Red Hat
Training and Certification

Student Workbook (ROLE)

Red Hat Fuse 7.10 AD221

Cloud-native Integration with Red Hat Fuse

Edition 1

AD221-RHF7.10-en-1-20220308

Join a community dedicated to learning open source

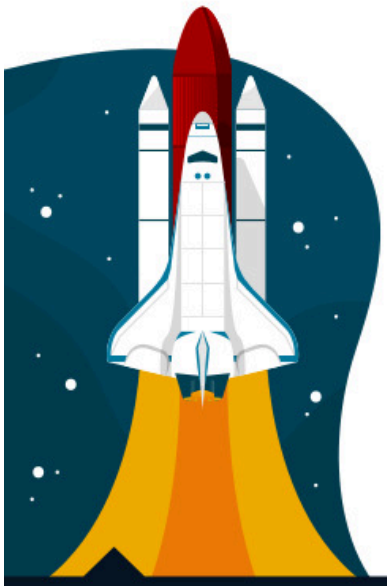
The Red Hat® Learning Community is a collaborative platform for users to accelerate open source skill adoption while working with Red Hat products and experts.

Network with tens of thousands of comunity members

Engage in thousands of active conversations and posts

Join and interact with hundreds of certified training instructors

Unlock badges as you participate and accomplish new goals



This knowledge-sharing platform creates a space where learners can connect, ask questions, and collaborate with other open source practitioners.

Access free Red Hat training videos

Discover the latest Red Hat Training and Certification news

Connect with your instructor - and your classmates - before, after, and during your training course.

Join peers as you explore Red Hat products

Join the conversation learn.redhat.com

Red Hat

Cloud-native Integration with Red Hat Fuse

Red Hat Fuse 7.10 AD221
Cloud-native Integration with Red Hat Fuse
Edition 1 20220308
Publication date 20220308

Authors:	Aykut Bulgu, Eduardo Ramírez Ronco, Jaime Ramírez Castillo, Pablo Solar Vilariño, Randy Thomas
Course Architect:	Zachary Gutterman
DevOps Engineer:	Richard Allred
Editor:	Sam Ffrench

Contributors: Heider Souza, Ted Singdahlsen

Document Conventions

This section describes various conventions and practices used throughout all Red Hat Training courses.

Admonitions

Red Hat Training courses use the following admonitions:

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

Note
These 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 something that makes your life easier.

Important
These provide details of information that is easily missed: configuration changes that only apply to the current session, or services that need restarting before an update will apply. Ignoring these admonitions will not cause data loss, but may cause irritation and frustration.

Warning
These should not be ignored. Ignoring these admonitions will most likely cause data loss.

Inclusive Language

Red Hat Training is currently reviewing its use of language in various areas to help remove any potentially offensive terms. This is an ongoing process and requires alignment with the products and services covered in Red Hat Training courses. Red Hat appreciates your patience during this process.

Introduction

Cloud-native Integration with Red Hat Fuse

Cloud-native Integration 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 cloud-native integrations in a simple and maintainable format.

Attendees will learn the skills required to develop, implement, test, and deploy applications with enterprise integration patterns (EIP) based on Apache Camel in Red Hat Fuse running on Spring Boot, Quarkus, and Camel K.

This course can assist in the preparation for the *Red Hat Certificate of Expertise in Camel Development* exam (EX221).

Course Objectives

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

Audience

- Java developers focused on implementing cloud-native integration solutions in an enterprise.

Prerequisites

- Experience with Java application development or Red Hat Application Development I: Programming in Java EE (AD183) is required.
- Experience with SQL and applications that use relational databases is required.
- Experience developing with the Spring Boot and Quarkus frameworks is strongly recommended, but not required.
- Experience building and packaging applications using Apache Maven is strongly recommended, but not required.
- Experience with OpenShift or Introduction to OpenShift Applications (DO101) is strongly recommended, but not required.
- Be proficient in using an IDE such as Red Hat® Developer Studio or VSCode.

Orientation to the Classroom Environment

Creating a Lab Environment

The Red Hat Online Learning (ROL) platform provides a RHEL 8 workstation environment in the cloud, which you can connect to remotely from your browser. To use this, click the **CREATE** button in the **Lab Environment** tab in the ROL interface.

For all classrooms provisioned for this course, ROL also provisions an account for you on a shared Red Hat OpenShift Container Platform (RHOCP) 4 cluster. When you provision your environment in the ROL interface, the system provides the cluster information. The interface gives you the OpenShift web console URL, your user name, and your password.

OpenShift Details

Username	**RHT_OCP4_DEV_USER**	your-user
Password	**RHT_OCP4_DEV_PASSWORD**	yourpassword
API Endpoint	**RHT_OCP4_MASTER_API**	https://api.CLUSTER.prod.nextcle.com:6443
Console Web Application		https://console-openshift-console.apps.CLUSTER.prod.nextcle.com
Cluster Id		37bd2501-4358-41b9-9d1b-56946f7ff765

The required tools are preinstalled in the `Cloud Workstation` classroom environment, which also includes VSCodium, a text editor that includes useful development features.

Cloud Workstation Classroom Overview

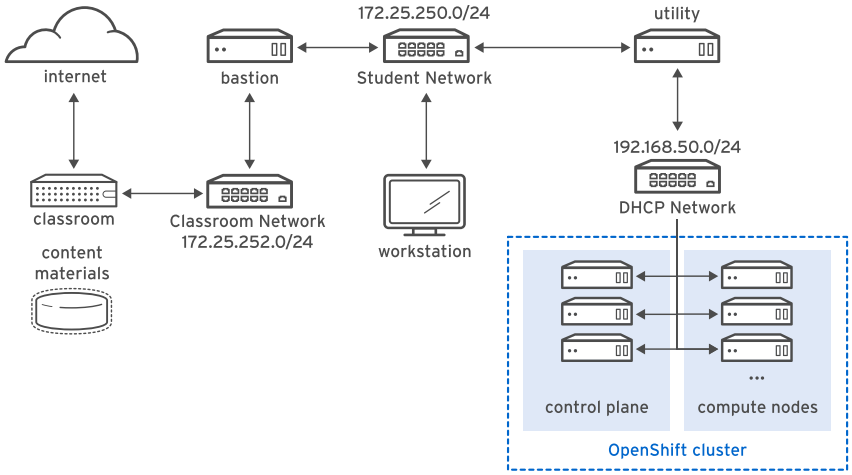


Figure 0.2: Cloud workstation classroom overview

In this environment, the main computer system used for hands-on learning activities is `workstation`. All virtual machines in the classroom environment 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	172.25.250.9	Graphical workstation used by students
bastion.lab.example.com	172.25.250.254	Router linking student's VMs to classroom servers
classroom.lab.example.com	172.25.252.254	Server hosting the classroom materials required by the course

The `bastion` system acts as a router between the network that connects the student machines and the classroom network. If `bastion` is down, then other student machines may not function properly or may even hang during boot.

Lab Directory Structure Considerations

The AD221 course uses a Python-based `lab` script that configures the directory structure for each guided exercise and lab activity. The *workspace* directory for this course is `/home/student/AD221`.

The workspace directory must contain the `AD221-apps` repository. This is the code repository that contains the necessary files for each activity in this course. The first guided exercise of the course guides you to clone this repository in `/home/student/AD221/AD221-apps`. The `lab` script uses the locally cloned `AD221-apps` repository to create a directory structure relevant to a particular guided exercise or lab activity.

For example, the `lab start route-messages` command does the following:

- Pulls the most recent code from the `AD221-apps` remote repository.
- Creates a `route-messages` directory in the workspace: `/home/student/AD221/route-messages`.
- Copies the `/home/student/AD221/AD221-apps/route-messages/apps/` source code subdirectories to the `/home/student/AD221/route-messages` directory.

You can find the solution for each activity in the `/home/student/AD221/AD221-apps` repository. For example, for the `route-messages` guided exercise, see the `/home/student/AD221/AD221-apps/route-messages/solutions` directory.

Troubleshooting Lab Scripts

If an error occurs while running the `lab` command, then you might want to check the following files:

- `/tmp/log/labs`: This directory contains log files. The `lab` script creates a unique log file for each activity. For example, the log file for the `lab start intro-setup` command is `/tmp/log/labs/intro_setup`
- `/home/student/.grading/config.yaml`: This file contains the course-specific configuration. Do not modify this file.
- `/home/student/.grading/ad221-workspace.json`: This file contains the user-specific lab configuration. The `ocp_*` properties must match the values provided in ROL. Do not manually modify this file. The file is created by the `lab start intro-setup` script.

Controlling Your Systems

Controlling Your Systems

You are assigned remote computers in a Red Hat Online Learning (ROLE) classroom. Self-paced courses are accessed through a web application that is hosted at rol.redhat.com [http://rol.redhat.com]. Log in to this site with your Red Hat Customer Portal user credentials.

Controlling the Virtual Machines

The virtual machines in your classroom environment are controlled through web page interface controls. The state of each classroom virtual machine is displayed on the **Lab Environment** tab.

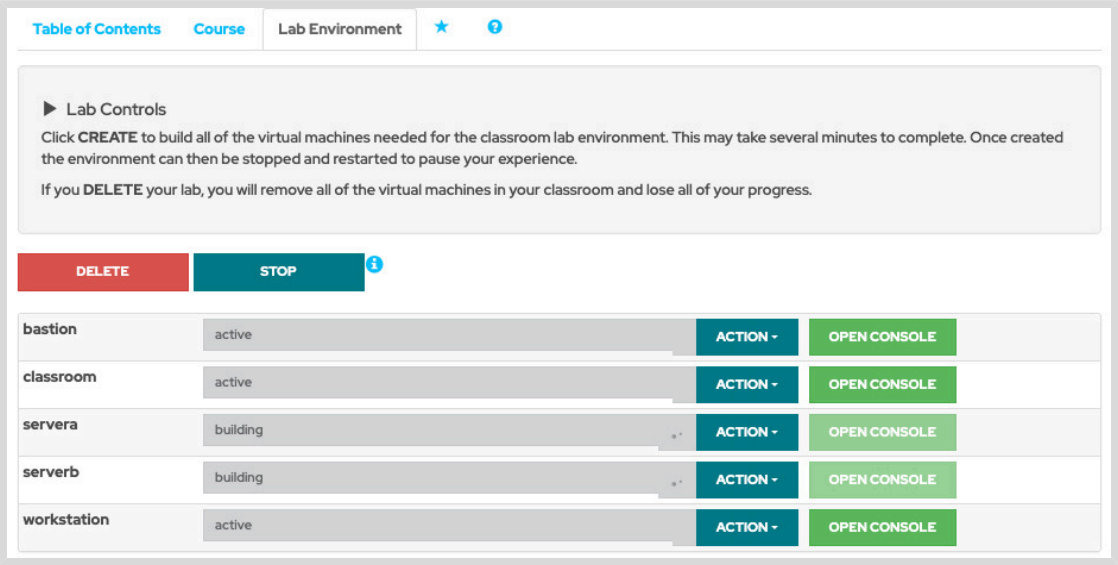


Figure 0.3: An example course Lab Environment management page

Machine States

Virtual Machine State	Description
building	The virtual machine is being created.
active	The virtual machine is running and available. If it just started, it still might be starting services.
stopped	The virtual machine is completely shut down. On starting, the virtual machine boots into the same state it was in before shutdown. The disk state is preserved.

Classroom Actions

Button or Action	Description
CREATE	Create the ROLE classroom. Creates and starts all the virtual machines needed for this classroom. Creation can take several minutes to complete.

Button or Action	Description
CREATING	The ROLE classroom virtual machines are being created. Creates and starts all the virtual machines that are needed for this classroom. Creation can take several minutes to complete.
DELETE	Delete the ROLE classroom. Destroys all virtual machines in the classroom. **All saved work on those systems' disks is lost.**
START	Start all virtual machines in the classroom.
STARTING	All virtual machines in the classroom are starting.
STOP	Stop all virtual machines in the classroom.

Machine Actions

Button or Action	Description
OPEN CONSOLE	Connect to the system console of the virtual machine in a new browser tab. You can log in directly to the virtual machine and run commands, when required. Normally, log in to the `workstation` virtual machine only, and from there, use `ssh` to connect to the other virtual machines.
ACTION > Start	Start (power on) the virtual machine.
ACTION > Shutdown	Gracefully shut down the virtual machine, preserving disk contents.
ACTION > Power Off	Forcefully shut down the virtual machine, while still preserving disk contents. This is equivalent to removing the power from a physical machine.
ACTION > Reset	Forcefully shut down the virtual machine and reset associated storage to its initial state. **All saved work on that system's disks is lost.**

At the start of an exercise, if instructed to reset a single virtual machine node, click ACTION > **Reset** for only that specific virtual machine.

At the start of an exercise, if instructed to reset all virtual machines, click ACTION > **Reset** on every virtual machine in the list.

If you want to return the classroom environment to its original state at the start of the course, then click DELETE to remove the entire classroom environment. After the lab has been deleted, then click CREATE to provision a new set of classroom systems.

Warning
The DELETE operation cannot be undone. All completed work in the classroom environment is lost.

The Auto-stop and Auto-destroy Timers

The Red Hat Online Learning enrollment entitles you to a set allotment of computer time. To help conserve your allotted time, the ROLE classroom uses timers, which shut down or delete the classroom environment when the appropriate timer expires.

To adjust the timers, locate the two + buttons at the bottom of the course management page. Click the auto-stop + button to add another hour to the auto-stop timer. Click the auto-destroy + button to add another day to the auto-destroy timer. Auto-stop has a maximum of 11 hours, and auto-destroy has a maximum of 14 days. Be careful to keep the timers set while you are working, so that your environment is not unexpectedly shut down. Be careful not to set the timers unnecessarily high, which could waste your subscription time allotment.

Chapter 1

Introducing Red Hat Fuse and Camel

Goal

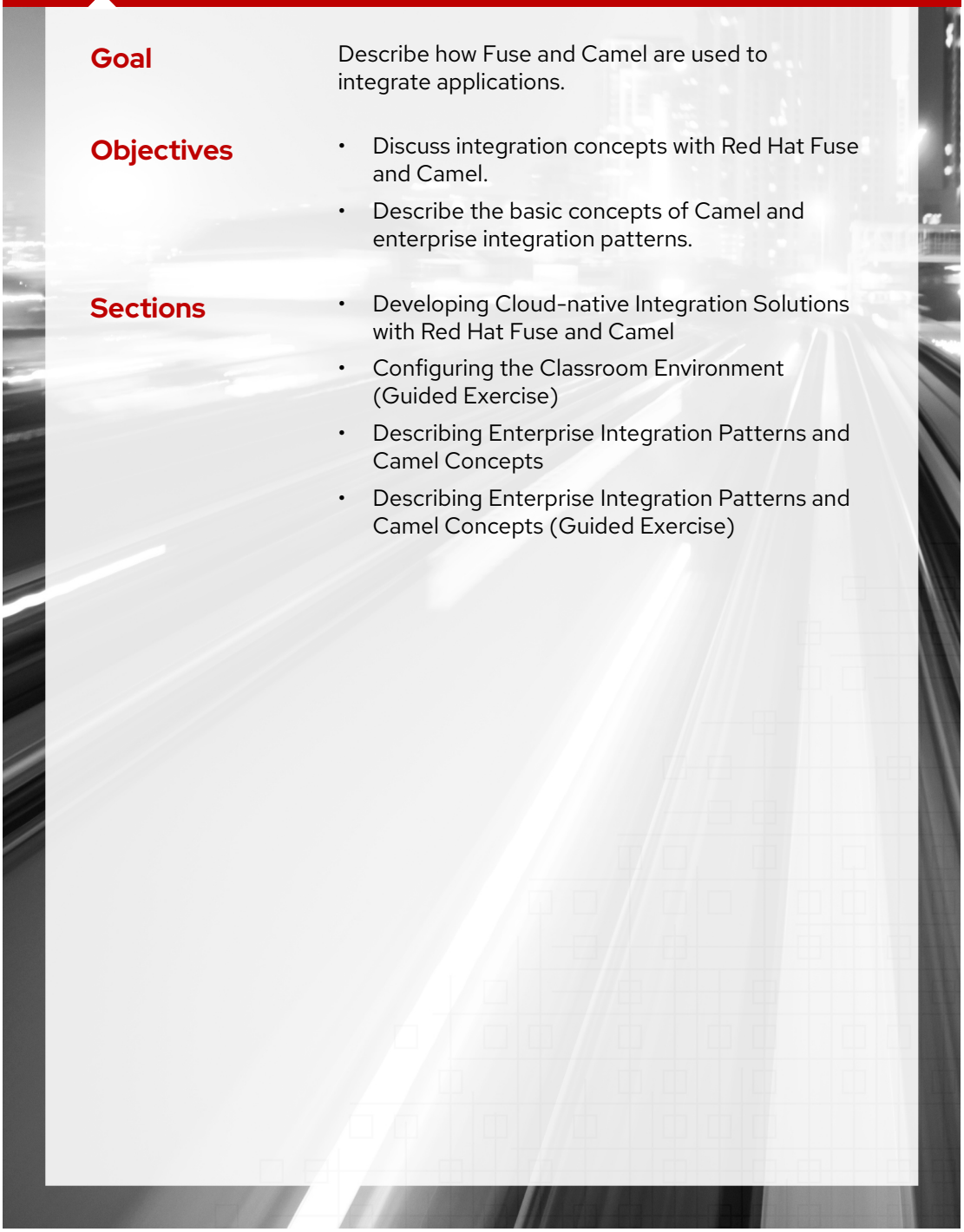
Describe how Fuse and Camel are used to integrate applications.

Objectives

- Discuss integration concepts with Red Hat Fuse and Camel.
- Describe the basic concepts of Camel and enterprise integration patterns.

Sections

- Developing Cloud-native Integration Solutions with Red Hat Fuse and Camel
- Configuring the Classroom Environment (Guided Exercise)
- Describing Enterprise Integration Patterns and Camel Concepts
- Describing Enterprise Integration Patterns and Camel Concepts (Guided Exercise)



Developing Cloud-native Integration Solutions with Red Hat Fuse and Camel

Objectives

After completing this section, you should be able to discuss integration concepts with Red Hat Fuse and Camel.

Introduction to the Traditional Integration Approach

Complex systems have traditionally used centralized integration mechanisms, such as an *Enterprise Service Bus* (ESB). An ESB is a middleware service, owned by IT infrastructure departments, which is in charge of routing, processing, and translating the data that flows between the services of a system.

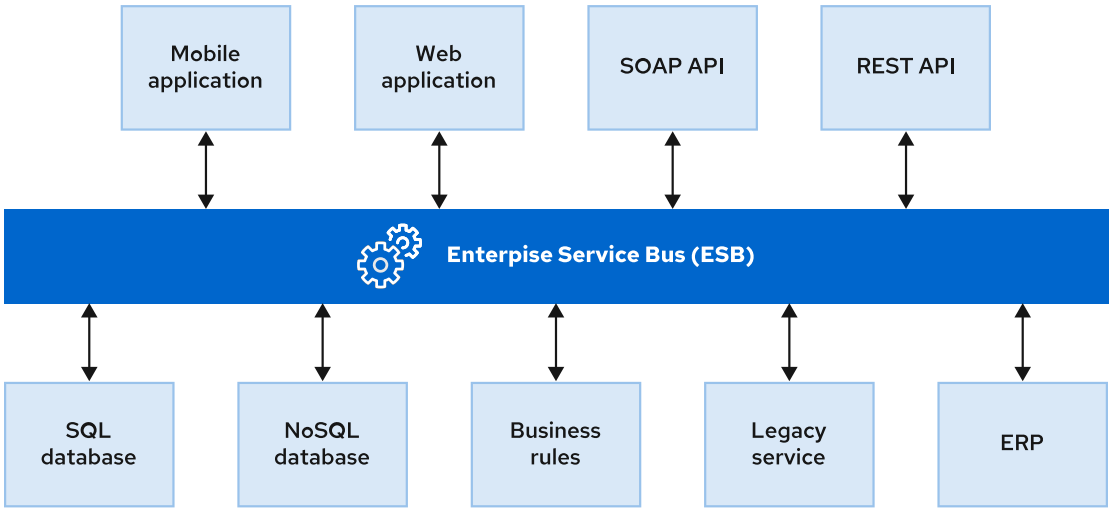


Figure 1.1: Example application using an ESB

This approach was a powerful solution to decouple services in the waterfall era, but presents scalability and elasticity problems in modern digital environments. These new environments require flexibility and faster delivery speed, which are critical to attain a competitive advantage in digital and continuously evolving markets.

Modern applications embrace the agile goals of adaptability and quick delivery. They take advantage of the flexibility that small components, usually microservices, provide in terms of agility. Therefore, using an isolated, monolithic piece of infrastructure, such as an ESB, impacts the ability of a modern system to scale.

Defining Agile Integration

To deliver software fast enough to meet market demands, organizations need an agile infrastructure plan. With this plan, development and infrastructure teams collaborate to break the monolithic integration infrastructure into distributed integration components and manage them as core parts of the application. This approach, called *agile integration*, is the result of combining the following capabilities:

The Pillars of Agile Integration

Distributed Integration

Organizations should distribute the integration efforts across multiple integration components instead of using a single bus. Components should be reusable, so that development teams can deploy the same component in multiple ways.

Application Programming Interfaces (APIs)

APIs provide developers with a common language that defines how to communicate with applications. With APIs, development teams can enforce policies to control communication, security, authorization, and usage.

Containers

Organizations should deploy integration services as containers on the cloud. Containers enable cloud-native deployments, and allow development and infrastructure teams to deploy distributed integrations and APIs as microservices.

By using these pillars, integration becomes more scalable and more flexible, providing organizations with better opportunities to quickly respond to customer needs.

Introduction to Agile Integration with Apache Camel

Apache Camel is an open-source integration framework that implements *Enterprise Integration Patterns* (EIPs) and makes integration easier. EIPs are proven solutions to recurring integration problems.

A Camel application normally implements how information flows from an origin endpoint into a destination endpoint. This is called a route.

A route uses Camel components to connect to the endpoints. Camel is based on an ecosystem of more than 300 pluggable components and a strong community that maintains these components. The following are examples of Camel components.

- Databases, middlewares, web and network protocols, such as FTP and MySQL
- Enterprise information systems, such as Salesforce
- Popular APIs, such as Slack and Twitter

For example, with Camel, you can define multiple, reusable integration applications and use them to integrate the different services of your application.

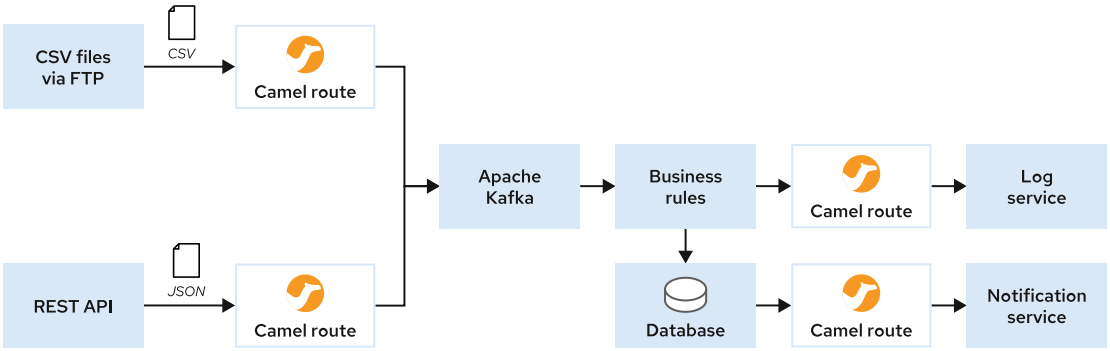


Figure 1.2: Example application using Apache Camel for distributed integration

Red Hat Fuse

Red Hat Fuse is a distributed integration platform that connects everything from legacy systems, APIs, partner networks, and Internet of things (IoT) devices. Fuse allows IT departments,

developers, and nontechnical users alike to create a unified solution that meets the principles of agile integration.

The core of Red Hat Fuse is Apache Camel, which gives Fuse its agile integration capabilities. On top of Camel, Fuse provides a set of capabilities and features to support and improve Camel development.

You can use Fuse in different ways, depending on your needs.

Fuse Standalone
: To package your application as a JAR and run it on a runtime, such as Spring Boot. This distribution supports Apache Karaf, Spring Boot, and Red Hat JBoss Enterprise Application Platform.

Fuse on OpenShift
: To build and deploy your application and its dependencies as a container image on OpenShift. In contrast to Fuse Standalone, Fuse on OpenShift packages all runtime components in a container image.

Note

This course focuses on Fuse Standalone with Spring Boot as the runtime. Additionally, some activities explore integration development with Quarkus and Camel K.

Apache Camel K

Apache Camel K is an open-source lightweight platform, which currently uses Camel version 3. Developers can easily run cloud-native integrations on OpenShift by using Camel K.

With Camel K, you do not need to develop and maintain an entire integration application. You only need to provide the Camel K CLI with a Java Camel DSL file. The following are just some of the key capabilities of Camel K:

- Cloud-native integrations on OpenShift and Kubernetes.
- Runtime based on Camel Quarkus, a Camel subproject.
- KNative support.
- Reusable high-level integrations with *Kamelets*. Kamelets are integration template documents in YAML format.

Camel K integration development is covered in more detail later in the course.

References

Apache Camel User Manual
https://camel.apache.org/manual/

What Is Agile Integration? - Red Hat Developers Blog
https://middlewareblog.redhat.com/2017/09/13/what-is-agile-integration/

For more information, refer to the *Fuse 7.10 Product Overview* chapter in the *Release Notes for Red Hat Fuse 7.10* guide at
https://access.redhat.com/documentation/en-us/red_hat_fuse/7.10/html-single/release_notes_for_red_hat_fuse_7.10/index#Product

For more information, refer to the *INTRODUCTION TO CAMEL K* chapter in the *Red Hat Integration Guide* at
https://access.redhat.com/documentation/en-us/red_hat_integration/2021.q4/html/getting_started_with_camel_k/introduction-to-camel-k

Guided Exercise

Configuring the Classroom Environment

In this exercise you will configure the lab environment required for the course exercises.

Outcomes

You should be able to configure the classroom environment and clone the sample code.

Before You Begin

To perform this exercise, ensure you have access to the Red Hat Online Learning Environment (ROL).

Instructions

The following procedure describes how to provision and configure a new classroom from the Red Hat Online Learning platform. You must complete the following procedure before attempting any of the course activities.

1. Provision a new lab environment.

   Click the **Lab Environment** tab on the ROL course page. Next, click **CREATE** to start provisioning the lab environment

2. Open the workstation console.

   After the lab environment is ready, click **OPEN CONSOLE** in the `workstation` virtual machine, and log in as the `student` user with the password `student`.

3. Configure the workspace.

   3.1. Open a new terminal window, and use the `lab` command to initialize the workspace and user-specific configuration.

```
[student@workstation ~]$ lab start intro-setup

This script configures the lab environment, and the connection parameters to
 access the OpenShift cluster.

Enter your workspace directory: [/home/student/AD221]:
...output omitted...
```

   When running the `lab` command, enter the OpenShift details provided in the **Lab Environment** tab. Do not change the workspace directory.

   3.2. Navigate to the `~/AD221` directory.

4. Clone the `AD221-apps` repository to your `AD221` workspace.

   4.1. From the `AD221` workspace directory, clone the `AD221-apps` repository to your workstation:

```
[student@workstation AD221]$ git clone https://github.com/RedHatTraining/AD221-
apps.git
Cloning into 'AD221-apps'...
...output omitted...
```

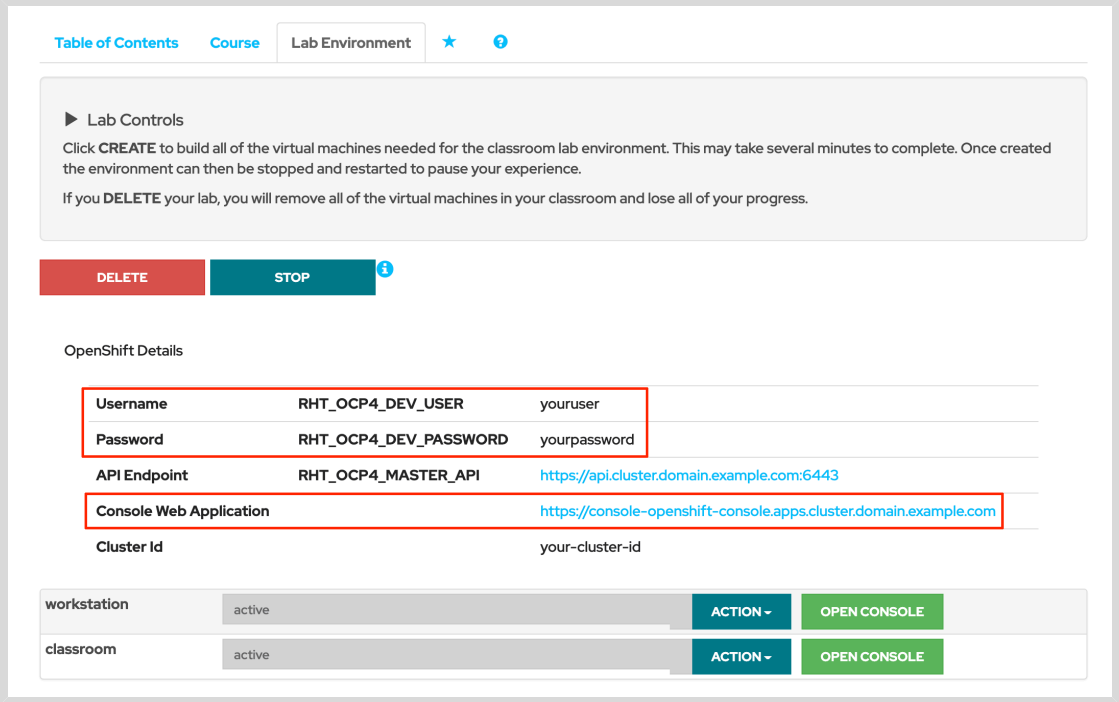
4.2. Verify that the cloned repository contains the repository content and return to the workspace directory.

```
[student@workstation AD221]$ cd AD221-apps
[student@workstation AD221-apps]$ head README.md
# AD221 Application Repository
...output omitted...
[student@workstation AD221-apps]$ cd ..
```

5. Install the Camel K command line interface (CLI).

5.1. Download the CLI application through the OpenShift web console.

Find the **Lab Environment** tab on your ROL course page. This table should be visible after you provision your online lab environment.



The **Console Web Application** link in the `OpenShift Details` table opens the web console for your dedicated Red Hat OpenShift Container Platform instance. Take note of your `Username` and `Password`, which you will be using in the next step.

5.2. From the `workstation` virtual machine, open a web browser and log in to the OpenShift web console using your `Username` and `Password`.

5.3. After logging in, click **?** in the upper-right corner, and then click **Command Line Tools**.

5.4. On the `Command Line Tools` page, download the Camel K CLI compressed binary file for Linux from the list **kamel - Red Hat Integration - Camel K - Command Line Interface**

5.5. Unpack the compressed archive file, and then copy the `kamel` binary to `/usr/local/bin`.

```
[student@workstation ~]$ sudo cp kamel /usr/local/bin/
[student@workstation ~]$ sudo chmod +x /usr/local/bin/kamel
```

5.6. Verify that the `kamel` binary works. Open a new command line terminal and run the following:

```
[student@workstation ~]$ kamel version
Camel K Client Red Hat 1.6.0
```

Note

Your output might be slightly different based on the version of the Camel K client that you downloaded.

Finish

This exercise has no command to finish it.

This concludes the guided exercise.

Describing Enterprise Integration Patterns and Camel Concepts

Objectives

After completing this section, you should be able to describe the basic concepts of Camel and enterprise integration patterns.

Describing Enterprise Integration Patterns

In 2003, as a solution to the integration problems that organizations deal with, Hohpe and Woolf published a book called `Enterprise Integration Patterns`. The book described 65 separate patterns that represent common approaches for designing integration solutions. Over the years, the `Enterprise Integration Patterns` became more and more popular, and frameworks came out as implementation solutions for integration problems. Apache Camel is one of the most popular of these implementations.

Camel supports most of the `Enterprise Integration Patterns` (EIP). The following are some of the most important EIPs.

Content Based Router

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

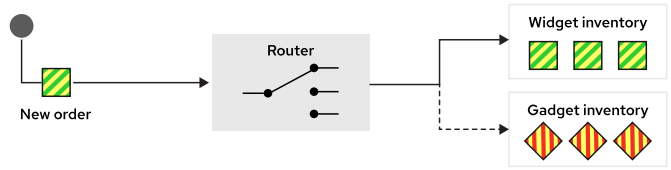


Figure 1.4: Content based router

Splitter

The Splitter pattern defines separating a list or collection of items in a message payload into individual messages. This pattern is useful when you need to process smaller, rather than larger, bulk messages.

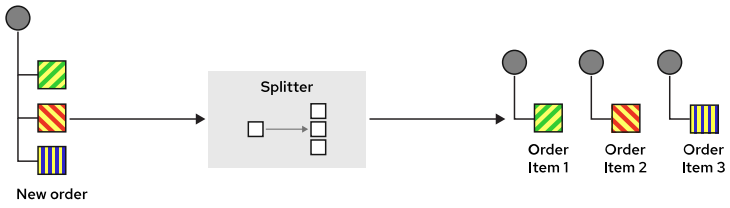


Figure 1.5: Splitter

Aggregator

The Aggregator pattern defines collapsing 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.

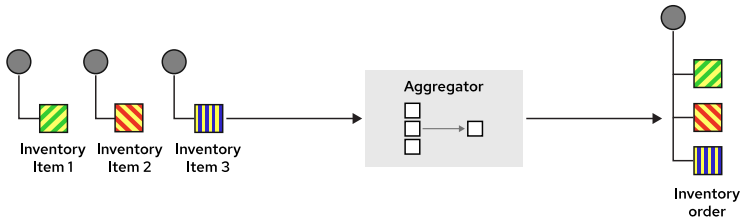


Figure 1.6: Aggregator

Content Enricher

The Content Enricher pattern defines enriching the data when sending messages from one system to the target system, which requires more information than the source system can provide.

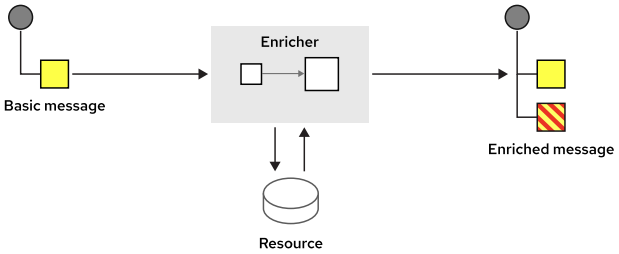


Figure 1.7: Content enricher

Content Filter

The Content Filter pattern defines filtering the data when sending messages from one system to the target system, which requires refined information.

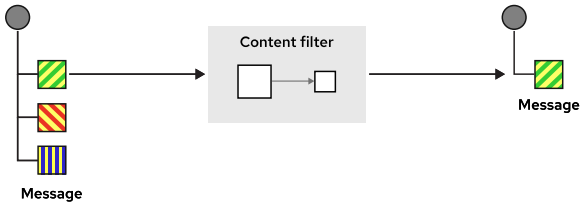


Figure 1.8: Content filter

Messaging Gateway

The Messaging Gateway pattern defines delegating 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.

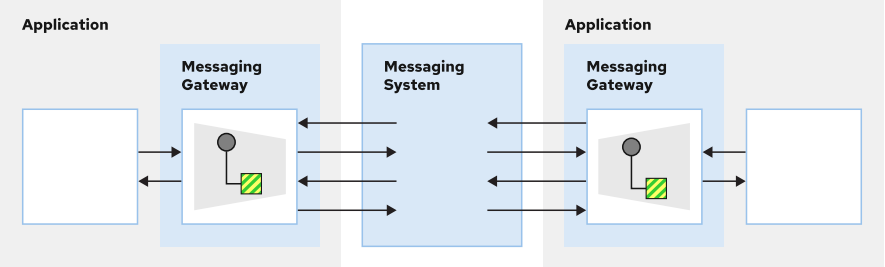


Figure 1.9: Messaging gateway

You can implement an EIP in many ways with Camel. It depends on the EIP and the suitable components of Camel.

You can examine the Camel architecture to have a better understanding of the Camel concepts and components.

Describing the Camel Architecture

Camel consists of many concepts. The following image shows a 10,000 feet snapshot of the Camel architecture.

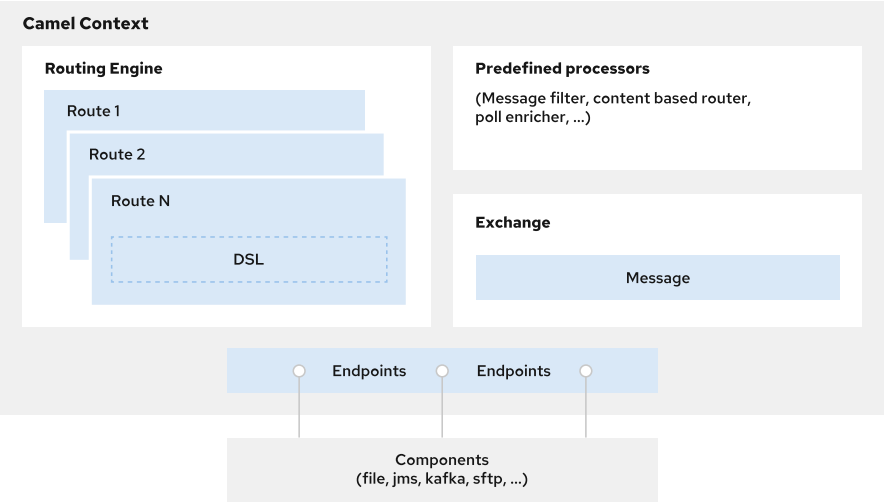


Figure 1.10: Camel architecture from 10,000 feet

Defining the Camel Concepts

Camel has a variety of concepts. Each of them play a different role in the Camel architecture. The following list presents short descriptions of each concept. These concepts make up the overall Camel Architecture.

Message

A `Message` is the smallest entity in a Camel architecture. It helps with the communication of external systems and Camel by carrying information. A `Message` can contain headers, attachments and a body.

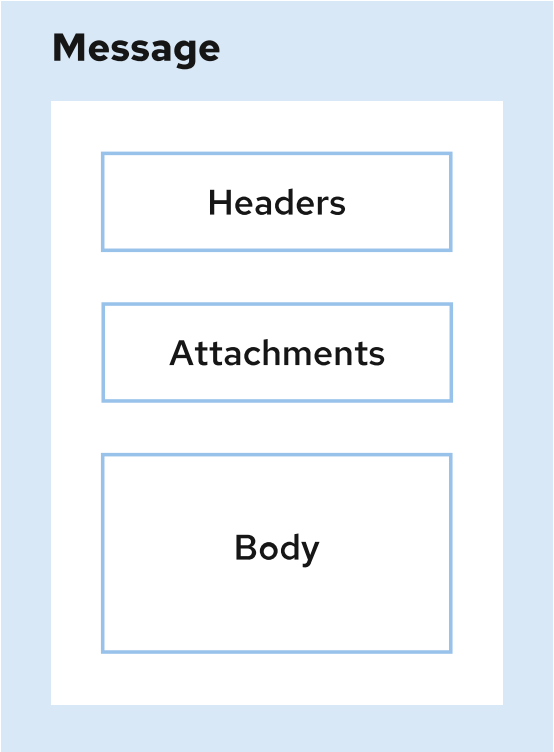


Figure 1.11: Message

- `Headers` are the name-value pairs that are related to the message. The values can be anything that is important for the message such as sender identifiers, content-encoding, and so on.
- `Attachments` are the optional fields, which Camel typically provides for web service and email components.
- `Body` is the body of the message, which is of `java.lang.Object` type. Thus, a message can be any kind of content and size.

Exchange

An `Exchange` is a container of the `Message` in Camel. The following image demonstrates the `Exchange` structure.

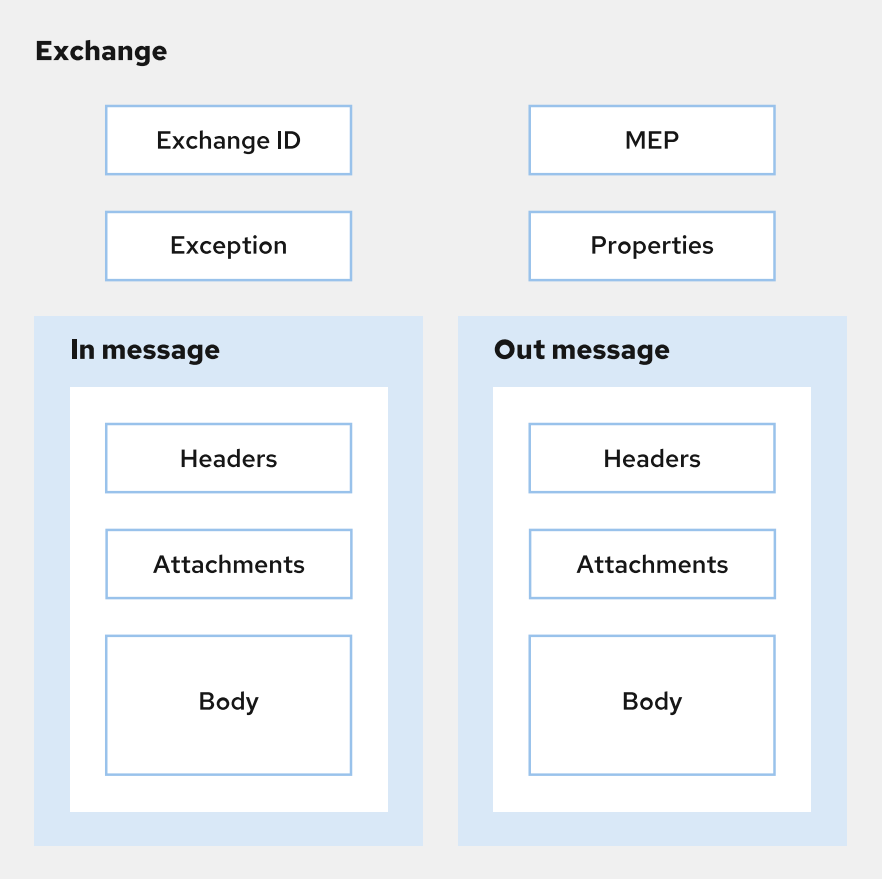


Figure 1.12: Exchange

- `Exchange ID:` The unique identifier of the exchange. Camel automatically generates this.
- `MEP:` Message Exchange Pattern. This is a pattern that describes various types of interactions between systems. You can use either the `InOnly` or `InOut` messaging style. `InOnly` is a one way message. A JMS (Java Message Service) message is an example of `InOnly` messaging. `InOut` is a request-response message. HTTP-based messaging is an example of `InOut` messaging.
- `Exception:` If an error occurs at any time, Camel sets the exception in this field.
- `Properties:` Properties are similar to message headers, but there are a few differences. Properties last for the duration of the entire exchange. Thus properties can contain global-level information. On the contrary, message headers are specific to a particular message.
- `In message:` Input message. This is a mandatory part of an exchange. The `In message` contains the request message.
- `Out message:` Output message. This is an optional part of an exchange and exists only if the MEP is InOut. The `Out message` contains the reply message.

CamelContext

The `CamelContext` is the Camel's runtime system. It is the context that keeps all the conceptual pieces together.

Routing Engine

The `Routing Engine` is the under-hood mechanism, which moves the messages. It ensures the messages are routed properly.

Routes

A `Route` is a chain of processors that delegates the message routing to the `Routing Engine`. You must create at least one `Route` to create an integration system with Camel. Routes have inputs and outputs. To define a `Route`, you must use a Domain-specific Language (DSL) in Camel.

Processor

A `Processor` is the unit of execution in Camel. It is capable of creating or modifying an incoming exchange. During a routing process, Camel passes the exchanges from one processor to another. Thus, the output of a processor is the input of another.

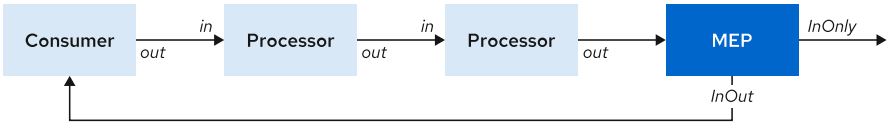


Figure 1.13: Processor in action

Component and Endpoint

`Components` are the main adapter points in Camel. To use a Camel component, you must define an `Endpoint` in a Camel route DSL. An `Endpoint` is an abstraction that constructs the end of a channel. A system can send or receive messages through this endpoint's channel.

Producer and Consumer

`Producers` and `Consumers` in Camel, work in an untypical way. A consumer is the service that receives messages from external systems. So in Camel, a flow starts with a consumer, not a producer, because an external system produces the messages.

A producer, which is totally a different concept than an external system producer, is the entity that sends messages to an endpoint. When the endpoint receives the message, the producer sends the message to the real system. As an example, `FileProducer` writes the message body to a file.

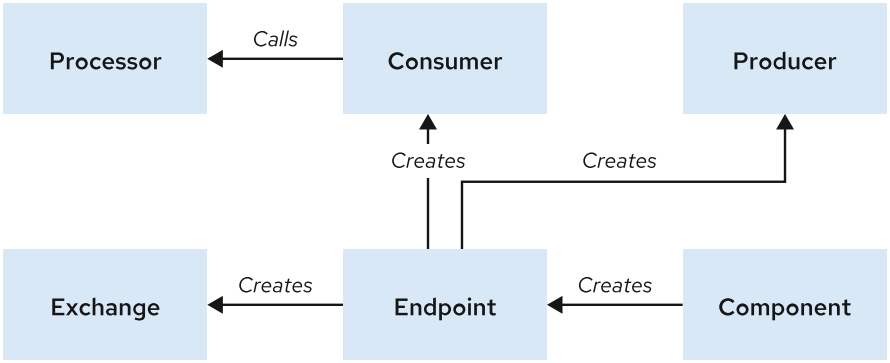


Figure 1.14: Producers and consumers

Defining the Camel Domain-specific Language (DSL)

Apache Camel has a domain-specific language, which makes integration development easier. The Camel DSL allows developers to focus on the integration problem rather than the programming language. Camel has two main official types of DSLs: XML DSL and Java DSL.

XML DSL

This DSL is in the form of XML. In Camel, developers can create XML DSLs in more than one form depending on the framework.

You can create an `OSGI Blueprint` XML that has the Camel DSL. Also, you can create a `Spring XML` if you are using `Spring Framework` or `Spring Boot Framework` with

Camel. This makes the routes usable as Spring beans in the context of Spring. It depends on the framework you prefer to use for Camel development.

The following code is a small example of an XML DSL.

```
<route>
  <from uri="file:path/orderInbox"/>
  <to uri="kafka:orders"/>
</route>
```

Note

In this course, even if we use `Spring Boot Framework` for most of our examples, we do not cover `XML DSL`. We use `Java DSL` in our examples and exercises.

Java DSL

The Java DSL is a fluent styled DSL, which uses chained method calls to define routes and components in Camel.

A route in a Java DSL contains no procedural code. If you need to embed complex conditional or transformation logic inside a route, then you can invoke a Java Bean or create a custom Camel processor.

The following code is a small example of a Java DSL.

```
from("file:path/orderInbox")
    .to("kafka:orders");
```

References

Enterprise Integration Patterns, by Gregor Hohpe and Bobby Woolf
https://www.enterpriseintegrationpatterns.com/

Apache Camel Official Website
https://camel.apache.org/

Claus Ibsen and Jonathan Anstey. (2018) Camel in Action, Second Edition. Manning. ISBN 978-1-617-29293-4.

Guided Exercise

Describing Enterprise Integration Patterns and Camel Concepts

In this exercise, you will observe how a health analytics company, called HealthGateway LLC., processes the public Covid-19 data of European countries and exposes that data via its UI and REST API.

HealthGateway LLC. must provide the latest Covid-19 cases and vaccination data to external developers and its customers. The developers of HealthGateway LLC. plan to consume data from external resources and expose them via a REST API. They decided to work with two official data sets provided by the European Union. One data set is a CSV file that has daily Covid-19 case data of the European countries. The second data set is an XML file that has the weekly Covid-19 vaccination data of the European countries.

Because of the different source formats and potential data conversion, transformation and integration scenarios, developers of HealthGateway LLC. decide to use a powerful integration technology; Red Hat Fuse.

With Red Hat Fuse, they can;

- Fetch the Covid-19 data from the files that reside in an SFTP server.
- Perform the suitable transformations of the data, thus handling the data as lists and objects.
- Filter the data, store it in a database and expose it via REST endpoints.
- Aggregate the Covid-19 cases and vaccination data and publish it to an Apache Kafka topic.
- Consume the aggregated data, and enrich it by using an external REST service, which provides the general data of European countries.
- Expose the final enriched Covid-19 data to be used by the front-end application and by the end-users.

Outcomes

In this exercise you should be able to examine a series of Red Hat Fuse capabilities such as:

- Reading messages from an SFTP server by using the `sftp` component.
- Data transformation from CSV, XML or JSON to custom Java object types.
- Implementing the `Splitter` enterprise integration pattern (EIP) by using the `split` component.
- Processing messages by using `parallelProcessing`.
- Persisting data to a relational database and reading the data from it by using the `jpa` component.
- Persisting data to a NoSQL database and reading the data from it by using the `mongodb` component.
- Using in-memory queues by using the `seda` component.
- Implementing the `Content Based Router` EIP.
- Implementing the `Content Enricher` EIP and aggregating different sets of the data by using the `pollEnrich` or `enrich` components.
- Using custom aggregation strategies.
- Implementing the `Content Filter` EIP and filtering data by using the `filter` component and custom filters.

- Sending and receiving messages from Apache Kafka by using the `kafka` component.
- Performing an HTTP call to another REST endpoint for fetching data by using the `http` component.
- Exposing the data in JSON format by using the `rest` component.

The code is available in the `~/AD221/AD221-apps/intro-demo/apps` directory for further examination.

Before You Begin

To perform this exercise, ensure you have the `AD221-apps` repository cloned in your workstation.

From your workspace directory use the `lab` command to start this exercise.

```
[student@workstation AD221]$ lab start intro-demo
```

The `lab` command copies the exercise files from the `~/AD221/AD221-apps/intro-demo/apps` directory, into the `~/AD221/intro-demo` directory.

Instructions

1. Examine the high-level architecture diagram.

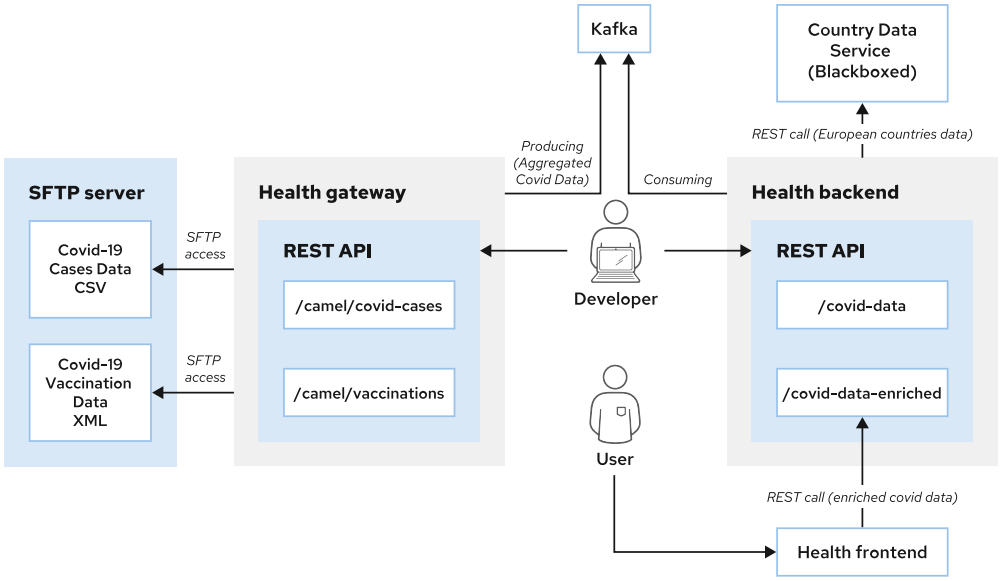


Figure 1.15: HealthGateway LLC. high-level architecture

2. Navigate to the `~/AD221/intro-demo` directory and open the project with an editor, such as VSCodium.

3. Compile and run the `health-gateway` application by using the `./mvnw package spring-boot:run` command.

   The application exposes two REST endpoints.

Method	REST Endpoint	Description
GET	`http://localhost:8080/camel/cases`	Provides the European Covid-19 case data for the last day of each week in 2021
GET	`http://localhost:8080/camel/vaccinations`	Provides the European Covid-19 weekly vaccination data for 2021

4. Explore how the `health-gateway` application works. The following image shows how the routes work in the application.

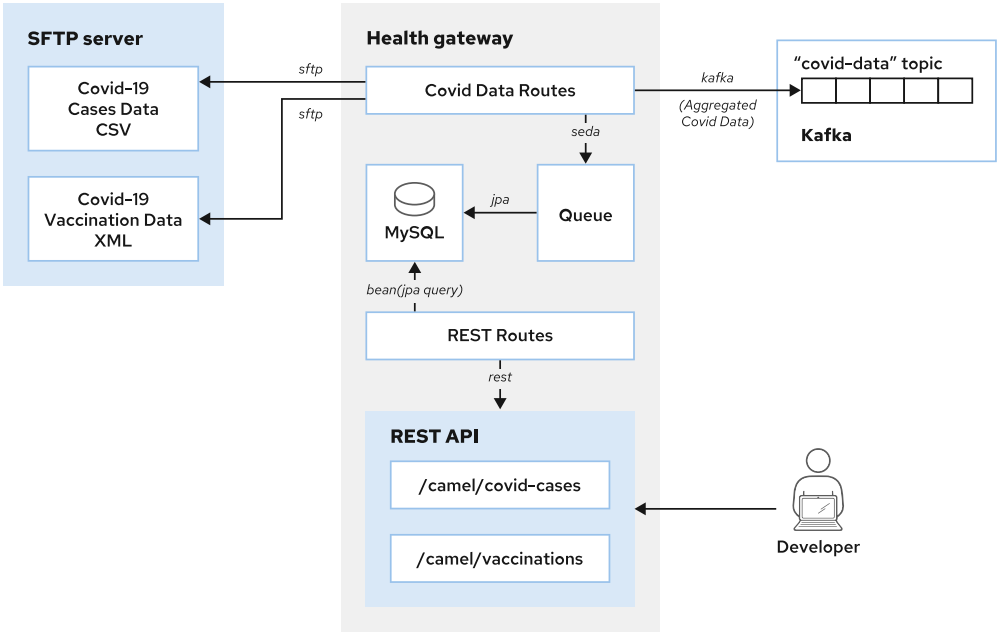


Figure 1.16: Health-gateway routes

5. Compile and run the `health-backend` application by using the `./mvnw package quarkus:dev` command.

   The application exposes two REST URL endpoints.

Method	REST Endpoint	Description
GET	`http://localhost:8081/covid-data`	Provides the aggregated Covid-19 case and vaccination data
GET	`http://localhost:8081/covid-data-enriched`	Provides the aggregated Covid-19 case and vaccination data, which is enriched with the general country data

6. Explore how the `health-backend` application works. The following image shows how the routes work in the application.

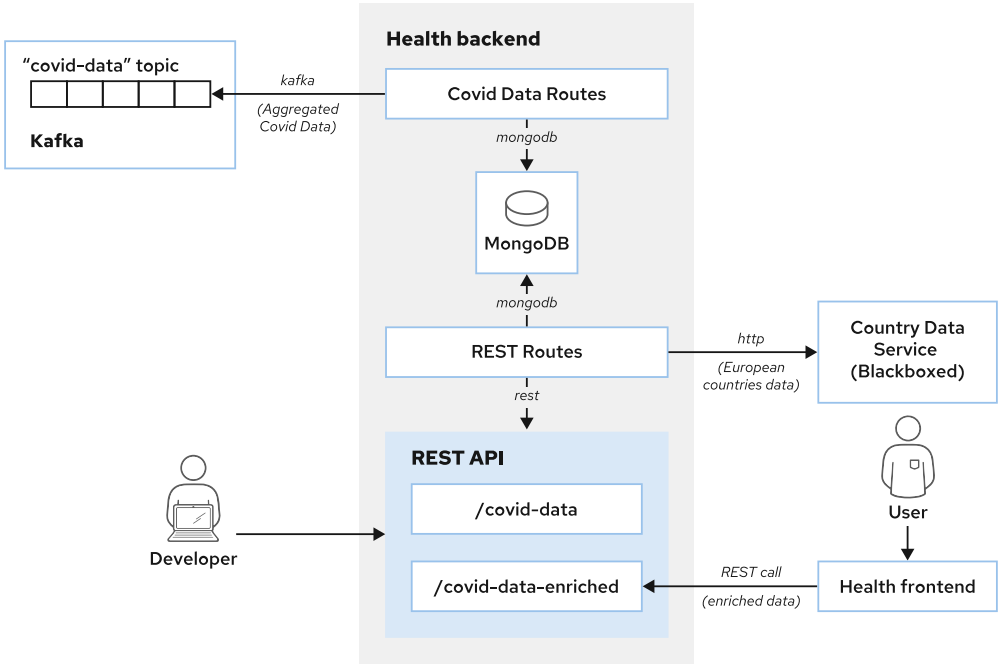


Figure 1.17: Health-backend routes

7. Run the `health-front` application by using the `python ~/AD221/intro-demo/scripts/serve-frontend.py` command from the exercise directory.

   Open a browser window and navigate to `http://localhost:8082` address.

8. Observe the Covid-19 data that the `health-front` application consumes and exposes in the web UI.

9. Stop the running applications.

> **Note**
> The SIGINT signal might not work in the `health-gateway` application because of pending inflight exchanges. In this case, send a SIGTERM or SIGKILL signal to stop the application.

Finish

Return to your workspace directory and use the `lab` command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation AD221]$ lab finish intro-demo
```

This concludes the guided exercise.

Summary

In this chapter, you learned:

- About Apache Camel and Red Hat Fuse, and their roles in Agile Integration.
- Apache Camel concepts and architecture.
- How Apache Camel supports Enterprise Integration Patterns.

Chapter 2

Creating Camel Routes

Goal

Develop Camel routes and process messages.

Objectives

- Create a route that reads and writes data to the file system.
- Create a route that reads from an FTP server.
- Implement routes that include dynamic routing.
- Use processors to manipulate exchange messages in routes.

Sections

- Creating Routes Using the Java and XML DSL (and Guided Exercise)
- Reading and Writing Files (and Guided Exercise)
- Routing Messages (and Guided Exercise)
- Developing a Camel Processor (and Guided Exercise)
- Creating Camel Routes (Quiz)

Creating Routes Using the Java and XML DSL

Objectives

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

Basic Developer Setup

To facilitate the development of Fuse applications, this course uses a set of tools including VSCodium, Maven, and Spring Boot.

Visual Studio Code Camel Support

The following extension applies both Visual Studio Code (VSCode) and VSCodium. VSCodium is used in the exercises of this course. Red Hat provides an Apache Camel extension pack for Visual Studio Code that includes multiple extensions related to Camel development. The pack enables code completion for both Java Domain-specific Language (DSL) and XML DSL. To install the extension, within VSCode, select `View > Extentions`. A search on `Camel` finds the following extension.

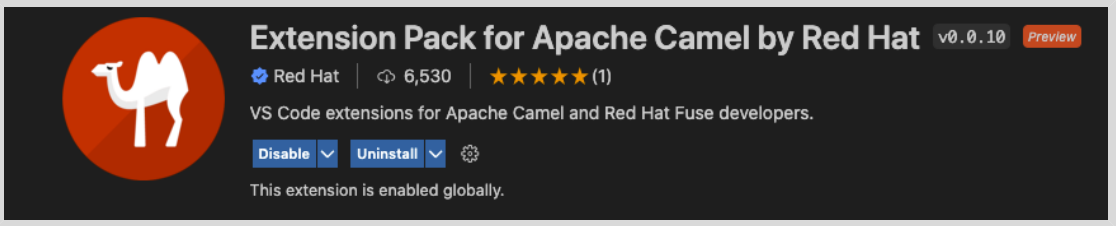


Figure 2.1: VSCode Camel extension

Visual Studio Code is not required for performing the lab exercises, but it is recommended.

Maven Configuration

Fuse applications are typically built with Maven. To access artifacts that are in Red Hat Maven repositories, you must add those repositories to Maven's settings.xml file in the .m2 directory of your home directory. The system-level `settings.xml` file at `M2_HOME/conf/settings.xml` is used if a user specific file is not found. Add the Red Hat repositories as illustrated in the following example.

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

  <profiles>
    <profile>
      <id>extra-repos</id>
      <activation>
        <activeByDefault>true</activeByDefault>
      </activation>
      <repositories>
```

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

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

Maven Support for Spring Boot

Spring Boot, discussed in the next section, makes it easy to create stand-alone, production-grade Spring based Applications. Since version 2.17, Apache Camel ships a Spring Boot Starter module. In order to build Spring Boot applications for Fuse, the Fuse Bill of Materials (BOM) is required. The BOM defines a curated set of Red Hat supported dependencies from the Red Hat Maven repository. The BOM exploits Maven's dependency management mechanism to define the appropriate versions of Maven dependencies. The BOM is provided by the following `pom.xml` configuration.

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.jboss.redhat-fuse</groupId>
      <artifactId>fuse-springboot-bom</artifactId>
      <version>${fuse.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

In addition to the BOM, the Spring Boot Maven Plugin is also required. The Spring Boot Maven Plugin implements the build process for a Spring Boot application in Maven. This plugin is responsible for packaging your Spring Boot application as an executable Jar file. The Spring Boot maven plugin is provided by the following pom.xml configuration.

```
<plugin>
    <groupId>org.jboss.redhat-fuse</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
    <version>${fuse.version}</version>
    <executions>
   <execution>
  <goals>
    <goal>repackage</goal>
  </goals>
   </execution>
    </executions>
</plugin>
```

Spring Boot

Red Hat Fuse includes a Spring Boot Starter module. With this module, you can use Camel in Spring Boot applications by using starters.

To use the starter, add the following dependency to your pom.xml file.

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-spring-boot-starter</artifactId>
    <version>${camel.version}</version> <!-- use the same version as your Camel
 core version -->
</dependency>
```

With Spring Boot, you can add classes with your Camel routes, by annotating a `RouteBuilder` class with the `org.springframework.stereotype.Component` annotation. The annotation allows Spring Boot to find the class, register it as a Java Bean, and start a camel context that includes the route. The following example defines a simple route:

```
package com.example;

import org.apache.camel.builder.RouteBuilder;
import org.springframework.stereotype.Component;

@Component
public class MyRoute extends RouteBuilder {

    @Override
    public void configure() throws Exception {
        from("timer:foo").to("log:bar");
    }
}
```

An `application.properties` file is used to customize Spring Boot. By design, you will often find that the default values are sufficient for your needs.

Camel Routes

In Camel, a route describes the path of a message from one endpoint (the origin) to another endpoint (the destination). The origin of a route is associated with the `from` method in the Java DSL and normally consumes messages from a source.

The `from` method uses an integration component configured as a consumer endpoint. Likewise, the destination is associated with the `to` method in the Java DSL and produces, or send messages to a destination. The `to` method uses an integration component configured as a producer endpoint.

Routes are a critical aspect of Camel because they define integration between endpoints. With the help of components, routes can 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 XML DSL.

A route starts with a consumer, which receives the data from a point of origin. With Camel, consider that the consumer is referring to where and how the initial message is being picked up. The origin determines which type of consumer endpoint Camel component is used, 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 destination. By abstracting the integration code, developers are able to implement Enterprise Exchange Patterns (EIPs) that manipulate or transform the data within the Camel route without requiring changes to either the origin or the destination.

Components

One of the most compelling reasons to use Camel is for the library of over 180 components. Each component typically has an exhaustive set of options that allow you to customize how the component interacts with the origin or destination. In this course a subset of these components is used for labs and demonstrations. The following items are examples of components used in this course.

- File component: read from or write to a file system
- JMS component: reads from or writes to a Java Messaging Service
- FTP component: integrates with the File Transfer Protocol
- Scheduler component: Generates messages on a given schedule

Some core Camel components are especially helpful to use when developing Camel applications.

The Direct component is a Camel core component that can be used to create consumer or producer endpoints for receiving and sending messages within the same `CamelContext`. External systems cannot send messages to `direct` component endpoints. In this example XML DSL route, the `from` element is using the `direct` component to receive messages from other routes running within the same `CamelContext` as this route. The `log_body` context provided in the `uri` attribute specifies the identity for this `direct` component. Another route can send a message to this `direct` component, within the same `CamelContext` by using a producing `direct` component with the same `uri` value.

```
<route id="XML DSL route">
  <from uri="direct:log_body"/>
  <log message="Message body: ${body}"/>
  <to uri="mock:next_service"/>
</route>
```

The producer in the `to` element in this route, is using a `mock` component. The `mock` component is used to make testing routes easier, by simulating a real component. The `mock` component is often used when a real component is not available.

A complete coverage of all components is out of scope for this course. It only takes a general understanding of how to use components, however, to be able to use any of the other 180 components. All of them conform to the same usage pattern. Refer to the Camel documentation for complete coverage on any component and all the options available for each component.

Endpoints

A Camel endpoint consists of a component and a URI. The URI defines how the component is used to consume new messages from an origin or produce exchange messages to a destination. The syntax of the URI endpoint consists of three parts: the scheme, the context path, and the options.

```
URI syntax:  scheme:context_path?options
```

For example:

```
ftp://services.lab.example.com?username=delete=true&include=order.*xml
```

In this example URI, the scheme instructs Camel to use the `ftp` component. The context path of `services.lab.example.com` provides the address of the ftp service to use. After the ? two options are specified and separated by the & character to provide additional details for how the component is to be used.

Each Camel route must have a consumer endpoint and can have multiple producer endpoints. The most powerful way of creating the routes is via Camel's Java Domain-specific Language (DSL).

Java DSL Routes

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, the `from` and `to` methods are used to define the route. The `org.springframework.stereotype.Component` annotation enables Spring Boot to recognize that this class is providing a Camel Route.

```
@Component
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. Likewise, the producer also uses a `file` component to send the file to the `orders/outgoing` path.

Inside a `RouteBuilder` class, each route can be uniquely identified by 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. Add additional calls to the `from` method to create multiple routes in the `configure` method.

```
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 endpoint options to further configure how the component should function at that endpoint. The endpoint options are listed after the ? character as illustrated in the next example. 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 XML files with a name starting with `order`.

(2) The producer component throws an exception if a given file already exists.

In addition to Java DSL, routes can be created via XML DSL files. Java DSL is a richer language to work with because you have the full power of the Java language at your fingertips. Often, messages require customized handling that is beyond the scope of Camel routes. Java provides an elegant solution as discussed later in this course. Also, some Java DSL features, such as value builders (for building expressions and predicates), are not available in the XML DSL.

On the other hand, using XML DSL routes gives a convenient alternative for externalizing route configurations.

XML DSL Routes

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 with Spring Boot, declare a `routes` element, using the custom Camel Spring namespace, inside an XML configuration file located in a `camel` folder on the Java classpath. Inside the `routes` 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.

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

References

Apache Camel Spring Boot Documentation
https://camel.apache.org/camel-spring-boot/3.12.x/spring-boot.html

For more information, refer to the *Fuse Tooling Support for Apache Camel* chapter in the *Red Hat Fuse 7.10 Release Notes* at
https://access.redhat.com/documentation/en-us/red_hat_fuse/7.10/html-single/release_notes_for_red_hat_fuse_7.10/index#fuse_tooling_support_for_apache_camel

For more information, refer to the *Getting Started with Fuse on Spring Boot* chapter in the *Red Hat Fuse 7.10 Documentation* at
https://access.redhat.com/documentation/en-us/red_hat_fuse/7.10/html-single/getting_started_with_fuse_on_spring_boot/index

Guided Exercise

Creating Routes Using the Java and XML DSL

In this exercise, you will create one camel route by using Java DSL and a second camel route by using XML DSL.

This exercise includes the following routes, methods, and components:

- A Java DSL route.
- An XML DSL route.
- The Scheduler component to create exchanges.
- The log method to create console output.
- The direct component to break larger routes into reusable segments.

Outcomes

You should be able to use Spring Boot to create Camel routes.

The solution files for this exercise are in the `AD221-apps` repository, within the `route-build/solutions` directory.

Before You Begin

To perform this exercise, ensure you have the `AD221-apps` repository cloned in your workstation.

From your workspace directory, use the `lab` command to prepare your system for this exercise.

```
[student@workstation AD221]$ lab start route-build
```

The `lab` command copies the exercise files from the `~/AD221/AD221-apps/route-build/apps` directory, into the `~/AD221/route-build` directory.

Instructions

1. Navigate to the `~/AD221/route-build` directory, and open the project with your editor of choice.

2. Open the project's POM file, and add the following dependencies:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-spring-boot-starter</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
```

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

3. Create the Java DSL route by updating the `SchedulerRouteBuilder` class.

    3.1. Open the `SchedulerRouteBuilder.java` file.

    Notice that the use of the `org.springframework.stereotype.Component` annotation is the marker that tells Spring Boot to pick up this class and use it to add a route to the CamelContext.

    3.2. Enable the route by extending the RouteBuilder superclass.

```
import org.apache.camel.builder.RouteBuilder;
...
public class SchedulerRouteBuilder extends RouteBuilder
```

    3.3. Set up the Scheduler component.

    The `scheduler` component comes with Camel's core library. In this exercise we are using the scheduler to generate message exchanges every 2 seconds. The name, `mySchedule`r, is an arbitrary name assigned to this instance of the component.

```
public void configure() throws Exception {
    from("scheduler:myScheduler?delay=2000")
```

    3.4. Set up a route ID.

    Each route can be uniquely identified by 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.

```
public void configure() throws Exception {
    from("scheduler:myScheduler?delay=2000")
        .routeId("Java DSL route")
```

    3.5. Add an exchange header.

    The `scheduler` component is generating exchange messages with empty bodies. In this step of the route, use the `setBody` method and a `simple` expression to add the message creation timestamp to the body of the message. The `simple` expression in this example copies the time value from a message header field.

```
public void configure() throws Exception {
    from("scheduler:myScheduler?delay=2000")
        .routeId("Java DSL route")
        .setBody().simple("Current time is ${header.CamelTimerFiredTime}")
```

    3.6. Log a message to the console with the `Log` method.

    Add a `log` step to the route. The log helps with tracking the progression of the message through the routes.

```
    public void configure() throws Exception {
        from("scheduler:myScheduler?delay=2000")
            .routeId("Java DSL route")
            .setBody().simple("Current time is ${header.CamelTimerFiredTime}")
            .log("Sending message to the body logging route")
```

3.7. Add a producer with the Direct component.

For the final step of this route, use the `direct` component to create a producer. The message produced by the `to` method in this route is consumed by the matching `direct` endpoint, which is created in the XML route discussed next in this lab.

```
public void configure() throws Exception {
    from("scheduler:myScheduler?delay=2000")
        .routeId("Java DSL route")
        .setBody().simple("Current time is ${header.CamelTimerFiredTime}")
        .log("Sending message to the body logging route")
        .to("direct:log_body");
```

3.8. Add an XML DSL route to receive the messages. The XML DSL route created in the next steps could have been added as a second route in the Java DSL. XML DSL is used instead for illustrative purposes.

4. Create the XML route by updating the camel-context.xml file.

4.1. Open the `src/main/resources/camel/camel-context.xml` file.

By default, Spring Boot inspects the classpath for a folder called `camel`. Any XML files with routes defined in that folder are automatically discovered and added to the CamelContext by Spring Boot.

4.2. Add the XML DSL route.

Notice that the XML elements closely match the Java DSL methods.

```
<route id="XML DSL route">
  <from uri="direct:log_body"/>
  <log message="Message body: ${body}"/>
  <to uri="mock:next_service"/>
</route>
```

In this example the `mock` component is used because a real component is not available.

5. Test the route.

Test the two routes by executing the application and observing the output in the console.

5.1. Run the `./mvnw clean package spring-boot:run` command to start the Spring Boot application.

5.2. Observe that the `Java DSL` route sends exchanges to the `XML DSL` route, and that the timestamps in the log messages differ by 2 seconds for every exchange.

```
Java DSL route  : Sending message to the body logging route
XML DSL route   : Message body: Current time is Thu Dec 02 17:39:33 EST 2021
Java DSL route  : Sending message to the body logging route
XML DSL route   : Message body: Current time is Thu Dec 02 17:39:35 EST 2021
```

Finish

Stop the Spring Boot application, return to your workspace directory and use the `lab` command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation AD221]$ lab finish route-build
```

This concludes the guided exercise.

Reading and Writing Files

Objectives

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

Introducing the File and FTP Components

Use the File and FTP Camel components to integrate your application with file systems, by reading and writing files. In particular, use the File component to work with local file systems. The FTP component, on the other hand, interacts with files in remote servers via the file transfer protocol (FTP) and its secure variants.

The following class is an example of a Camel route that uses the `file` and `ftp` components. The route downloads files from an FTP server to a local directory.

```
public class FileRouteBuilder extends RouteBuilder {

    @Override
    public void configure() throws Exception {
        from("ftp://localhost:21/documents")
        .to("file:downloads/docs");
    }

}
```

The File Component

Use the `file` component to read files from the file system or write files to the file system. To use the `file` component, you must specify the URI endpoint as follows:

```
file:directoryName
```

- The `directoryName` part is required. It specifies the base directory to use for the file endpoint.
- Additionally, you can specify endpoint options. For a complete list of endpoint options, refer to the File component documentation.

For example, to copy files from one directory to another, you can use the following implementation:

```
public class FileRouteBuilder extends RouteBuilder {

    @Override
    public void configure() throws Exception {

        from("file:orders/incoming") (1)
        .to("file:orders/outgoing"); (2)
```

```
    }
}
```

Note how the example uses the `file` component both as the origin and destination of messages.

1. The `file` component reads all the files from the `orders/incoming` directory.
2. The `file` component writes all the files to the `orders/outgoing` directory.

Filtering Files

The `file` component requires a directory as the only endpoint parameter. If you want to process a specific file, either as the origin or destination of a route, then you must use the `fileName` option, as the following example shows.

```
from("file:datasets?fileName=covid_cases.csv")
.to("file:tmp/data/");
```

Likewise, you can configure the endpoint to include a subset of files, by using the `filter` option. The following example demonstrates a `file` component endpoint that only reads product `json` files.

```
from("file:warehouse/incoming?include=product.*json")
.to("file:warehouse/outgoing");
```

Note

The `include` parameter supports regex patterns, both for the `file` and `ftp` components.

Message Headers

Similar to other components, the `file` component provides producer and consumer headers in the message. For example, you can use the `CamelFileLastModified` header to inspect the last modified time of each consumed file.

```
from("file:orders/")
.log("File: ${header.CamelFileLastModified}")
```

The File component documentation provides the full list of supported headers.

The FTP Component

Use the `ftp` component to read files from and write files to an FTP server. The URI endpoint for this component is as follows:

```
ftp://[username@]hostname[:port]/directoryName
```

- The component allows the use of `ftp`, `sftp`, and `ftps` protocols.

- The `hostname` parameter is required, but the username and port are optional. You can also specify the username as an endpoint query option.
- The `directoryName` parameter must be a relative directory path.
- Similar to the `file` component, you can include further options in the endpoint. For example, you can use options to specify additional connection parameters, such as the authentication password.

The following example demonstrates the use of the `ftp` component.

```
from(
    "ftp://localhost:21/documents?" +
    "username=myuser&password=mypass"
)
.to("file:docs/");
```

The preceding Camel route reads files from the `documents` directory in an FTP server and copies the files into the `docs` directory of the local file system. Likewise, you can use the `ftp` component as a producer, to write files to an FTP endpoint.

Installing the FTP component

The `ftp` component is not included in `camel-core`. To use this component, you must specify the `camel-ftp` artifact as a Maven dependency, as follows:

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

Filtering Files

Similar to the `file` component, you can use the `fileName` and `include` parameters to select or filter specific files in an FTP server. For example, you can select a subset of the files as the following example shows:

```
from(
    "ftp://localhost:21/documents?" +
    "username=myuser&password=mypass&" +
    "include=recipe.*txt"
)
.to("file:docs/recipes");
```

FTP Connection Mode

By default, the `ftp` component uses the FTP `active` connection mode. In this mode, the FTP server can initiate requests to the client, which means that the client must be publicly accessible to the FTP server. If the FTP server cannot reach the client to start a connection, then you must switch to the FTP `passive` mode. In passive mode, only the client starts the connection to the FTP server.

To activate the passive mode, set the `passiveMode` endpoint option to `true`, as follows:

```
from(
    "ftp://localhost:21/documents?" +
    "username=myuser&password=mypass&" +
    "include=recipe.*txt&" +
    "passiveMode=true"
)
.to("file:docs/");
```

Message Headers

The `ftp` component provides producer and consumer headers for each message. For example, you can use the `CamelFileName` header to inspect the produced or consumed file name.

```
from("ftp://localhost:21/?include=record.*txt&")
.log("File: ${header.CamelFileName}")
.to("file:records");
```

The FTP component documentation provides the full list of supported headers.

References

For more information, refer to the *File Component* chapter in the *Red Hat Fuse 7.10 Apache Camel Component Reference* at https://access.redhat.com/documentation/en-us/red_hat_fuse/7.10/html-single/apache_camel_component_reference/index#file-component

For more information, refer to the *FTP Component* chapter in the *Red Hat Fuse 7.10 Apache Camel Component Reference* at https://access.redhat.com/documentation/en-us/red_hat_fuse/7.10/html-single/apache_camel_component_reference/index#ftp-component

Guided Exercise

Reading and Writing Files

In this exercise, you will route the files of a telecommunications company customer support system, from an FTP server to the local file system.

The company stores customer support requests as text files in an FTP server. Their data science team uses these files to train a natural language processing model. The team must download the files every time they retrain the model, to make sure that they use the latest available data in the training process. This is a repetitive, network-intensive, time-consuming task.

You must create a Camel route to make the latest data continuously available to the data science team. This route must copy the files from the FTP server to the local file system.

Outcomes

You should be able to create a Camel route that reads files from an FTP server and writes the files to the local file system.

The solution files for this exercise are in the `AD221-apps` repository, within the `route-files/solutions` directory.

Before You Begin

To perform this exercise, ensure you have the `AD221-apps` repository cloned in your workstation.

From your workspace directory, use the `lab` command to start this exercise.

```
[student@workstation AD221]$ lab start route-files
```

The `lab` command copies the exercise files from the `~/AD221/AD221-apps/route-files/apps` directory, into the `~/AD221/route-files` directory.

Instructions

1. Navigate to the `~/AD221/route-files` directory and open the project with an editor, such as VSCodium.

2. Add the Camel FTP component dependency in the project's POM file.

3. Edit the `FtpToFileRouteBuilder` class. Extend the `RouteBuilder` class and override the `configure` method.

4. To define the route, consume files from an FTP endpoint. Specify the FTP endpoint URI by using the following parameters:

Parameter	Value
Host	`localhost`
Port	`21721`
Username	`datauser`
Password	`fuse`
Filter	File names matching the `ticket.*txt` pattern
FTP mode	Passive

5. Set the ID of the route to `ftpRoute`. The unit tests expect your route to use this ID. Any other ID would make unit tests fail.

6. Log the processed file names. To log the file name, use the `header.CamelFileName` header.

7. Write the files to the local file system. You must write the files to the `customer_requests/` directory, within the project root.

8. Open a new terminal window and run the FTP server by using the following command:

```
[student@workstation customer-support-requests]$ python3 ftp/run.py
...output omitted...

FTP server running...
```

The FTP server serves the files contained in the `ftp/data` directory of the project root. Note that this folder contains a `README.txt` file. Your route must ignore this file.

9. Run the Spring Boot application by using `./mvnw clean spring-boot:run`. Verify that the application writes all the ticket files but not the `README.txt` file in the `customer_requests` directory.

10. Run `./mvnw clean test` to execute the unit tests. Verify that one unit test passes.

11. Stop the Spring Boot application and the FTP server.

Finish

Return to your workspace directory and use the `lab` command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation AD221]$ lab finish route-files
```

This concludes the guided exercise.

Routing Messages

Objectives

After completing this section, you should be able to implement routes that include dynamic routing.

Routing Messages Dynamically

One common integration problem is when routes are not static, and not always predetermined. Camel provides several enterprise integration patterns (EIP) to solve this problem.

Content-based Router

The *Content-based Router* (CBR) EIP routes messages to the correct destination based on the message contents.

For example, a logistics system that processes fulfillment orders can implement this EIP. A fulfillment order for the ACME provider goes to the "schema:acme-destination" endpoint, and the orders for the PACME provider goes to the "schema:pacme-destination" endpoint.

To support this pattern, Camel has the `choice` DSL element. The `choice` DSL element contains multiple `when` DSL elements, and optionally an `otherwise` DSL element.

```
from("schema:origin")
    .choice()
        .when(predicate1) (1)
            .to("schema:acme-destination")
        .when(predicate2)
            .to("schema:pacme-destination")
        ...
        .otherwise() (2)
            .to("schema:destinationN");
```

(1) The `when` DSL element requires a predicate that, when true, sends the message to a specific destination. If the predicate is false, then the route flow moves to the next `when` element. The predicate can use Simple language expression, XPath expressions, or any other expression language supported by Camel.

(2) The `otherwise` DSL element defines a destination for messages that fail to match any of the `when` predicates.

Routing Slip

The *Routing Slip* EIP routes a message consecutively through a series of processing steps. The sequence of steps is unknown at design time, and varies for each message.

For example, a pipeline that processes security checks for a credit card company might have different processing steps. A purchase from the same country requires standard security checks. But, an international purchase requires additional endpoints to verify that the purchase is not fraudulent.

In this pattern, a header field (the slip) contains the list of endpoints required in the processing steps. At run time, Apache Camel reads this header and constructs the pipeline.

Note
A pipeline is a route in which all the intermediate steps are endpoints.

To support this pattern, Camel has the `routingSlip` method.

```
from("schema:origin")
    .routingSlip( (1)
        header("destination") (2)
    );
```

(1) The `routingSlip` method requires a comma-separated list of endpoints.

(2) The `header` method extracts the list of endpoints from the `destination` header.

You can use a bean to compute the header that contains the list of endpoints.

```
from("schema:origin")
    .setHeader("destination") (1)
        .method(MyBeanImplementation.class) (2)
    .routingSlip(header("destination")); (3)
```

(1) The `setHeader` method creates a header named `destination` to store the list of endpoints.

(2) The `method` method uses the `MyBeanImplementation` Java bean to calculate the sequence of endpoints, and adds the result to the `destination` header.

(3) The `routingSlip` method constructs a pipeline from the list of endpoints stored in the `destination` header, and sends the message to those endpoints.

Dynamic Router

The *Dynamic Router* EIP routes a message consecutively through a series of processing steps. This pattern does not require the series of steps to be predetermined, as with the Routing Slip EIP. Each time the message returns from an endpoint, the dynamic router recalculates the next endpoint in the route.

To use the Dynamic Router EIP, create a Java bean with the logic that determines where the message should go next. Each time the endpoint process finishes, the route uses the same method to recalculate the next step.

```
from("schema:origin")
    .dynamicRouter(
        method(MyBeanImplementation.class, "calculateDestination")
    );
```

The preceding example uses the `calculateDestination` method of the `MyBeanImplementation` bean to calculate where the message should go next.

Dynamic To

The `toD` DSL method allows you to send a message to a single dynamically computed endpoint. In this method, the parameter must be a String, or a Simple language expression that resolves to a destination.

The following example resolves the key named `destination` from the exchange header to identify the destination.

```
from("schema:origin")
    .toD("${header.destination}");
```

References

For more information, refer to the *Bean Integration* chapter in the *Apache Camel Development Guide* at
https://access.redhat.com/documentation/en-us/red_hat_fuse/7.10/html-single/apache_camel_development_guide/index#BasicPrinciples-BeanIntegration

For more information, refer to the *Content-based Router* chapter in the *Apache Camel Development Guide* at
https://access.redhat.com/documentation/en-us/red_hat_fuse/7.10/html-single/apache_camel_development_guide/index#MsgRout-ContentBased

For more information, refer to the *Routing Slip* chapter in the *Apache Camel Development Guide* at
https://access.redhat.com/documentation/en-us/red_hat_fuse/7.10/html-single/apache_camel_development_guide/index#MsgRout-RoutingSlip

For more information, refer to the *Dynamic Router* chapter in the *Apache Camel Development Guide* at
https://access.redhat.com/documentation/en-us/red_hat_fuse/7.10/html-single/apache_camel_development_guide/index#DynamicRouter

For more information, refer to the *Dynamic To* chapter in the *Apache Camel Development Guide* at
https://access.redhat.com/documentation/en-us/red_hat_fuse/7.10/html-single/apache_camel_development_guide/index#topic-dynamicto

Guided Exercise

Routing Messages

In this exercise, you will develop a pipeline that routes messages for a book publishing company dynamically.

The company stores the books as DocBook files, and uses a shared file system for the publishing process. A team of editors and graphic designers review the book manuscripts before they are ready for printing.

At this moment, the company publishes technical, and novel books. Editors review all types of books, and graphic designers only the technical ones.

Selecting the books to review for each one of the teams is a repetitive, manual, and time-consuming task. You must use Red Hat Fuse, and create a Camel route to route the correct type of book to the correct team.

The company also has printing services. The printing services use different machines depending on the book type. You must create a Camel route to dynamically route the reviewed books to the correct printing system.

Outcomes

You should be able to create Camel routes that route messages dynamically by using the Routing Slip EIP, and the `toD` component.

The solution files for this exercise are in the `AD221-apps` repository, within the `route-messages/solutions` directory.

Before You Begin

To perform this exercise, ensure you have the `AD221-apps` repository cloned in your workstation.

From your workspace directory, use the `lab` command to start this exercise.

```
[student@workstation AD221]$ lab start route-messages
```

The `lab` command copies the exercise files from the `~/AD221/AD221-apps/route-messages/apps` directory, into the `~/AD221/route-messages` directory.

Instructions

1. Navigate to the `~/AD221/route-messages` directory, open the project with your editor of choice, and examine the code.

2. Create a destination algorithm for the book review pipeline. This algorithm must use the book type to decide the destination endpoint. Edit the `RoutingSlipStrategy` class, and implement an algorithm that implements the following rules:

Book type	Route destinations
`technical`	`file://data/pipeline/graphic-designer` and `file://data/pipeline/editor`
`novel`	`file://data/pipeline/novel`

3. Create a route for the book review pipeline in the `BookReviewPipelineRouteBuilder` class. The route must collect all the books located in the `file://data/manuscripts` endpoint, with the `noop` option activated, and use the Routing Slip EIP. Use the `RoutingSlipStrategy` bean to compute the routing slip header, and set `book-review-pipeline` as the route ID.

4. Verify the correctness of the `book-review-pipeline` route by executing the unit tests. Run the `./mvnw clean -Dtest=BookReviewPipelineRouteBuilderTest test` command, and verify that three unit tests pass.

5. Create a destination algorithm for the book printing pipeline in the `DynamicRoutingStrategy` class. This algorithm must use the book type to dynamically decide the destination endpoint. The computed destination must be `file://data/printing-services/`*`BOOK_TYPE`*.

6. Create a route for the book printing pipeline in the `BookPrintingPipelineRouteBuilder` class. The route must collect all the books located in the `file://data/pipeline/ready-for-printing` endpoint, with the `noop` option activated. Set `book-printing-pipeline` as the route ID, add a message header with the computed destination, and use the `toD` component to send the messages to the computed endpoint.

7. Run the Spring Boot application by using the `./mvnw clean package spring-boot:run` command.

8. Wait for the application to process the books stored in the `file://data/manuscripts` endpoint, and manually verify the correctness of the book review pipeline:

   - The `data/pipeline/editor` directory contains the files: `book-01.xml`, `book-02.xml`, and `book-03.xml`.
   - The `data/pipeline/graphic-designer` directory contains the files: `book-01.xml`, and `book-03.xml`.

9. To simulate the end of the book review process by the editors, copy the `book-01.xml` and `book-02.xml` files from the `data/pipeline/editor` directory to the `data/ready-for-printing` directory. Wait for the application to process the files, and manually verify the correctness of the book printing pipeline:

   - The `data/printing-services/novel` directory contains the file `book-02.xml`.
   - The `data/printing-services/technical` directory contains the file `book-01.xml`.

10. Stop the Spring Boot application, run the tests by executing the `./mvnw test` command, and verify that five unit tests pass.

Finish

Return to your workspace directory, and use the `lab` command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation AD221]$ lab finish route-messages
```

This concludes the guided exercise.

Developing a Camel Processor

Objectives

After completing this section, you should be able to use processors to manipulate exchange messages in routes.

Defining a Camel Processor

Camel routes often need to do more than pass a message from a consumer to a producer. A processor is a command you insert into the route to perform the arbitrary processing of messages. The Message Translator EIP is implemented by a Camel Processor.

Describing the Message Translator EIP

With the Message Translator EIP, a message enters a route with an initial format. An intermediate step changes the format of the message. The message then exits the route with the new format.

Figure 2.2: Message translation

Camel provides many ways to implement the `Translation` step in a route. If a built-in Camel component is available to perform the required function, then this option is the recommended approach. However, when custom Java code is required to provide the needed business logic, a Camel Processor can be used to make changes to any part of an exchange message.

Defining the Processor Types

A processor is a command you can insert into a route to perform the arbitrary processing of messages that flow through the route. There are two types of processors, Built-in processors and custom processors.

Built-in Processors

Red Hat Fuse provides many built-in processors that perform a wide variety of functions. The following list is a sample of these built-in processors.

bean
: Processes the current exchange by invoking a method on a Java object (or bean).

convertBodyTo
: Converts the In message body to the specified type.

filter
: Uses a predicate expression to filter incoming exchanges.

log
: Logs a message to the console.

marshal
Transforms into a low-level or binary format by using the specified data format, in preparation for sending over a particular transport protocol.

unmarshal
Transforms the In message body from a low-level or binary format to a high-level format, by using the specified data format.

setBody
Sets the message body of the exchange's In message.

setHeader
Sets the specified header in the exchange's In message.

removeHeaders
Removes the headers matching the specified pattern from the exchange's In message. The pattern can have the form `prefix\*` – in which case it matches every name starting with `prefix`– otherwise, it is interpreted as a regular expression.

Custom Processors

Through integration with the Camel API, a custom processor can access any part of the camel exchange message or even the broader camel context.

Developing a Custom Processor

A built-in processor should be used whenever possible. However, when developing complex Camel routes, it might be necessary to apply business logic to a message. A custom Camel processor is one way of adding the functionality, providing complete control over the exchange message.

Implementing the Processor Interface

You must implement the `Processor` interface to create a Camel Processor. 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 called `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 take advantage of other Camel features, such as data type converters and fluent expression builders.

```
import org.apache.camel.Exchange;
import org.apache.camel.Processor;

public class MyProcessor implements Processor {
    @Override
    public void process(Exchange exchange) throws Exception {
        String custom = exchange.getIn()
            .getBody(String.class); (1)
        String date = custom.substring(10, 20);
        String formatedOrderDate = getFormattedDate(date);
```

```
            exchange.getIn().setHeader("orderDate", formatedOrderDate); (2)
        }
    }
```

(1) Extracts the body of the input message as a String using `getBody`

(2) Adds a message header called `orderDate` with the value of `formatedOrderDate` variable.

The `exchange` object instance includes an output message that can be accessed using the `getOut` method. In practice, the outgoing message is often not used because it does not include the message headers and attachments. You can copy the headers and attachments from the incoming message to the outgoing message, but this can be tedious. The alternative is to set the changes directly on the incoming message, and to not use the outgoing message as illustrated in the preceding example.

To use a processor inside a route, insert the `process` method for Java DSL:

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

Before writing your own `Processor` implementation, determine first whether there are ready-to-use Camel components that might provide the same result with less custom code, for example:

1. The `transform` component allows for changing of a message body by using any expression language supported by Camel.
2. The `setHeader` component allows the changing of header values.
3. The `bean` component allows the calling of 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. To 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.

References

For more information, refer to the *Processors* chapter in the *Red Hat Apache Camel Development Guide* at
https://access.redhat.com/documentation/en-us/red_hat_fuse/7.10/html-single/apache_camel_development_guide/index#FMRS-P

For more information, refer to the *Implementing a Processor* chapter in the *Red Hat Apache Camel Development Guide* at
https://access.redhat.com/documentation/en-us/red_hat_fuse/7.10/html-single/apache_camel_development_guide/index#Processors

Guided Exercise

Developing a Camel Processor

In this exercise, you will process incoming files and add a line number at the beginning of each line.

Your department produces text files that contain the details of processed orders. In these files, each line corresponds to an order. Recently, the business intelligence team of your company has required the lines of these files to be numbered. Therefore, you must process the input files to add a line number to each line.

Outcomes

You should be able to modify the message content to meet the requirements of the next component in the Route.

The solution files for this exercise are in the `AD221-apps` repository, within the `route-processor/solutions` directory.

Before You Begin

To perform this exercise, ensure you have the `AD221-apps` repository cloned in your workstation.

From your workspace directory, use the `lab` command to start this exercise.

```
[student@workstation AD221]$ lab start route-processor
```

The `lab` command copies the exercise files from the `~/AD221/AD221-apps/route-processor/apps` directory, into the `~/AD221/route-processor` directory.

Instructions

1. Navigate to the `~/AD221/route-processor` directory, open the project with your editor of choice, and examine the code.

2. Review format of the input data.

```
[student@workstation route-processor]$ cat orders/incoming/orders1.csv
DPR,DPR,100,AD-982-708-895-
F-6C894FB,52039657,1312378,83290718932496,04/12/2018,2,200.0,-200.0,0.0,...
RJF,Product P,28 / A / MTM,83-490-
E49-8C8-8-3B100BC,56914686,3715657,36253792848113,01/04/2019,2,...
CLH,Product B,32 / B / FtO,68-ECA-BC7-3B2-A-
E73DE1B,24064862,9533448,73094559597229,05/11/2018,0,...
...output omitted...
```

3. Update the route `configure` method of the `FileRouteBuilder` class to add the Processor code.

3.1. Add the processor step to the route pipeline and create a new `org.apache.camel.Processor` instance as the parameter:

```
.process( new Processor() {
    public void process( Exchange exchange ) {
    }
} )
```

3.2. Inside the `process` method get the input message:

```
String inputMessage = exchange.getIn().getBody( String.class );
```

3.3. Create an external thread safe counter to use in the lambda function:

```
AtomicReference<Long> counter = new AtomicReference<>(1L);
```

3.4. Using the counter, process the lines to add the counter value at the beginning:

```
String processedLines = Stream.of(inputMessage.split(separator))
    .map( l -> counter.getAndUpdate( p -> p + 1 ).toString() + "," + l )
    .collect(Collectors.joining(separator));
```

3.5. Set the modified lines as the content of the output message.

```
exchange.getIn().setBody( processedLines );
```

The whole section of the `process` function should look like the following lines:

```
.process( new Processor() {
    public void process( Exchange exchange ) {
        String inputMessage = exchange.getIn().getBody( String.class );

        AtomicReference<Long> counter = new AtomicReference<>(1L);

        String processedLines = Stream.of(inputMessage.split(separator))
            .map( l -> counter.getAndUpdate( p -> p + 1 ).toString() + "," + l )
            .collect(Collectors.joining(separator));

        exchange.getIn().setBody( processedLines );
    }
} )
```

4. Test the route that processes the data.

4.1. Run the route by using the `./mvnw clean spring-boot:run` Maven goal.

```
[student@workstation route-processor]$ ./mvnw spring-boot:run
...output omitted...
... Route: route1 started and consuming from: file://orders/incoming?noop=true
... Total 1 routes, of which 1 are started
...output omitted...
```

4.2. Give the route a few moments to process the incoming files, and then terminate it by using Ctrl+C.

4.3. Inspect the output folder to verify that the output files contain line numbers at the beginning of each line. Expected output is:

```
[student@workstation route-processor]$ cat orders/outgoing/orders2.csv
1,DPR,DPR,100,AD-982-708-895-
F-6C894FB,52039657,1312378,83290718932496,04/12/2018,2,200.0,-200.0 ...
2,RJF,Product P,28 / A / MTM,83-490-
E49-8C8-8-3B100BC,56914686,3715657,36253792848113,01/04/2019,2,190.0,-190.0 ...
3,CLH,Product B,32 / B / FtO,68-ECA-BC7-3B2-A-
E73DE1B,24064862,9533448,73094559597229,05/11/2018,0,164.8,-156.56,-8.24 ...
...output omitted...
```

Notice the line number at the beginning of each line of the processed output.

Note

If you need to start over to test something or in case of a retry due to a mistake, then run the following `lab finish` command and begin again with the `lab start`.

Finish

Return to your workspace directory and use the `lab` command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation AD221]$ lab finish route-processor
```

This concludes the guided exercise.

Quiz

Creating Camel Routes

In this quiz, consider you are developing a system that processes orders from a file stored in an FTP server and distributes them among two partners depending on the type. One partner uses a different date format, so you need to modify the date on each order before sending it to that specific partner. You must write each partner's orders in its own file so they can read their orders.

Choose the correct answers to the following questions:

1. The first step is to connect to the FTP server to retrieve the orders. Regarding the following line, which two of the sentences are true? (Choose two.):

```
from( "ftp://ftpserver/integration?include?include=incoming_orders.*csv" )
```

a. Connects to the FTP server in `localhost`.
b. Connects to the FTP server in `ftpserver`.
c. Picks up the file `incoming_orders.csv` in the `integration` directory.
d. Picks up all the CSV files in the `integration` directory that start with `incoming_orders`.

2. The next step is to split orders by type. Regarding the following code, which three sentences are true? (Choose three.)

```
.choice()
    .when( header( "CamelFileName" ).contains( "parts") )
        .to( "direct:parts" )
    .when( header( "CamelFileName" ).contains( "accessories" ) )
        .to( "direct:accessories" )
    .otherwise()
        .log( "File: ${header.CamelFileName} does not have correct type." )
```

a. Files containing `parts` in their name are logged in the `parts` log.
b. Files containing `parts` in their name are sent to the `direct:parts` endpoint.
c. Files containing `accessories` in their name are sent to the `direct:accessories` endpoint.
d. Files with neither `parts` nor `accessories` in their name are logged.
e. All files are logged.

3. After the splitting step, orders in the `accesories` consumer need to be processed to attend to the specifications of a legacy system. Regarding the following processing, which of the following sentences is true?

```
from( "direct:accessories" )
    .process( new Processor() {
        String inputMessage = exchange.getIn().getBody( String.class );

        String output = inputMessage.replace( "\n", "\r\n" );

        exchange.getIn().setBody( output );
    })
.to( "direct:accessories2" )
```

a. The processor replaces each newline character with legacy characters in messages received from the `accessories` consumer, and the route sends the result to the `accessories2` producer.
b. The processor replaces each newline character with legacy characters in the `accessories` consumer.
c. The processor replaces each newline character with legacy characters in the `accessories2` producer.
d. The processor replaces each return carriage character with newline characters in the `accessories` consumer.
e. The processor replaces each newline character with legacy characters in the `accessories` consumer to the `accessories2` producer.

4. To finish the route, you must store each type of order in files. Regarding the following code, which two sentences are true? (Choose two.)

```
from( "direct:accessories2" )
.to( "file:/shared/?fileName=accessories_${date:now:yyyyMMdd}.csv" )

from( "direct:parts" )
.to( "file:/shared/?fileName=parts_${date:now:yyyyMMdd}.csv" )
```

a. The route stores orders in the `accessories` and `parts` consumers into their respective files.
b. The route stores orders in the `accessories2` and `parts` producers into their respective files.
c. The route stores orders in the `accessories2` and `parts` consumers into their respective files.
d. The route appends the current date to each file name.

Solution

Creating Camel Routes

In this quiz, consider you are developing a system that processes orders from a file stored in an FTP server and distributes them among two partners depending on the type. One partner uses a different date format, so you need to modify the date on each order before sending it to that specific partner. You must write each partner's orders in its own file so they can read their orders.

Choose the correct answers to the following questions:

1. The first step is to connect to the FTP server to retrieve the orders. Regarding the following line, which two of the sentences are true? (Choose two.):

```
from( "ftp://ftpserver/integration?include?include=incoming_orders.*csv" )
```

a. Connects to the FTP server in `localhost`.
b. Connects to the FTP server in `ftpserver`.
c. Picks up the file `incoming_orders.csv` in the `integration` directory.
d. Picks up all the CSV files in the `integration` directory that start with `incoming_orders`.

2. The next step is to split orders by type. Regarding the following code, which three sentences are true? (Choose three.)

```
.choice()
    .when( header( "CamelFileName" ).contains( "parts") )
        .to( "direct:parts" )
    .when( header( "CamelFileName" ).contains( "accessories" ) )
        .to( "direct:accessories" )
    .otherwise()
        .log( "File: ${header.CamelFileName} does not have correct type." )
```

a. Files containing `parts` in their name are logged in the `parts` log.
b. Files containing `parts` in their name are sent to the `direct:parts` endpoint.
c. Files containing `accessories` in their name are sent to the `direct:accessories` endpoint.
d. Files with neither `parts` nor `accessories` in their name are logged.
e. All files are logged.

3. After the splitting step, orders in the `accesories` consumer need to be processed to attend to the specifications of a legacy system. Regarding the following processing, which of the following sentences is true?

```
from( "direct:accessories" )
    .process( new Processor() {
        String inputMessage = exchange.getIn().getBody( String.class );

        String output = inputMessage.replace( "\n", "\r\n" );

        exchange.getIn().setBody( output );
    })
.to( "direct:accessories2" )
```

a. The processor replaces each newline character with legacy characters in messages received from the `accessories` consumer, and the route sends the result to the `accessories2` producer.
b. The processor replaces each newline character with legacy characters in the `accessories` consumer.
c. The processor replaces each newline character with legacy characters in the `accessories2` producer.
d. The processor replaces each return carriage character with newline characters in the `accessories` consumer.
e. The processor replaces each newline character with legacy characters in the `accessories` consumer to the `accessories2` producer.

4. To finish the route, you must store each type of order in files. Regarding the following code, which two sentences are true? (Choose two.)

```
from( "direct:accessories2" )
.to( "file:/shared/?fileName=accessories_${date:now:yyyyMMdd}.csv" )

from( "direct:parts" )
.to( "file:/shared/?fileName=parts_${date:now:yyyyMMdd}.csv" )
```

a. The route stores orders in the `accessories` and `parts` consumers into their respective files.
b. The route stores orders in the `accessories2` and `parts` producers into their respective files.
c. The route stores orders in the `accessories2` and `parts` consumers into their respective files.
d. The route appends the current date to each file name.

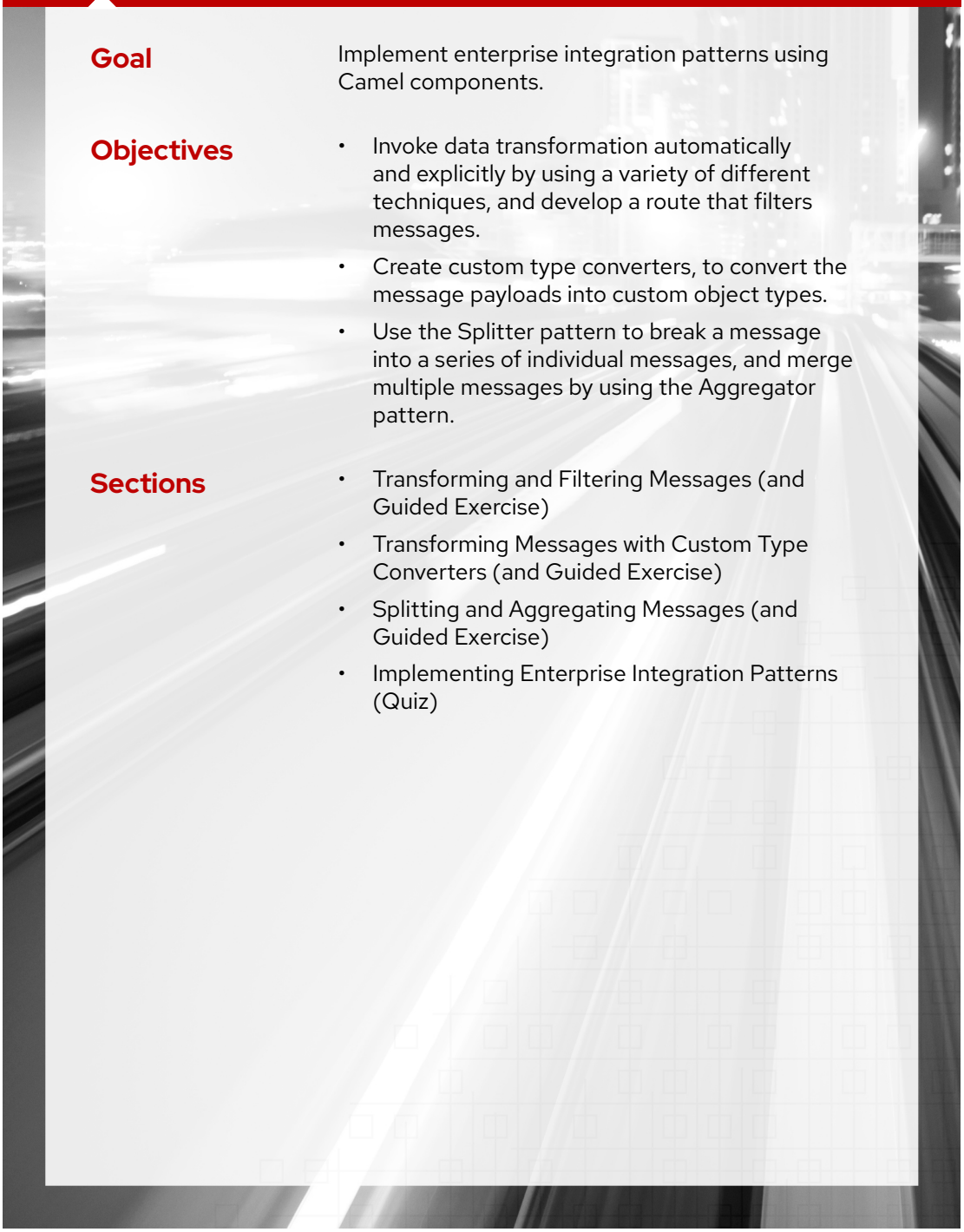
Summary

In this chapter, you learned:

- A Camel route describes the path of a message from an origin endpoint to a destination endpoint.
- You can define routes with the Java DSL or the XML DSL.
- To use the File or FTP components to read and write files.
- For complex message routing, Camel provides built-in enterprise integration pattern (EIP) implementations.
- Routes might include message processing logic.
- You can use built-in or custom processors for message processing.
- Processors are intended for integration logic, not business logic.

Chapter 3

Implementing Enterprise Integration Patterns



Goal

Implement enterprise integration patterns using Camel components.

Objectives

- Invoke data transformation automatically and explicitly by using a variety of different techniques, and develop a route that filters messages.
- Create custom type converters, to convert the message payloads into custom object types.
- Use the Splitter pattern to break a message into a series of individual messages, and merge multiple messages by using the Aggregator pattern.

Sections

- Transforming and Filtering Messages (and Guided Exercise)
- Transforming Messages with Custom Type Converters (and Guided Exercise)
- Splitting and Aggregating Messages (and Guided Exercise)
- Implementing Enterprise Integration Patterns (Quiz)

Transforming and Filtering Messages

Objectives

After completing this section, you should be able to invoke data transformation automatically and explicitly by using a variety of different techniques, and develop a route that filters messages.

Camel Data Transformation Terms

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 must transform the data from one system before another system can correctly process it. 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.

The following terms are commonly used to describe transformation features in Camel.

Data Formats
: `DataFormats` in Camel are the various forms that your data can be represented, either in binary or text. Each data format has a class you must instantiate and optionally configure before you can use it in a route.

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.

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.

Transforming XML Data by 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 must use to annotate your model class. JAXB matches fields on the model class to elements and attributes contained in the XML data by 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;
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAttribute;

@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class Order implements Serializable {  (1)

    private static final long serialVersionUID = 5851038813219503043L;

    @XmlAttribute
    private String id;

    @XmlAttribute
    private String description;

    @XmlAttribute
    private double value;

    @XmlAttribute
    private double tax;
}
```

(1) Note that this model class implements the java.io.Serializable interface, which is required by JAXB to execute the marshaling and unmarshaling process.

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 Java objects in the body of a message and then sends the message to an ActiveMQ queue called `itemInput`. The Camel route consumes the messages, and JAXB marshals the object to XML data and replaces the contents of the exchange body with the corresponding XML equivalent. The route then sends the XML data in a message to an ActiveMQ queue called `itemOutput`.

```
from("activemq:queue:itemInput)
  .marshal().jaxb()
  .to("activemq:queue:itemOutput);
```

Notice that in the previous example 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 marshal to the XML data.

Transforming JSON Data by Using 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 must 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,
    ...
```

The following example is a Jackson-annotated model class which you can use to marshal the JSON data:

```
package com.redhat.training;

import com.fasterxml.jackson.annotation.JsonIgnore;
import com.fasterxml.jackson.annotation.JsonProperty;

public class Order implements Serializable{

    private static final long serialVersionUID = 5851038813219503043L;

    @JsonProperty("ID") (1)
    private String id;

    @JsonIgnore (2)
    private String description;

    private double value;
```

```
    private double tax;
}
```

1. Use the `@JsonProperty` annotation to override the field name with 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.

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

As discussed previously, Camel supports data formats to perform both XML and JSON-related conversions. Both of these transformations, however, 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` library, the terminologies `marshaling` and `unmarshaling` are not as obvious because there are no Java objects involved. The library 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.

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 must 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>
</dependency>
```

Note

The XOM library cannot be included by default due to an incompatible license with Apache Software Foundation. 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");
```

Filtering Messages

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")
```

Implementing Predicates

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

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 called `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'))]")
```

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

```
from("file:orders/incoming?include=order.*xml")
    .filter(xpath("/order/orderItems/orderItem/orderItemQty > 1"))
    .to("file:orders/outgoing/?fileExist=Fail");
```

An expression language (EL) used to identify Java objects is the `Simple` EL. It uses a syntax that resembles many other scripting languages, using dots to step through nested Java objects. For example, in a class called Order, with an address attribute that contains a ZIP code, the `Simple` EL to search for the zip code 33212 is:

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

Implementing the Wire Tap Pattern in a Camel Route

The *Wire Tap* EIP is an integration pattern that creates a duplicate copy of each processed message, and then forwards the duplicate message to a secondary destination. This functionality is useful for inspecting messages as they travel along a Camel route without impacting the route execution itself.

To implement this pattern, Camel provides the `wireTap` component.

```
from("activemq:queue:orders.in")
  .wireTap("file:backup")
  .to("direct:start");
```

The preceding example uses the Wire Tap pattern, and sends a copy of every message received on the `activemq:queue:orders.in` endpoint to the `file:backup` endpoint.

References

For more information, refer to the *JAXB DataFormat* chapter in the *Apache Camel Component Reference Guide* at
https://access.redhat.com/documentation/en-us/red_hat_fuse/7.10/html-single/apache_camel_component_reference/index#jaxb-dataformat

For more information, refer to the *JSON Jackson DataFormat* chapter in the *Apache Camel Component Reference Guide* at
https://access.redhat.com/documentation/en-us/red_hat_fuse/7.10/html-single/apache_camel_component_reference/index#json-jackson-dataformat

For more information, refer to the *Message Filter* chapter in the *Apache Camel Development Guide* at
https://access.redhat.com/documentation/en-us/red_hat_fuse/7.10/html-single/apache_camel_development_guide/index#MsgRout-MsgFilter

For more information, refer to the *Wire Tap* chapter in the *Apache Camel Development Guide*
https://access.redhat.com/documentation/en-us/red_hat_fuse/7.10/html-single/apache_camel_development_guide/index#WireTap

Guided Exercise

Transforming and Filtering Messages

In this exercise, you are working for an e-commerce company. You are required to transform orders to XML and to JSON because these formats are required for further processing by other systems. You are receiving orders from a Red Hat AMQ messaging broker. The orders are then picked up for transformation by your Fuse route.

Outcomes

In this exercise you should be able to:

- Read the incoming Order objects from an ActiveMQ queue.
- Marshal the Order objects 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.

The solution files for this exercise are in the `AD221-apps` repository, within the `pattern-filter/solutions` directory.

Before You Begin

To perform this exercise, ensure you have the `AD221-apps` repository cloned in your workstation.

From your workspace directory, use the `lab` command to start this exercise.

```
[student@workstation AD221]$ lab start pattern-filter
```

The `lab` command copies the exercise files from the `~/AD221/AD221-apps/pattern-filter/apps` directory, into the `~/AD221/pattern-filter` directory. The `Lab` command also creates a Red Hat AMQ instance.

Note

You can inspect the logs for the AMQ instance at any time with the following command:

```
[student@workstation AD221]$ podman logs artemis
```

Instructions

1. Navigate to the `~/AD221/pattern-filter` directory and open the project with an editor, such as VSCodium.

2. Open the project POM file and add the dependencies that are required.

Dependencies to Add

XML to JSON Transformations	`<dependency>`
	`  <groupId>org.apache.camel</groupId>`
	`  <artifactId>camel-xmljson</artifactId>`
	`  </dependency>`
	`<dependency>`
	`  <groupId>xom</groupId>`
	`  <artifactId>xom</artifactId>`
	`  <version>1.3.7</version>`
	`</dependency>`

JSON Paths	`<dependency>`
	`  <groupId>org.apache.camel</groupId>`
	`  <artifactId>camel-jsonpath</artifactId>`
	`</dependency>`

XML Transformation	`<dependency>`
	`  <groupId>org.apache.camel</groupId>`
	`  <artifactId>camel-jaxb</artifactId>`
	`</dependency>`

3. Open the `TransformRouteBuilder` class and inspect the route definition:

   The `jms` consumer retrieves `Order` objects from the `orderInput` queue in the Red Hat AMQ broker. The `routeId` adds a label to the route making it easier to trace in the log files. JAXB is then used to marshal the `Order` object into XML, which can be viewed in the console.

4. Build and run the application to verify that the route and view the XML for the five sample Order objects.

```
[student@workstation pattern-filter]$ ./mvnw clean spring-boot:run
```

5. Update the route to convert XML messages to the JSON format.

   Create the XmlJsonDataFormat instance.

```
XmlJsonDataFormat xmlJson = new XmlJsonDataFormat();
```

   This class is used by the marshal method of the Java DSL to convert the XML data into JSON data.

6. Marshal the XML data into JSON.

   In the route definition, add the code to convert the XML to JSON and log the result to the console:

```
from("jms:queue:orderInput")
  .routeId("Transforming Orders")
  .marshal().jaxb()
  .log("XML Body: ${body}")
  .marshal(xmlJson)
  .log("JSON Body: ${body}")
  .to("mock:fufillmentSystem");
```

7. Using the `delivered` field on the Order object, filter out any orders that were already delivered.

   Add the following method to filter out `Order` objects that have a property of `delivered` set to `true`.

```
from("jms:queue:orderInput")
  .routeId("Transforming Orders")
  .marshal().jaxb()
  .log("XML Body: ${body}")
  .marshal(xmlJson)
  .log("JSON Body: ${body}")
  .filter().jsonpath("$[?(@.delivered !='true')]")
  .to("mock:fufillmentSystem");
```

   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`.

8. Send all undelivered orders to a mock endpoint representing an order logging system.

   Add the `wireTap` DSL method to the route definition to send a copy of all undelivered orders to a separate direct endpoint:

```
from("jms:queue:orderInput")
  .routeId("Transforming Orders")
  .marshal().jaxb()
  .log("XML Body: ${body}")
  .marshal(xmlJson)
  .log("JSON Body: ${body}")
  .filter().jsonpath("$[?(@.delivered !='true')]")
  .wireTap("direct:jsonOrderLog")
  .to("mock:fufillmentSystem");
```

9. Create a new `direct` route to a mock order logging system.

   Open the `OrderLogRouteBuilder` class and add a route that logs the orders:

```
from("direct:jsonOrderLog")
  .routeId("Log Orders")
  .log("Order received: ${body}")
  .to("mock:orderLog");
```

10. Build and run the application with the `./mvnw clean spring-boot:run` command.

11. Examine the console log.

    For each Order with a `delivered` value of `false`, there is a log entry similar to the following.

```
WireTap] Log Orders : Order received: {"orderItems":[{"extPrice":" ...
```

    In the sample, two of the five orders have the `delivered` field set to `false`. Thus you should find two of these `Log Orders` resulting from the wiretap.

12. Use the `./mvnw clean test` command to run the unit tests, and verify that the two tests pass.

```
Results :

Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
```

Finish

Return to your workspace directory, and use the lab command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation AD221]$ lab finish pattern-filter
```

This concludes the guided exercise.

Transforming Messages with Custom Type Converters

Objectives

After completing this section, you should be able to create custom type converters, to convert the message payloads into custom object types.

Camel Type Converters

Camel provides a data transformation mechanism to transform particular data formats. Camel `data transformation` covers two types of transformation:

- `Data format transformation:` The data format of the message body transforms. `XML` to `JSON` transformation is an example of this. In *Transforming and Filtering Messages* you read about how to apply data format transformation in Camel.

- `Data type transformation:` The data type of the message body transforms. `java.lang.String` to `javax.jms.TextMessage` transformation is an example of this.

For the data type transformation, Camel has a built-in type-converter system that automatically converts between well-known types. This system allows Camel components to work together without having type mismatches.

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. Conversions frequently occur between the following types:

- `File`

- `String`

- `byte[]` and `ByteBuffer`

- `InputStream` and `OutputStream`

- `Reader` and `Writer`

- XML payloads such as `Document` and `Source`

The `Message` interface defines the `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.

If you needed to route files to a JMS queue using `javax.jmx.TextMessage` objects then you must convert each file to a `String`, which forces the JMS component to use the `TextMessage` class. Use the `convertBodyTo` method to convert to `String`:

```
from("file://orders/inbox")
  .convertBodyTo(String.class)
  .to("activemq:queue:inbox");
```

Examining the Camel Type Conversion

Camel implements the type conversion strategy with the `TypeConverter` interface. The `TypeConverter` interface has a method called `convertTo` that enables Camel to convert one type to another.

```
<T> T convertTo(Class<T> type, Exchange exchange, Object value)
                throws TypeConversionException;
```

Note

Camel implements more than 350 out-of-the-box type converters, which are capable of applying type conversion for the most commonly used types.

A Camel context can have more than one converter active. Camel keeps track of these converters in a registry called `TypeConverterRegistry` where all the type converters are registered when Camel is started. This allows Camel to pick up type converters not only from camel-core, but also from any of the Camel components, including the custom converters in your Camel applications.

Figure 3.1: Type converter registry

Camel scans and loads the type converters by using the `org.apache.camel.impl.converter.AnnotationTypeConverterLoader` class and registers them in the `TypeConverterRegistry`.

Creating a Custom Type Converter

In some use cases, you might need to use a particular data format that is not supported by Camel by default. For example, you might work with a vendor that uses proprietary encryption technology, which requires a custom API to encrypt/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. A custom converter must have a static method whose signature must meet the following requirements:

- The return value must be a type that is compatible with the target object type you are converting to.
- 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.

You must annotate both the converter class and the static method by using the `@Converter` annotation.

The following code is an example of a custom type converter class that converts an `Order` object to an encrypted `InputStream`.

```
package com.redhat.training.ad221.converters;

// imports omitted

@Converter (1)
public class OrderConverter {

    private CustomEncryptor encryptor = EncryptorAPI.createEncryptor(); (2)

    @Converter (3)
    public static InputStream toInputStream(Order order) { (4)
        String encryptedOrderInfo = encryptor.encrypt(order.toString());
        return new ByteArrayInputStream(encryptedOrderInfo.getBytes()); (5)
    }
}
```

1. Class level `@Converter` annotation.
2. A custom encryptor instance that comes from a vendor API.
3. Method level `@Converter` annotation.
4. The `toInputStream` method takes an `Order` object instance as a source and returns an `InputStream` as the target of the conversion.
5. Returns a `ByteArrayInputStream` that has the encrypted order information.

Note

In this example, the `Order` class has a `toString` method that returns the details of an order in `String` format. The `OrderConverter` class uses this method to get the order information as `String` and encrypt it.

The following code is an example of an `Order` class.

```
// imports omitted

public class Order implements Serializable{

    private int id;
    private String description;
    private double price;
    private double tax;

// constructor(s) omitted

  @Override
  public String toString() {
      return "Order [description=" + description + ", id=" + id + ", price=" +
 price + ", tax=" + tax + "]";
  }

// getters and setters omitted
}
```

Additionally, to allow Camel to register your custom type converter classes in the `TypeConverterRegistry`, you must include your custom converter package name in the `META-INF/services/org/apache/camel/TypeConverter` file.

`TypeConverter` is a service discovery file that has a list of fully qualified class names or packages that contain Camel type converters. Each record in this file must be on a new line.

The following code snippet is an example of `TypeConverter` file content:

```
com.redhat.training.ad221.MyConverter (1)
com.redhat.training.ad221.MyOtherConverter

com.redhat.training.ad221.converters (2)
```

(1) A fully qualified custom converter class definition.

(2) A converters package definition that might have custom converters in it. This enables the `OrderConverter` in the preceding example to be discovered.

After setting up the type converter registry for your converters, you can create a route that uses the `OrderConverter` implicitly as follows:

```
// imports omitted

public class OrderRouteBuilder extends RouteBuilder {
```

```
@Override
public void configure() throws Exception {
    from("kafka:orders") (1)
    .unmarshal(new JacksonDataFormat(Order.class)) (2)
        .to("http4://localhost:8081/orders"); (3)
}
```

(1) The route consumes messages from the `orders` Kafka topic in JSON format.

(2) Unmarshals the JSON data to the `Order` data.

(3) The `http4` component requires `InputStream` as the input data format. In the previous step, you unmarshaled the JSON data to `Order` data, so a conversion must happen before this step.

Camel searches for a converter that converts from an `Order` type to an `InputStream` type. Camel finds your custom converter `OrderConverter` in the type converter registry and runs the conversion implicitly.

References

For more information, refer to the *Type Converters* chapter in the *Apache Camel Development Guide* at
https://access.redhat.com/documentation/en-us/red_hat_fuse/7.10/html-single/apache_camel_development_guide/index#TypeConv

Claus Ibsen and Jonathan Anstey. (2018) Camel in Action, Second Edition. Manning. ISBN 978-1-617-29293-4.

Guided Exercise

Transforming Messages with Custom Type Converters

In this exercise, you will update an integration service of a smart home IoT software company.

The IoT software company sends commands to air purifier devices for configuration. The current air purifier devices accept these commands in JSON data format. However, they renew the devices and use a second version of the air purifier devices. The new versioned devices accept a custom format instead of the JSON format.

You must update the Camel route in the `command-router` application to send the configuration commands to the version-2 air purifier devices in the required custom format.

Outcomes

In this exercise you should be able to:

- Create a Camel route that reads the configuration data from a CSV file.
- Unmarshal the configuration data into Java objects.
- Create a custom type converter for converting the objects to the required command object format.

The solution files for this exercise are in the `AD221-apps` repository, within the `pattern-converter/solutions` directory.

Before You Begin

To perform this exercise, ensure you have the `AD221-apps` repository cloned in your workstation.

From your workspace directory, use the `lab` command to start this exercise.

```
[student@workstation AD221]$ lab start pattern-converter
```

The `lab` command copies the exercise files from the `~/AD221/AD221-apps/pattern-converter/apps` directory, into the `~/AD221/pattern-converter` directory.

Instructions

1. Navigate to the `~/AD221/pattern-converter/air-purifier-v2` directory, compile and run the application by using `./mvnw package quarkus:dev`.

   This is the application that represents the new version of the air purifier device.

2. Navigate to the `~/AD221/pattern-converter` directory and open the `command-router` project with an editor, such as VSCodium.

3. In the `command-router` application directory, run `./mvnw clean test` to execute the test.

   Verify that the unit test fails.

The unit test fails because the current `command-router` application sends JSON formatted messages to the `air-purifier-v2` application. The `air-purifier-v2` application does not accept JSON format, it accepts a custom data format instead, so the Quarkus application throws an exception.

4. Examine the `CommandConfiguration` class. This class acts as a Data Transfer Object (DTO) for preparing the data in the right custom format. The `toString` method of the class creates a formatted String similar to the following:

```
name=A Command Name|sentDate=26/02/2022|type=A COMMAND TYPE|value=2
```

This is the format that the `air-purifier-v2` application accepts.

5. Open the `CommandConfigurationConverter` class and add the `@Converter` annotations both for the class level and the method level.

Implement the `convertToCommandConfiguration` method by setting the attributes of the `commandConfig` object instance.

The `commandConfig` is an instance of the `CommandConfiguration` class. Use the `toString` method to obtain the formatted results.

Use the `csvRecord` object instance to obtain the attribute values.

6. Navigate to the `src/main/resources/META-INF/services/org/apache/camel/TypeConverter` file and add the package definition for the converter you created.

The package definition of the `CommandConfigurationConverter` is `com.redhat.training.route.converter`. Camel scans this package and activates the `CommandConfigurationConverter` class in the relevant package.

7. Open the `CommandConfigurationRouteBuilder` class.

Comment out or remove the JSON type conversion.

You do not need this conversion because Camel uses the `CommandConfigurationConverter` for a custom conversion instead of a JSON conversion. The custom conversion happens automatically before sending the data to the `http4` component.

8. Run `./mvnw clean test` to execute the test of the `command-router` application again.

Verify that the test passes.

9. Stop the `air-purifier-v2` application.

Finish

Return to your workspace directory and use the `lab` command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation AD221]$ lab finish pattern-converter
```

This concludes the guided exercise.

Splitting and Aggregating Messages

Objectives

After completing this section, you should be able to use the Splitter pattern to break a message into a series of individual messages, and merge multiple messages by using the Aggregator pattern.

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 to create order items from it and process the order items for shipment.

To implement this pattern, Camel provides the `split` method, which you can use when creating the route. The following example demonstrates the use of 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");
```

The `split` function also supports the 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"); (1)

from("direct:splitUsingHeader").split(header("foo")).to("mock:result"); (2)
```

(1) This example splits the body containing an Iterable Java object (List, Set, Map, etc) into separate exchanges.

(2) This example splits a header containing an Iterable Java object into separate exchanges with the same body. For example, if the exchange header contains a list of users then 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, by using a specified delimiter pattern. You can use the tokenizer expression with the `split` method to split the exchange body by using a token. This allows you to split your text content without the need to first unmarshal it into Java objects.

The following code snippet is an example of a route that uses the tokenizer expression by using the `tokenize` method. The route reads a text-based file and splits the messages per line, and then sends it to a Kafka topic one by one.

```
from("file:messages.txt")
  .split(body().tokenize("\n"))
  .to("kafka:messages");
```

Additionally, if you are splitting XML data, Camel provides the DSL method `tokenizeXML` as an optimized version of the tokenizer.

```
from("file:inbox")
  .split().tokenizeXML("order")
  .to("activemq:queue:order");
```

Addressing Memory Usage Issues in the Splitter Pattern with Streaming

When consuming large pieces of data with Camel, the Splitter pattern is used to divide up this data into smaller, more manageable pieces. But this large amount of data can make the Route consume large amounts of memory. To avoid this problem, Camel offers an option called streaming, which alleviates memory issues by not loading the entire piece of data into memory.

If streaming is enabled then Camel splits the input message into chunks, instead of attempting to load the entire body into memory at once, and then splitting it. This reduces the memory required for each invocation of the route.

When dealing with extremely large payloads, it is recommended that you enable streaming. If data is small enough to fit in memory, streaming is an unnecessary overhead.

You can split streams by enabling the streaming mode using the streaming builder method.

```
from("direct:streaming")
    .split(body().tokenizeXML("order")).streaming()
    .to("activemq:queue.order");
```

If the data you are splitting is in XML format then be sure to use `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.

The Aggregator Pattern

You can group fragmented data from multiple source messages into a single unique message by using the Aggregator pattern. 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. By 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
The correlation value is a field obtained by 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 by using the correlation value.

The complete condition
You must define the complete condition, which uses a predicate or time condition to instruct Camel to validate 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, you must define a completion condition to specify when to send the aggregated exchange. You can define the completion condition by using methods from the Java domain-specific language (DSL). The Java DSL takes place in the subsequent parts of this section.

For example, in the following route, the messages with a matching header field called `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 is always `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 verify 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, Exchange newExchange) { (1) (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 verify if this is the first message processed by this `AggregationStrategy` implementation. The first invocation of the `aggregate` method always has 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 transforms 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. The following are the six most commonly used methods that Camel provides 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 hangs 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, which ever condition is met first triggers 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.

References

For more information, refer to the *Splitter* chapter in the *Apache Camel Development Guide*
https://access.redhat.com/documentation/en-us/red_hat_fuse/7.10/html-single/apache_camel_development_guide/index#MsgRout-Splitte

For more information, refer to the *Aggregator* chapter in the *Apache Camel Development Guide*
https://access.redhat.com/documentation/en-us/red_hat_fuse/7.10/html-single/apache_camel_development_guide/index#MsgRout-Aggregator

Claus Ibsen and Jonathan Anstey. (2018) Camel in Action, Second Edition. Manning. ISBN 978-1-617-29293-4.

Guided Exercise

Splitting and Aggregating Messages

In this exercise, you will split the incoming CSV file to work on each line individually and then batch the operations every 10 lines.

Your department produces text files that contain the details of processed orders. The number of orders that each file contains is too large to be processed by other departments in your company. Each line of an orders file represents an order. You must split the input files by order, and then produce batches of 10 orders, so that other departments can easily process the data.

Outcomes

You should be able to split incoming messages into smaller parts and implement a custom aggregation strategy to aggregate batches of lines every 10 lines.

The solution files for this exercise are in the `AD221-apps` repository, within the `pattern-combine/solutions` directory.

Before You Begin

To perform this exercise, ensure you have the `AD221-apps` repository cloned in your workstation.

From your workspace directory use the `lab` command to start this exercise.

```
[student@workstation AD221]$ lab start pattern-combine
```

The `lab` command copies the exercise files from the `~/AD221/AD221-apps/pattern-combine/apps` directory, into the `~/AD221/pattern-combine` directory.

Instructions

1. Navigate to the `~/AD221/pattern-combine` directory, open the project with your editor of choice, and examine the code.

2. Update the route `configure` method of the `CombineRouteBuilder` class to add the splitting and aggregation code.

    2.1. Add the split step to the route pipeline that uses the system line separator as the tokenizer for the message body:

```
.split( body().tokenize( SEPARATOR ) )
```

    2.2. Add the aggregating step to the route pipeline and create a new `org.apache.camel.processor.aggregate.AggregationStrategy` instance as the parameter:

```
.aggregate( constant(true), new AggregationStrategy() {
    public Exchange aggregate(Exchange oldExchange, Exchange newExchange) {
    }
} )
```

2.3. Inside the `aggregate` method, return the next element in the queue if it is the first one:

```
if (oldExchange == null) {
    return newExchange;
}
```

2.4. Otherwise, concatenate the previous message body with the new one by using the line separator:

```
String oldBody = oldExchange.getIn().getBody( String.class );
String newBody = newExchange.getIn().getBody( String.class );
oldExchange.getIn().setBody( oldBody + SEPARATOR + newBody );
```

2.5. Return the old message with the concatenated content.

```
return oldExchange;
```

3. Accumulate the lines, batching them every 10 lines. After the aggregate method, call the `completionSize` method with a batch size of 10.

```
.completionSize( 10 )
```

4. Test the `CombineRouteBuilder` class. The project contains a test for the route that you can execute to verify that the execution is correct.

```
[student@workstation pattern-combine]$ ./mvnw clean test
...output omitted...
Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
...output omitted...
```

Finish

Return to your workspace directory and use the `lab` command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation AD221]$ lab finish pattern-combine
```

This concludes the guided exercise.

Quiz

Implementing Enterprise Integration Patterns

Consider that you have recently joined a company. The company has an online Software as a Service (SaaS) solution for file conversion. The company's Apache Camel based services have all the conversion and integration logic.

In some integration services, the legacy developer left some unfinished code for you to complete. You must choose the correct answers to the following questions to complete the relevant part of the services:

1. In the file service, you must implement a custom converter that converts any metric type to an `InputStream`. The `Metric` class, which represents the metric type, has a `toString` method that returns the metric values in `String` format. Regarding the following code, which two of the following options are true? (Choose two.)

```
...imports omitted...

// TODO: Annotate the class as converter
public class MetricConverter {

    // TODO: Annotate the method as converter
    public static InputStream toInputStream(Metric metric) {
        // TODO: Add logic to convert the metric to an InputStream
    }
}
```

a. You must annotate the class with `@ClassConverter` and the method with `@Converter`.

b. The return statement must be as follows: `return new ByteArrayInputStream(metric.toString().getBytes());`.

c. The return statement must be as follows: `return metric.toString();`.

d. You must both annotate the class and the method with `@Converter`.

e. The return statement must be as follows: `return new ByteArrayInputStream(metric);`.

2. **In the XML converter service, you must refactor a Camel route that converts XML data to JSON. The legacy developer implemented this by first converting the XML to an object, and converting that object to JSON. The following code shows the Camel route the legacy developer developed before. You must refactor the route to use the `camel-xmljson` library to directly convert from XML to JSON. The legacy developer added all the required dependencies in the pom.xml file. Regarding this code, which of the following options is true for the new implementation? (Choose one.)**

```
...code omitted...

JaxbDataFormat jaxbDataFormat =
    new JaxbDataFormat(JAXBContext.newInstance(CustomObject.class)); //[1]

from("file:xml-in")
.unmarshal(jaxbDataFormat) //[2]
.marshal().json(JsonLibrary.Jackson) //[3]
.to("kafka:json-out");

...code omitted...
```

a. You must replace the code marked as `[1]` with `XmlJsonDataFormat xmlJsonFormat = new XmlJsonDataFormat();`. Also you must remove the codes marked as `[2]` and `[3]` and leave the code like that.

b. You must only remove the code marked as `[2]` and `[3]` and add `.unmarshal()` instead.

c. You must remove the code marked as `[1]` and put `XmlJsonDataFormat xmlJsonFormat = new XmlJsonDataFormat();` instead. Also you must remove the codes marked as `[2]` and `[3]` and add `.unmarshal(xmlJsonFormat)` instead.

d. You must keep the code marked as `[2]` but delete the codes marked as `[1]` and `[3]`.

e. You must only add `.unmarshal(xmlJsonFormat)` after the code marked as `[3]`.

3. In the XML converter service, you have another route that reads big XML files. The route is not complete, because it has to process the data one by one for each `<object>` of the XML file. The route should filter the split XML, and skip all the `<object>` elements that have the `type` attribute empty. Regarding the following code, which two of the following options are true? (Choose two.)

```
...code omitted...

from("file:objects-xml")
 // TODO: Split the XML
 // TODO: Filter the split XML
 .to("activemq:queue:filtered-objects");

...code omitted...
```

a. You must split the XML by using `.split().tokenizeXML("object")`.
b. You must split the XML by using `.split("object")`.
c. You must filter the split XML by using `.filter("/type/text() != ''")`.
d. You must filter the split XML by using `.filter(xpath("/type/text() != ''"))`.
e. You must split and filter the XML by using `.split("object").filter("type!='')`.

4. A text service is responsible for appending the different document contents, especially PDFs, to create merged documents from the separate ones. In the text service, a route performs the content appending by using an `AggregationStrategy` called `AppenderAggregationStrategy`. Regarding the following route that uses the `AppenderAggregationStrategy`, which of the following options is false? (Choose one.)

```
...code omitted...

from("file:source-file")
 .aggregate(header("content"),new AppenderAggregationStrategy())
 .to("file:appended-file");

...code omitted...
```

a. The `AppenderAggregationStrategy` must have an `aggregate` method that has the old exchange and the new exchange as its parameters.

b. You must validate if the old exchange is `null` to prevent any errors, because when the route receives first message, it does not have an old message or an exchange.

c. The `aggregate` method must have an `org.apache.camel.Message` return type.

d. After the concatenation of the exchange bodies, the resulting `String` value must be the new exchange's body. `newExchange.getIn().setBody(newBody);` is a possible implementation of it.

Solution

Implementing Enterprise Integration Patterns

Consider that you have recently joined a company. The company has an online Software as a Service (SaaS) solution for file conversion. The company's Apache Camel based services have all the conversion and integration logic.

In some integration services, the legacy developer left some unfinished code for you to complete. You must choose the correct answers to the following questions to complete the relevant part of the services:

1. In the file service, you must implement a custom converter that converts any metric type to an `InputStream`. The `Metric` class, which represents the metric type, has a `toString` method that returns the metric values in `String` format. Regarding the following code, which two of the following options are true? (Choose two.)

```
...imports omitted...

// TODO: Annotate the class as converter
public class MetricConverter {

    // TODO: Annotate the method as converter
    public static InputStream toInputStream(Metric metric) {
        // TODO: Add logic to convert the metric to an InputStream
    }
}
```

a. You must annotate the class with `@ClassConverter` and the method with `@Converter`.

b. The return statement must be as follows: `return new ByteArrayInputStream(metric.toString().getBytes());`.

c. The return statement must be as follows: `return metric.toString();`.

d. You must both annotate the class and the method with `@Converter`.

e. The return statement must be as follows: `return new ByteArrayInputStream(metric);`.

2. **In the XML converter service, you must refactor a Camel route that converts XML data to JSON. The legacy developer implemented this by first converting the XML to an object, and converting that object to JSON. The following code shows the Camel route the legacy developer developed before. You must refactor the route to use the `camel-xmljson` library to directly convert from XML to JSON. The legacy developer added all the required dependencies in the pom.xml file. Regarding this code, which of the following options is true for the new implementation? (Choose one.)**

```
...code omitted...

JaxbDataFormat jaxbDataFormat =
    new JaxbDataFormat(JAXBContext.newInstance(CustomObject.class)); //[1]

from("file:xml-in")
.unmarshal(jaxbDataFormat) //[2]
.marshal().json(JsonLibrary.Jackson) //[3]
.to("kafka:json-out");

...code omitted...
```

a. You must replace the code marked as `[1]` with `XmlJsonDataFormat xmlJsonFormat = new XmlJsonDataFormat();`. Also you must remove the codes marked as `[2]` and `[3]` and leave the code like that.

b. You must only remove the code marked as `[2]` and `[3]` and add `.unmarshal()` instead.

c. You must remove the code marked as `[1]` and put `XmlJsonDataFormat xmlJsonFormat = new XmlJsonDataFormat();` instead. Also you must remove the codes marked as `[2]` and `[3]` and add `.unmarshal(xmlJsonFormat)` instead.

d. You must keep the code marked as `[2]` but delete the codes marked as `[1]` and `[3]`.

e. You must only add `.unmarshal(xmlJsonFormat)` after the code marked as `[3]`.

3. In the XML converter service, you have another route that reads big XML files. The route is not complete, because it has to process the data one by one for each `<object>` of the XML file. The route should filter the split XML, and skip all the `<object>` elements that have the `type` attribute empty. Regarding the following code, which two of the following options are true? (Choose two.)

```
...code omitted...

from("file:objects-xml")
 // TODO: Split the XML
 // TODO: Filter the split XML
 .to("activemq:queue:filtered-objects");

...code omitted...
```

a. You must split the XML by using `.split().tokenizeXML("object")`.

b. You must split the XML by using `.split("object")`.

c. You must filter the split XML by using `.filter("/type/text() != ''")`.

d. You must filter the split XML by using `.filter(xpath("/type/text() != ''"))`.

e. You must split and filter the XML by using `.split("object").filter("type!='")`.

4. A text service is responsible for appending the different document contents, especially PDFs, to create merged documents from the separate ones. In the text service, a route performs the content appending by using an `AggregationStrategy` called `AppenderAggregationStrategy`. Regarding the following route that uses the `AppenderAggregationStrategy`, which of the following options is false? (Choose one.)

```
...code omitted...

from("file:source-file")
 .aggregate(header("content"),new AppenderAggregationStrategy())
 .to("file:appended-file");

...code omitted...
```

a. The `AppenderAggregationStrategy` must have an `aggregate` method that has the old exchange and the new exchange as its parameters.

b. You must validate if the old exchange is `null` to prevent any errors, because when the route receives first message, it does not have an old message or an exchange.

c. The `aggregate` method must have an `org.apache.camel.Message` return type.

d. After the concatenation of the exchange bodies, the resulting `String` value must be the new exchange's body. `newExchange.getIn().setBody(newBody);` is a possible implementation of it.

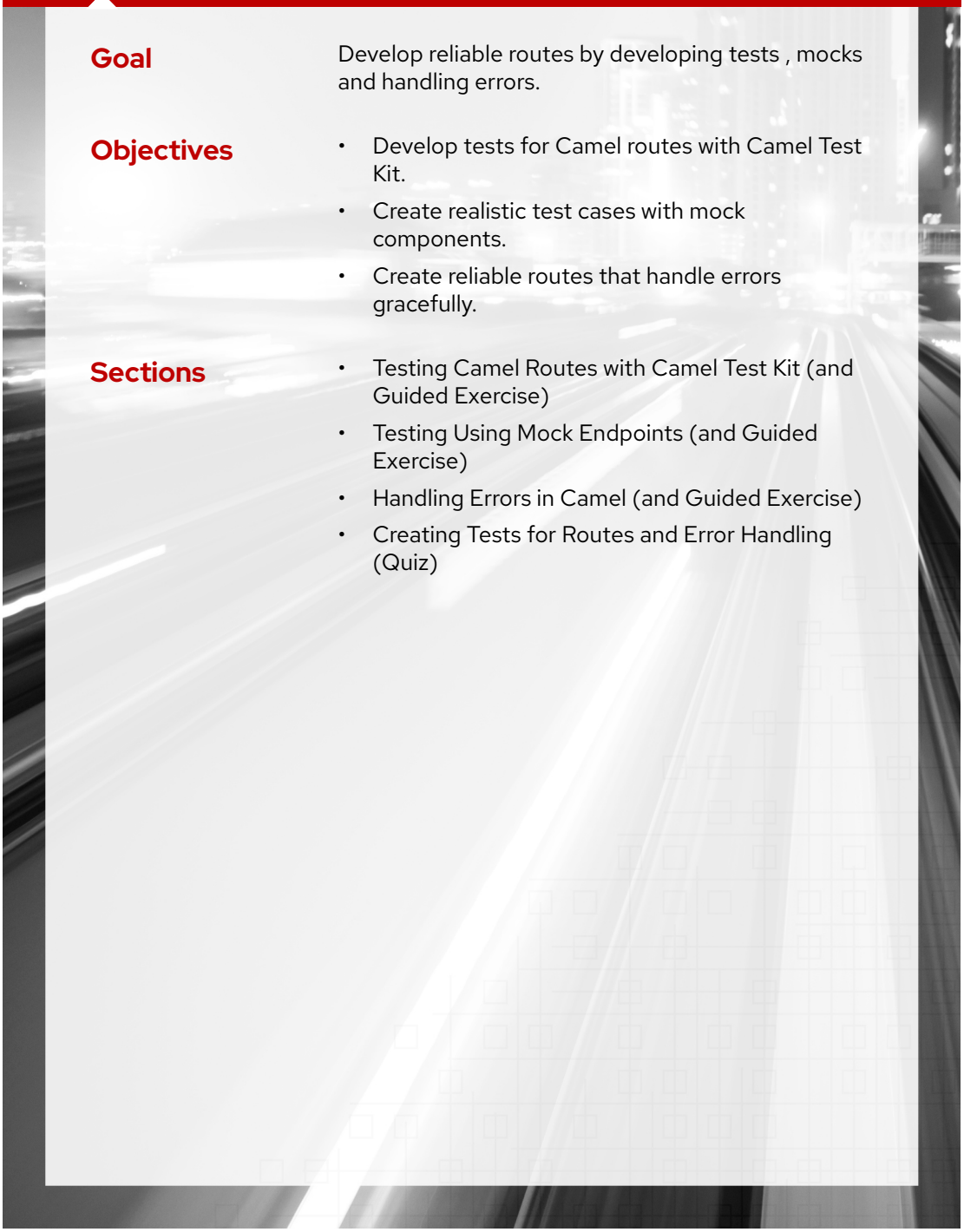
Summary

In this chapter, you learned:

- You can use the JAXB library to transform messages to and from XML.
- You can use the Jackson library to transform messages to and from JSON.
- A message filter selectively removes message exchanges from a route based upon message contents.
- Camel type converters run implicitly to convert data from the source data format to the target data format.
- You can create a custom type converter and register it with Camel's type conversion system.
- Camel supports the Splitter pattern, enabling you to break messages into chunks for separate processing.
- You can use the Aggregator pattern to combine multiple related messages into a single message.

Chapter 4

Creating Tests for Routes and Error Handling



Goal

Develop reliable routes by developing tests , mocks 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 Camel Routes with Camel Test Kit (and Guided Exercise)
- Testing Using Mock Endpoints (and Guided Exercise)
- Handling Errors in Camel (and Guided Exercise)
- Creating Tests for Routes and Error Handling (Quiz)

Testing Camel Routes with Camel Test Kit

Objectives

After completing this section, you should be able to develop tests for Camel routes with Camel Test Kit.

Introduction to the Camel Test Kit

Apache Camel provides a collection of modules and JUnit extension classes known as the Camel Test Kit. This kit is intended to make the testing of Camel routes easier for developers.

Similar to testing any other piece of software, you should distribute your testing efforts by using various types of tests. The required effort to adopt each test type is defined by the testing pyramid. The testing pyramid suggests an agile testing strategy including unit, integration, and end-to-end acceptance tests.

With the Camel Test Kit, you can implement unit and integration tests. Higher-level tests, such as user interface or end-to-end tests require additional testing frameworks, which are not covered in this course.

The following are the available Camel Test Kit modules in Red Hat Fuse 7.10 for Spring Boot.

camel-test
: The main testing module, which provides a number of helpers for writing JUnit tests for Camel routes.

camel-test-spring
: This module wraps `camel-test` to make the Camel testing helpers available in Spring and Spring Boot tests.

Note

Camel 3 replaces the preceding libraries with `camel-test-junit5` and `camel-test-spring-junit5`. These JUnit 5 libraries are not covered in this course, because Red Hat Fuse 7.10 is based on Camel 2.

Configuring Camel Testing in Spring Boot

1. Add the `spring-boot-starter-test` dependency to your POM file. This module gathers a number of testing libraries, including JUnit, making it easier for Spring Boot developers to manage testing dependencies.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
```

2. Add the `camel-test-spring` dependency to your POM file. This module also adds `camel-test` as a transitive dependency.

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

Important

If you do not use the Fuse Spring Boot BOM to control the versions of your dependencies, then you must explicitly declare the `spring-boot-starter-test` and `camel-test-spring` versions.

Implementing Camel Tests in Spring Boot

Consider a Camel route that receives a double type number, and sends a 2-decimal string representation to the file endpoint. You can implement this route in Spring Boot as the following example shows:

```
@Component
public class DoubleNumbersRouteBuilder extends RouteBuilder {

    @Override
    public void configure() throws Exception {
        from( "direct:doubleNumber")
            .process(exchange -> {
                Double number = exchange.getIn().getBody(Double.class);
                exchange.getIn().setBody(String.format("%,.2f", number));
            })
            .to("file:formatted");
    }

}
```

Note that the `@Component` Spring annotation is required for your Spring Boot tests to discover the route.

Next, consider that you want to test that, given a double number, the preceding route produces the correct string representation. You can write a Spring Boot test case as follows:

```
@RunWith( CamelSpringBootRunner.class ) (1)
@SpringBootTest( classes = Application.class ) (2)
public class DoubleNumbersRouteBuilderTest {

    @Autowired
    private ProducerTemplate producerTemplate; (3)

    @Autowired
    private ConsumerTemplate consumerTemplate; (4)
```

```
    @Test
    public void testRouteParsesLatestWarningText() {
        producerTemplate.sendBody("direct:doubleNumber", Math.PI); (5)

        String formatted = consumerTemplate.receiveBody( (6)
            "file:formatted",
            String.class
        );

        assertEquals("3.14", formatted); (7)
    }
}
```

(1) `@CamelSpringBootRunner` enables the Camel test helpers in Spring Boot test cases and automatically handles the Camel context.

(2) `@SpringBootTest` specifies the configuration class to use for the Spring Boot application context. Note that you must also annotate the main configuration class of your application with `@SpringBootApplication`.

(3) `@Autowired` injects the `ProducerTemplate` object. Use this template to send messages to a route endpoint.

(4) `@Autowired` injects the `ConsumerTemplate` object. Use this template to receive messages from a route endpoint.

(5) The producer template sends the Pi number to the `direct:doubleNumber` input endpoint. The `ProducerTemplate` class provides additional methods, other than `sendBody`, to easily send messages to endpoints.

(6) The consumer template receives the resulting message body from the `file:formatted` destination endpoint. The `ConsumerTemplate` class provides additional methods, other than `receiveBody`, to easily receive messages from endpoints.

(7) The test verifies that the resulting message body is a 2-decimal string representation of Pi.

Injecting the Camel Context

Although `CamelSpringBootRunner` is useful for automating the management of Camel context, in some cases a test case might need greater control over the Camel context. To get direct access to the Camel context in your test cases, inject the `CamelContext` dependency with the `@Autowired` Spring annotation.

```
@RunWith( CamelSpringBootRunner.class )
@SpringBootTest( classes = Application.class )
public class MyRouteBuilderTest {

    @Autowired
    private CamelContext context;

    @Test
    public void testRouteParsesLatestWarningText() {
        Collection<Endpoint> endpoints = context.getEndpoints();
```

```
        ...
    }
}
```

Testing Utilities

The `TestSupport` class provides a number of static testing utility methods. For example, if you want to create a directory before each test runs, then you can use the `createDirectory` method.

```
@BeforeEach
public void setUp() {
    TestSupport.createDirectory( "my/testing/dir" );
}
```

Similarly, you can delete a directory after each test, with the `deleteDirectory` method.

```
@AfterEach
public void clean() {
    TestSupport.deleteDirectory( "my/testing/dir" );
}
```

References

The Practical Test Pyramid
https://martinfowler.com/articles/practical-test-pyramid.html

TestSupport Class Javadoc
https://www.javadoc.io/static/org.apache.camel/camel-test/2.23.2/org/apache/camel/test/junit4/TestSupport.html

For more information, refer to the *Testing with Camel Spring Boot* section in the *Red Hat Fuse 7.10 - Deploying into Spring Boot Guide* at
https://access.redhat.com/documentation/en-us/red_hat_fuse/7.10/html-single/deploying_into_spring_boot/index#test-with-camel-spring-boot

Guided Exercise

Testing Camel Routes with Camel Test Kit

In this exercise, you will implement a test case for a route based on the Content Based Router EIP.

Consider a legacy application that exposes its errors and warnings as HTML files. The provided code for this exercise implements a route that parses HTML markup and extracts the text of the most recent error messages. The routing logic is as follows:

- For errors, the route must write the latest error text to the `out/latest-error.txt` file.
- For warnings, the route must write the latest warning text to the `out/latest-warning.txt` file.
- The HTML markup for errors is different from the markup for warnings.

Outcomes

You should be able to implement a test case for a Camel route, by using the Camel Test Kit.

The solution files for this exercise are in the `AD221-apps` repository, within the `test-kit/solutions` directory.

Before You Begin

To perform this exercise, ensure you have the `AD221-apps` repository cloned in your workstation.

From your workspace directory, use the `lab` command to prepare your system for this exercise.

```
[student@workstation AD221]$ lab start test-kit
```

The `lab` command copies the exercise files from the `~/AD221/AD221-apps/test-kit/apps` directory, into the `~/AD221/test-kit` directory.

Instructions

1. Navigate to the `~/AD221/test-kit` directory and open the project with an editor, such as VSCodium.

2. Add the `spring-boot-starter-test` and `camel-test-spring` dependencies to the project. These libraries are required to test Camel Spring Boot applications.

3. Review the Camel route implemented in the `HtmlRouteBuilder` class.

   The route uses the `language` component and XPath to parse the HTML content and select the destination file. Note how the XPath expression for warnings is different from the one for errors.

4. In the `src/main/resources` directory, inspect the `test_errors.html` and `test_warnings.html` files. These files are examples of the HTML markup that the route

should parse, including errors and warnings from different microservices. The tests use these files to verify the route.

5. Review the test cases in the `HtmlRouteBuilderTest` class.

   - `testRouteWritesLatestWarningToFile` verifies that the route creates the `out/latest-warning.txt` file.
   - `testRouteWritesLatestErrorToFile` verifies that the route creates the `out/latest-error.txt` file.
   - `testRouteParsesLatestWarningText` verifies that the route parses the first warning text.
   - `testRouteParsesLatestWarningText` should verify that the route parses the first error text. You must implement this method.

6. Run the tests with `./mvnw clean test` and verify that tests fail.

```
...output omitted...

Tests in error:
testRouteParsesLatestWarningText(...)
testRouteWritesLatestWarningToFile(...)
testRouteWritesLatestErrorToFile(...)

Tests run: 4, Failures: 1, Errors: 3, Skipped: 0
```

Tests do not pass because the test class is missing the annotations required for Spring Boot and Camel Test Kit.

7. Annotate the `HtmlRouteBuilderTest` class to use the Camel Test Kit in Spring Boot.

   7.1. Add the following annotations to the `HtmlRouteBuilderTest` class.

   - `@RunWith( CamelSpringBootRunner.class )`
   - `@SpringBootTest( classes = Application.class )`

   7.2. Rerun the tests. Only the `testRouteParsesLatestErrorText` test case should fail.

```
Failed tests: testRouteParsesLatestErrorText(...)

Tests run: 4, Failures: 1, Errors: 0, Skipped: 0
```

8. Implement the `testRouteParsesLatestErrorText` test method. This test must validate that, given the HTML content of the `test_errors.html` file, the route parses and extracts the text from the first `<article>` HTML tag.

   8.1. Read the contents from the `test_errors.html` file resource.

```
// TODO: read errors file
String errorsHtml = new String( errors.getInputStream().readAllBytes() );
```

8.2. Send the errors HTML as the body to the `direct:parseHtmlErrors` endpoint. Use the `producerTemplate.sendBody` method.

8.3. Read the resulting body from the `file:out` endpoint. Use the `consumerTemplate.receiveBody` method.

8.4. Assert that the resulting body contains the expected fragment of the first `<article>` in the `test_errors.html` file.

```
// TODO: assert body contains a portion of the first article
assertTrue(
    body.contains(
        "Exception occurred during execution on the exchange"
    )
);
```

8.5. Verify that all tests pass.

Finish

Return to your workspace directory and use the `lab` command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation AD221]$ lab finish test-kit
```

This concludes the guided exercise.

Testing Using Mock Endpoints

Objectives

After completing this section, you should be able to create realistic test cases with mock components.

Introduction to the Mock Component

When implementing route tests, especially unit tests for routes, you should replace endpoints with mocks or other test doubles to decouple tests from route dependencies, such as external web services or databases. With the Camel `mock` component, you can replace actual endpoints with fake endpoints and define simulated test scenarios. This is useful to implement tests, but also to develop routes that depend on external systems that are not yet available.

The `mock` endpoint only supports producer endpoints. Use this endpoint by specifying the `mock:` prefix, followed by a name that identifies the mock. The following example shows how to create a mock endpoint called `out`.

```
from("direct:in")
.to("mock:out");
```

Injecting Mock Endpoints in Test Cases

Before defining the expectations and behaviors of a mock, you must inject the corresponding `MockEndpoint` object in your test class. To inject a mock endpoint object into a test class, use the `EndpointInject` annotation, as follows:

```
public class YourRouteBuilderTest {
    @EndpointInject(uri = "mock:out")
    MockEndpoint outEndpoint;

    ...implementation omitted...
}
```

The `uri` parameter of the `@EndpointInject` annotation determines the specific mock endpoint to inject. This URI must match the URI of the endpoint that you want to test.

In this particular example, the `uri` parameter is `mock:out`, which matches the endpoint defined in the preceding route example. Therefore, the `outEndpoint` variable is the `MockEndpoint` instance that defines the behaviors and expectations of the `mock:out` endpoint.

Defining and Asserting Expectations

After injecting the mock endpoint, you can define the expectations of the mock in your test cases. Define mock expectations by using one of the `MockEndpoint#expectXYZ` methods, as demonstrated in the following example:

```
@Test
public void yourTestCase() throws InterruptedException {
    outEndpoint.expectedMessageCount(1);
    outEndpoint.expectedBodiesReceived("Hello Eduardo"); (1)

    template.sendBody("direct:in", "Hello Eduardo"); (2)

    outEndpoint.assertIsSatisfied(); (3)
}
```

(1) Define the mock expectations as preconditions before sending messages to the route.

(2) Send a message to the route under test.

(3) Assert that the mock has met the expectations.

The following methods are some of the most common expectations:

`expectedMessageCount`	Expects a specific number of received messages
`expectedBodiesReceived`	Expects a message with a specific body or bodies
`expectedHeaderReceived`	Expects a message with a specific header
`expectsNoDuplicates`	Expects no duplicated messages

Defining Behavior

You can also tell a mock how to behave to simulate test scenarios. For example, you can configure an HTTP mock endpoint to simulate a specific HTTP response body, and later make your test verify how the route processes this response.

```
@Test
public void yourTestCase() throws InterruptedException {
    httpMockEndpoint.whenAnyExchangeReceived(e -> {
        e.getOut().setBody("Hello, Randy!");
    });

    template.sendBody("direct:start", null);

    ...assertions omitted...
}
```

With the `whenAnyExchangeReceived` function, you can define the preprogrammed behavior and responses required to set up the test scenario.

The `whenAnyExchangeReceived` method receives a `Proccessor` as a parameter, which allows you to manipulate the exchange object.

Note

The `mock` component exposes numerous methods to define expectations and behaviors, other than the ones covered in this lecture. Refer to the `mock` component documentation for a complete list of methods.

Replacing Endpoints with Mocks

Manually replacing real endpoints with mocks in your code each time you run a test is tedious and prone to errors. You should be able to test any route without including any external system in the test, even if your route code points to production systems.

Camel supports the replacement of endpoints in routes with the `adviceWith` feature. With this feature, you can *advise* the route to intercept messages and substitute endpoints with mock endpoints, before starting the Camel context.

The following example illustrates the use of `adviceWith`.

```
@RunWith( CamelSpringBootRunner.class )
@SpringBootTest
@UseAdviceWith (1)
public class YourTest {
    @EndpointInject(uri = "mock:file:customer_requests")
    MockEndpoint fileMock;

    ...other injections omitted...

    @Before
    public void setUp() throws Exception {
        context
            .getRouteDefinition("myRoute")
            .adviceWith(context, new AdviceWithRouteBuilder() { (2)
                @Override
                public void configure() {
                    replaceFromWith("direct:origin); (3)

                    interceptSendToEndpoint("file:.*customer_requests.*") (4)
                            .skipSendToOriginalEndpoint() (5)
                            .to("mock:file:customer_requests"); (6)
                }
            });

        context.start(); (7)
    }

    @After
    public void tearDown() throws Exception {
        context.stop(); (8)
    }

    ...test cases omitted...
}
```

1. The `@UseAdviceWith` annotation marks the use of `adviceWith` in the test class. This annotation deactivates the automatic Camel context start/stop feature. Deactivating the context autostart is a prerequisite, because Camel needs to know route advice before starting the context.
2. `adviceWith` requires an `AdviceWithRouteBuilder` object. You must override the `configure` method of this class to advise the route.
3. `replaceFromWith` replaces the `from` endpoint with another component.
4. `interceptSendToEndpoint` intercepts messages sent to endpoints that match the pattern.
5. `skipSendToOriginalEndpoint` skips the original endpoint and just sends messages to the mock.
6. The mock endpoint where Camel should send intercepted messages. Note that this endpoint must match the URI of the `MockEndpoint` object injected in the test class.
7. After advising the routes, start the context.
8. After running your tests, stop the context.

Automocking Endpoints

Instead of using `interceptSendToEndpoint`, you can use the simpler `mockEndpointsAndSkip` call to easily replace endpoints with mocks. For example:

```
context
    .getRouteDefinition( "myRoute" )
    .adviceWith( context, new AdviceWithRouteBuilder() {
        @Override
        public void configure() {
            replaceFromWith( "direct:start" );

            mockEndpointsAndSkip("file:.*customer_requests.*");
        }
    } );
```

The `mockEndpointsAndSkip` method replaces all the endpoints that match the given pattern by following these steps:

1. Adds the `mock:` prefix to the original endpoint.
2. Replaces the first `://` occurrence with `:`.
3. Strips off additional endpoint parameters.

For example, given the `http://localhost:8888/users?sort=name` URI, `mockEndpointsAndSkip` replaces the http endpoint with the `mock:http:localhost:8888/users` endpoint.

Automocking in Spring Boot

If you use the `camel-test-spring` package, then you can use the `@MockEndpointsAndSkip` annotation to replace all the endpoints matching a pattern. The result is the same as what you get by advising the route with the `mockEndpointsAndSkip` method.

```
@MockEndpointsAndSkip("file:.*customer_requests.*")
public class YourTest {
    ...implementation omitted...
```

By using `@MockEndpointsAndSkip`, you can quickly mock endpoints without having to implement `adviceWith` code or manually starting and stopping the Camel context.

Using Property Placeholders

Property placeholders are another useful tool to decouple your routes from specific endpoints. Instead of hard coding endpoints in your route, use property placeholders, as shown in the following example:

```
from("{{myroute.queue}}")
    .to("{{myroute.api}}")
```

Next, you can use your `application.properties` file to define values for these properties. Additionally, you can define specific property values for your tests, such as mock or direct endpoints, as follows:

```
@SpringBootTest(properties = {
    "myroute.queue=direct:start",
    "myroute.api=mock:api"
})
public class YourTest {
    ...implementation omitted...
}
```

References

For more information, refer to the *Mock Component* chapter in the *Red Hat Fuse 7.10 Apache Camel Component Reference* at https://access.redhat.com/documentation/en-us/red_hat_fuse/7.10/html-single/apache_camel_component_reference/index#mock-component

For more information, refer to the *Property Placeholders* section in the *Red Hat Fuse 7.10 Apache Camel Development Guide* at https://access.redhat.com/documentation/en-us/red_hat_fuse/7.10/html/apache_camel_development_guide/basicprinciples#BasicPrinciples-PropPlaceholders

Guided Exercise

Testing Using Mock Endpoints

In this exercise, you will use mocks to test a Camel route in isolation.

Assume you have developed a route that reads data from an HTTP endpoint and writes the result to a file. You must test this route in isolation from the external HTTP service and the file system.

Outcomes

You should be able to use mocks to test routes in isolation and decoupled from external systems.

The solution files for this exercise are in the `AD221-apps` repository, within the `test-mock/solutions` directory.

Before You Begin

To perform this exercise, ensure you have the `AD221-apps` repository cloned in your workstation.

From your workspace directory, use the `lab` command to prepare your system for this exercise.

```
[student@workstation AD221]$ lab start test-mock
```

The `lab` command copies the exercise files from the `~/AD221/AD221-apps/test-kit/apps` directory, into the `~/AD221/test-mock` directory.

Instructions

1. Navigate to the `~/AD221/test-mock` directory and open the project with an editor, such as VSCodium.

2. Open the `HttpRouteBuilder` class and inspect the route code.

```
from("direct:start") (1)
    .to("http4://my-external-service/greeting") (2)
    .to("file:out?fileName=response.txt"); (3)
}
```

(1) The `direct` component invokes the route.

(2) The `http4` component calls the `http://my-external-service/greeting` endpoint and reads the response.

(3) The `file` component writes the response body to the `out/response.txt` endpoint.

3. Open the `HttpRouteBuilderTest` class and configure the test to have access to the mock endpoint instances required for testing the route.

   Use the `@EndpointInject` annotation to inject the mock endpoints. You must inject a mock for the HTTP endpoint and a mock for the file endpoint. The mock URIs should be as follows:

   - HTTP endpoint: `mock:http4:my-external-service/greeting`.
   - File endpoint: `mock:file:out`.

4. In the same class, complete the `testFileRecievesContentFromHttpClient` test case by setting the behavior and expectations of the mocks.

   4.1. Configure the HTTP mock endpoint to return the `Hello test!` response body.

```
// TODO: add httpMockEndpoint behaviour
httpMockEndpoint.whenAnyExchangeReceived(e -> {
    e.getOut().setBody("Hello test!");
});
```

   4.2. Configure the `fileMockEndpoint` mock to expect one message.

   4.3. Configure the `fileMockEndpoint` mock to expect the `Hello test!` message body.

   4.4. Assert that `fileMockEndpoint` satisfies the expectations.

5. Run `./mvnw test`. The test fails with an `UnknownHostException` error because the route is still using the `http://my-external-service/greeting` endpoint instead of `mock:http:my-external-service/greeting`.

6. Enable the automock feature to automatically advise the route endpoints and replace them with mock endpoints.

   6.1. Annotate the `HttpRouteBuilderTest` class with `@MockEndpointsAndSkip("http.*|file:out.*")`.

   6.2. Run tests and verify that tests pass.

   6.3. Review the test logs. Verify that the logs show that the original endpoints have been advised with mock endpoints

```
Adviced endpoint [http4://my-external-service/greeting] with mock endpoint
 [mock:http4:my-external-service/greeting]
Adviced endpoint [file://out?fileName=response.txt] with mock endpoint
 [mock:file:out]
```

7. Use property placeholders to decouple the route from specific URIs and components.

   7.1. In the `HttpRouteBuilder` class, replace the `from` and HTTP endpoint URIs with property placeholders.

```
from("{{http_route.start}}")
    .to("{{http_route.server}}/greeting")
    .to("file:out?fileName=response.txt");
```

7.2. Open the `HttpRouteBuilderTest` class and assign the property values specific to the test.

```
@SpringBootTest(properties = {
    "http_route.start=direct:start",
    "http_route.server=http4://test-fake"
})
public class PropertiesTest {
  ...
```

7.3. In the same class, update the URI of the injected HTTP mock to use the correct URI. Remember that `@MockEndpointsAndSkip` replaces `http4://test-fake/greeting` with `mock:http4:test-fake/greeting`.

7.4. Run tests to verify that they pass.

Finish

Return to your workspace directory, and use the `lab` command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation AD221]$ lab finish test-mock
```

This concludes the guided exercise.

Handling Errors in Camel

Objectives

After completing this section, you should be able to create reliable routes that handle errors gracefully.

Understanding Errors in Camel

Many different errors can occur when messages flow through a Camel route, such as network outages or incorrect file system permissions. Thankfully, Camel implements error handling mechanisms to help you to capture and handle these errors.

There are two categories of errors in a Camel route:

Recoverable errors
: Errors caused by transient problems that can be solved by retrying the delivery, such as a connection timeout. Normally, these errors generate a Java exception, which Camel attaches to the `Exchange` object transmitted by a route.

Irrecoverable errors
: Failures without an immediate solution, such as file system failure. These errors set a fault message in the `Exchange` object of a route. By default, Camel does not handle these errors.

A channel is a Camel piece that routes messages from one route step to the next one. If an error occurs in a step, then the channel that precedes the step captures the error and passes the error to the error handler.

Customizing Camel Error Handling

If you do not explicitly set an error handler, then Camel applies a default error handler strategy. To change the default behavior, you can use the `errorHandler` method at the context and route levels, as shown in the following example:

```
public class MyRoute extends RouteBuilder{
  public void configure() throws Exception{
    errorHandler(loggingErrorHandler()); (1)

    from("file:inputDir") (2)
      .routeId("first")
      .to(...)

    from("file:anotherDir")
      .routeId("second")
      .errorHandler(defaultErrorHandler()) (3)
      .to(...)
  }
}
```

(1) The Camel context uses the `LoggingErrorHandler` class as the error handler.

2. The first route inherits `LoggingErrorHandler` from the context as the error handler.

3. The second route uses the `DefaultErrorHandler` class as the error handler.

Camel Error Handlers

When configuring error handlers, you can use any of the four error handlers available in Camel:

DefaultErrorHandler
The default error handling strategy if none is explicitly set. This handler 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).

Trapping Exceptions by Exception Type

Camel error handlers represent a standard approach to handle every error thrown in Camel routes. Unfortunately, this approach does not allow custom error handling based on specific exception types. For example, you might want to redeliver a message in the event of a network error, but not when a business rule error occurs.

Camel implements a type-specific exception trapping policy with the `onException` method. With this method, you can trap exceptions by type, and customize the exchange, routing, and redelivery policies. The following example illustrates the use of the `onException` method.

```
public class JavaRouteBuilder extends RouteBuilder {

  @Override
  public void configure() throws Exception {
    onException(LogException.class) 1
      .to("file:log") 2
      .handled(true) 3
      .maximumRedeliveries(3); 4

    from("file:in")
    ...
  }

}
```

1. Traps all `LogException` errors raised in this context.

2. Sends the message to another endpoint.

3. Sets the exception as handled. Keep in mind that, by default, the `onException` clause does not mark exceptions as handled.

4. Attempts message redelivery three times at most.

The `onException` call returns an instance of the `OnExceptionDefinitionClass` class. This class implements a rich API to trap and handle errors, apart from the methods shown in the preceding example. For more details about the available methods, refer to the `onException` documentation.

Note

Notice the difference between **trapping** an exception and **catching** an exception.

- Catching implies handling errors raised by a specific code fragment, as you would do with a regular Java try/catch block.
- Trapping implies handling errors raised at any point of the context or the route. With the `onException` method, you trap exceptions.

Catching Exceptions with the Try/Catch DSL

If you need to control errors in specific parts of a route, such as a processor step, then trapping exceptions is not suitable. In this case, you must use the Camel Try/Catch DSL.

In Camel routes, you cannot use the ordinary try/catch/finally Java mechanism. This is because Camel executes the `configure` method of your route builder only once, when registering the route definition in the context. Instead, you must use the Camel `doTry/doCatch/doFinally` syntax, as follows:

```
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())
      .doFinally()
        .to("mock:finally")
      .end();
      ...
  }
}
```

As the preceding example shows, you can use the `doTry/doCatch/doFinally` Camel mechanism in a similar way to regular Java try/catch/finally blocks. This syntax presents a few differences, when compared to `onException`:

- The `doTry/doCatch/doFinally` syntax handles exceptions. The `onException` clause does not mark exceptions as handled by default.
- The `doTry/doCatch/doFinally` syntax is restricted to the route where you use it. In contrast, the `onException` clause, applies to the context and route levels.

References

Apache Camel Manual - Exception Clause
https://camel.apache.org/manual/exception-clause.html

Apache Camel Manual - Try...Catch...Finally
https://camel.apache.org/manual/try-catch-finally.html

For more information, refer to the *Exception Handling* section in the *Red Hat Fuse 7.10 Apache Camel Development Guide* at
https://access.redhat.com/documentation/en-us/red_hat_fuse/7.10/html-single/apache_camel_development_guide/index#BasicPrinciples-ExceptionHandling

Guided Exercise

Handling Errors in Camel

In this exercise, you will improve an application created for the payroll department of your company.

The company wants to integrate the internal financial system with an external service that generates employee payslips. To validate the generated payslips, you will add error handling to the application, and flag the payslips that contain errors.

Outcomes

You should be able to provide exception management capabilities by using Camel's `doTry/doCatch` blocks, error handlers, and `onException` mechanisms.

The solution files for this exercise are in the `AD221-apps` repository, within the `test-error/solutions` directory.

Before You Begin

To perform this exercise, ensure you have the `AD221-apps` repository cloned in your workstation.

From your workspace directory, use the `lab` command to start this exercise.

```
[student@workstation AD221]$ lab start test-error
```

The `lab` command copies the exercise files from the `~/AD221/AD221-apps/test-error/apps` directory, into the `~/AD221/test-error` directory.

Instructions

1. Navigate to the `~/AD221/test-error` directory, open the project with your editor of choice, and examine the code.

2. Execute the `./mvnw -DskipTests clean package spring-boot:run` command to start the Spring Boot application.

   Wait for the application to process the messages stored in the `data/payslips` directory, and evaluate the output log.

   Stop the route execution after verifying that the application log displays exceptions and warnings for the files: `payslip-1.xml`, `payslip-2.xml`, and `payslip-6.xml`.

```
java.lang.NumberFormatException: For input string: "NA"
    ...output omitted...
    at java.base/java.util.concurrent...
    at java.base/java.lang.Thread.run(Thread.java:832) ~[na:na]
```

```
2022-03-01 10:21:18.287  WARN 27036 --- [//data/payslips]
 o.a.c.c.file.GenericFileOnCompletion: Rollback file strategy:
 org.apache.camel.component.file.strategy.GenericFileRenameProcessStrategy@101f3b6
 for file: GenericFile[payslip-1.xml]

...output omitted...

2022-03-01 10:21:17.783  WARN 27036 --- [//data/payslips]
 o.a.c.c.file.GenericFileOnCompletion: Rollback file strategy:
 org.apache.camel.component.file.strategy.GenericFileRenameProcessStrategy@101f3b6
 for file: GenericFile[payslip-2.xml]

...output omitted...

2022-03-01 10:21:18.294  WARN 27036 --- [//data/payslips]
 o.a.c.c.file.GenericFileOnCompletion: Rollback file strategy:
 org.apache.camel.component.file.strategy.GenericFileRenameProcessStrategy@101f3b6
 for file: GenericFile[payslip-6.xml]
```

3. Open the `PayslipValidationRouteBuilder` class. Add a `doTry/doCatch` block to the route with ID `amount-process`. Capture the `NumberFormatException` exception, and send the messages with invalid values to the `file://data/validation/error-amount` endpoint.

4. Verify the correctness of the changes made to the route by executing the unit tests. Run the `./mvnw clean -Dtest=AmountProcessRouteTest test` command, and verify that two unit tests pass.

5. Handle the `NumberFormatException` exception raised from the `PriceProcessor` processor by using the `onException` clause. Capture the exception, and send the messages with invalid values to the `file://data/validation/error-price` endpoint.

6. Implement an error handler that uses the Dead Letter EIP. Capture any exception raised, and send the messages to the `file://data/validation/error-dead-letter` endpoint. You must disable the redelivery policy.

7. Verify the correctness of the changes made to the route by executing the unit tests. Run the `./mvnw clean test` command, and verify that five unit tests pass.

8. Execute the `./mvnw clean package spring-boot:run` command to start the Spring Boot application.

9. Wait for the application to process the payslips stored in the `file://data/payslips` endpoint, and manually verify the correctness of the application logic:

   - The `data/validation/correct` directory contains the files: `payslip-3.xml`, `payslip-4.xml`, and `payslip-5.xml`.
   - The `data/validation/error-amount` directory contains the file `payslip-1.xml`.
   - The `data/validation/error-dead-letter` directory contains the file `payslip-6.xml`.
   - The `data/validation/error-price` directory contains the file `payslip-2.xml`.

Finish

Stop the Spring Boot application, return to your workspace directory, and use the `lab` command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation AD221]$ lab finish test-error
```

This concludes the guided exercise.

Quiz

Creating Tests for Routes and Error Handling

Consider a Camel route that retrieves weather forecasts from an API and sends the data to ActiveMQ. The route parses the API response by converting this raw response into a format supported by your application. Finally, the route sends the processed forecast to ActiveMQ, which delivers the forecast to other parts of your application.

Choose the correct answers to the following questions:

1. You must develop your first test case for the Camel route in Spring Boot. Which two of the following annotations are required for the test class? (Choose two.)

a. `@Autowired`
b. `@RunWith( CamelSpringBootRunner.class )`
c. `@Test`
d. `@BeforeEach`
e. `@SpringBootTest( classes = Application.class )`

2. Assuming that the route code is as follows, what statement should you use in your test to trigger the execution of the route?

```
from("direct:start")
  .to("http://api.example.com/weather/forecast")
  .process(new WeatherApiResponseProcessor())
  .to("activemq:weatherForecasts");
```

a. Use the Java File API to send test data to the `direct:start` endpoint.
b. Inject a `ProducerTemplate` instance in the test class, and send a message body to the `activemq:weatherForecasts` endpoint.
c. Inject a `ProducerTemplate` instance in the test class, and send a message body to the `direct:start` endpoint.
d. Inject a `ProducerTemplate` instance in the test class, and send a message body to the `file:direct:start` endpoint.

3. **Consider you have replaced the `activemq:weatherForecasts` with the `mock:activemq:weatherForecasts` endpoint in your test class. You have also replaced the `http://api.example.com/weather/forecast` endpoint with the `mock:http:api.example.com/weather/forecast` endpoint. Which two expectations and behaviors do you need to set in these mocks to validate that messages received from the http endpoint are received in the ActiveMQ endpoint? (Choose two.)**

```
class YourTest {

    @EndpointInject(uri = "mock:http:api.example.com/weather/forecast")
    MockEndpoint mockHttp;

    @EndpointInject(uri = "mock:activemq:weatherForecasts")
    MockEndpoint mockActiveMQ;

    @Test
    public void test...() {
        // Expectations/Behaviours definition

        template.sendBody("direct:start", null);

        mockActiveMQ.assertIsSatisfied();
    }
}
```

a. Use the `mockHttp.whenAnyExchangeReceived` method to simulate a response from the HTTP mock endpoint.
b. Use the `mockActiveMQ.expectedBodiesReceived` method to verify that the route sends the processed HTTP response to the ActiveMQ mock endpoint.
c. Use the `mockActiveMQ.whenAnyExchangeReceived` method to simulate a response from the ActiveMQ mock endpoint.
d. Use the `mockHttp.expectedBodiesReceived` method to verify that the route sends the processed HTTP response to the HTTP mock endpoint.

4. **The HTTP endpoint sometimes returns invalid message bodies. You observe that these messages generate `InvalidResponseException` errors in your route. You want to handle these exceptions and send invalid messages to a file for subsequent processing. How should you handle this?**

   a. Use `onException(InvalidResponseException.class).to("file:invalid-responses")`

   b. Use `onException(InvalidResponseException.class).to("file:invalid-responses").handled(true)`

   c. Use `errorHandler(loggingErrorHandler())`

   d. Use `errorHandler(defaultErrorHandler())`

Solution

Creating Tests for Routes and Error Handling

Consider a Camel route that retrieves weather forecasts from an API and sends the data to ActiveMQ. The route parses the API response by converting this raw response into a format supported by your application. Finally, the route sends the processed forecast to ActiveMQ, which delivers the forecast to other parts of your application.

Choose the correct answers to the following questions:

1. You must develop your first test case for the Camel route in Spring Boot. Which two of the following annotations are required for the test class? (Choose two.)

a. `@Autowired`

b. `@RunWith( CamelSpringBootRunner.class )`

c. `@Test`

d. `@BeforeEach`

e. `@SpringBootTest( classes = Application.class )`

2. Assuming that the route code is as follows, what statement should you use in your test to trigger the execution of the route?

```
from("direct:start")
  .to("http://api.example.com/weather/forecast")
  .process(new WeatherApiResponseProcessor())
  .to("activemq:weatherForecasts");
```

a. Use the Java File API to send test data to the `direct:start` endpoint.

b. Inject a `ProducerTemplate` instance in the test class, and send a message body to the `activemq:weatherForecasts` endpoint.

c. Inject a `ProducerTemplate` instance in the test class, and send a message body to the `direct:start` endpoint.

d. Inject a `ProducerTemplate` instance in the test class, and send a message body to the `file:direct:start` endpoint.

3. **Consider you have replaced the `activemq:weatherForecasts` with the `mock:activemq:weatherForecasts` endpoint in your test class. You have also replaced the `http://api.example.com/weather/forecast` endpoint with the `mock:http:api.example.com/weather/forecast` endpoint. Which two expectations and behaviors do you need to set in these mocks to validate that messages received from the http endpoint are received in the ActiveMQ endpoint? (Choose two.)**

```
class YourTest {

    @EndpointInject(uri = "mock:http:api.example.com/weather/forecast")
    MockEndpoint mockHttp;

    @EndpointInject(uri = "mock:activemq:weatherForecasts")
    MockEndpoint mockActiveMQ;

    @Test
    public void test...() {
        // Expectations/Behaviours definition

        template.sendBody("direct:start", null);

        mockActiveMQ.assertIsSatisfied();
    }
}
```

a. Use the `mockHttp.whenAnyExchangeReceived` method to simulate a response from the HTTP mock endpoint.
b. Use the `mockActiveMQ.expectedBodiesReceived` method to verify that the route sends the processed HTTP response to the ActiveMQ mock endpoint.
c. Use the `mockActiveMQ.whenAnyExchangeReceived` method to simulate a response from the ActiveMQ mock endpoint.
d. Use the `mockHttp.expectedBodiesReceived` method to verify that the route sends the processed HTTP response to the HTTP mock endpoint.

4. The HTTP endpoint sometimes returns invalid message bodies. You observe that these messages generate `InvalidResponseException` errors in your route. You want to handle these exceptions and send invalid messages to a file for subsequent processing. How should you handle this?

a. Use `onException(InvalidResponseException.class).to("file:invalid-responses")`

b. Use `onException(InvalidResponseException.class).to("file:invalid-responses").handled(true)`

c. Use `errorHandler(loggingErrorHandler())`

d. Use `errorHandler(defaultErrorHandler())`

Summary

In this chapter, you learned:

- Apache Camel provides a collection of modules and JUnit extension classes known as the Camel Test Kit. With this kit, you can implement unit and integration tests of Camel routes.
- When implementing route tests, you should replace endpoints with mocks or other test doubles to decouple tests from route dependencies.
- Apache Camel implements error handling mechanisms such as `errorHandler`, `onException`, and `doTry/doCatch` to capture and handle errors.

Chapter 5

Integrating Services using Asynchronous Messaging

Goal

Integrate services using Apache Kafka and JMS.

Objectives

- Create routes that use the JMS and AMQP components to receive and send asynchronous messages.
- Create routes that use the Kafka component to send and receive durable asynchronous messages.

Sections

- Integrating Services Using JMS (and Guided Exercise)
- Integrating Camel with Kafka (and Guided Exercise)
- Integrating Services Using Asynchronous Messaging (Quiz)

Integrating Services Using JMS

Objectives

After completing this section, you should be able to create routes that use the JMS and AMQP components to receive and send asynchronous messages.

Java Messaging Service (JMS) on Red Hat Fuse

Java Message Service (JMS) is an API that allows JVM-based application components to create, send, receive, and read messages.

JMS acts as a generic wrapper layer that is used to access many different kinds of messaging systems, and supports the JMS API. For example, ActiveMQ, MQSeries, Tibco, Sonic, and more all implement JMS. Red Hat Fuse uses a JMS component to send and receive messages to any of these messaging systems.

JMS uses queues or topics for message channels. The syntax of the endpoint URI, for the JMS component, uses the following format:

```
jms:[queue:|topic:]destinationName[?options]
```

For example, the following Java DSL receives order objects from a queue called `jms_order_input`, transforms the message body to JSON, and sends the JSON formatted order to a queue called `json_order_input`.

```
from("jms:queue:jms_order_input")
  .routeId("ROUTE_NAME")
  .marshal().json(JsonLibrary.Jackson)
  .to("jms:queue:json_order_input");
```

The `camel-jms` library provides the JMS component. Spring Boot users use the `camel-jms-starter` and can configure options by specifying `camel.component.jms.*` properties in the `application.properties` file.

For example, to consume messages concurrently in multiple threads, add an entry like the following:

```
camel.component.jms.concurrent-consumers = 20
```

There are more than 80 configuration options for the JMS component. See the component documentation for a complete list of component options.

JMS Connection Factories

The JMS component requires a connection factory object to make a client connection to a given messaging system. The following example shows how to provide a connection factory to a Red Hat AMQ instance.

```
...imports omitted...

@Configuration (1)
public class MessagingConnectionFactory {

 @Bean
 public JmsComponent jmsComponent() throws JMSException { (2)
   // Creates the connectionfactory that connects to Artemis
   ActiveMQConnectionFactory connectionFactory = new
 ActiveMQConnectionFactory(); (3)
   connectionFactory.setBrokerURL("tcp://localhost:61616");
   connectionFactory.setUser("admin");
   connectionFactory.setPassword("admin");

   // Creates the Camel JMS component and wires it to the Artemis
 connectionfactory
   JmsComponent jms = new JmsComponent(); (4)
   jms.setConnectionFactory(connectionFactory);

   return jms;
 }
}
```

(1) The `Configuration` annotation for defining Spring framework beans. This class can contain one or more bean methods that you must annotate by using `@Bean`.

(2) Spring framework takes the name of the method as the bean ID. In this case the bean's name is `jmsComponent`.

(3) The connection factory `ActiveMQConnectionFactory`, defined for the JMS component.

(4) Initialization of the `JmsComponent`. The method makes the `JmsComponent` available as a bean.

To use a defined JMS component with a connection factory configured, you must add the `jmsComponent` bean in the route endpoint as follows:

```
from("jmsComponent:queue:jms_order_input")
  .routeId("ROUTE_NAME")
  .marshal().json(JsonLibrary.Jackson)
  .to("jmsComponent:queue:json_order_input");
```

To integrate multiple brokers with Camel, you must define multiple JMS component beans. For example, if you want to use two different brokers in a Camel context, you must define two JMS component beans. This is because you must configure different connection factories for each broker.

For the `org.apache.activemq.artemis.jms.client.ActiveMQConnectionFactory` class, the application must have the following Maven dependency:

```
<dependency>
    <groupId>org.apache.activemq</groupId>
    <artifactId>artemis-jms-client</artifactId>
</dependency>
```

Advanced Message Queuing Protocol (AMQP) on Red Hat Fuse

AMQP is an open standard application layer protocol for delivering messages. The main difference between JMS and AMQP is that JMS is a Jakarta EE compliant API, whereas AMQP is a wire-level messaging protocol that is platform independent.

Camel implements the AMQP component by inheriting the JMS component. The AMQP component provides access to messaging providers that require the AMQP protocol. The component supports the AMQP 1.0 protocol by using the JMS Client API of the Qpid project.

The `camel-amqp` library provides the AMQP component. The URI syntax for the AMQP component is identical to the syntax for the JMS component and supports all of the options of the JMS component.

```
amqp:[queue:|topic:]destinationName[?options]
```

To configure options in the `application.properties` configuration file, `amqp` properties use the `camel.component.amqp.*` prefix instead of `camel.component.jms.*`.

See the component documentation for details on configuration options for the AMQP component.

References

Apache Camel Component Documentation
https://camel.apache.org/components/2.x/index.html

For more information, refer to the *JMS Component* chapter in the *Apache Camel Component Reference* at
https://access.redhat.com/documentation/en-us/red_hat_fuse/7.10/html-single/apache_camel_component_reference/index#jms-component

For more information, refer to the *AMQP Component* chapter in the *Apache Camel Component Reference* at
https://access.redhat.com/documentation/en-us/red_hat_fuse/7.10/html-single/apache_camel_component_reference/index#amqp-component

Guided Exercise

Integrating Services Using JMS

In this exercise, you work for Acme Inc, and you need to integrate with two messaging systems to process orders. An upstream fast fulfillment system processes and delivers orders for our most popular products. You must send these orders, in JSON format to the order logging route for audit purposes. Otherwise, you will pass non-delivered orders to a fulfillment system that only receives JSON formatted messages via AMQP.

The first system allows integration via the JMS protocol, while the other system uses the AMQP specification. In both cases you will use Red Hat Fuse to simplify the integration via the messaging components.

Outcomes

In this exercise you should be able to:

- Receive and send messages using the `JMS` component
- Configure the required `JMS` connection factory
- Receive and send messages using an `AMQP` component

The solution files for this exercise are in the `AD221-apps` repository, within the `async-jms/solutions` directory.

Before You Begin

To perform this exercise, ensure you have the `AD221-apps` repository cloned in your workstation.

From your workspace directory, use the `lab` command to start this exercise.

```
[student@workstation AD221]$ lab start async-jms
```

The `lab` command copies the exercise files from the `~/AD221/AD221-apps/async-jms/apps` directory, into the `~/AD221/async-jms` directory. The `Lab` command also creates a Red Hat AMQ instance.

You can inspect the logs for the AMQ instance at any time with the following command:

```
[student@workstation AD221]$ podman logs artemis
```

Instructions

1. Navigate to the `~/AD221/async-jms` directory and open the project with an editor, such as VSCodium.

2. Open the project's POM file, and add the required dependencies.

Dependencies to Add

Purpose	Group ID	Artifact ID
JMS dependency	org.apache.activemq	artemis-jms-client
JMS dependency	org.apache.camel	camel-jms-starter
AMQP dependency	org.apache.camel	camel-amqp

3. Open the `JmsRouteBuilder` class, and add a route to receive orders from the `jms:queue:jms_order_input` endpoint. Marshall the message body into a JSON format.

   Send the message to the `direct:log_orders` endpoint if the delivered field is set to true. Otherwise, forward the order to the `amqp:queue:amqp_order_input` endpoint. Set the ID of the route to `jms-order-input` and use the log method to track the progress of messages through the route.

Note

You can use the following JSON path expression to filter orders by the delivered field: `$[?(@.Delivered == false)]`

4. The `jms` component requires a connection factory to acquire a connection to the messaging broker instance. Add the connection factory to the `MessagingConnectionFactory` class. The connection factory type is `ActiveMQConnectionFactory`. Set the following additional properties on the connection factory object to connect to the AMQ Broker instance:

Connection Factory Properties

BrokerURL	User	Password
tcp://localhost:61616	admin	admin

5. Open the `AMQPRouteBuilder` class, and add a route to receive orders from the `amqp:queue:amqp_order_input` endpoint. Send the messages to the `direct:log_orders` endpoint. You can use the log method to track the progress of the route, and set `amqp-order-input` as the ID of the route.

   Red Hat AMQ supports the AMQP 1.0 specification. The AMQ broker, used in this lab, accepts AMQP connections on port 61616. Thus, the `amqp` component can use the connection factory provided in the previous step.

6. Open the `OrderLogRouteBuilder` class and observe the steps of this route.

7. Use the `./mvnw clean test` command to run the unit tests. The application has a unit test for each route.

```
Results :

Tests run: 3, Failures: 0, Errors: 0, Skipped: 0
```

8. Build and run the application. Inspect the logs, and verify that the logs display received orders.

```
[student@workstation async-jms]$ ./mvnw clean spring-boot:run
```

Finish

Stop the Spring Boot application, return to your workspace directory and use the `lab` command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation AD221]$ lab finish async-jms
```

This concludes the guided exercise.

Integrating Camel with Kafka

Objectives

After completing this section, you should be able to create routes that use the Kafka component to send and receive durable asynchronous messages.

Defining Apache Kafka

Apache Kafka is an open source distributed system composed of servers and clients that communicate through the TCP protocol. Kafka is a high-performance messaging system, and it is referred to as a distributed commit log system, or as a distributed streaming platform.

Kafka is composed of several servers that are called brokers. To be horizontally scalable, Kafka distributes messages with copies through brokers. These messages are also known as `records`.

Kafka categorizes messages into `topics`. Messages within the same topic are usually related. That is why a topic is conceptually similar to the `table` of a relational database.

Topics consist of `partitions` which are distributed into brokers. Kafka distributes messages into `partitions` for scalability.

Kafka has clients that either write messages to the topics or read messages from topics. A client that writes messages to a topic is called a `producer`, and a client that reads messages from a topic is called a `consumer`.

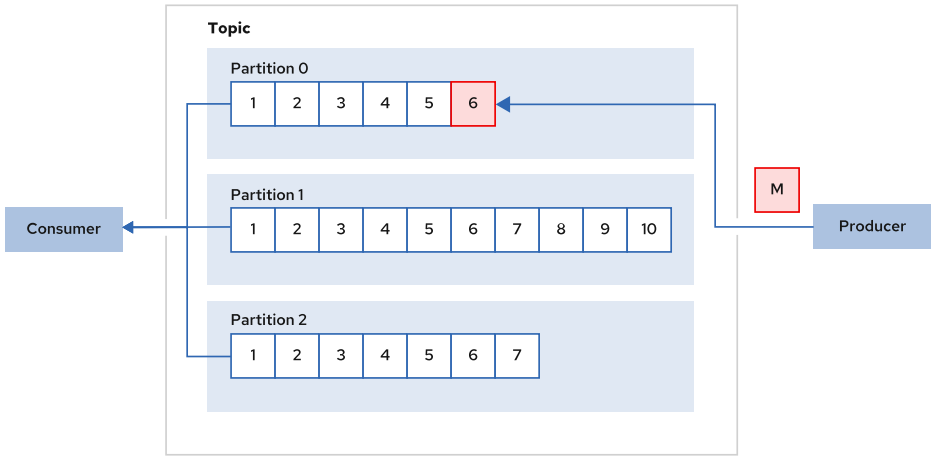


Figure 5.1: The structure of a Kafka topic

Kafka accepts messages in the binary format. That is why it provides a `serialization and deserialization (SerDe)` mechanism with its client API. Producers, serialize messages before sending them, and consumers deserialize messages after receiving them.

You can either use the `SerDe` classes provided by the Kafka client API for some basic types such as `String`, or you can create your custom SerDe classes depending on your requirements.

Figure 5.2: The Kafka SerDe

The Camel Kafka Component

Camel provides a Kafka component that enables Kafka usage in a Camel based integration system.

With the Kafka component, Camel can take advantage of Kafka benefits such as resilience, high-performance and durability, which traditional message brokers usually lack.

As an example, Kafka is durable, so it provides a message replay feature. You can use this feature to consume previous messages in case of a delivery failure. With a traditional broker implementation of Camel, such as JMS or AMQP, you have to implement the `Dead Letter Channel` enterprise integration pattern for resiliency. You do not have to implement the same pattern when using the Kafka component.

Configuring the Kafka Component

To use the Kafka component in a Camel route, you must add the relevant Maven dependency for the component. Depending on the requirements, you can either use the core dependency or the Spring Boot Starter dependency for Kafka.

Using the core dependency

This is the core dependency that provides the bare minimum to use the Kafka component in a Camel context. You can use this dependency in any Maven-based Java application that runs a Camel context.

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

Using the Spring Boot starter dependency

This is the Spring Boot starter dependency that brings Spring Boot autoconfiguration capabilities to Camel.

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

The Spring Boot starter dependency uses the core dependency as well but extends it for Spring Boot usage. This dependency is very useful when adding Kafka configurations for a Spring Boot based Camel application. The subsequent parts of this lecture cover more about how to use the autoconfiguration.

By adding the Maven dependency, the Kafka component becomes ready to use and ready to configure. The Camel URI format for the Kafka component is as follows:

```
kafka:my-topic[?options]
```

The endpoint URI must start with the `kafka:` prefix. Then as a mandatory path parameter, the topic name must follow. In the example the topic name is `my-topic`. The `options` come next and each option must follow the URI parameter format.

These options are the parameters that configure the Camel Kafka component. For example, you can define the `brokers` and an optional consumer parameter `autoCommitEnable` as follows:

```
kafka:my-topic?brokers=localhost:9092&autoCommitEnable=false
```

If you are using the Spring Boot starter dependency, then you do not have to specify the options in the URI. You can define the configurations in the `application.properties` file of the application. The Spring Boot based Red Hat Fuse application uses its autoconfiguration mechanism to apply the configuration. The following snippet applies the same configuration of the preceding example:

```
camel.component.kafka.configuration.brokers=localhost:9092
camel.component.kafka.configuration.auto-commit-enable=false
```

You must use the `camel.component.kafka.configuration.*` autoconfiguration prefix to add the Kafka component configurations.

Note

For more information about the configuration options of the Kafka component, refer to the `Camel Kafka Component` reference, which is in the references list of this lecture.

Using the Kafka Component

A Camel route acts as a client for Kafka. This client either acts as a consumer or a producer.

Consuming messages

You can consume messages from Kafka by using the `from` method of Camel. The following code snippet is a minimal example of a route that reads messages from Kafka.

```
from("kafka:my-topic")
    .log("Message received from Kafka : ${body}") (1)
    .log("on the topic ${headers[kafka.TOPIC]}") (2)
    .log("on the partition ${headers[kafka.PARTITION]}")
    .log("with the offset ${headers[kafka.OFFSET]}")
    .log("with the key ${headers[kafka.KEY]}");
```

(1) The Camel message body, which is also the received Kafka message.

(2) The Camel Kafka component carries the information that returns for the consumed Kafka message by using the message header. You can access this information by using the `headers` array and the `kafka.*` prefixed keys. In this example, the client returns the topic name, the partition, offset and key information of the consumed message.

You can consume from more than one topic with a single Camel Kafka component by separating the topic names with commas:

```
from("kafka:my-topic,other-topic,another-topic")
    .log("Message received from Kafka : ${body}");
```

Producing messages

You can produce messages to Kafka by using the `to` method of Camel. The following code snippet is a minimal example of a route that writes messages to Kafka.

```
from("direct:kafka-producer")
    .setBody(constant("Message from Camel")) (1)
    .setHeader(KafkaConstants.KEY, constant("Camel")) (2)
    .to("kafka:my-topic"); (3)
```

(1) A String message to send to Kafka.

(2) A String key that the Camel Kafka component must carry in the message header. A key is an optional part of a message so this setting is not mandatory. Headers have an important role for carrying message related data for producers, like they do for the consumers.

(3) The producing part of the route. The route sends the defined message and the key to the `my-topic` topic.

Note

For the preceding examples of consumer and producer routes, you might notice there are no configuration parameters in the URIs. Suppose that you use Spring Boot autoconfiguration for the examples.

References

For more information, refer to the *Kafka Component* chapter in the *Apache Camel Component Reference Guide* at
https://access.redhat.com/documentation/en-us/red_hat_fuse/7.10/html-single/apache_camel_component_reference/index#kafka-component

Apache Kafka Official Documentation
https://kafka.apache.org/documentation

Dead Letter Channel
https://www.enterpriseintegrationpatterns.com/DeadLetterChannel.html

Camel Kafka Component
https://camel.apache.org/components/2.x/kafka-component.html

Guided Exercise

Integrating Camel with Kafka

In this exercise, you will help a search and rescue association, FriendWilson Inc. association. The association has an integration service that reads rescue location data from a telecommunication operator's file system, and saves the data into their database.

The association requires the integration service to be asynchronous and resilient. They hired you to refactor the integration service, which is based on Apache Camel, to use Apache Kafka as a message backbone. In this guided exercise you are expected to make the relevant changes in the integration service.

Outcomes

You should be able to configure the Camel application for Kafka broker access, create a Camel route that uses the Kafka component and set it up for sending messages to a particular Kafka topic.

The solution files for this exercise are in the `AD221-apps` repository, within the `async-kafka/solutions` directory.

Before You Begin

To perform this exercise, ensure you have the `AD221-apps` repository cloned in your workstation.

From your workspace directory use the `lab` command to start this exercise. This will make a Kafka cluster and a MySQL instance available for the exercise.

```
[student@workstation AD221]$ lab start async-kafka
```

The `lab` command copies the exercise files from the `~/AD221/AD221-apps/async-kafka/apps` directory, into the `~/AD221/async-kafka` directory.

Instructions

1. Navigate to the `~/AD221/async-kafka/emergency-location-service` project directory and examine the `EmergencyLocationRouteBuilder` class.

2. Run the following command to execute the test for the `emergency-location-route` route.

```
[student@workstation emergency-location-service]$ ./mvnw \
-Dtest=EmergencyLocationRouteBuilderTest#testEmergencyLocationRoute \
clean test
```

   The test must run successfully. The `emergency-location-route` route gets the location data from the vendor file system and saves the data in the MySQL database.

3. Run the `podman stop async-kafka_mysql_1` command to stop the MySQL database that you started with the startup lab script. This simulates a database service down situation.

4. Run the following test command again.

```
[student@workstation emergency-location-service]$ ./mvnw \
-Dtest=EmergencyLocationRouteBuilderTest#testEmergencyLocationRoute \
clean test
```

The test must fail this time.

5. Open the `pom.xml` file and add the `camel-kafka-starter` dependency.

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

6. Open the `src/main/resources/application.properties` file, and add the following `camel.component.kafka.configuration.*` properties.

Property	Value
`brokers`	`localhost:9092`
`auto-offset-reset`	`earliest`
`value-deserializer`	`com.redhat.training.emergency.serde.LocationDeserializer`

7. In the `emergency-location-route` route, add a `to` method with a suitable endpoint to produce messages into the `locations` Kafka topic.

Delete the JDBC statement and the relevant line that has the insert statement.

8. Run the following test command again.

```
[student@workstation emergency-location-service]$ ./mvnw \
-Dtest=EmergencyLocationRouteBuilderTest#testEmergencyLocationRoute \
clean test
```

The test must run successfully.

9. Add a Kafka consumer route with the route id `kafka-consumer-route`.

This route must consume the location data from the `locations` Kafka topic and save it into the database.

10. Run the `podman start async-kafka_mysql_1` command to start the database.

11. Run the following command to execute the test for the `kafka-consumer-route` route.

```
[student@workstation emergency-location-service]$ ./mvnw \
-Dtest=EmergencyLocationRouteBuilderTest#testKafkaConsumerRoute \
clean test
```

The test must run successfully.

The test verifies if there are records created in the database and creates a console output for it such as "The locations table has n records". This means that, the consumer route safely persisted the queued data in the Kafka cluster when the database is back online. Using Kafka prevented data loss and provided resilience.

Finish

Return to your workspace directory and use the `lab` command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation AD221]$ lab finish async-kafka
```

This concludes the guided exercise.

Quiz

Integrating Services Using Asynchronous Messaging

Consider you are working on a development team for a startup company called The Q LLC. This company provides queues as a service, and publish-subscribe APIs for commercial companies.

The development team has to implement some important parts of the application before the deadline. You must choose the correct answers to the following questions to complete the project:

1. **The `registration microservice` has a Camel route that consumes the registration data of customers from a Kafka topic called `registrations`. Another service called `accountancy microservice`, which can only use the JMS protocol, must access this data but Kafka does not support the JMS protocol. You must send the consumed Kafka messages to the ActiveMQ broker's `registrations` queue by using the JMS protocol. Regarding this information, which two of the following options are true? (Choose two.)**

   a. You must create the outgoing endpoint of the route as `to("jms:queue:registrations")`.

   b. You must create the outgoing endpoint of the route as `to("jms:topic:registrations")`.

   c. The incoming endpoint of the route must be `from("kafka:topic:queue:registrations")`.

   d. The incoming endpoint of the route must be `from("kafka:topic:registrations")`.

   e. You must configure a serializer for the `jms` component as `camel.component.jms.configuration.value-serializer=com.theq.serde.RegistrationSerializer` in the application.properties file.

2. **Your team has recently converted one of the microservices to a Spring Boot application, but they have not changed the Camel route code or applied the configuration in the `application.properties` file. You must refactor the following Camel route and apply the required configuration. Regarding this information, which of the following options is FALSE? (Choose one.)**

```
from("kafka:announcements?brokers=broker1:9092
&valueDeserializer=com.theq.services.kafka.serde.AnnouncementDeserializer
&heartbeatIntervalMs=1500")
.to("mongodb:camelMongoClient");
```

   a. You must set the broker configuration as `camel.component.kafka.configuration.brokers=broker1:9092`.
   b. You must set the topic configuration as `camel.component.kafka.configuration.topic=announcements`.
   c. You must set the value deserializer configuration as `camel.component.kafka.configuration.value-deserializer=com.theq.services.kafka.serde.AnnouncementDeserializer`.
   d. After applying all the configurations in the `application.properties` file, you can refactor the route start to `from("kafka:announcements")`.

3. **You must complete a Camel route code that uses the Camel AMQP component. Which of the following options is true for the AMQP component? (Choose one.)**
   a. You can use the `camel-amqp-runner` dependency to configure the Camel AMQP component in a Spring Boot microservice.
   b. The AMQP component uses the Kafka client API to provide messaging topics.
   c. `camel.component.amqp.concurrent-consumers=2` is a valid Spring Boot autoconfiguration for the route that uses the AMQP component. This configuration sets the concurrent consumers count to two.
   d. `amqp:application-topic` is a valid URI for the AMQP component.
   e. You can consume from more than one topic such as `from("amqp:topic:applicants-topic,registration-topic,accounts-topic")`.

4. The team realized that the monitoring system shows a failure in one of the Camel microservices occasionally. The Camel route gets the applied patient data from the government system and saves it in a relational database by sorting it by the case urgency. The database occasionally becomes unresponsive and there is no time to fix or replace it. Regarding this information, which two of the following options are true? (Choose two.)

a. You can solve this issue by using an in-memory queue mechanism. The SEDA component makes the system resilient.

b. You use the Kafka component to queue and sort the messages in a Kafka system before saving them to the database. Kafka's durability makes the system resilient.

c. You can use the AMQP component to queue the messages in an ActiveMQ Broker system and sort them before saving to the database. You can use the Dead Letter Channel implementation to provide resiliency.

d. You can use the AMQP component to queue and sort the messages in an ActiveMQ Broker system before saving to the database. ActiveMQ's durability makes the system resilient.

Solution

Integrating Services Using Asynchronous Messaging

Consider you are working on a development team for a startup company called The Q LLC. This company provides queues as a service, and publish-subscribe APIs for commercial companies.

The development team has to implement some important parts of the application before the deadline. You must choose the correct answers to the following questions to complete the project:

1. The `registration microservice` has a Camel route that consumes the registration data of customers from a Kafka topic called `registrations`. Another service called `accountancy microservice`, which can only use the JMS protocol, must access this data but Kafka does not support the JMS protocol. You must send the consumed Kafka messages to the ActiveMQ broker's `registrations` queue by using the JMS protocol. Regarding this information, which two of the following options are true? (Choose two.)

a. You must create the outgoing endpoint of the route as `to("jms:queue:registrations")`.

b. You must create the outgoing endpoint of the route as `to("jms:topic:registrations")`.

c. The incoming endpoint of the route must be `from("kafka:topic:queue:registrations")`.

d. The incoming endpoint of the route must be `from("kafka:topic:registrations")`.

e. You must configure a serializer for the `jms` component as `camel.component.jms.configuration.value-serializer=com.theq.serde.RegistrationSerializer` in the application.properties file.

2. **Your team has recently converted one of the microservices to a Spring Boot application, but they have not changed the Camel route code or applied the configuration in the `application.properties` file. You must refactor the following Camel route and apply the required configuration. Regarding this information, which of the following options is FALSE? (Choose one.)**

```
from("kafka:announcements?brokers=broker1:9092
&valueDeserializer=com.theq.services.kafka.serde.AnnouncementDeserializer
&heartbeatIntervalMs=1500")
.to("mongodb:camelMongoClient");
```

   a. You must set the broker configuration as `camel.component.kafka.configuration.brokers=broker1:9092`.
   b. You must set the topic configuration as `camel.component.kafka.configuration.topic=announcements`.
   c. You must set the value deserializer configuration as `camel.component.kafka.configuration.value-deserializer=com.theq.services.kafka.serde.AnnouncementDeserializer`.
   d. After applying all the configurations in the `application.properties` file, you can refactor the route start to `from("kafka:announcements")`.

3. **You must complete a Camel route code that uses the Camel AMQP component. Which of the following options is true for the AMQP component? (Choose one.)**
   a. You can use the `camel-amqp-runner` dependency to configure the Camel AMQP component in a Spring Boot microservice.
   b. The AMQP component uses the Kafka client API to provide messaging topics.
   c. `camel.component.amqp.concurrent-consumers=2` is a valid Spring Boot autoconfiguration for the route that uses the AMQP component. This configuration sets the concurrent consumers count to two.
   d. `amqp:application-topic` is a valid URI for the AMQP component.
   e. You can consume from more than one topic such as `from("amqp:topic:applicants-topic,registration-topic,accounts-topic")`.

4. The team realized that the monitoring system shows a failure in one of the Camel microservices occasionally. The Camel route gets the applied patient data from the government system and saves it in a relational database by sorting it by the case urgency. The database occasionally becomes unresponsive and there is no time to fix or replace it. Regarding this information, which two of the following options are true? (Choose two.)

a. You can solve this issue by using an in-memory queue mechanism. The SEDA component makes the system resilient.

b. You use the Kafka component to queue and sort the messages in a Kafka system before saving them to the database. Kafka's durability makes the system resilient.

c. You can use the AMQP component to queue the messages in an ActiveMQ Broker system and sort them before saving to the database. You can use the Dead Letter Channel implementation to provide resiliency.

d. You can use the AMQP component to queue and sort the messages in an ActiveMQ Broker system before saving to the database. ActiveMQ's durability makes the system resilient.

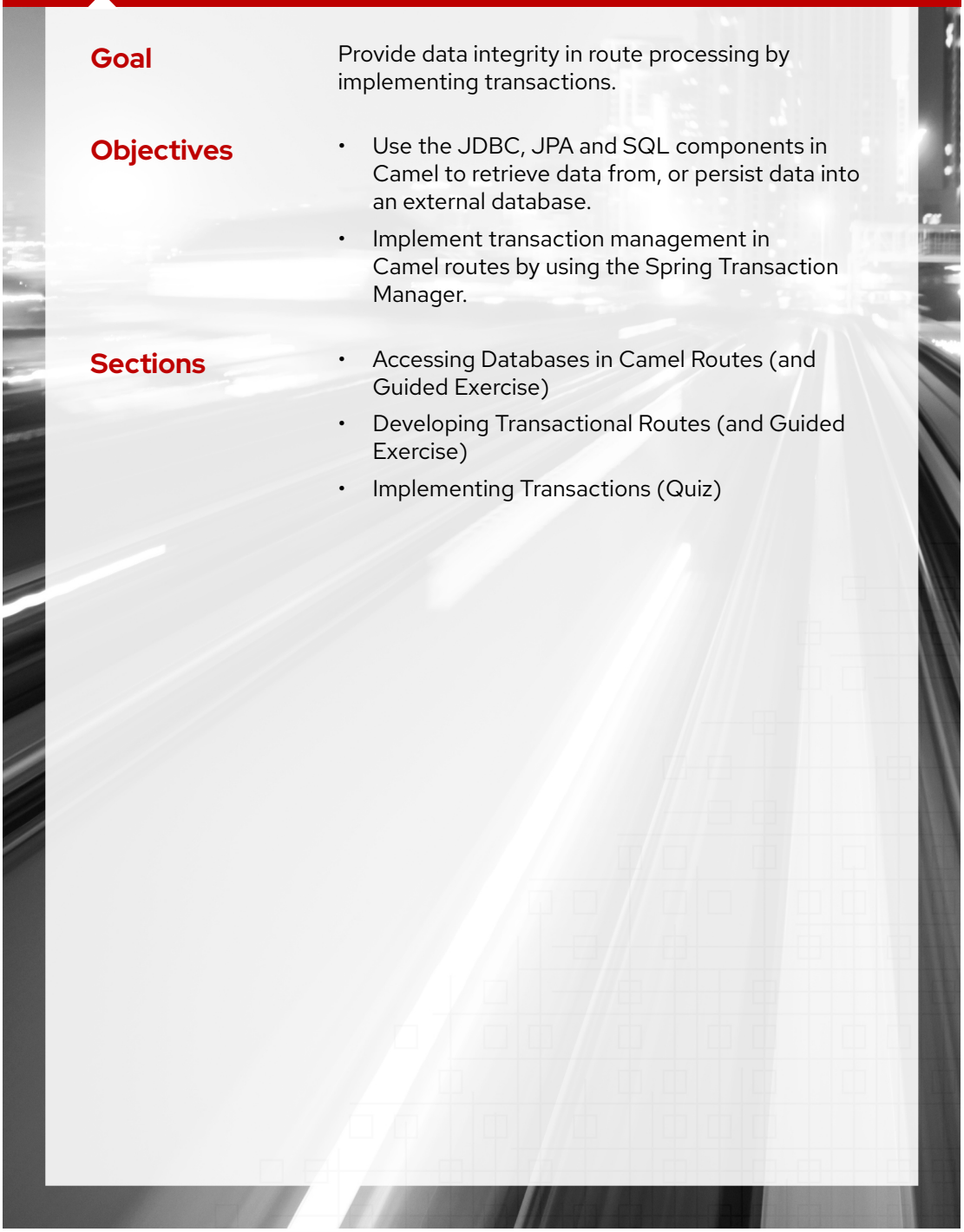
Summary

In this chapter, you learned:

- You can send and receive messages by using the Camel JMS component.
- You can configure the Camel JMS component for the connection factory.
- You can send and receive messages by using the Camel AMQP component, and configure it in a Spring Boot application.
- You can create a producer for writing messages to Kafka and a consumer for reading messages from Kafka, by using the Camel Kafka component.
- You can configure the Camel Kafka component for almost any client option.
- Kafka is durable, and its messages are re-playable. This provides a resilient system for the Camel applications.

Chapter 6

Implementing Transactions



Goal

Provide data integrity in route processing by implementing transactions.

Objectives

- Use the JDBC, JPA and SQL components in Camel to retrieve data from, or persist data into an external database.
- Implement transaction management in Camel routes by using the Spring Transaction Manager.

Sections

- Accessing Databases in Camel Routes (and Guided Exercise)
- Developing Transactional Routes (and Guided Exercise)
- Implementing Transactions (Quiz)

Accessing Databases in Camel Routes

Objectives

After completing this section, you should be able to use the JDBC, JPA and SQL components in Camel to retrieve data from, or persist data into an external database.

Accessing Databases by Using the JDBC, JPA, and SQL Components

Camel provides multiple components to access relational databases:

JDBC
: Provided by the `camel-jdbc` library. This component uses Java Database Connectivity (JDBC) queries through the JDBC API. When using this component, you specify the database query as the message body. By default, the component returns query records in the message body as a list of `Map` objects.

SQL
: Provided by the `camel-sql` library. This component uses JDBC queries through the `spring-jdbc` dependency. By default, it returns the result of queries as a list of `Map` objects. In contrast to the JDBC component, which uses the Camel message body for queries, the SQL component uses the Camel endpoint to specify the query. The use of the JDBC, or the SQL component depends on the integration use case. For example, if you have static or simple queries that only require a few parameters, then the SQL component is easier to use and maintain.

Java Persistence API (JPA)
: Provided by the `camel-jpa` library. Similar to any JPA use case, you can use this component to access databases by using an Object Relational Mapping (ORM) layer.

Configuring a Default Data Source in Spring Boot

Spring Boot developers can configure database connection parameters for Camel components by specifying `spring.datasource.*` properties in the `application.properties` file. The following example shows basic connection parameters for a MySQL database:

```
spring.datasource.url=jdbc:mysql://localhost/my_database
spring.datasource.username=dbuser
spring.datasource.password=dbpass
spring.datasource.platform=mysql
```

Using the JDBC Component

The URI syntax is as follows:

```
jdbc:dataSourceName[?options]
```

Camel looks for the `dataSourceName` bean in the Camel registry. If `dataSourceName` is `dataSource` or `default`, then Camel attempts to use the default data source from the

Camel registry. You can define the default data source by using the `spring.datasource.*` configuration properties.

The `jdbc` component only supports producer endpoints, as the following example shows:

```
from("direct:getUsers")
    .setBody(constant("select * from users")) (1)
    .to("jdbc:dataSource") (2)
    .to("direct:processUsers") (3)
```

(1) The message body contains the query. This particular query selects all users from the database.

(2) The `jdbc` component runs the query. The component uses the default data source.

(3) The route sends the resulting list of users to another endpoint, for further processing.

Using the SQL Component

The URI syntax of the `sql` component is as follows:

```
sql:query[?options]
```

You can use exchange properties in a query, by prepending the property name with the # character. For example, you can select a user by using the syntax shown in the following example.

```
sql:select * from users where id=:#userId
```

If the Camel message body is an instance of `java.util.Map`, then Camel looks for the `userId` property in the body. If the `userId` property is not in the body, or if the body is not a `Map` object, then Camel looks for the `userId` property in the message headers.

You can also use expressions as parameters:

```
sql:update users set email =:#${headers.newEmail} where id=:#userId
```

Alternatively, you can use an external file to define your query by using the `classpath:file_path` expression:

```
sql:classpath:path/to/my_query_file.sql
```

Similar to the `jdbc` component, the result of a `SELECT` query is a list of `Map` objects. Both the `jdbc` and `sql` components, however, provide options to control the output type, such as `outputType` and `outputClass`. Refer to the documentation for more details about these options and the output of other SQL statements, such as `INSERT` or `UPDATE`.

Note

In Spring Boot applications, you can use the `camel-sql-starter` dependency to extend the `camel-sql` library capabilities.

The Spring Boot starter dependency enables Spring Boot autoconfiguration capabilities for this component. This means that users can configure the component by specifying `camel.component.sql.*` properties in the `application.properties` file.

Using the JPA Component

The URI syntax of the `jpa` component is as follows:

```
jpa:entityClassName[?options]
```

The `entityClassName` parameter must be a valid JPA entity class name.

The following example shows how to periodically read `Order` entities from a database.

```
from("jpa:com.redhat.training.entity.Order" (1)
        + "persistenceUnit=mysql" (2)
        + "&consumeDelete=false" (3)
        + "&consumer.namedQuery=getPending" (4)
        + "&maximumResults=5" (5)
        + "&consumer.delay=3000" (6)
        + "&consumeLockEntity=false" (7)
    )
    .process(new OrderProcessor()) (8)
    .to("file:out");
```

(1) The `Order` class is a JPA entity used to query orders from the database.

(2) The Persistence unit is MySQL.

(3) Do not delete records from the database after consumption.

(4) Use a named query from the JPA entity. In this particular example, the `getPending` named query of the `Order` entity retrieves pending orders only. If you do not specify this option, then the `jpa` component selects all records.

(5) Retrieve at most five records in each poll.

(6) Poll the database every three seconds.

(7) Do not set an exclusive lock on each entity bean while processing the results from polling.

(8) The processor receives instances of `Order` as message bodies.

Similar to the preceding components, the `jpa` component provides options to control the output of different operations and queries. Refer to the documentation for a full list of configuration options.

Note

In Spring Boot applications, you can use the `camel-jpa-starter` dependency to extend the `camel-jpa` library capabilities.

You can set `camel.component.jpa.*` properties in the `application.properties` file to auto configure the component.

References

For more information, refer to the *JDBC Component* chapter in the *Red Hat Fuse 7.10 Apache Camel Component Reference* at https://access.redhat.com/documentation/en-us/red_hat_fuse/7.10/html-single/apache_camel_component_reference/index#jdbc-component

For more information, refer to the *SQL Component* chapter in the *Red Hat Fuse 7.10 Apache Camel Component Reference* at https://access.redhat.com/documentation/en-us/red_hat_fuse/7.10/html-single/apache_camel_component_reference/index#sql-component

For more information, refer to the *JPA Component* chapter in the *Red Hat Fuse 7.10 Apache Camel Component Reference* at https://access.redhat.com/documentation/en-us/red_hat_fuse/7.10/html-single/apache_camel_component_reference/index#jpa-component

Guided Exercise

Accessing Databases in Camel Routes

In this exercise, you will use the `camel-jpa-starter` and `camel-sql-starter` components to read and write data in a relational database.

Your company is working on a payment fraud detection system. Payments are stored in a database table called `payments`. The outcome of the fraud detection algorithm is stored in a table called `payment_analysis`.

You must develop a Camel integration that retrieves payments from the `payments` table, processes each payment by running the fraud detection algorithm, and stores the results in the `payment_analysis` table.

Outcomes

You should be able to implement a route that retrieves payments from a database table by using the `camel-jpa` component, and updates another table by using the `camel-sql` component.

The solution files for this exercise are in the `AD221-apps` repository, within the `transaction-database/solutions` directory.

Before You Begin

To perform this exercise, ensure you have the `AD221-apps` repository cloned in your workstation.

From your workspace directory, use the `lab` command to prepare your system for this exercise.

```
[student@workstation AD221]$ lab start transaction-database
```

The `lab` command copies the exercise files from the `~/AD221/AD221-apps/transaction-database/apps` directory, into the `~/AD221/transaction-database` directory. This command also starts a containerized MySQL server and creates a database called `payments`.

Instructions

1. Navigate to the `~/AD221/transaction-database` directory, and open the project with your editor of choice.

2. Inspect the application.

   - The `PaymentAnalysisRouteBuilder` class must implement a route to run the fraud detection algorithm for each payment.
   - The `src/main/resources` directory contains SQL files to create and seed the `payments` and `payment_analysis` tables. Spring Boot executes these SQL scripts on application startup.
   - The `Payment` class implements a Java Persistence API (JPA) entity, which maps to the `payments` table.
   - The `PaymentFraudAnalyzer` class is a Camel processor that runs a mock fraud detection algorithm. This processor computes a fraud score and saves the score in the `fraudScore` exchange header.
   - The `application.properties` file contains the database connection settings.

3. Add the `camel-jpa-starter` dependency to your POM file.

4. Add the JPA endpoint to read payments from the database.

   Edit the `PaymentAnalysisRouteBuilder` class and add the consumer JPA endpoint. The endpoint requirements are as follows:

   - Consume `com.redhat.training.payments.Payment` entities.
   - Use the MySQL JPA persistence unit.
   - Do not delete payments from the table after they are consumed.
   - Limit the number of payments consumed to five.
   - Run the query every three seconds.
   - Deactivate the `consumeLockEntity` option.

5. Use the `./mvnw clean spring-boot:run -DskipTests` command to run the application. Inspect the logs and verify that the logs display consumed payments.

```
...output omitted...
Payment [id=1, userId=11, amount=41.0, currency=EUR]
Payment [id=2, userId=12, amount=500000.0, currency=USD]
...output omitted...
```

6. Stop the application.

7. Add the `camel-sql-starter` dependency to your POM file.

8. Add the SQL endpoint to update the fraud scores value for each payment in the `payment_analysis` table.

   Edit the `PaymentAnalysisRouteBuilder` class and add the producer SQL endpoint to update each `payment_analysis` row. For each payment, you must do the following:

   - The `WHERE` clause must select rows with a `payment_id` field equal to the payment ID of the message body.
   - Set the `fraud_score` field to the value of the `fraudScore` header. This header is set by the `PaymentFraudAnalyzer` processor.
   - Set the `analysis_status` field to `Completed`.

9. Run the tests to verify that the route has written fraud scores to the `payment_analysis` table of the database. Use the `./mvnw clean test` command for this.

   Verify that two tests pass.

Finish

Return to your workspace directory and use the `lab` command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation AD221]$ lab finish transaction-database
```

This concludes the guided exercise.

Developing Transactional Routes

Objectives

After completing this section, you should be able to implement transaction management in Camel routes by using the Spring Transaction Manager.

Defining Transactional Routes

A *transaction* is an operation that conceptually consists of a single step, but the implementation requires a series of steps. In this type of operation, all the steps must succeed or fail, to avoid leaving the system in an inconsistent state.

For example, a route that processes bank transfers between accounts must require that each message can complete different endpoints without data loss. Using transactions during route processing alleviates many common integration issues by rolling back the transfer operation.

When developing a Camel route with transactions, each component must support transactions, and all of them must use the same transaction manager. The JMS messaging, and SQL database components support the use of transactions by implementing the Transactional Client EIP.

There are two types of transactions:

Local transaction
A transaction that spans over one single resource, such as one database.

Global transaction
A transaction that spans over multiple resources, such as one database and one messaging system.

Note
Camel supports transactions by using Spring Transactions, or a JTA transaction manager.

Transaction Managers

A *transaction manager* is the part of an application responsible for coordinating transactions across endpoints. In an integration pipeline, the 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. The transaction is committed when the entire route execution is complete. At this point, all the systems are ready to process the transaction.

Spring offers a number of transaction managers for local transactions. The following snippet creates a `DataSourceTransactionManager` instance to use as the application local transaction manager in Spring Boot.

```
@Configuration
public class MyCustomTransactionManager {
    @Bean
    public PlatformTransactionManager transactionManager(DataSource dataSource) {
        return new DataSourceTransactionManager(dataSource);
    }
}
```

To implement global transactions, you must use a third-party JTA transaction manager, or implement an event-driven architecture.

Transactions Implementation

To enable transaction management in a route, the route must have a `transacted` DSL method after the `from` method.

```
from("schema:origin")
    .transacted()
    ...
```

Transactions Rollback

If a route throws an exception, then Camel might not roll back the transaction. `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.

```
onException(ConnectionException.class)
    .handled(true)
    .to("schema:failure-destination")
    .markRollbackOnly();
```

Defining Transaction Propagation Policies

Transaction policies influence the way a transactional client creates new transactions. For example, you can decide the behavior of a transactional client that detects the existence of a transaction associated with the current thread.

Camel supports predefined and custom transaction propagation policies. The following list contains the most commonly used propagation policies:

PROPAGATION_REQUIRED
All Camel processors from a route must use the same transaction.

PROPAGATION_REQUIRES_NEW
Each processor creates its own transaction.

PROPAGATION_NOT_SUPPORTED
Does not support a current transaction.

The following snippet defines a custom transaction policy as a Java bean in Spring Boot.

```
@Bean(name = "myTransactionPolicy")
public SpringTransactionPolicy propagationRequired(
        PlatformTransactionManager transactionManager
) {

    SpringTransactionPolicy policy = new SpringTransactionPolicy();

    policy.setTransactionManager(transactionManager);
    policy.setPropagationBehaviorName("PROPAGATION_REQUIRED");

    return policy;
}
```

The preceding example creates a custom transaction management policy called `myTransactionPolicy`. This policy, sets the propagation behavior as `PROPAGATION_REQUIRED`, meaning that all Camel processors in the route must use the same transaction.

The following snippet uses the custom transaction management policy called `myTransactionPolicy` in the route transaction.

```
from("schema:origin")
    .transacted("myTransactionPolicy")
    .to("schema:destination");
```

In Camel, a route can only have exactly one transaction policy. If you need to change transaction propagation, for example on nested transactions, then you must use a new route.

Testing Transactional 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, routes must process test data to verify the integrity across multiple services.

Camel supports a set of mechanisms to emulate these requirements, such as the Camel Test Kit (CTK) to mock external systems to run integration tests with live systems, or extension capabilities to plug external technologies to substitute services.

For testing purposes, tests must use a transaction manager to create a runtime environment as accurate as possible to a real world environment. To test whether a transaction fails during a route execution, the Camel Test Kit supports the capability to throw route exceptions, and evaluate the results of the error as well as the success or failure of any transaction rollbacks.

```
@Before
public void setUp() throws Exception {
    context
        .getRouteDefinition("route-one") (1)
        .adviceWith(context, new AdviceWithRouteBuilder() {
            @Override
            public void configure() {
                interceptSendToEndpoint("jpa:*") (2)
                    .throwException( (3)
                        new SQLException("Cannot connect to the database")
```

```
                    );
            }
        });

    context.start();
}
```

1. Updates the `route-one` route.
2. Intercepts any message sent to any JPA endpoint.
3. Throws an `SQLException` exception whenever a message is sent to the database.

On rollbacks, the route execution must not affect resources, such as external databases. To verify if a rollback was successful, tests must query the resources for changes. Alternatively, tests can send invalid content to generate a transaction error.

Avoiding Duplicates with Idempotent Consumers

An algorithm is idempotent if it generates the same result for identical calls, regardless of how many times the algorithm is executed. When an operation is not idempotent, multiple, identical calls might generate different results.

In Camel, idempotent consumers prevent processing the same message multiple times. Camel provides the `idempotentConsumer` processor, which implements the Idempotent Consumer EIP to filter out duplicates.

```
from("direct:start")
    .idempotentConsumer(
        header("paymentId"), (1)
        MemoryIdempotentRepository.memoryIdempotentRepository() (2)
    )
    .log("Unique message ${body}") (3)
    .to("direct:process_unique_messages");
```

1. The unique key is the `paymentId` header. The idempotent consumer verifies this header to filter out duplicates.
2. In-memory implementation of the `org.apache.camel.spi.IdempotentRepository` interface. Idempotent consumers require an instance of an idempotent repository to keep track of unique messages. As the route processes messages, the idempotent consumer queries the repository to verify whether the current message, in this case identified by the `paymentId` header, has been processed before.
3. The rest of the route only processes unique messages.

Camel provides multiple built-in implementations of the `IdempotentRepository` interface, such as the `org.apache.camel.processor.idempotent.MemoryIdempotentRepository` class used in the preceding example. You can, however, create your own repository implementations.

References

Spring Transaction Management
https://docs.spring.io/spring-framework/docs/5.2.15.RELEASE/spring-framework-reference/data-access.html#transaction

For more information, refer to the *Transactional Client* section in the *Red Hat Fuse 7.10 Apache Camel Development Guide* at
https://access.redhat.com/documentation/en-us/red_hat_fuse/7.10/html-single/apache_camel_development_guide/index#MsgEnd-Transactional

For more information, refer to the *Idempotent Consumer* section in the *Red Hat Fuse 7.10 Apache Camel Development Guide* at
https://access.redhat.com/documentation/en-us/red_hat_fuse/7.10/html-single/apache_camel_development_guide/index#MsgEnd-Idempotent

Guided Exercise

Developing Transactional Routes

In this exercise, you will develop an integration for the payments received in an e-commerce platform.

The e-commerce system stores the payments in separate XML files. The payments can have a wrong order ID, or an invalid email.

You must develop a Camel integration to notify customers about processed payments. The integration must retrieve payments from a folder, store them into a database, validate the payment data, and finally send the payments to a queue for further processing. The payments in the database and in the queue must be the ones with valid data.

Outcomes

You should be able to implement transactional routes in Camel.

The solution files for this exercise are in the `AD221-apps` repository, within the `transaction-routes/solutions` directory.

Before You Begin

To perform this exercise, ensure you have the `AD221-apps` repository cloned in your workstation.

From your workspace directory, use the `lab` command to prepare your system for this exercise.

```
[student@workstation AD221]$ lab start transaction-routes
```

The `lab` command copies the exercise files from the `~/AD221/AD221-apps/transaction-routes/apps` directory, into the `~/AD221/transaction-routes` directory. It also starts a containerized MySQL server, creates a database called `payments`, and starts a containerized Red Hat AMQ broker.

Instructions

1. Navigate to the `~/AD221/transaction-routes` directory, open the project with your editor of choice, and examine the code.

2. Run the `./mvnw -DskipTests=true clean package spring-boot:run` command to start the Spring Boot application.

   Notice that the Camel route returns an exception every time it tries to process the `payment-2.xml`, and `payment-3.xml` files. On each attempt, the JPA component saves a new record in the `payments` database because the exception happens after the storage of the data.

3. Open a new terminal window, and execute the following command to verify the existence of multiple records in the database for the invalid payments. Any payment with a negative order ID, or with an empty email, is an invalid order.

```
[student@workstation transaction-routes]$ podman exec database \
mysql -u dbuser -pdbpass payments -e "SELECT * FROM payments;"
id  user_id  amount  order_id  currency  email             notified
...output omitted...
54  3        300     -1        EUR       fail@example.com  NULL
55  2        200     200       EUR                         NULL
56  3        300     -1        EUR       fail@example.com  NULL
57  2        200     200       EUR                         NULL
58  3        300     -1        EUR       fail@example.com  NULL
```

4. Stop the application.

5. Open the `PaymentsTransactionManager` class, and create a transaction manager instance.

   Add a Java bean method that returns a `PlatformTransactionManager` implementation. The method must accept a `DataSource` object as parameter, and return a new instance of the `DataSourceTransactionManager` type.

6. Open the `PaymentsTransactionPolicy` class, and create a transaction policy.

   Add a Java bean method that returns a `SpringTransactionPolicy` instance. The method must accept a `PlatformTransactionManager` object as parameter, and return a new instance of the `SpringTransactionPolicy` type. Set `PROPAGATION_REQUIRED` as the propagation behavior name and the Java bean name.

7. Open the `PaymentRouteBuilder` class. Handle the `IllegalStateException` and `InvalidEmailException` exceptions raised from the `NotificationProcessor` processor by using the `onException` clause. Capture the exceptions, set the maximum re deliveries to 1, and mark the exception as handled. Send the messages with invalid values to the `jms:queue:dead-letter` endpoint, and mark the transaction as `markRollbackOnly`.

8. Update the `payments-process` route to support transactions. Mark the route as transacted, and use the `PROPAGATION_REQUIRED` policy.

9. Verify the correctness of the changes made to the route by executing the unit tests. Run the `./mvnw clean test` command, and verify that three unit tests pass.

10. Run the `./mvnw clean package spring-boot:run` command to start the Spring Boot application. Wait for the application to process the payments, and execute the following command to verify the existence of two records in the database.

```
[student@workstation transaction-routes]$ podman exec database \
mysql -u dbuser -pdbpass payments -e "SELECT * FROM payments;"
id  user_id  amount  order_id  currency  email              notified
1   1        100     100       EUR       user@example.com   1
4   4        400     400       USD       hello@example.com  1
```

   Stop the Spring Boot application.

Finish

Return to your workspace directory, and use the `lab` command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation AD221]$ lab finish transaction-routes
```

This concludes the guided exercise.

Quiz

Implementing Transactions

In this quiz, consider that you are developing Camel routes for data integration and processing in a video streaming platform. Your Camel routes communicate with a relational database and use Spring Boot as the runtime.

Choose the correct answers to the following questions:

1. **The data science team has provided you with a complex and fine-tuned SQL query to extract popularity statistics about each show from the database. They would like you to extract the results of the query and post the data to the REST API of a data lake for further processing. Which two of the following components are suitable components to execute the query? (Choose two.)**
   a. The CSV component
   b. The JDBC component
   c. The SQL component
   d. The JMS component

2. **One of your Camel routes consumes the `Documentary` JPA entity from the database for analysis. This route must read data without deleting records from the database after consumption. Which one of the following options should you use? (Choose one.)**
   a. `consumeDelete=false`
   b. `consumer.delete=true`
   c. `consumer.delete=false`
   d. `consumeDelete=true`

3. **The application runs a statistical analysis on subscription payment data to find potential frauds and anomalies. In the Camel route that implements this mechanism, each message body contains the payment object as the body. The headers of each message are `paymentId`, which uniquely identifies the payment, and `subscriptionId`, which refers to the subscription associated to the payment. The route must process each payment once, however, the existing implementation contains a bug. Which one of the following options fixes the bug? (Choose one.)**

```
MemoryIdempotentRepository repository =
 MemoryIdempotentRepository.memoryIdempotentRepository();

from("direct:paymentAnalysis")
    .idempotentConsumer(
        body("subscriptionId"),
        repository
    )
    .process(new PaymentAnalyzer())
    .to("jms:queue:analisys_results");
```

   a. Replace the repository with a database repository, to ensure that uniqueness is persisted.
   b. Use the `header("paymentId")` expression as the unique key.
   c. Run the `PaymentAnalyzer` processor before the `idempotentConsumer` processor.
   d. Use the `header("subscriptionId")` expression as the unique key.

4. **One of the Camel integration's of the application requires the use of the same transaction to control all Camel processors in route. Which one of the following options should you choose as the transaction policy? (Choose one.)**
   a. `PROPAGATION_REQUIRED`
   b. `PROPAGATION_REQUIRES_NEW`
   c. `PROPAGATION_NOT_SUPPORTED`
   d. `PROPAGATION_SUPPORTED`

Solution

Implementing Transactions

In this quiz, consider that you are developing Camel routes for data integration and processing in a video streaming platform. Your Camel routes communicate with a relational database and use Spring Boot as the runtime.

Choose the correct answers to the following questions:

1. The data science team has provided you with a complex and fine-tuned SQL query to extract popularity statistics about each show from the database. They would like you to extract the results of the query and post the data to the REST API of a data lake for further processing. Which two of the following components are suitable components to execute the query? (Choose two.)

a. The CSV component

b. The JDBC component

c. The SQL component

d. The JMS component

2. One of your Camel routes consumes the `Documentary` JPA entity from the database for analysis. This route must read data without deleting records from the database after consumption. Which one of the following options should you use? (Choose one.)

a. `consumeDelete=false`

b. `consumer.delete=true`

c. `consumer.delete=false`

d. `consumeDelete=true`

3. **The application runs a statistical analysis on subscription payment data to find potential frauds and anomalies. In the Camel route that implements this mechanism, each message body contains the payment object as the body. The headers of each message are `paymentId`, which uniquely identifies the payment, and `subscriptionId`, which refers to the subscription associated to the payment. The route must process each payment once, however, the existing implementation contains a bug. Which one of the following options fixes the bug? (Choose one.)**

```
MemoryIdempotentRepository repository =
 MemoryIdempotentRepository.memoryIdempotentRepository();

from("direct:paymentAnalysis")
    .idempotentConsumer(
        body("subscriptionId"),
        repository
    )
    .process(new PaymentAnalyzer())
    .to("jms:queue:analisys_results");
```

   a. Replace the repository with a database repository, to ensure that uniqueness is persisted.
   b. Use the `header("paymentId")` expression as the unique key.
   c. Run the `PaymentAnalyzer` processor before the `idempotentConsumer` processor.
   d. Use the `header("subscriptionId")` expression as the unique key.

4. **One of the Camel integration's of the application requires the use of the same transaction to control all Camel processors in route. Which one of the following options should you choose as the transaction policy? (Choose one.)**
   a. `PROPAGATION_REQUIRED`
   b. `PROPAGATION_REQUIRES_NEW`
   c. `PROPAGATION_NOT_SUPPORTED`
   d. `PROPAGATION_SUPPORTED`

Summary

In this chapter, you learned:

- You can use the JDBC, SQL, and JPA components in Camel to integrate with databases.
- In Spring Boot, the data source of JDBC, SQL, and JPA components is configurable via `spring.datasource.*` properties.
- Camel implements the Idempotent Consumer EIP to handle duplicate messages.
- Transactions alleviate common Camel integration problems by rolling back operations and leaving the system in a consistent state.

Chapter 7

Building and Consuming REST Services

Goal

Implement and consume REST services with Camel.

Objectives

- Create a route that hosts a REST service by using the REST DSL, and customize a REST service to use various data bindings.
- Develop a Camel route that uses Camel's HTTP component to enrich a message exchange.

Sections

- Implementing REST Services with the REST DSL (and Guided Exercise)
- Consuming HTTP Services (and Guided Exercise)
- Building and Consuming REST Services (Quiz)

Implementing REST Services with the REST DSL

Objectives

After completing this section, you should be able to create a route that hosts a REST service by using the REST DSL, and customize a REST service to use various data bindings.

Introducing the REST DSL API

Representational state transfer (REST) is a way to architect and design web services for simplicity, scalability, and portability. RESTful web services associate URLs and HTTP methods such as `GET` and `POST` to actions on entities or resources that elicit a response from the service in the form of a response payload. This response is either an HTTP status, an XML, or a JSON payload with the response body.

For example, a `GET` request to the `/users/1` endpoint can either return a `404 - Not Found` status code if the user is not present in the system, or it can return a JSON representation of the user if it exists.

Beginning in version 2.14, Camel offers a REST DSL that developers can use in route definitions to build REST web services. You can use the REST DSL to define REST services in Camel routes by using verbs that align with the REST HTTP protocol, such as `GET`, `POST`, `DELETE`, and so on.

The benefit of using this DSL is that it drastically reduces the amount of development time necessary to build REST services into your Camel routes. This reduction comes from eliminating a lot of the boilerplate networking code and enables you to focus on the business logic that supports the REST service.

The DSL builds REST endpoints as consumers for Camel routes. The REST DSL requires an underlying REST implementation provided by components such as Restlet, Spark, and other components that include REST integration.

The following is an example of the REST DSL:

```
public class HelloRoute extends RouteBuilder {

    public void configure() throws Exception {
        restConfiguration()
           .component("servlet") (1)
           .port(8080);

        rest("/speak") (2)
           .get("/hello")
              .transform().constant("Hello World");
    }
}
```

(1) Uses the `camel-servlet` component to implement the REST service.

(2) Returns a `Hello World` message on requests to the `/speak/hello` GET request.

The REST DSL works as an extension to the existing Camel routing DSL, by using specialized keywords to more closely resemble the underlying REST and HTTP technologies.

The REST DSL provides a simple syntax that extends Camel's existing DSL by mapping each keyword to a method. This also means that all existing functionality of a 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 must specify which of the REST DSL capable components should 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 fundamentally the same. This course focuses on `servlet`, however, 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-servlet`: Uses Servlets
- `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

To specify the REST implementation to use, the REST DSL provides the `restConfiguration` method. By using this method, you can control the resulting REST service created by Camel, as shown in the following example:

```
restConfiguration()
    .component("servlet") (1)
    .contextPath("/restService") (2)
    .port(8080); (3)
```

(1) Camel REST component to use

(2) Root context path for all endpoints

(3) Port to bind the REST endpoints

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 `servlet`, `jetty`, `restlet`, `spark-rest`, and `undertow`.
`schema`	The HTTP schema to use, HTTP (default) or HTTPS.
`hostname`	The host name or IP where the HTTP server is bound.
`port`	The port number to use for the HTTP server.

Option	Description
`contextPath`	The base context path for the HTTP server.

You can also set options on the `restConfiguration` DSL method to configure the intended Component, Endpoint, and Consumer. Because options vary from Component to Component, to set them you must use a generic DSL method, as shown in the following table:

REST Configuration Generic Options

Option type	DSL method
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.

The following example sets the `minThreads` and `maxThreads` properties for the Jetty web server:

```
restConfiguration()
    .component("jetty")
    .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 throws a runtime error.

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` method, and set the context path that is specific to this set of services.

You can then define individual services by using REST DSL methods such as `get`, `post`, `put`, and `delete`. You can also define path parameters by using the `{}` syntax for each service.

The following example shows how to use the REST DSL to define multiple services:

```
public class OrderRoute extends RouteBuilder {
    public void configure() throws Exception {

    restConfiguration()
        .component("servlet")
        .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 by using Camel's data formats. This means that incoming JSON or XML data is automatically unmarshaled into model objects, and any processing done inside the service can use the 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.

The following table lists the supported binding modes in the REST DSL, which are defined in the `org.apache.camel.model.rest.RestBindingMode` enumeration:

REST DSL Binding Modes

Mode	Description
`off`	Turns off automated binding. This is the default.
`auto`	Binding is automatic, assuming class path contains the necessary data formats. Typically based on the `Content-Type` header.
`json`	Enables binding to and from JSON, requires `camel-jackson` on the class path.
`xml`	Enables binding to and from XML, requires `camel-jaxb` on the class path.
`json_xml`	Enables binding to and from JSON and XML. Requires class path containing both data formats.

Similar to other configurations for REST DSL, the `restConfiguration` method sets the binding mode, as shown in the following example:

```
restConfiguration()
    .component("spark-rest").port(8080)
    .bindingMode(RestBindingMode.json) (1)
    .dataFormatProperty("prettyPrint", "true");
```

(1) Sets the REST binding mode to JSON

Similar to component or endpoint properties, data format properties specific to the data format you are using can be set generically by using `dataFormatProperty`. In the previous example, Jackson's `prettyPrint` option is set to `true` by using a data format property that formats the JSON output in a human-readable format.

References

For more information, refer to the *Defining REST Services* chapter in the *Apache Camel Component Reference Guide* at https://access.redhat.com/documentation/en-us/red_hat_fuse/7.10/html-single/apache_camel_development_guide/index#RestServices

Claus Ibsen and Jonathan Anstey. (2018) Camel in Action, Second Edition. Manning. ISBN 978-1-617-29293-4.

Guided Exercise

Implementing REST Services with the REST DSL

In this exercise, you will develop a REST API for a retail company.

The company requires you to create two REST endpoints. One to expose all customer payments, and another to expose all the payments for a specific customer.

The internal database has all the customer payments in the `payments` table, and the `userId` field identifies each customer. The company also requires you to use the JPA component to interact with the database, so you can reuse code from a previous integration.

Outcomes

You should be able to implement a route that hosts a REST service that implements two use cases:

- Retrieve all Payments in the database.
- Retrieve all Payments made by one User.

The solution files for this exercise are in the `AD221-apps` repository, within the `rest-dsl/solutions` directory.

Before You Begin

To perform this exercise, ensure you have the `AD221-apps` repository cloned in your workstation.

From your workspace directory, use the `lab` command to prepare your system for this exercise.

This command ensures that the MySQL database is already running before you proceed with the exercise.

```
[student@workstation AD221]$ lab start rest-dsl
```

The `lab` command copies the exercise files from the `~/AD221/AD221-apps/rest-dsl/apps` directory, into the `~/AD221/rest-dsl` directory. The `lab` command also creates a MySQL instance with the data.

Instructions

1. Navigate to the `~/AD221/rest-dsl` directory and open the project with an editor, such as VSCodium.

2. Configure the REST component inside the RouteBuilder.

   Use the `servlet` REST component and configure it to use the port `8080` and use the JSON `RestBindingMode`.

```
restConfiguration()
    // to use servlet component and run on port 8080
    .component("servlet")
    .port(8080)
    .bindingMode(RestBindingMode.json);
```

3. Add two GET REST endpoints by using REST DSL under the `/payments` path that fetches data from the `payments` table.

    3.1. The first one on the / subpath redirects to a Direct component that fetches all the Payments.

    3.2. The second one on the `/{userId}` subpath redirects to a Direct component that fetches all the Payments that belong to the specified user ID.

4. Create the Direct routes to retrieve the data for the REST endpoints.

    4.1. Create a direct route to retrieve all the Payments in the database by using this query.

```
.to("jpa:com.redhat.training.payments.Payment?query=select p from
 com.redhat.training.rest.Payment p")
```

    4.2. Create a direct route to retrieve all the Payments of a specific user ID. Use this query:

```
.toD("jpa:com.redhat.training.payments.Payment?query=select p from
 com.redhat.training.rest.Payment p where p.userId = ${header.userId}")
```

5. Run the Route with `./mvnw spring-boot:run` and use the `curl` command to verify that the route is working. The URL to get the payments is `localhost:8080/camel/payments`. If the application works successfully, then you should see a list of payments.

6. To verify that the Route is working as expected, there is a test in the project that you can use. Use `./mvnw test` to verify that the route matches the expected behavior. If you still have the Route running, use Ctrl+C to terminate it before running the tests.

Finish

Return to your workspace directory, and use the lab command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation AD221]$ lab finish rest-dsl
```

This concludes the guided exercise.

Consuming HTTP Services

Objectives

After completing this section, you should be able to develop a Camel route that uses Camel's HTTP component to enrich a message exchange.

Consuming HTTP Services

Camel provides the `http4` component to integrate with technologies over HTTP. This component can consume contents from an HTTP server, or even use GET and POST HTTP methods with a REST service to retrieve or create data. The `http4` component provides an easy way to consume HTTP services, but cannot produce a REST service. The `http4` component is provided by the `camel-http4` library and uses the following endpoint URI format:

```
http[s]4://hostname[:port][/resourceURI][?options]
```

By default, the `http4` component uses port 80 for HTTP or port 443 for HTTPS.

To import the `camel-http4` library, include the following configuration in the `pom.xml` file:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-http4</artifactId>
</dependency>
```

Alternatively, use the `camel-http4-starter` library to enable autoconfiguration with Spring Boot.

Camel always uses the `InOut` message exchange protocol because of the HTTP protocol nature (based on a request/response paradigm).

Note

You can only produce to endpoints generated by the `http4` 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 the header field called `Exchange.HTTP_METHOD`.
2. GET, if a query string is provided in the header `Exchange.HTTP_QUERY`.
3. GET, if the 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 a GET or POST request as follows:

- Sends an HTTP POST request to the URL by 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 then it sends an HTTP GET request to the URL and returns the response as the `outMessage` object on the exchange.

Endpoint options and HTTP query parameters have the same syntax. You must use the `Exchange.HTTP_QUERY` header to set HTTP query parameters. For example, to make the `http://example.com?order=123&detail=short` GET request, use the following:

```
from("direct:start")
  .setHeader(Exchange.HTTP_QUERY, constant("order=123&detail=short"))
  .to("http4://example.com");
```

You can also use the `connectTimeout` endpoint option in the `http4` endpoint, as the following example shows:

```
from("direct:start")
  .setHeader(Exchange.HTTP_QUERY, constant("order=123&detail=short"))
  .to("http4://example.com?connectTimeout=2000");
```

Handling HTTP Errors

When you use the `camel-http4` component, Camel can throw an exception depending on the HTTP response code returned by the external resource:

A response code from 100 to 299 is a success.

If the response code is 300 or greater, then Camel 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("http4://example.com?throwExceptionOnFailure=false");
```

Enriching Message Exchanges

When sending data from one system to another, it is common for the receiving system to require more information than the source can provide. For example, the source system's data might only contain a customer ID, but the receiving system actually requires the customer name and address. Similarly, an order message sent by the order management system might only contain an order number. However, you must 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. The `http4` component is a common option for acquiring the additional data.

Camel supports the enrich EIP by using the `enrich` DSL method to enrich the message. The enrich DSL method has two parameters. The first is the URI of the producer Camel must invoke to retrieve the enrichment data. The second parameter is an optional instance of the `AggregationStrategy` implementation, which you must provide to Camel for use when combining the original message exchange with the enrichment data. If you do not provide an aggregation strategy, then Camel uses the body obtained from the resource as the enriched message exchange.

The enrich method synchronously retrieves additional data from a resource endpoint to enrich an incoming message (contained in the original 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;
    }
}
```

You can use the `http4` 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("http4://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. Setting the body of the exchange to null causes the `http4` component to send 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.

Consuming SOAP Services

The Simple Object Access Protocol (SOAP) is an application communication protocol for sending and receiving messages using XML in a way that is platform independent. SOAP provides a way to communicate between applications running on different operating systems, with different technologies and programming languages.

While the `http4` component could be used to query a SOAP application, additional tools are required to build SOAP requests. The additional tools are provided by the `camel-cxf` library. The `camel-cxf` library provides a `cxf` component that is a wrapper for Apache CXF, a Java library for working with web services. To invoke a SOAP service in a camel route, take the following three steps:

1. Create client classes from the WSDL
2. Create the request payload
3. Set up the CXF (SOAP) endpoint in the route

Creating Client Classes from the WSDL

The Web Service Description Language (WSDL) is an XML based definition language. The WSDL file describes the functionality of a SOAP-based web service.

Use the `cxf-codgen-plugin` for Maven to create the Java classes from your WSDL file. To use this feature, first include the following in your project's `pom.xml`:

```
<plugin>
   <groupId>org.apache.cxf</groupId>
   <artifactId>cxf-codegen-plugin</artifactId> (1)
```

```
    <version>${cxf.version}</version>
    <executions>
      <execution>
        <id>generate-sources</id>
        <phase>generate-sources</phase>
        <configuration>
          <wsdlOptions>
            <wsdlOption>
              <wsdl>src/main/resources/wsdl/Footprint.wsdl</wsdl> (2)
            </wsdlOption>
          </wsdlOptions>
        </configuration>
        <goals>
          <goal>wsdl2java</goal>
        </goals>
      </execution>
    </executions>
  </plugin>
```

(1) The `cxf-codgen-plugin` creates and compiles Java classes from the WSDL file to the `target/generated-sources/cxf` directory.

(2) Obtain the WSDL file from the web service and copy it to the `src/main/resources/wsdl` directory.

To generate classes from the WSDL use the following Maven command:

```
mvn generate-sources
```

Creating the Request Payload

Now that you have the WSDL classes generated, use a Java bean to create a Java object of the classes that represent the service's request type. For the `Footprint` service, the request type might be `GetFootprintRequest`. An example class follows:

```
public class GetFootprintBuilder {

     public GetFootprintRequest getFootprint(String id) {
         GetFootprintRequest request = new GetFootprintRequest();
         request.setID(id);

         return request;
     }
 }
```

This class can now be used in the route.

Setting up the CXF (SOAP) Endpoint in the Route

In the route, invoke the Java bean from the previous step. The following example invokes the `GetFootprintBuilder` bean.

```
from("direct:start")
     .setBody(constant("12"))  (1)
     .bean(GetFootprintBuilder.class)  (2)
...
```

(1) The value of `12` in the message body is set to use in the `id` parameter.

(2) When the GetFootprintBuilder bean is invoked, camel uses `bean parameter binding` to pass the value 12 as the argument to the `getFootprint(String id)` method. The bean replaces the exchange body with a `GetFootPrintRequest` object with an `id` value of `12`.

Identify the generated `serviceClass` for the SOAP operation. The class is an `interface` that includes the values that must match the `cxf` component configuration. Use the `OPERATION_NAME` and `OPERATION_NAMESPACE` in the message header, as illustrated in the following example:

```
from("direct:start")
     .setBody(constant("12"))
     .bean(GetFootprintBuilder.class)
     .setHeader(CxfConstants.OPERATION_NAME, constant("GetFootprint"))
     .setHeader(CxfConstants.OPERATION_NAMESPACE,
      constant("http://training.redhat.com/FootprintService/"))

...
```

Finally, the `cxf` endpoint requires the following parameters.

```
from("direct:start")
     .setBody(constant("12"))
     .bean(GetFootprintBuilder.class)
     .setHeader(CxfConstants.OPERATION_NAME, constant("GetFootprint"))
     .setHeader(CxfConstants.OPERATION_NAMESPACE,
      constant("http://training.redhat.com/FootprintService/"))
     .to("cxf://http://localhost:8423/ws"  (1)
     + "?serviceClass=com.redhattraining.service.FootprintServiceEndpoint")  (2)
```

(1) URL for the service

(2) Interface generated by the `cxf-codegen-plugin` plug-in that represents the SOAP service endpoint.

References

Consuming a SOAP service with Apache Camel
https://tomd.xyz/camel-consume-soap-service/

For more information, refer to the *Http4 Component* chapter in the *Apache Camel Component Reference* at
https://access.redhat.com/documentation/en-us/red_hat_fuse/7.10/html-single/apache_camel_component_reference/index#http4-component

For more information, refer to the *CXF Component* chapter in the *Apache Camel Component Reference* at
https://access.redhat.com/documentation/en-us/red_hat_fuse/7.10/html-single/apache_camel_component_reference/index#cxf-component

Guided Exercise

Consuming HTTP Services

In this exercise, assume you work for Acme Inc, which is testing new ways to reduce their carbon footprint.

The company has a SOAP service that calculates the carbon footprint of the customers, based on their previous orders. The company requires you to enrich the received orders with an additional header, which includes the carbon footprint value.

Outcomes

In this exercise you should be able to:

- Consume HTTP content by using the HTTP4 component
- Consume a SOAP service by using the CXF component
- Use the Enrich EIP to augment a message with additional content

Before You Begin

To perform this exercise, ensure you have the `AD221-apps` repository cloned in your workstation.

From your workspace directory, use the `lab` command to start this exercise.

```
[student@workstation AD221]$ lab start rest-http
```

The `lab` command copies the exercise files from the `~/AD221/AD221-apps/rest-http/apps` directory, into the `~/AD221/rest-http` directory. The `Lab` command also creates a Red Hat AMQ instance, and a SOAP service.

Note

You can inspect the logs for the AMQ instance at any time with the following command:

```
[student@workstation AD221]$ podman logs artemis
```

You can inspect the logs for the SOAP server instance at any time with the following command:

```
[student@workstation AD221]$ podman logs soap_server
```

Instructions

1. Navigate to the `~/AD221/rest-http` directory, open the project with your editor of choice, and examine the code.

2. Open the project's POM file, and add the `camel-http4` dependency from the `org.apache.camel` group.

3. Pull the WSDL file by using the HTTP4 component.

   Open the `WSDLRetrievalRouteBuilder` class, and retrieve the WSDL file from the `http4://localhost:8080/footprints.php` endpoint.

4. Verify the correctness of the route by executing the unit tests.

   Run the `./mvnw clean -Dtest=WSDLRetrievalRouteBuilderTest test` command, and verify that one unit test passes.

5. Open the project's POM file to add the required CXF dependencies and plug-ins.

   5.1. Add the `camel-cxf` dependency from the `org.apache.camel` group.

   5.2. Add the `cxf-codegen-plugin` plug-in.

6. Execute the `./mvnw generate-sources` command to generate client classes from the WSDL.

   Open the `target/generated-sources/cxf/com/redhat/training/carbonfootprintservice/` folder to view the generated files.

7. Create a Java bean called `GetFootprintBuilder` to create the SOAP request object.

```
package com.redhat.training.messaging;

import com.redhat.training.carbonfootprintservice.CarbonFootprintRequest;

public class GetFootprintBuilder {

     public CarbonFootprintRequest getFootprint(String id) {
         CarbonFootprintRequest request = new CarbonFootprintRequest();
         request.setID(id);

         return request;
     }
}
```

8. Create a route that uses the CXF component to query the SOAP service.

   Open the `SoapRouteBuilder` class, and edit the class based on the following requirements:

SoapRouteBuilder requirements

Description	Value
Java Bean to build the SOAP request	`GetFootprintBuilder.class`
SOAP Operation Name	`CarbonFootprint`
SOAP Operation Namespace	`http://training.redhat.com/CarbonFootprintService/`
URI for SOAP service	`http://localhost:8080/footprints.php`
SOAP service interface	`com.redhat.training.carbonfootprintservice.CarbonFootprintEndpoint`

You must import the `org.apache.camel.component.cxf.common.message.CxfConstants` class.

9. Verify the correctness of the route by executing the unit tests.

   Run the `./mvnw clean -Dtest=SoapRouteBuilderTest test` command, and verify that one unit test passes.

10. Implement an aggregation strategy to merge the content of two exchanges. The implementation must add a header to an exchange based on the content of another exchange.

    Create the `HttpAggregationStrategy` class with the following contents:

```
package com.redhat.training.messaging;

import org.apache.camel.processor.aggregate.AggregationStrategy;
import org.apache.camel.Exchange;
import com.redhat.training.carbonfootprintservice.CarbonFootprintResponse;

public class HttpAggregationStrategy implements AggregationStrategy {

    public static final String FOOTPRINT_HEADER = "FOOT_PRINT";

    public Exchange aggregate(Exchange original, Exchange resource) {
        CarbonFootprintResponse carbonFootprintResponse =
 resource.getIn().getBody(CarbonFootprintResponse.class);
        original.getIn().setHeader(FOOTPRINT_HEADER,
 carbonFootprintResponse.getCarbonFootprint());

        return original;
    }
}
```

11. Create a route to receive messages with orders, and enrich the message with data from the SOAP service.

    Open the `EnrichRouteBuilder` class, and use the `enrich` method to pull data from the `direct:soap` endpoint. Use the `HttpAggregationStrategy` class to enrich the original message with a new header value.

12. Verify the correctness of the route by executing the unit tests.

    Run the `./mvnw clean -Dtest=EnrichRouteBuilderTest test` command, and verify that one unit test passes.

13. Build and run the application with the `./mvnw clean spring-boot:run` command.

    Verify in the console output that the `enrich-route` route did not process any messages, and stop the application.

14. Open the `JmsRouteBuilder` class, and edit the route to send messages to the `direct:enrich` endpoint instead of the `direct:soap` endpoint.

15. Build and run the application with the `./mvnw clean spring-boot:run` command.

    Verify in the console output that the `enrich-route` route processed messages.

```
...output omitted...
... enrich-route : Order sent to fulfillment: {"ID":2 ... "customer-b"}
... enrich-route : New Header value: 16428.22
```

Finish

Stop the Spring Boot application, return to your workspace directory, and use the `lab` command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation AD221]$ lab finish rest-http
```

This concludes the guided exercise.

Quiz

Building and Consuming REST Services

Consider that you have recently joined a real estate startup. The startup has an online platform that enriches the information available for the properties with external data.

You must choose the correct answers to the following questions to complete the relevant part of the integration services:

1. The company has a REST service that exposes the information available for the properties. Regarding the following code, which sentence is true? (Choose one.)

```
restConfiguration()
    .component("servlet")
        .port(8080)
        .bindingMode(RestBindingMode.json);

rest("/house")
    .get("/{maxPrice}")
        .route()
            .choice()
                .when().simple("${maxPrize} > 5000")
                    .to("direct:partnerHouses")
                .otherwise()
                    .to("direct:companyHouses)
    .get("/")
        .to("direct:allHouses")
    .get("/{id}/details")
        .to("direct:houseDetails");
```

a. The service exposes all the available houses in the `/` GET REST endpoint.
b. The `direct:partnerHouses` endpoint is responsible for processing GET requests for houses with a `maxPrice` greater than 5000.
c. A GET request to the `/123/details` endpoint returns details for the property with ID `123`.
d. The code has a bug in the definition and usage of the REST path parameters.

2. **The company has a service responsible for enriching the information available for the properties with external data. Regarding the following integration code, which sentence is true? (Choose one.)**

```
from("direct:origin-a")
  .enrich("direct:origin-b", aggregationStrategy)
  .to("direct:result");
```

   a. The `enrich` DSL method invokes asynchronously the `direct:origin-b` endpoint to obtain some additional data.
   b. Camel invokes the `direct:origin-b` endpoint to enrich incoming messages from the `direct:origin-a` endpoint.
   c. Camel invokes the `direct:origin-a` endpoint to enrich incoming messages from the `direct:origin-b` endpoint, and uses the `aggregationStrategy` instance to combine the data.
   d. The `aggregationStrategy` instance is responsible for invoking the `direct:origin-b` endpoint, and combining the enrichment data with the original exchange.

3. **A big partner of the startup has a REST API exposed in the `api.partner.example.com` hostname. The API has a `properties` endpoint that returns all the properties available. The endpoint also supports the `details` parameter to limit the amount of information that it returns. You want to build an integration with the partner and use the `short` details option. Which two options can you use to implement this requirement? (Choose two.)**
   a. Consume from the `http4://api.partner.example.com/properties?details=short` endpoint.
   b. Produce to the `http4://api.partner.example.com/properties?details=short` endpoint.
   c. Consume from the `http4://api.partner.example.com/properties` endpoint with an HTTP_QUERY header with `details=short` as the value.
   d. Set an HTTP_QUERY header with `details=short` as the value, and produce to the `http4://api.partner.example.com/properties` endpoint.

4. The startup enriches the data retrieved from the partners with statistical information about the property location. A SOAP service available in the `stats.example.com` host provides the statistical information. Regarding the following code, which two sentences are true? (Choose two.)

```
from("direct:start")
    .setBody(constant("Liverpool"))
    .bean(GetStatsRequestBuilder.class)
    .setHeader(CxfConstants.OPERATION_NAME, constant("GetStats"))
    .setHeader(CxfConstants.OPERATION_NAMESPACE,
        constant("http://stats.example.com/"))
    .to("cxf://http://stats.example.com:8423"
        + "?serviceClass=com.example.StatsService"
        + "&wsdlURL=/stats.wsdl")
    .log("The population in the area is : ${body[0].stats.population}");
```

a. The code has a bug in the CFX URI.

b. The code writes in the log the population of Liverpool.

c. The implementation uses a WSDL located in the `http://stats.example.com/stats.wsdl` URL.

d. The implementation uses a WSDL located in the file system.

Solution

Building and Consuming REST Services

Consider that you have recently joined a real estate startup. The startup has an online platform that enriches the information available for the properties with external data.

You must choose the correct answers to the following questions to complete the relevant part of the integration services:

1. The company has a REST service that exposes the information available for the properties. Regarding the following code, which sentence is true? (Choose one.)

```
restConfiguration()
    .component("servlet")
        .port(8080)
        .bindingMode(RestBindingMode.json);

rest("/house")
    .get("/{maxPrice}")
        .route()
            .choice()
                .when().simple("${maxPrize} > 5000")
                    .to("direct:partnerHouses")
                .otherwise()
                    .to("direct:companyHouses)
    .get("/")
        .to("direct:allHouses")
    .get("/{id}/details")
        .to("direct:houseDetails");
```

a. The service exposes all the available houses in the / GET REST endpoint.

b. The `direct:partnerHouses` endpoint is responsible for processing GET requests for houses with a `maxPrice` greater than 5000.

c. A GET request to the `/123/details` endpoint returns details for the property with ID 123.

d. The code has a bug in the definition and usage of the REST path parameters.

2. **The company has a service responsible for enriching the information available for the properties with external data. Regarding the following integration code, which sentence is true? (Choose one.)**

```
from("direct:origin-a")
  .enrich("direct:origin-b", aggregationStrategy)
  .to("direct:result");
```

   a. The `enrich` DSL method invokes asynchronously the `direct:origin-b` endpoint to obtain some additional data.
   b. Camel invokes the `direct:origin-b` endpoint to enrich incoming messages from the `direct:origin-a` endpoint.
   c. Camel invokes the `direct:origin-a` endpoint to enrich incoming messages from the `direct:origin-b` endpoint, and uses the `aggregationStrategy` instance to combine the data.
   d. The `aggregationStrategy` instance is responsible for invoking the `direct:origin-b` endpoint, and combining the enrichment data with the original exchange.

3. **A big partner of the startup has a REST API exposed in the `api.partner.example.com` hostname. The API has a `properties` endpoint that returns all the properties available. The endpoint also supports the `details` parameter to limit the amount of information that it returns. You want to build an integration with the partner and use the `short` details option. Which two options can you use to implement this requirement? (Choose two.)**
   a. Consume from the `http4://api.partner.example.com/properties?details=short` endpoint.
   b. Produce to the `http4://api.partner.example.com/properties?details=short` endpoint.
   c. Consume from the `http4://api.partner.example.com/properties` endpoint with an HTTP_QUERY header with `details=short` as the value.
   d. Set an HTTP_QUERY header with `details=short` as the value, and produce to the `http4://api.partner.example.com/properties` endpoint.

4. The startup enriches the data retrieved from the partners with statistical information about the property location. A SOAP service available in the `stats.example.com` host provides the statistical information. Regarding the following code, which two sentences are true? (Choose two.)

```
from("direct:start")
    .setBody(constant("Liverpool"))
    .bean(GetStatsRequestBuilder.class)
    .setHeader(CxfConstants.OPERATION_NAME, constant("GetStats"))
    .setHeader(CxfConstants.OPERATION_NAMESPACE,
        constant("http://stats.example.com/"))
    .to("cxf://http://stats.example.com:8423"
        + "?serviceClass=com.example.StatsService"
        + "&wsdlURL=/stats.wsdl")
    .log("The population in the area is : ${body[0].stats.population}");
```

a. The code has a bug in the CFX URI.

b. The code writes in the log the population of Liverpool.

c. The implementation uses a WSDL located in the `http://stats.example.com/stats.wsdl` URL.

d. The implementation uses a WSDL located in the file system.

Summary

In this chapter, you learned:

- REST DSL is a wrapper layer that provides REST DSL methods such as `get`, `post`, `put`, and `delete`.
- You can define and configure the REST implementation to use with the `restConfiguration` method.
- The content enricher pattern enhances your message exchange with additional data.
- The HTTP4 component provides HTTP based endpoints for calling external HTTP resources.
- You can use the CXF component to communicate with a SOAP service.

Chapter 8

Integrating Cloud-native Services

Goal

Deploy cloud-native integration services based on Camel Routes to OpenShift.

Objectives

- Deploy Spring Boot Camel applications to OpenShift.
- Develop Camel routes with Quarkus.
- Create cloud-native integrations with Camel K.

Sections

- Deploying Camel Applications to Red Hat OpenShift (and Guided Exercise)
- Integrating Cloud-native Services Using Camel Quarkus (and Guided Exercise)
- Integrating Cloud-native Services Using Camel K (and Guided Exercise)
- Integrating Cloud-native Services (Quiz)

Deploying Camel Applications to Red Hat OpenShift

Objectives

After completing this section, you should be able to deploy Spring Boot Camel applications to OpenShift.

Deploying Spring Boot Camel Applications to Red Hat OpenShift Container Platform

There are multiple ways to deploy Spring Boot applications to the Red Hat OpenShift Container Platform (RHOCP). You can take responsibility for the whole process: building a container image, pushing the image to a container registry, and creating all the required RHOCP resources to deploy the image to the cluster.

Alternatively, you can use the RHOCP Source-to-Image (S2I) build process to offload parts of the deployment process to RHOCP. The following options are some common deployment workflows:

- Trigger an S2I build from the source code.
- Trigger an S2I build from a binary artifact, such as a JAR file.
- Do not use S2I. Instead, build the container image, push the image to a registry, and create the deployment in RHOCP.

This course focuses on S2I builds from the source code triggered by JKube. JKube is a project of the Eclipse foundation, which provides components to simplify the deployment of cloud-native Java applications, relieving developers from repetitive deployment tasks. JKube uses S2I to generate the container image for you, also creating the rest of RHOCP and Kubernetes resources required to deploy the application.

Configuring Deployments with JKube

For OpenShift deployments, JKube provides the `openshift-maven-plugin` module. To use this plug-in to deploy your application, you must create a build profile in you POM file, as follows:

```
<profile>
    <id>openshift</id>
    <properties>
        <jkube.generator.from>
            registry.redhat.io/fuse7/fuse-java-openshift-rhel8:1.10
        </jkube.generator.from> (1)
    </properties>
    <build>
        <plugins>
            <plugin>
                <groupId>org.jboss.redhat-fuse</groupId>
                <artifactId>openshift-maven-plugin</artifactId> (2)
                <version>7.10.0.fuse-sb2-7_10_0-00014-redhat-00001</version>
                <executions>
                    <execution>
```

```
                        <goals>
                            <goal>resource</goal> (3)
                            <goal>build</goal> (4)
                            <goal>apply</goal> (5)
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</profile>
```

1. The base image used to generate the container image where the application runs. The `registry.redhat.io/fuse7/fuse-java-openshift-rhel8` image provides all the required dependencies to run Spring Boot Camel applications on OpenShift.
2. The name of the plug-in to use in this profile: `openshift-maven-plugin`. The plug-in provides build goals to generate Kubernetes and RHOCP artifacts and resources.
3. The `resource` goal generates the OpenShift resources required for your application, based on the contents of your project's `src/main/jkube` directory.
4. The `build` goal builds the container image for your application.
5. The `apply` goal applies the resources produced by the `resource` goal to your RHOCP cluster.

Additionally, if you use Java 11 with Fuse 7.10, you might want to specify a Java 11 profile to simplify the build process.

Note

Alternatively, you can use the `spring-boot-2-camel-xml` quick start, provided by Red Hat Fuse. The archetype creates a Spring Boot Camel project ready to be deployed to RHOCP.

To deploy your application to RHOCP, use the following command from the root of your Spring Boot project:

```
[student@workstation your-project] mvn oc:deploy -Popenshift
```

Important

Before running `mvn oc:deploy`, you must use the `oc` CLI to log in to the cluster, and select the project in which you want to deploy your application.

Monitoring Health with Probes

In Kubernetes and RHOCP, you can monitor the health of your application by using container health probes. You must define probes at the container level, when declaring the pods of your application. There are three types of probes:

Readiness Probes
A *readiness probe* determines whether a container is ready to service requests. If the probe fails, then RHOCP stops sending traffic to the container.

Liveness Probes
A *liveness probe* determines whether a container is still running. If the probe fails, then RHOCP kills the container, which is subjected to its restart policy.

Startup probe
A *startup probe* verifies whether an application in a container has started. Startup probes automatically disable readiness and liveness probes until the application starts. If the startup probe fails, then RHOCP kills the container, which is subjected to its restart policy.

To configure probes with JKube, specify the configuration of the probes in the `src/main/jkube/deployment.yml` file of your project. The following snippet shows an example of this file including probes configuration:

```
spec:
  template:
    spec:
      containers:
        - readinessProbe:
            httpGet:
              path: /health/ready
              port: 80
              scheme: HTTP
            timeoutSeconds: 5
          livenessProbe:
            exec:
              command:
              - cat
              - /tmp/health
            failureThreshold: 4
        ...container spec omitted...
```

The preceding example defines a readiness and a liveness probe.

- The readiness probe makes requests to the `/health/ready` endpoint and port 80 of the container host.
- The liveness probe executes the `cat /tmp/health` command in the container.

Probes can verify the health status by using checks such as *HTTP endpoint checks*, and *Container execution checks*. You can also set up configuration options, such as failure thresholds and timeouts. Refer to the RHOCP health monitoring documentation for more details about different types of checks and configurations.

Exposing Health Endpoints with Spring Boot Actuator

The `spring-boot-actuator` module implements a set of production-ready features for Spring Boot applications. With Actuator, you can easily expose endpoints to audit and monitor applications.

Spring Boot Actuator is preconfigured with default health checks exposed through `/actuator/health/*` endpoints. An endpoint reporting a healthy state returns a `200 OK` HTTP code. Otherwise, the endpoint returns a `503 Service Unavailable` HTTP code.

In the Actuator, beans that implement health checks are called *health indicators*. Each indicator implements the `org.springframework.boot.actuate.health.HealthIndicator` interface. By implementing this interface, you can define custom health indicators. For example, you can define an indicator to verify the health of a Camel route, as the following example demonstrates:

```
@Component
public class MyCamelRouteHealthIndicator implements HealthIndicator {

    ...code omitted...

    @Override
    public Health health() {
        if (myRouteChecker.isDown()) {
            return Health
                    .down()
                    .withDetail(
                        "Route failed",
                        CamelRouteHealth.getErrorMessage())
                    .build();
        }
        return Health.up().build();
    }

}
```

You can use a simple bean instance, such as `myRouteChecker` in the preceding example, to store the health status of a route. From your route, you can use this bean to set the health status of the route.

Actuator Configuration

You can configure Actuator endpoints and health checks in your `application.properties` file. For example:

```
management.endpoint.health.show-details = always (1)
management.health.probes.enabled=true (2)
```

(1) Shows additional details in the `/actuator/health/` endpoint for all requests. When `show-details` is activated, Actuator returns more detailed information, such as the uptime, and the result of each registered indicator.

(2) Activates container health probes.

Liveness and Readiness Probes

When the `management.health.probes` setting is enabled, Actuator enables the following endpoints:

- `/actuator/health/liveness`
- `/actuator/health/readiness`

These endpoints correspond to the liveness and readiness *health groups*. A health group is a way to organize multiple indicators together.

By default, the `readiness` group only includes the `readinessState` indicator. The `liveness` group only includes the `livenessState` indicator.

You can add additional indicators to a group with the corresponding configuration parameter, as the following example shows:

```
management.endpoint.health.group.readiness.include=myCustomCheck,readinessState
```

The preceding example uses the `myCustomCheck` and `readinessState` indicators for the `readiness` group. If any of these indicators report that the health status is `down`, then the `/actuator/health/readiness` endpoint returns a 503 HTTP error, indicating that the application is not ready.

To map the class name of an indicator to an indicator identifier, Actuator removes the trailing `HealthIndicator` from the class name. Therefore, for the previous example, `myCustomCheck` resolves to the `MyCustomCheckHealthIndicator` bean.

References

JKube openshift-maven-plugin
https://www.eclipse.org/jkube/docs/openshift-maven-plugin

Spring Boot 2.3.12.RELEASE - Health Information
https://docs.spring.io/spring-boot/docs/2.3.12.RELEASE/reference/html/production-ready-features.html#production-ready-health

For more information, refer to the *Monitoring application health by using health checks* chapter in the *OpenShift Container Platform 4.6 Applications Guide* at https://access.redhat.com/documentation/en-us/openshift_container_platform/4.6/html-single/applications/index#application-health

For more information, refer to the *Creating and deploying applications on Fuse on OpenShift* section in the *Fuse on OpenShift Guide* at https://access.redhat.com/documentation/en-us/red_hat_fuse/7.10/html-single/fuse_on_openshift_guide/index#create-and-deploy-applications

For more information, refer to the *OpenShift Maven plugin* appendix in the *Fuse on OpenShift Guide* at https://access.redhat.com/documentation/en-us/red_hat_fuse/7.10/html-single/fuse_on_openshift_guide/index#openshift-maven-plugin

Guided Exercise

Deploying Camel Applications to Red Hat OpenShift

In this exercise, you will deploy a Spring Boot Camel application to the Red Hat OpenShift Container Platform (RHOCP) with health checks.

You must deploy a Spring Boot Camel application called `temperatures-route`, which exposes a set of temperature measurements in the Fahrenheit scale. The measurements originally come from a Node.js gateway service, called `temperatures-celsius-app`, which serves Celsius values gathered from temperature sensor devices. The `temperatures-route` application implements Camel routes that read the Celsius values from `temperatures-celsius-app`, converts them to Fahrenheit values, and exposes the values through a REST endpoint.

The `temperatures-celsius-app` service is already deployed in the RHOCP cluster. You must deploy the `temperatures-route` application. You must also configure a readiness health check to verify that the Camel route is up and running.

Outcomes

You should be able to deploy a Spring Boot Camel application to RHOCP, and configure health checks for a Camel application.

The solution files for this exercise are in the `AD221-apps` repository, within the `cloud-deploy/solutions` directory.

Before You Begin

To perform this exercise, ensure you have the following:

- The `AD221-apps` repository cloned in your workstation.
- Access to a configured and running RHOCP cluster.
- The RHOCP CLI (`oc`) installed.

From your workspace directory, use the `lab` command to prepare your system for this exercise.

```
[student@workstation AD221]$ lab start cloud-deploy
```

The `lab` command copies the exercise files from the `~/AD221/AD221-apps/cloud-deploy/apps` directory, into the `~/AD221/cloud-deploy` directory. This command also logs you in to the RHOCP cluster and deploys the `temperatures-celsius-app` service.

Instructions

1. Verify that the `temperatures-celsius-app` service is deployed and returning temperature values.

    1.1. Get the route of the service by running the following command:

```
[student@workstation AD221]$ oc get route temperatures-celsius-app \
-o jsonpath="{'http://'}{.spec.host}{'/temperatures\n'}"
...output omitted...
```

1.2. Send a GET request to the URL returned by the preceding command. Verify that the `temperatures-celsius-app` service returns a list of temperature values. You can use the `curl` command for this.

```
[student@workstation AD221]$] curl http://temperatures-celsius-app-user-cloud-
deploy.apps.cluster.example.com/temperatures
...output omitted...
```

2. Simulate a scenario in which the `temperatures-celsius-app` service is not available.

2.1. Scale down the deployment to zero replicas to simulate that the service is not ready. Use the `oc scale deployment temperatures-celsius-app --replicas=0` command for this.

2.2. Make another request to the service to verify that the application is not available.

3. Navigate to the `~/AD221/cloud-deploy` directory, and open the project with your editor of choice. Inspect the Camel routes in the `TemperaturesRESTRouteBuilder` class.

The builder class implements a route that reads Celsius temperature values from the `temperatures-celsius-app` service, converts them to Fahrenheit, and exposes the result through the `/camel/temperatures/fahrenheit` REST endpoint.

4. Configure the OpenShift deployment in the POM file, by using the `openshift-maven-plugin` plug-in.

Open the project's POM file and uncomment the `openshift` profile. Inspect the profile. Note that the profile uses the `openshift-maven-plugin` plug-in from JKube, and the `fuse-java-openshift-jdk11-rhel8` image for the S2I deployment.

5. Run the `./mvnw oc:deploy -Popenshift` command to deploy the `temperatures-route` application to RHOCP. Wait for the application to be deployed. The deployment might take several minutes. You can verify the status of the deployment with the `oc status` command.

6. Verify that the Camel route of the `temperatures-route` application fails.

6.1. Get the URL of the `/camel/temperatures/fahrenheit` REST path of the application, by running the following command.

```
[student@workstation cloud-deploy]$ oc get route temperatures-route \
-o jsonpath="{'http://'}{.spec.host}{'/camel/temperatures/fahrenheit\n'}"
...output omitted...
```

6.2. Send a request to the URL of the REST endpoint. Notice that the Camel route returns an exception. You might get either a `HttpHostConnectException` or a `ConnectTimeoutException` error.

```
[student@workstation cloud-deploy]$ curl http://temperatures-route-user-cloud-
deploy.apps.cluster.example.com/camel/temperatures/fahrenheit
org.apache.http.conn.HttpHostConnectException: Connect to temperatures-celsius-
app:3000
...output omitted...
```

RHOCP is sending traffic to the Spring Boot application, even though the Camel route is not healthy.

7. Update the application code to expose the readiness status of the Camel route through an HTTP endpoint.

   7.1. Inspect the `application.properties` file. The readiness group includes the `camelRoute` health check. Invoking the `/actuator/health/readiness` endpoint means that the `camelRoute` health check is executed.

   7.2. Inspect the `CamelRouteHealthIndicator` class, which defines the `camelRoute` health check. This class uses the `RouteHealth` bean to get the health status of the Camel route.

   7.3. Inspect the `RouteHealth` class. This class implements the `up` and `down` methods. You must use these methods from the Camel route to set the health state of the route.

   7.4. Edit the `TemperaturesRESTRouteBuilder` class. In the `processException` route, after the `process` call, use the `down` method of the `route-health` bean to set the health status to `down`.

   7.5. In the `celsiusToFahrenheit` route, use the Wire Tap EIP to call the `up` method of the `route-health` bean.

   7.6. Configure the container readiness check.

   Edit the `src/main/jkube/deployment.yaml` file. Change the path of the readiness probe to check the `/actuator/health/readiness` HTTP path.

8. Redeploy the `temperatures-route` application. Wait until the application is redeployed, and send a request to the `/camel/temperatures/fahrenheit` path of the application.

   Verify that RHOCP is not sending traffic to the application because the probe detected that the application is not ready.

9. Verify that the application pod is set as `Unhealthy`. Run the `oc describe pod -l app=temperatures-route` command to verify that the readiness probe failed.

```
[student@workstation cloud-deploy]$ oc describe pod -l app=temperatures-route
...output omitted...
Events:
  Type     Reason      Age        From         Message
  ----     ------      ----       ----         -------
  ...
  Warning  Unhealthy   16s        kubelet...   Readiness probe failed...
```

10. Activate the `temperatures-celsius-app` service. Run the `oc scale deployment temperatures-celsius-app --replicas=1` command.

11. Verify that the `temperatures-route` application is ready. Send a request to the `/camel/temperatures/fahrenheit` REST endpoint of the application and verify that the response contains a list of temperatures. You might need to wait a few seconds until the readiness probe succeeds and the application state is set as `Healthy`.

Finish

Return to your workspace directory and use the `lab` command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation AD221]$ lab finish cloud-deploy
```

This concludes the guided exercise.

Integrating Cloud-native Services Using Camel Quarkus

Objectives

After completing this section, you should be able to develop Camel routes with Quarkus.

Describing Apache Camel Quarkus

Apache Camel Quarkus adds the integration capabilities and component library of Apache Camel to the Quarkus runtime.

The benefits of using Camel Quarkus include the following:

- Takes advantage of the performance benefits, developer joy, and the container first ethos provided by Quarkus.
- Takes advantage of the performance improvements made in Camel 3, which results in a lower memory footprint, less reliance on reflection, and faster startup times.

Developing Camel Quarkus Applications

The process of developing Camel Quarkus applications is similar to the process you follow when you use the Spring Boot framework.

Create the Skeleton

You can use the Quarkus online project generator located at https://code.quarkus.redhat.com to generate a Quarkus Maven project. In this project generator, all the Camel Quarkus extensions are under the **Integration** category. Another option to generate the skeleton project is to use the Quarkus Maven plug-in.

Define Camel Routes

The most common way of defining Camel routes is by extending the `org.apache.camel.builder.RouteBuilder` class.

```
import org.apache.camel.builder.RouteBuilder;

public class QuarkusRoute extends RouteBuilder {

    @Override
    public void configure() throws Exception {
        from("schema:origin")
            .log("Hello World");
    }
}
```

The `camel-quarkus-core` artifact contains builder methods for all Camel components. You still need to add the component's extension as a dependency for the route to work properly. You can tune Camel Quarkus extensions by using `quarkus.camel.*` properties.

Camel Quarkus automatically configures and deploys a Camel Context bean. By default, the lifecycle of this bean is tied to the Quarkus application lifecycle.

Note
To test routes in the context of Quarkus, the recommendation is to write integration tests.

Deploying to Red Hat OpenShift Container Platform

You can deploy your application to Red Hat OpenShift Container Platform (RHOCP) by using any of the following build strategies:

- Docker build
- Source to Image (S2I)
- S2I Binary

Adding Health and Liveness Checks

Health checks are methods to determine the state of an application from another machine. In cloud environments, health checks determine when a computing node requires intervention. For example, replacing a computing node with a new healthy instance.

The `camel-quarkus-microprofile-health` extension provides support for health and liveness checks.

You can create custom heath check implementations. Any checks provided by your application are automatically discovered, and bound to the Camel registry. They are available in the `/q/health/live`, and `/q/health/ready` Quarkus health endpoints.

The following checks are automatically registered for your application.

Camel Context Health
Inspects the Camel Context status. If the context status is anything other than `Started`, then it sets the health check status to `DOWN`.

Camel Route Health
Inspects the status of each route. If any route status is not `Started`, then it sets the health check status to `DOWN`.

Deploying to RHOCP

As a developer, you can deploy your Quarkus applications to RHOCP by using a Maven command. This functionality is provided by the `quarkus-openshift` extension, which supports multiple deployment options. The default deployment option is S2I, but you can configure the deployment strategy by using the `quarkus.openshift.build-strategy` property.

Prior to the deployment, you are required to use the OpenShift CLI to log in, and select the project in which you want to deploy your application.

The following example packages and deploys your Quarkus application to the current RHOCP project:

```
[user@host project]$ ./mvnw clean package -Dquarkus.kubernetes.deploy=true
```

References

Red Hat Build of Quarkus Online Project Generator
https://code.quarkus.redhat.com/

For more information, refer to the *Extensions Overview* chapter in the *Camel Extensions for Quarkus Reference* at
https://access.redhat.com/documentation/en-us/red_hat_integration/2021.q4/html-single/camel_extensions_for_quarkus_reference/index#camel-quarkus-extensions-overview

Guided Exercise

Integrating Cloud-native Services Using Camel Quarkus

In this exercise, you will develop a Quarkus application that uses Camel to route messages for a book publishing company.

The company stores the books as DocBook files, and uses a shared file system for the publishing process. A team of editors and graphic designers review the book manuscripts before they are ready for printing.

The company currently publishes technical, and novel books. Editors review all types of books, and graphic designers only the technical ones.

Selecting the books to review for each one of the teams is a repetitive, manual, and time-consuming task. You must use Red Hat Fuse, and create a Camel route to send the correct type of book to the correct team.

The company also requires you to expose the books assigned to each one of the teams in REST endpoints. You must deploy the Quarkus application to Red Hat OpenShift Container Platform (RHOCP).

Outcomes

You should be able to create routes in Quarkus, add health checks, and deploy the application to RHOCP.

The solution files for this exercise are in the `AD221-apps` repository, within the `cloud-quarkus/solutions` directory.

Before You Begin

To perform this exercise, ensure you have the following:

- The `AD221-apps` repository cloned in your workstation.
- Access to a configured and running RHOCP cluster.
- The OpenShift CLI (`oc`) installed.

From your workspace directory, use the `lab` command to start this exercise.

```
[student@workstation AD221]$ lab start cloud-quarkus
```

The `lab` command copies the exercise files from the `~/AD221/AD221-apps/cloud-quarkus/apps` directory, into the `~/AD221/cloud-quarkus` directory. It also verifies the connection to the RHOCP shared cluster, and creates a project to deploy the application.

Instructions

1. Navigate to the `~/AD221/cloud-quarkus` directory, open the project with your editor of choice, and examine the code.

2. Open the project's POM file, and add the following dependencies from the `org.apache.camel.quarkus` group:

   - `camel-quarkus-bean`
   - `camel-quarkus-file`
   - `camel-quarkus-jackson`
   - `camel-quarkus-jacksonxml`
   - `camel-quarkus-rest`
   - `camel-quarkus-xpath`

3. Create a route for the book review pipeline in the `BookReviewPipelineRouteBuilder` class.

   The route must collect all the books located in the `file://data/manuscripts` endpoint, and use the Routing Slip EIP. Use the `RoutingSlipStrategy` bean to set the routing slip `destination` header, set `book-review-pipeline` as the route ID, and use the `noop` option in the consumer endpoint. You must record the file processing progress by logging a message with the following format: `"File: ${header.CamelFileName} - Destination: ${header.%s}"`.

4. Create a route in the `EditorRestRouteBuilder` class to store all the books assigned to the editors into an in-memory storage.

   The route must collect all the books located in the `file://data/pipeline/editor` endpoint, with the `noop` option activated. Next, unmarshal the data, and convert the unmarshalled data to JSON with the `jacksonxml` processor. Process the JSON data, and store it as an `Object` type in the `inMemoryBooksForEditor` variable. You must set `pipeline-editor` as the route ID.

5. Create a Camel REST route in the `EditorRestRouteBuilder` class, and expose the books assigned to the editors.

   The route must define a GET endpoint in the `/pipeline/editor` path. Return the content of the `inMemoryBooksForEditor` variable in the response body, and set `rest-pipeline-editor` as the route ID.

6. Verify the correctness of the routes by executing the unit tests.

   Run the `./mvnw clean -Dtest=EditorPipelineEndpointTest test` command, and verify that one unit test passes.

7. Run the Quarkus application by using the `./mvnw -DskipTests clean package quarkus:dev` command, and wait for the application to process the books stored in the `file://data/manuscripts` endpoint.

   Open a browser window, and navigate to `http://localhost:8080/pipeline/editor`. Notice that the application returns a JSON with the three books assigned to the editors.

8. Open the project's POM file, and add the following dependencies to deploy the application to RHOCP.

Group ID	Artifact ID
`org.apache.camel.quarkus`	`camel-quarkus-microprofile-health`
`io.quarkus`	`quarkus-openshift`

9. Wait for Quarkus to perform a live reload, and verify the status of the application in the health check endpoint.

   In a browser window, navigate to `http://localhost:8080/q/health`. Notice that the application returns a JSON response with the `status` field set to `UP`, and stop the application.

10. Verify the correctness of the complete application by executing the unit tests.

    Run the `./mvnw clean test` command, and verify that three unit tests pass.

11. Run the `./mvnw clean package -Dquarkus.kubernetes.deploy=true` command to deploy the Quarkus application to RHOCP. Wait for the RHOCP deployment process to finish.

12. By using the `oc describe pod` command, verify that the deployed application is running, and it has the health checks configured.

```
[student@workstation AD221]$ oc describe pod \
-l app.kubernetes.io/name=book-publishing
Name:         book-publishing-2-7twcn
...content omitted...
Status:       Running
...content omitted...
Containers:
  book-publishing:
    ...content omitted...
    Liveness:   http-get http://:8080/q/health/live ...
    Readiness:  http-get http://:8080/q/health/ready ...
    ...content omitted...
```

Finish

Return to your workspace directory, and use the `lab` command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation AD221]$ lab finish cloud-quarkus
```

This concludes the guided exercise.

Integrating Cloud-native Services Using Camel K

Objectives

After completing this section, you should be able to create cloud-native integrations with Camel K.

Describing Apache Camel K

Apache Camel K is a lightweight integration framework built from Apache Camel. With this framework, developers can build serverless, and microservice architectures. It runs natively in the cloud on Red Hat OpenShift, and uses the Kubernetes Operator SDK to automate the deployment process of your integrations.

The Camel K runtime is a Java application based on Camel Quarkus. The main goal of the runtime is to run a Camel Quarkus application, and configure the routes defined by the developer.

With Camel K, you write your integration code in a single file, and use the Camel K CLI to run the code immediately in the cloud. As a developer, you concentrate on developing only the integration source code; minimizing the costs of maintaining a complete application, and reducing the deployment complexity.

Describing Kamelets

Kamelets are an alternative approach to application integration. They are reusable route components, implemented as Kubernetes resources, you can use to connect to external systems.

There are three types of Kamelets.

Source
Consumes data from an external system.

Sink
Sends data to an external system.

Action
Executes a specific action to manipulate data while it passes from a source to a sink Kamelet.

Developing Camel K Applications

Camel K supports multiple languages for writing your integrations, such as Java, XML, and YAML. Writing an integration in Java is the same as defining your routing rules in Camel. You can use any Camel component directly in your integration routes.

Camel K automatically handles the dependency management, and imports the required libraries by using code inspection. With Camel K, you do not need to build, and package your integration.

Important
The automatic resolution of dependencies only works with dependencies from the Camel catalog (`camel-*` artifacts), and with routes that do not specify dynamic URIs.

Defining Camel Routes

The most common way of defining Camel routes is by coding your integration in a single file. In this file, you must create a class that extends from the `org.apache.camel.builder.RouteBuilder` class.

```
import org.apache.camel.builder.RouteBuilder;

public class CamelKRoute extends RouteBuilder {

    @Override
    public void configure() throws Exception {
        from("schema:origin")
            .log("Hello World");
    }
}
```

Running Integrations on Red Hat OpenShift

The Camel K CLI (`kamel`) is the main entry point for running Camel K integrations on OpenShift.

The following table is a non-comprehensive list of commands supported by the Camel K CLI.

Camel K CLI Commands

Command	Description
`run`	Runs an integration in OpenShift
`delete`	Deletes an integration deployed to OpenShift
`describe`	Gets detailed information about a Camel K resource
`get`	Gets the status of the integrations deployed to OpenShift

You can run a Camel K integration in developer mode by adding the `--dev` parameter to the `run` command.

```
[user@host project]$ kamel run MyIntegration.java --dev
```

The preceding example runs the integration defined in the `MyIntegration.java` file in developer mode. The developer mode watches the file for changes, and automatically refreshes the integration deployed in the cloud. That means that you can make live updates to the integration code, and view the results instantly.

Integration Configuration

There are two configuration phases in a Camel K integration life cycle: build time, and runtime. You can provide configuration values to the `kamel run` command to customize the different phases.

Build time

Use the `--build-property` option to provide the property values required in the build process.

Runtime

Use the `--property`, `--config`, or `--resource` options to provide the property values required when the integration is running.

Traits are high-level features of Camel K that you can configure to customize the behaviour of your integration. You can define traits by using the `--trait` option on the `kamel run` command.

Camel K Modeline

Camel K modeline is a feature that processes integration options defined in code comments. The following table is a non-comprehensive list of modeline options.

Camel K Modeline Options

Option	Description
`config`	Adds runtime configuration from a ConfigMap, a Secret, or a file
`dependency`	Includes an explicit external library
`env`	Sets an environment variable in the integration container
`resource`	Adds a runtime resource from a ConfigMap, a Secret, or a file

The following example uses modeline hooks to customize a Java integration.

```
// camel-k: dependency=camel-quarkus-jacksonxml (1)
// camel-k: resource=file:./path/to/file.txt (2)

import org.apache.camel.builder.RouteBuilder;

public class CamelKRoute extends RouteBuilder {

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

(1) Includes the `camel-quarkus-jacksonxml` library

(2) Adds the `file.txt` file to the integration

In an initial phase, the `kamel run` command inspects the integration file to detect modeline options. Then, it transforms the modeline options into arguments that are later executed by the `run` command.

Differences with Apache Camel

Although Apache Camel K is built from Apache Camel and runs Camel integrations, Camel K presents significant differences with Camel:

Apache Camel compared to Apache Camel K

Apache Camel	Apache Camel K
Runs on popular Java frameworks	Runs natively on the cloud
Requires a Java application, and preferably, a framework such as Spring Boot or Quarkus	Requires an integration definition file
You must write integrations in Java	You can write the integration file in multiple languages, such as Java, Groovy, or JavaScript
Traditional Java development workflow	Cloud-native development workflow, tailored to serverless integration
Reusability based on Camel components	Reusability based on Camel components and Kamelets

Based on the context, scope, and constraints of your project, you might want to use either Camel or Camel K. The following is a non-comprehensive list of use cases and recommendations about when to use one or the other.

- You might want to use Apache Camel when working with traditional Java development workflows. This includes non-cloud-native projects and Java projects with a strong dependency on a specific runtime, such as Spring Boot.
- If you do not want to deal with a Java runtime, the details of a specific Java framework, or if you want to define integrations in a different language than Java, then use Camel K.
- Similarly, if you want to add Camel integration capabilities to other technology stacks, then use Camel K. An example of this is the Rayvens Python project [https://github.com/project-codeflare/rayvens].
- Camel is specifically designed for serverless applications. If you are comfortable with serverless and KNative, then you might want to consider Camel K. KNative allows you to optimize how Camel K integrations use cluster resources and other capabilities, such as eventing.

References

For more information, refer to the *Kamelets Reference* chapter in the *Red Hat Integration Guide* at
https://access.redhat.com/documentation/en-us/red_hat_integration/2021.q4/html-single/kamelets_reference/index

For more information, refer to the *Configuring Camel K Integrations* chapter in the *Developing and Managing Integrations Using Camel K Guide* at
https://access.redhat.com/documentation/en-us/red_hat_integration/2021.q4/html-single/developing_and_managing_integrations_using_camel_k/index#configuring-camel-k

For more information, refer to the *Camel K trait configuration reference* chapter in the *Developing and Managing Integrations Using Camel K Guide* at
https://access.redhat.com/documentation/en-us/red_hat_integration/2021.q4/html-single/developing_and_managing_integrations_using_camel_k/index#camel-k-traits-reference

Guided Exercise

Integrating Cloud-native Services Using Camel K

In this exercise, you will develop a Camel K integration for a book publishing company.

The company requires you to expose the book manuscripts in a REST endpoint. You must deploy the integration application to Red Hat OpenShift.

Outcomes

You should be able to:

- Deploy an integration in the cloud by using Camel K.
- Leverage Camel K development mode to receive feedback while developing.

The solution files for this exercise are in the `AD221-apps` repository, within the `cloud-camelk/solutions` directory.

Before You Begin

To perform this exercise, ensure you have the following:

- The `AD221-apps` repository cloned in your workstation.
- Access to a configured and running OpenShift cluster.
- The OpenShift CLI (`oc`) installed.
- The Camel K CLI (`kamel`) installed.

From your workspace directory, use the `lab` command to start this exercise.

```
[student@workstation AD221]$ lab start cloud-camelk
```

The `lab` command copies the exercise files from the `~/AD221/AD221-apps/cloud-camelk/apps` directory, into the `~/AD221/cloud-camelk` directory. It also verifies the connection to the OpenShift shared cluster, and creates a project to deploy the application.

Instructions

1. Navigate to the `~/AD221/cloud-camelk` directory, open the project with your editor of choice, and examine the code.

2. Run the Camel K application in develop mode by using the `kamel run ManuscriptsApi.java --dev` command. Wait for the deployment to finish.

```
...output omitted...
... [io.quarkus] (main) camel-k-integration 1.6.3 ... started in 2.707s. Listening
 on: http://0.0.0.0:8080
...output omitted...
```

Important

The initial Camel K deployment might take a few minutes to finish. Subsequent redeployments should be faster.

3. Open a new terminal and use the `oc get route` command to get the route assigned to the Camel K application.

```
[student@workstation cloud-camelk]$ oc get route -o \
jsonpath="{'http://'}{..spec.host}{'/manuscripts'}{'\n'}"
```

Open your web browser, and navigate to the REST endpoint. Verify that the application throws an internal server error because the integration has some missing parts.

4. Open the `ManuscriptsApi` class, and add the following dependencies:

   - `camel-quarkus-jackson`
   - `camel-quarkus-jacksonxml`

5. Add the the following runtime resources to the integration pod:

   - `./data/manuscripts/book-01.xml`
   - `./data/manuscripts/book-02.xml`

   Camel K copies your local runtime resources into the `/etc/camel/resources` directory of the integration pod container.

6. Create a route to store all the books assigned to the editors into an in-memory storage.

   The route must collect all the books located in the `file:/etc/camel/resources` endpoint, and unmarshal the data to convert it to the JSON format. Process the JSON data, and store it as an `Object` type in the `inMemoryBooks` variable. You must set `manuscripts` as the route ID, and use the `noop` option in the consumer endpoint.

Note

You can use the `log` method in the route to track the progress of the file processing.

7. Save the changes to trigger a new deployment of the Camel K application, and wait for the integration to process the book files.

```
...output omitted...
... Processing file: book-01.xml
... Processing file: book-02.xml
```

Important

The Camel K development mode command might hang if you introduce compilation errors. If this happens, then stop and restart the command.

8. Return to the browser, and reload the Camel K application URL. Notice that the application returns a JSON response with the information about two books.

Finish

Stop the Camel K application, return to your workspace directory, and use the `lab` command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises.

```
[student@workstation AD221]$ lab finish cloud-camelk
```

This concludes the guided exercise.

Quiz

Integrating Cloud-native Services

Consider that you have developed a Spring Boot Camel application to route weather condition measurements from garden sensors to a REST API. The application is ready to be deployed to production.

Choose the correct answers to the following questions:

1. Which two of the following are valid strategies to deploy the application to the Red Hat OpenShift Container Platform (RHOCP)? (Choose two.)

a. Run a Source-to-Image (S2I) build to deploy the application from the source code.

b. Run an S2I build to deploy the application from the source code. Next, build the container image, and push the image to a registry.

c. Generate a JAR file. Next, run an S2I build to deploy the application from the JAR file.

d. Run an S2I build to deploy the application from the source code. Next, build the container image, push the image to a registry, and create the deployment.

2. You are developing an additional integration in Camel K to route garden sensor data to a relational database. While you develop the integration in the `GardenToSql.java` file, you would like to run the route, show logs, and watch for changes to automatically refresh the integration as you add more code. Which command of the following should you use?

a. `kamel dev GardenToSql.java`

b. `kamel start GardenToSql.java`

c. `kamel start GardenToSql.java --dev`

d. `kamel run GardenToSql.java --dev`

3. The garden sensors lose network connection occasionally. When the sensors are unreachable, the Camel route becomes unavailable and throws connection errors. How can you configure the deployment so that, under these conditions, RHOCP marks the application as not `Ready`, without killing the application container?

a. Configure a startup probe.

b. Configure a readiness probe.

c. Configure a liveness probe.

d. Configure a container execution probe.

4. Your team is considering a switch to Camel Quarkus for the next version of the application. Which two of the following statements are true regarding Camel Quarkus? (Choose two.)

a. You do not need to add component dependencies to Quarkus. Quarkus handles all dependencies for you.

b. You can tune Camel Quarkus extensions by using `quarkus.camel.*` properties.

c. You must add the `camel-quarkus-core` artifact as a dependency of your Quarkus project.

d. Quarkus provides support for liveness probes only.

Solution

Integrating Cloud-native Services

Consider that you have developed a Spring Boot Camel application to route weather condition measurements from garden sensors to a REST API. The application is ready to be deployed to production.

Choose the correct answers to the following questions:

1. Which two of the following are valid strategies to deploy the application to the Red Hat OpenShift Container Platform (RHOCP)? (Choose two.)

a. Run a Source-to-Image (S2I) build to deploy the application from the source code.

b. Run an S2I build to deploy the application from the source code. Next, build the container image, and push the image to a registry.

c. Generate a JAR file. Next, run an S2I build to deploy the application from the JAR file.

d. Run an S2I build to deploy the application from the source code. Next, build the container image, push the image to a registry, and create the deployment.

2. You are developing an additional integration in Camel K to route garden sensor data to a relational database. While you develop the integration in the `GardenToSql.java` file, you would like to run the route, show logs, and watch for changes to automatically refresh the integration as you add more code. Which command of the following should you use?

a. `kamel dev GardenToSql.java`

b. `kamel start GardenToSql.java`

c. `kamel start GardenToSql.java --dev`

d. `kamel run GardenToSql.java --dev`

3. The garden sensors lose network connection occasionally. When the sensors are unreachable, the Camel route becomes unavailable and throws connection errors. How can you configure the deployment so that, under these conditions, RHOCP marks the application as not Ready, without killing the application container?

a. Configure a startup probe.

b. Configure a readiness probe.

c. Configure a liveness probe.

d. Configure a container execution probe.

4. Your team is considering a switch to Camel Quarkus for the next version of the application. Which two of the following statements are true regarding Camel Quarkus? (Choose two.)

a. You do not need to add component dependencies to Quarkus. Quarkus handles all dependencies for you.

b. You can tune Camel Quarkus extensions by using `quarkus.camel.*` properties.

c. You must add the `camel-quarkus-core` artifact as a dependency of your Quarkus project.

d. Quarkus provides support for liveness probes only.

Summary

In this chapter, you learned:

- You can deploy Spring Boot Camel applications to Red Hat OpenShift Container Platform (RHOCP) with the JKube `openshift-maven-plugin` plug-in.
- Developing Camel integrations in Quarkus is similar to developing Camel integrations in Spring Boot.
- You can deploy Camel Quarkus applications to RHOCP with the `quarkus-openshift` Maven extension.
- You can monitor the health of Camel routes and contexts both in Quarkus and Spring Boot applications.
- Camel K is a light, cloud-native integration platform, based on Camel.