OrderReceived** to an **com.redhat.training.model.Order** object.

```
//TODO convert the received object (an OrderReceived object to an Order object)
.convertBodyTo(Order.class)
```

4. In the route with a starting URI of **activemq:queue:orders**, define the destination endpoint as the JPA endpoint to store the orders in the database.

Update the route definition to include a call to the JPA component to store the orders in the database:

```
// TODO store in the db using JPA
.to("jpa:com.redhat.training.model.Order");
```

5. Convert the output from the database to XML using JAXB.

- 5.1. Instantiate the **org.apache.camel.model.dataformat.JaxbDataFormat** class in the beginning of the **configure** method from the route.

```
//TODO Instantiate the JaxbDataFormat
JaxbDataFormat jaxb = new JaxbDataFormat(true);
```

- 5.2. In the route with a starting URI of **jpa:com.redhat.training.model.Order**, add the following code before the recipient list DSL:

```
//TODO convert to XML
.marshal(jaxb)
```

6. In the route with a starting URI of **jpa:com.redhat.training.model.Order**, in the **TransactedRouteBuilder** class, configure the destination URI using the recipient list EIP to define the correct queue.

The **com.redhat.training.beans.DestinationBean** class identifies the order amount and defines the destination. The URI must be added to an exchange header with the name **recipients**.

- 6.1. To call the bean, add the following method call to the route.

```
//TODO define the destination URI for the appropriate activemq queue in the  
recipients header.  
.setHeader("recipients", method(DestinationBean.class))
```

- 6.2. Call the recipient list EIP. In the final step, add the following method call, recalling that the destination was added to the header and named as recipients.

```
// TODO send the order to the correct destination using the Recipient list EIP  
.recipientList(header("recipients"));
```

7. Implement the error handling for the **java.net.ConnectException** exception, log the message **Connection to database failed. Will re-attempt.**, and roll back the transaction if the transaction fails.

Update the route definition to include the **onException** method to handle the **ConnectException**:

```
onException(ConnectException.class)  
.handled(true)  
.log("Connection to database failed. Will re-attempt.")  
.markRollbackOnly();
```

Evaluation

As the **student** user on **workstation**, run the **lab review3** script with the **grade** argument, to confirm success on this exercise. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab review3 grade
```

This concludes the comprehensive review lab.

► Lab

Deploying Microservices With Fuse on OpenShift

In this lab, you will deploy two REST microservices implemented using the Camel REST DSL and Spring Boot with Fuse on OpenShift.

Outcomes

After completing this exercise, you should be able to:

- Complete the catalog-service project's POM file to dependencies required by Camel and Spring Boot.
- Complete the Fabric8 Maven Plug-in configuration to use the supported Fuse Spring Boot container image.
- Complete the Fabric8 resource fragment files that defines a route to use a short host name and health checks.
- Update the catalog-service route definition to include a circuit breaker for the call to the vendor-service using the **hystrix** EIP provided by Camel.
- Test the applications running on OpenShift using the **curl** command and verify the circuit breaker and fallback behavior.

Before You Begin

Log in to **workstation** as **student**, and run the following command:

```
[student@workstation ~]$ lab review4 setup
```

Instructions

The vendor-service REST service provides a single endpoint that retrieves vendor data from the bookstore database by ID. The data resides inside a MySQL database server located on the **services** machine.

The vendor-service REST endpoint takes a single argument, the vendor ID, and returns JSON data. The resource URI is **/camel/vendor/{id}**.

The vendor-service REST service is ready for deployment. You need to deploy it on the **review4-lab** project. The project is provided with all the source code needed for this laboratory. It is available in the **/home/student/JB421/labs/review4/vendor-service** directory.

The catalog-service REST service provides a single endpoint that retrieves catalog item data in the bookstore database. The catalog-service REST endpoint takes a single argument, the catalog item ID. The resource URI is **/camel/catalog/{id}**.

The catalog-service REST service retrieves the vendor name from the vendor-service REST service. A starter project is provided with all the source code needed for this laboratory. It is available in the **/home/student/JB421/labs/review4/catalog-service** directory.

- Create a new project called **review4-lab**.
- Deploy the vendor-service REST service on OpenShift.
- Update the **pom.xml** configuration file in the catalog-service project to deploy the application using the Fabric8 Maven Plug-in and to configure the required dependencies from Spring Boot.
- Configure the Fabric8 Maven Plug-in in the catalog-service REST service:
 - Configure the liveness probe to refer to the **/health** endpoint. This endpoint is available in the **8180** port.
 - Configure the readiness probe to refer to the **/health** endpoint. This endpoint is available in the **8180** port.
 - Configure the route to access the service by using the `http://catalog.apps.lab.example.com` host name.
- Implement a database health check. The application is considered healthy if the application can run a simple SQL script.
- Update the catalog-service REST service to call the vendor-service using the Camel HTTP4 component and the provided **vendorHost** and **vendorPort** properties.
- Add the circuit breaker pattern to the route to protect the call from the **catalog-service** to the vendor-service. Include the following configuration for the circuit breaker:
 - Any execution over three seconds must time out.
 - The circuit breaker must receive a minimum of two requests before the circuit can open.
 - The circuit breaker must only monitor a rolling window of the last 60 seconds of requests when deciding whether to open.
 - When opened, the circuit breaker must wait 20 seconds before attempting to close again.
 - The circuit breaker must open if greater than 50 percent of the requests are failing.
 - If there is a failure, or the circuit is open, the message **Vendor Not Available!** must be used as a fallback.

Steps

1. Create the **review4-lab** project on an OpenShift cluster and deploy the **vendor-service** microservice.
2. Complete the Fabric8 Maven Plug-in configuration to use the supported Fuse container image for the **catalog-service** microservice. Use the **openshift/fuse7-java-openshift:1.1** image stream from the **openshift** project.
3. Complete the Fabric8 Maven Plug-in resource fragments:
 - Use the **/health** endpoint to configure the readiness probe. This endpoint is available in the **8182** port.

- Use the **/health** endpoint to configure the liveness probe. This endpoint is available in the **8182** port.
 - Use the `catalog.apps.lab.example.com` host name to access the application.
4. Update the **DatabaseHealthCheck** class to create a health endpoint. This class returns a healthy status if the "**select 1**" SQL command executes successfully.
 5. Update the **RestRouteBuilder** class to invoke the **SqlProcessor** processor right after the **sql** component. This processor adds the returned columns from the database to Camel headers that will be required later.
 6. Update the **RestRouteBuilder** class to invoke the **vendor-service** microservice using the Camel HTTP4 component and the provided **vendorHost** and **vendorPort** properties. Recover the vendor id from the **catalog_vendor_id** header defined on the **SqlProcessor** processor.
 7. Update the **RestRouteBuilder** class to invoke the **VendorProcessor** processor right after the **HTTP4** component. This processor adds the vendor name to a Camel header.
 8. Update the **RestRouteBuilder** class to add the circuit breaker pattern to the call from the **catalog-service** to the vendor-service. Include the following configuration for the circuit breaker:
 - Any execution over three seconds must time out.
 - The circuit breaker must receive a minimum of two requests before the circuit can open.
 - The circuit breaker must only monitor a rolling window of the last 60 seconds of requests when deciding whether to open.
 - When opened, the circuit breaker must wait 20 seconds before attempting to close again.
 - The circuit breaker must open if greater than 50 percent of the requests are failing.
 - If there is a failure, or the circuit is open, the message **Vendor Not Available!** must be used as a fallback.
 9. Update the **RestRouteBuilder** class to invoke the **ResponseProcessor** processor. This processor will create a **ResponseVO** object from the headers and define it as the body of the router.
 10. Deploy the **catalog-service** microservice.
 11. Test the microservice applications on OpenShift.

11.1. Wait for the application pod to be ready and running.

Run the **oc get pod** command until the output resembles the following:

[student@workstation catalog-service]\$ oc get pod					
NAME	READY	STATUS	RESTARTS	AGE	
catalog-service-1-mwvl8	1/1	Running	0	33s	
catalog-service-s2i-1-build	0/1	Completed	0	58s	
vendor-service-1-w62kv	1/1	Running	0	13m	
vendor-service-s2i-1-build	0/1	Completed	0	13m	

11.2. Open a Firefox window and, using the RESTClient plug-in, execute the following request:

Send an HTTP GET to `http://vendor.apps.lab.example.com/camel/vendor/1`. This returns a 200 HTTP response code. Look at the **Response** body tab:

```
{"id":1,"name":"Bookmart, Inc."}
```

11.3. Send an HTTP GET to `http://catalog.apps.lab.example.com/camel/catalog/1`. This returns a 200 HTTP response code. Look at the **Response** body:

```
{"id":1,"description":"description 1","author":"Lt. Howard Payson","vendorName":"Bookmart, Inc."}
```

11.4. Run the **block-vendor.sh** bash script to block connections to the **vendor-service** microservice.

```
[student@workstation catalog-service]$ cd ..
[student@workstation review4]$ ./block-vendor.sh
```

11.5. Send an HTTP GET to `http://catalog.apps.lab.example.com/camel/catalog/1`. This returns a 500 HTTP response code. Look at the **Response** body:

```
ERROR Locating Vendor
```

Evaluation

As the **student** user on **workstation**, run the **lab review4** script with the **grade** argument, to confirm success on this exercise. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab review4 grade
```

This concludes the comprehensive review lab.

► Solution

Deploying Microservices With Fuse on OpenShift

In this lab, you will deploy two REST microservices implemented using the Camel REST DSL and Spring Boot with Fuse on OpenShift.

Outcomes

After completing this exercise, you should be able to:

- Complete the catalog-service project's POM file to dependencies required by Camel and Spring Boot.
- Complete the Fabric8 Maven Plug-in configuration to use the supported Fuse Spring Boot container image.
- Complete the Fabric8 resource fragment files that defines a route to use a short host name and health checks.
- Update the catalog-service route definition to include a circuit breaker for the call to the vendor-service using the **hystrix** EIP provided by Camel.
- Test the applications running on OpenShift using the **curl** command and verify the circuit breaker and fallback behavior.

Before You Begin

Log in to **workstation** as **student**, and run the following command:

```
[student@workstation ~]$ lab review4 setup
```

Instructions

The vendor-service REST service provides a single endpoint that retrieves vendor data from the bookstore database by ID. The data resides inside a MySQL database server located on the **services** machine.

The vendor-service REST endpoint takes a single argument, the vendor ID, and returns JSON data. The resource URI is **/camel/vendor/{id}**.

The vendor-service REST service is ready for deployment. You need to deploy it on the **review4-lab** project. The project is provided with all the source code needed for this laboratory. It is available in the **/home/student/JB421/labs/review4/vendor-service** directory.

The catalog-service REST service provides a single endpoint that retrieves catalog item data in the bookstore database. The catalog-service REST endpoint takes a single argument, the catalog item ID. The resource URI is **/camel/catalog/{id}**.

The catalog-service REST service retrieves the vendor name from the vendor-service REST service. A starter project is provided with all the source code needed for this laboratory. It is available in the **/home/student/JB421/labs/review4/catalog-service** directory.

- Create a new project called **review4-lab**.
- Deploy the vendor-service REST service on OpenShift.
- Update the **pom.xml** configuration file in the catalog-service project to deploy the application using the Fabric8 Maven Plug-in and to configure the required dependencies from Spring Boot.
- Configure the Fabric8 Maven Plug-in in the catalog-service REST service:
 - Configure the liveness probe to refer to the **/health** endpoint. This endpoint is available in the **8180** port.
 - Configure the readiness probe to refer to the **/health** endpoint. This endpoint is available in the **8180** port.
 - Configure the route to access the service by using the `http://catalog.apps.lab.example.com` host name.
- Implement a database health check. The application is considered healthy if the application can run a simple SQL script.
- Update the catalog-service REST service to call the vendor-service using the Camel HTTP4 component and the provided **vendorHost** and **vendorPort** properties.
- Add the circuit breaker pattern to the route to protect the call from the **catalog-service** to the vendor-service. Include the following configuration for the circuit breaker:
 - Any execution over three seconds must time out.
 - The circuit breaker must receive a minimum of two requests before the circuit can open.
 - The circuit breaker must only monitor a rolling window of the last 60 seconds of requests when deciding whether to open.
 - When opened, the circuit breaker must wait 20 seconds before attempting to close again.
 - The circuit breaker must open if greater than 50 percent of the requests are failing.
 - If there is a failure, or the circuit is open, the message **Vendor Not Available!** must be used as a fallback.

Steps

1. Create the **review4-lab** project on an OpenShift cluster and deploy the **vendor-service** microservice.
 - 1.1. Log in to the OpenShift cluster at `master.lab.example.com` as the **developer** user with the password **redhat**.
Open a terminal window and run the **oc login** command. If the **oc login** command prompts you about using insecure connections, answer **y**:

```
[student@workstation ~]$ oc login -u developer -p redhat \
https://master.lab.example.com
...
Use insecure connections? (y/n): y
Login successful.
...
```

- 1.2. Create the **review4-lab** project on OpenShift.

Run the **oc new-project** command:

```
[student@workstation ~]$ oc new-project review4-lab
Now using project "review4-lab" on server "https://master.lab.example.com:443".
...
```

- 1.3. Use the Fabric8 Maven Plug-in to build a container image for the **vendor-service** microservice and create the required resources on OpenShift.

Enter the **~/JB421/labs/review4/vendor-service** folder and invoke the **fabric8:deploy** Maven goal, from the **openshift** Maven profile:

```
[student@workstation ~]$ cd ~/JB421/labs/review4/vendor-service
[student@workstation vendor-service]$ mvn -Popenshift fabric8:deploy
...
[INFO] -----
[INFO] Building Review: Deploying Microservices With Fuse on OpenShift - Vendor
Service 1.0
[INFO] -----
...
[INFO] --- fabric8-maven-plugin:3.5.33.fuse-710023-redhat-00002:resource (default)
@ vendor-service ---
[INFO] F8: Running in OpenShift mode
[INFO] F8: Using docker image name of namespace: review4-lab
[INFO] F8: Running generator spring-boot
[INFO] F8: spring-boot: Using ImageStreamTag 'fuse7-java-openshift:1.1' from
namespace 'openshift' as builder image
...
[INFO] F8: Creating BuildServiceConfig vendor-service-s2i for Source build
[INFO] F8: Creating ImageStream vendor-service
[INFO] F8: Starting Build vendor-service-s2i
```

You can safely ignore warnings and errors similar to the following from the **mvn** command. Notice that the image build step is successful despite the error that follows it.

```
...
[INFO] F8: Starting S2I Java Build .....
...
[INFO] F8: {"kind":"Status","apiVersion":"v1","metadata":{},
"status":"Failure","message":"container \"sti-build\" in pod
\"vendor-service-solution-s2i-1-build\" is waiting to start:
PodInitializing","reason":"BadRequest","code":400}
```

```
[INFO] Current reconnect backoff is 1000 milliseconds (T0)
[INFO] Current reconnect backoff is 2000 milliseconds (T1)
[INFO] Current reconnect backoff is 4000 milliseconds (T2)
[INFO] Current reconnect backoff is 8000 milliseconds (T3)
[INFO] Current reconnect backoff is 16000 milliseconds (T4)
[INFO] Current reconnect backoff is 32000 milliseconds (T5)
[INFO] Current reconnect backoff is 32000 milliseconds (T5)

[ERROR] Exception in reconnect
java.util.concurrent.RejectedExecutionException: Task
java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutureTask@5f123860
rejected from java.util.concurrent.ScheduledThreadPoolExecutor@1a10ade6[Shutting
down, pool size = 1, active threads = 1, queued tasks = 0, completed tasks = 12]
...
[INFO] BUILD SUCCESS
...
```

- 1.4. Run the **mvn** command from the previous step again if you see this error during the Maven build:

```
...
[ERROR] Exception in reconnect
java.util.concurrent.RejectedExecutionException: ...
...
[ERROR] F8: Failed to execute the build [Unable to build the image using the
OpenShift build service]
...
```

2. Complete the Fabric8 Maven Plug-in configuration to use the supported Fuse container image for the **catalog-service** microservice. Use the **openshift/fuse7-java-openshift:1.1** image stream from the **openshift** project.

Use the following listing as a reference for the changes to make to the POM file:

```
...
<plugins>
  <plugin>
    <groupId>io.fabric8</groupId>
    <artifactId>fabric8-maven-plugin</artifactId>
    <version>${fabric8.maven.plugin.version}</version>
    <configuration>
      <generator>
        <config>
          <spring-boot>
            <!-- TODO: configure the image stream name -->
            <fromMode>istag</fromMode>
            <from>openshift/fuse7-java-openshift:1.1</from>
          </spring-boot>
        </config>
      </generator>
    </configuration>
    <executions>
      ...
    </executions>
  </plugin>
</plugins>
```

3. Complete the Fabric8 Maven Plug-in resource fragments:

- Use the **/health** endpoint to configure the readiness probe. This endpoint is available in the **8182** port.
- Use the **/health** endpoint to configure the liveness probe. This endpoint is available in the **8182** port.
- Use the `catalog.apps.lab.example.com` host name to access the application.

3.1. Configure the readiness probe.

In the **Project Explorer** view, expand **catalog-service → src/main/fabric8**. Double-click the **deployment.yml** file.

Change the **path** attribute to refer to the **/health** endpoint.

Change the **port** attribute to **8182**.

Use the following listing as a reference for the changes to make to the route resource fragment file:

```
livenessProbe:
  failureThreshold: 3
  httpGet:
    #TODO change the readiness probe path
    path: /health
    #TODO change the readiness probe port
    port: 8182
    scheme: HTTP
```

3.2. Configure the liveness probe.

Change the **path** attribute to refer to the **/health** endpoint.

Change the **port** attribute to **8182**.

Use the following listing as a reference for the changes to make to the route resource fragment file:

```
livenessProbe:
  failureThreshold: 3
  httpGet:
    #TODO change the liveness probe path
    path: /health
    #TODO change the liveness probe port
    port: 8182
    scheme: HTTP
```

3.3. Complete the route resource fragment file.

In the **Project Explorer** view, expand the **catalog-service → src/main/fabric8**. Double-click the **route.yml** file.

Change the **host** attribute to refer to the `catalog.apps.lab.example.com` host name.

Use the following listing as a reference for the changes to make to the route resource fragment file:

```
spec:  
  port:  
    targetPort: 8082  
  to:  
    kind: Service  
    name: ${project.artifactId}  
  #TODO change the host name of the OpenShift route  
  host: catalog.apps.lab.example.com
```

4. Update the **DatabaseHealthCheck** class to create a health endpoint. This class returns a healthy status if the "select 1" SQL command executes successfully.

- 4.1. Double-click the **com.redhat.training.jb421.health.DatabaseHealthCheck** class and add implements the **HealthIndicator** interface:

```
...  
public class DatabaseHealthCheck implements HealthIndicator {  
...
```

- 4.2. Update the **health()** method to check if the application is healthy.

```
public Health health() {  
  
    try {  
        EntityManager entityManager = entityManagerFactory.createEntityManager();  
        Query q = entityManager.createNativeQuery("select 1");  
        q.getFirstResult();  
        return Health.up().build();  
    } catch(Exception e) {  
        return Health.down(e).build();  
    }  
}
```

5. Update the **RestRouteBuilder** class to invoke the **SqlProcessor** processor right after the **sql** component. This processor adds the returned columns from the database to Camel headers that will be required later.

Double-click the **com.redhat.training.jb421.RestRouteBuilder** class and invoke the **SqlProcessor** processor.

```
.to("sql:select id, author, description, vendor_id as vendorId from  
bookstore.CatalogItem where id=:#catalogId"  
+ "?dataSource=mysqlDataSource&outputType=SelectOne"  
+ "&outputClass=com.redhat.training.jb421.model.CatalogItem")  
  
//TODO: invoke the SqlProcessor  
.process(new SqlProcessor())
```

6. Update the **RestRouteBuilder** class to invoke the **vendor-service** microservice using the Camel HTTP4 component and the provided **vendorHost** and **vendorPort** properties. Recover the vendor id from the **catalog_vendor_id** header defined on the **SqlProcessor** processor.

```

from("direct:getVendor")
    .removeHeader(Exchange.HTTP_URI)
    //TODO: Add the catalog_vendor_id header
    .setHeader(Exchange.HTTP_PATH, simple("${header.catalog_vendor_id}"))
    .setHeader(Exchange.HTTP_METHOD, simple("GET"))
    //TODO: Invoke the vendor-service microservice
    .to("http4:"+ vendorHost ":"+ vendorPort +"/camel/vendor")

```

7. Update the **RestRouteBuilder** class to invoke the **VendorProcessor** processor right after the **HTTP4** component. This processor adds the vendor name to a Camel header.

```

//TODO: Invoke the vendor-service microservice
.to("http4:"+ vendorHost ":"+ vendorPort +"/camel/vendor")
//TODO: Invoke the VendorProcessor
.process(new VendorProcessor())

```

8. Update the **RestRouteBuilder** class to add the circuit breaker pattern to the call from the **catalog-service** to the vendor-service. Include the following configuration for the circuit breaker:

- Any execution over three seconds must time out.
- The circuit breaker must receive a minimum of two requests before the circuit can open.
- The circuit breaker must only monitor a rolling window of the last 60 seconds of requests when deciding whether to open.
- When opened, the circuit breaker must wait 20 seconds before attempting to close again.
- The circuit breaker must open if greater than 50 percent of the requests are failing.
- If there is a failure, or the circuit is open, the message **Vendor Not Available!** must be used as a fallback.

```

//TODO: Add circuit breaker pattern
.hystrix()
    .hystrixConfiguration()
        .executionTimeoutInMilliseconds(3000)
        .circuitBreakerRequestVolumeThreshold(2)
        .metricsRollingPercentileWindowInMilliseconds(60000)
        .circuitBreakerSleepWindowInMilliseconds(20000)
        .circuitBreakerErrorThresholdPercentage(50)
    .end()
    //TODO: Invoke the vendor-service microservice
    .to("http4:"+ vendorHost ":"+ vendorPort +"/camel/vendor")
    //TODO: Invoke the VendorProcessor
    .process(new VendorProcessor())
    .onFallback()
        .transform(constant(VENDOR_ERROR_MSG))
    .endHystrix();

```

9. Update the **RestRouteBuilder** class to invoke the **ResponseProcessor** processor. This processor will create a **ResponseVO** object from the headers and define it as the body of the router.

```
.otherwise()
//TODO: invoke the ResponseProcessor
.process(new ResponseProcessor())
```

10. Deploy the **catalog-service** microservice.

- 10.1. Use the Fabric8 Maven Plug-in to build a container image for the Spring Boot application and create the required resources on OpenShift.

Enter the **~/JB421/labs/review4/catalog-service** folder and invoke the **fabric8:deploy** Maven goal, from the **openshift** Maven profile:

```
[student@workstation ~]$ cd ~/JB421/labs/review4/catalog-service
[student@workstation catalog-service]$ mvn -Popenshift fabric8:deploy
...
[INFO] -----
[INFO] Building Review: Deploying Microservices With Fuse on OpenShift - Catalog Service 1.0
[INFO] -----
...
[INFO] --- fabric8-maven-plugin:3.5.33.fuse-710023-redhat-00002:resource (default)
@ catalog-service ---
[INFO] F8: Running in OpenShift mode
[INFO] F8: Using docker image name of namespace: review4-lab
[INFO] F8: Running generator spring-boot
[INFO] F8: spring-boot: Using ImageStreamTag 'fuse7-java-openshift:1.1' from namespace 'openshift' as builder image
...
[INFO] F8: Creating BuildServiceConfig catalog-service-s2i for Source build
[INFO] F8: Creating ImageStream catalog-service
[INFO] F8: Starting Build catalog-service-s2i
```

You can safely ignore warnings and errors similar to the following from the **mvn** command. Notice that the image build step is successful despite the error that follows it.

```
...
[INFO] F8: Starting S2I Java Build .....
...
[INFO] F8: {"kind":"Status","apiVersion":"v1","metadata":{},"status":"Failure","message":"container \"vendor-service-s2i-1\" in pod \"vendor-service-s2i-1-build\" is waiting to start: PodInitializing","reason":"BadRequest","code":400}
[INFO] Current reconnect backoff is 1000 milliseconds (T0)
[INFO] Current reconnect backoff is 2000 milliseconds (T1)
[INFO] Current reconnect backoff is 4000 milliseconds (T2)
[INFO] Current reconnect backoff is 8000 milliseconds (T3)
[INFO] Current reconnect backoff is 16000 milliseconds (T4)
[INFO] Current reconnect backoff is 32000 milliseconds (T5)
[INFO] F8: Build review4/vendor-service-s2i-1 Complete
[INFO] Current reconnect backoff is 32000 milliseconds (T5)
```

```
[ERROR] Exception in reconnect  
java.util.concurrent.RejectedExecutionException: Task  
java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutureTask@5f123860  
rejected from java.util.concurrent.ScheduledThreadPoolExecutor@1a10ade6[Shutting  
down, pool size = 1, active threads = 1, queued tasks = 0, completed tasks = 12]  
...  
[INFO] BUILD SUCCESS  
...
```

- 10.2. Run the **mvn** command from the previous step again if you see this error during the Maven build:

```
...  
[ERROR] Exception in reconnect  
java.util.concurrent.RejectedExecutionException: ...  
...  
[ERROR] F8: Failed to execute the build [Unable to build the image using the  
OpenShift build service]  
...
```

11. Test the microservice applications on OpenShift.

- 11.1. Wait for the application pod to be ready and running.

Run the **oc get pod** command until the output resembles the following:

```
[student@workstation catalog-service]$ oc get pod  
NAME                      READY   STATUS    RESTARTS   AGE  
catalog-service-1-mwvl8     1/1     Running   0          33s  
catalog-service-s2i-1-build 0/1     Completed  0          58s  
vendor-service-1-w62kv      1/1     Running   0          13m  
vendor-service-s2i-1-build  0/1     Completed  0          13m
```

- 11.2. Open a Firefox window and, using the RESTClient plug-in, execute the following request:

Send an HTTP GET to <http://vendor.apps.lab.example.com/camel/vendor/1>. This returns a 200 HTTP response code. Look at the **Response** body tab:

```
{"id":1,"name":"Bookmart, Inc."}
```

- 11.3. Send an HTTP GET to <http://catalog.apps.lab.example.com/camel/catalog/1>. This returns a 200 HTTP response code. Look at the **Response** body:

```
{"id":1,"description":"description 1","author":"Lt. Howard  
Payson","vendorName":"Bookmart, Inc."}
```

- 11.4. Run the **block-vendor.sh** bash script to block connections to the **vendor-service** microservice.

```
[student@workstation catalog-service]$ cd ..  
[student@workstation review4]$ ./block-vendor.sh
```

- 11.5. Send an HTTP GET to `http://catalog.apps.lab.example.com/camel/catalog/1`. This returns a 500 HTTP response code. Look at the **Response** body:

```
ERROR Locating Vendor
```

Evaluation

As the **student** user on **workstation**, run the **lab review4** script with the **grade** argument, to confirm success on this exercise. Correct any reported failures and rerun the script until successful.

```
[student@workstation ~]$ lab review4 grade
```

This concludes the comprehensive review lab.